I JORNADAS SOBRE PROGRAMACIÓN Y LENGUAJES

Editores: Fernando Orejas
          Fernando Cuartero
          Diego Cazorla
PRIMERAS JORNADAS SOBRE PROGRAMACIÓN Y LENGUAJES (PROLE 2001)

Almagro, 23 y 24 de Noviembre de 2001

Coordinador
Dr. D. Fernando Orejas

Organiza
Grupo Alarcos
Universidad de Castilla - La Mancha
ÍNDICE

A Demand Driven Strategy for Lazy Narrowing with Multisets  1
  M. Abengózar, P. Arenas, A. Gil

Computational properties of term rewriting with strategy annotations 17
  S. Lucas

A Proof of Correctness for the STG Machine 33
  A. de la Encina, R. Peña

Improving the Efficiency of Non-Deterministics Computations 49
  S. Antoy, P. Julián, B. Massey

Dynamic Programming of Partial Parses 63
  D. Cabrero, J. Vilares, M. Vilares

Applying Frameworks and Object-Oriented Techniques for developing
  Language Processors Tools 77
  M. C. Luengo, D. Basanta, J. E. Labra, N. García, J. M. Cueva

Language Prototyping in the Maude Metalanguage 93
  M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Olilet, J. Meseguer

Panta Rei (Everything Flows): A Calculus of Channels 111
  F. Arbab

INDALOG: A Declarative Deductive Database Language 113
  J. M. Almendros, A. Becerra

A declarative debugger of wrong answers for lazy functional logic pro-
  grams 129
  R. Caballero, M. Rodríguez Artalejo

Transforming Layered Specifications into Operational Specifications 145
  F. J. Galán, J. M. Cañete

Abstract Satisfiability of Linear Temporal Logic 163
  M. del Mar Gallardo, P. Merino, E. Pimentel

Declarative Reflection and its Application as a Pattern Language 179
  J. J. Moreno-Navarro, A. Herranz-Nieva

Un lenguaje modelo para la definición y análisis de refactorizaciones 199
  Y. Crespo, V. Cardeñoso, J. M. Marqués
El Visitante Genérico Extensible ........................................ 217
F. Moya, J. C. López

Generic Description, Behavior and Animation of Visual Modeling Languages ........................................ 227
H. Ehrig, R. Bardohl, C. Ernel

Modelling implementations as institution morphisms ........... 229
C. Domínguez, L. Lambán, V. Pascual, J.Rubio

Tight and loose semantics for transformation systems .......... 241
F. Orejas, H. Ehrig, E. Pino

A Comparison between three Non-determinism Analyses in a Parallel-Functional Language ....................... 263
R. Peña, C. Segura

Sincronización Multiproceso en Programas Concurrentes: Selección Complemente k-Justa de Interacciones ........ 279
D. Ruiz, R. Cordueto, J. A. Pérez, M. Toro

Compatibility of Linda Interfaces ................................. 295
A. Brogi, E. Pimentel, A. M. Roklán

A Pattern-based Language to Coordinate HPF Tasks .......... 309
M. Díaz, B. Rubio, E. Soler y J. M. Troya

Reasoning about Probabilistic and Nondeterministic Processes ... 325
D. Cazorla, F. Cuartero, V. Valero, F.L. Pelayo
INTRODUCCIÓN

Este volumen recoge los trabajos aceptados para su presentación en las Primeras Jornadas sobre Programación y Lenguajes (PROLE 2001) celebradas en Almagro los días 23 y 24 de Noviembre de 2001, bajo la organización del grupo Alarcos de la Universidad de Castilla - La Mancha. También recoge los resúmenes de las dos conferencias invitadas impartidas por los profesores Farhad Arbab (Amsterdam) y Hartmut Ehrig (Berlín).

La idea de la organización de estas Jornadas partió de Juano Moreno que me convenció para que las pusiera en marcha. El objetivo era crear un espacio de reunión de los grupos españoles que trabajan en temas ligados a la programación y a los lenguajes de programación. Con la organización de este evento, se ha pretendido que los, cada vez más numerosos, grupos que trabajan en España en este ámbito puedan conocerse mejor, intercambiar experiencias y resultados de investigación y, posiblemente, iniciar líneas de cooperación.

El ámbito establecido para PROLE comprende trabajo tanto teórico como práctico alrededor de la especificación, diseño, implementación, análisis y verificación de programas y lenguajes de programación. Más concretamente, una lista no exhaustiva de los temas de PROLE incluirá:

- Paradigmas de programación (funcional, lógico, concurrente, orientado a objetos, visual...) y su integración.
- Especificación y lenguajes de especificación - Semántica y su aplicación al diseño, análisis y verificación, transformación y aprendizaje de programas.
- Técnicas de análisis de programas - Compilación y herramientas de implementación de lenguajes de programación

En un tiempo relativamente escaso se organizó un Comité de Programa formado por Jesús Almendros (Universidad de Almería), María Alpueyte Universidad Politécnica de Valencia), Javier Esparza (Universidad de Edimburgo) Rafael Corchuelo (Universidad de Sevilla), Fernando Cuartero (Universidad de Castilla-La Mancha), Juan Manuel Cueva (Universidad de Oviedo), David de Frutos (Universidad Complutense), Manuel Hermenegildo (Universidad Politécnica de
Madrid], Juan Hernández (Universidad de Extremadura), Pascual Julián Iranzo (Universidad de Castilla-La Mancha), Jordi Levy IIIA-CSIC), Jim Lipton (Wesleyan University), Juanjo Moreno (Universidad Politécnica de Madrid), Marisa Navarro (Universidad del País Vasco), Robert Nieuwenhuis (Universidad Politécnica de Cataluña), Fernando Orejas (Universidad Politécnica de Cataluña), Ricardo Peña (Universidad Complutense), Ernesto Pimentel (Universidad de Málaga), Isidro Ramos (Universidad Politécnica de Valencia), Mario Rodríguez Artalejo (Universidad Complutense), Ambrosio Tosal (Universidad de Murcia) y Manuel Vilares (Universidad de Coruña). A continuación se difundió una llamada para el envío de trabajos, que tuvo considerable éxito, dado el poco tiempo disponible, y se seleccionaron los trabajos que incluye este volumen.

Estas jornadas se realizan conjuntamente con varios eventos, entre los que tiene especial relevancia las, ya consolidadas, VI Jornadas de Ingeniería del Software y Bases de Datos (JISBD’2001). Consideramos que la considerable intersección común, en temas y personas, de ambas Jornadas apoya su realización conjunta. Esperamos que PROLE pueda tener continuidad en el futuro, manteniendo su organización conjunta con las JISBD.

Deseo agradecer, especialmente, al Grupo Alarcos haber aceptado incluir a PROLE en el paquete de eventos que organizan, a Fernando Cuartero su colaboración en la edición e impresión de estas actas.

Fernando Orejas
Presidente del Comité de Programa de PROLE 2001
A Demand Driven Strategy for Lazy Narrowing with Multisets*

M. Abengózar-Carreros, P. Arenas-Sánchez, A. Gil-Luezas

Dept. Sistemas Informáticos y Prog. (U. Complutense de Madrid)
Facultad de CC. Matemáticas, Av. Complutense s/n, 28040 Madrid, Spain
email: {mercedes.puri,ana.gil}@sip.ucm.es

Abstract. In recent works, a lazy functional logic language with multisets and constraints, named SETA, was described, including a Prolog-based implementation and a well-defined declarative and operational semantics. In the present paper, we have changed the current naive lazy narrowing strategy implemented for SETA, incorporating a new demand driven strategy which avoids, in some cases, unnecessary computations, improving the behaviour of the strategy. Furthermore, we have developed a complete Prolog-based implementation of the language and we have incorporated it into the lazy functional logic system TCO, comparing the old SETA implementation with the new one. Experimental results have shown good performance.

Keywords: Functional logic programming, lazy narrowing, multisets, implementation.

1 Introduction

The integration of Functional and Logic Programming (FLP in short) aims to combine into a single paradigm the nicest properties of both functional and Logic Programming (see [13] for a survey). In many approaches to FLP programs are seen as constructor-based conditional rewrite systems. This is the case of the CRWL-framework presented in [11], where classical equational logic is replaced by a suitable constructor-based rewriting logic which expresses properly the semantics of reduction for lazy, partial and possibly non-deterministic functions.

Most approaches to FLP, including CRWL, are based on free constructors. However, in some cases the use of non-free constructors (in particular multisets) is very suitable to describe some kind of problems, such as action and change problems [12], [20] or the General Abstract Model for Multiset Manipulation (GAMMA) [8]. In [7], authors extend the CRWL-framework incorporating algebraic polymorphic datatypes. The new resulting framework, named ACRWL, allows to deal with any kind of non-free constructors (including multisets). However, like CRWL, ACRWL only supports strict equalities as possible constraints.

The language SETA, firstly proposed in [5], is an attempt to integrate lazy functional logic programming with multisets, as well as various symbolic constraints over datatypes (equality, disequality, membership and non-membership). More concretely, the work [5] describes an executable Prolog specification of the operational semantics of SETA, which can be executed in any Prolog system with the ability to solve simple arithmetic constraints (needed to solve cardinality constraints). The operational and declarative semantics of SETA, extended with arithmetic constraints, was presented in [6], where authors shown that SETA is in fact an extended instance of ACRWL. It is an instance in the sense that

* This research has been partially supported by the the Spanish CICYT (project CICYT-TIC98-0445-C03-02/97 "TREND").
the multiset constructor is the unique non-free constructor considered, but it is an extension since it incorporates primitive built-in data (the real numbers) and constraints including equality, disequality, membership and non-membership (CRWL and ACRWL only support strict equality). Some other interesting extensions of logic programming with (multi) sets are [20], [10], [15], [16], which use set unification as a particular instance of unification modulo equations [14]; but no concern on programming with lazy functions is found in these proposals. With respect to functional logic programming, we can find the higher-order language Escher [17], where sets and multisets can be represented as functions rather than using data constructors (as in SETA). The advantages of each representation are problem dependent (e.g., see [17]).

The aim of the present paper is to describe a new lazy narrowing strategy for the language SETA, more lazy and efficient than the presented in [5]. In [5] authors use a naive lazy narrowing strategy, similar to that presented in [18], but changing the unification in order to take into account the commutativity axiom for multisets. As commented in [18], [5], the naive strategy performs a lot of redundant work, even more in presence of multisets, where the existence of a unique meet general unifier is lost. What we have done in the present paper is to replace the naive strategy by a demand driven strategy which improves substantially the behaviour of the language. The lazy narrowing strategy that we present uses ideas in the spirit of needed narrowing [18], [2], [3] but with important modifications. In fact, the demand driven strategy in [18], modifying exclusively the unification to take into account multisets, is as bad as the naive. We have also extended the syntax of SETA including cardinality constraints, which are really needed in order to detect failures and successes as soon as possible, without evaluating any expression. Although, due to the lack of space, in this paper we focus on the description of a demand driven strategy for lazy narrowing with multisets, we have also developed a complete Prolog-based implementation of the language, incorporating it into the lazy functional logic system TOY [19]. In fact, we have three different implementations, one supporting the naive strategy described in [5], other with the demand driven strategy of [18], [2], [3] modifying the concept of unification, and the last with the demand driven strategy for multisets presented here. Experimental results have shown good performance of the last one.

The rest of the paper is organized as follows: Sect. 2 describes the language SETA as presented in [5], including a simple but illustrative example. Sect. 3 presents the demand driven strategy for lazy narrowing with multisets. Sect. 4 comments very briefly on the Prolog-based implementation for SETA and shows several experimental results. Some conclusions are pointed in Sect. 5.

2 The language SETA

Our presentation of SETA follows exactly the lines of [5]. In the current implementation, cardinality constraints have been incorporated to close SETA’s syntax to computed answers (see next section).

We assume a countable set \( TV \) of type variables \( A, B, \ldots, \) etc. and a countable ranked alphabet \( TC = \bigcup_{n \geq 0} TC^n \) of type constructors, including the multiset type constructor \( mset \in TC^1 \). Polymorphic types \( \tau, \tau^* \in T_{TC}(TV) \) are built from \( TV \) and \( TC \). In the following \( \text{var}(\tau) \) stands for the set of type variables occurring in \( \tau \). We define a polymorphic signature \( \Sigma \) over \( TC \) as \( \langle TC, DC, FS, PS \rangle \), where:
\[ DC \text{ is a set of type declarations for data constructors, of the form } c : (\tau_1, \ldots, \tau_n) \rightarrow \tau_0, \text{ with } \bigcup_{i=1}^{n} tvar(\tau_i) \subseteq tvar(\tau_0). \text{ We assume that } DC \text{ contains the type declarations } \{ \} : mset(A) \text{ (representing the empty multiset), and } (A, mset(A)) : mset(A) \text{ (representing the multiset constructor).} \]

The multiset constructor is governed by the equation \( (\text{MSET}) : [X, Y | Xs] \cong [Y, X | Xs] \). Here, we have used \( [X, Y | Xs] \) as an abbreviation for \( [X][Y | Xs] \). In the sequel we will continue using such notation.

\[ FS \text{ is a set of type declarations for function symbols, of the form } f : (\tau_1, \ldots, \tau_n) \rightarrow \tau. \]

\[ PS \equiv \{ =, \neq, (A, A), \in, \notin: (A, mset(A)) \} \text{ is a set of type declarations for predicate symbols. } = \text{ and } \neq \text{ stand for strict equality and disequality respectively; whereas } \in, \notin \text{ represent membership and non-membership respectively.} \]

We will write \( h \in DC^o \cup FS^o \) to indicate the arity of a symbol according to its type declaration. Assuming another countable set \( DV \) of data variables \( X, Y, \ldots \), expressions \( e, r, l, \ldots \in E^o_\Sigma(DV) \) are built from \( DV, DC \) and \( FS \). Terms \( T^o_\Sigma(DV) \subseteq E^o_\Sigma(DV) \) are built by using only \( DV \) and \( DC \). A constraint \( \varphi \in R^o_\Sigma(DV) \) is defined as \( \varphi := e_1 \neq e_2 \), where \( e_1, e_2 \in E^o_\Sigma(DV) \) and \( \in \equiv \{ =, \neq, \in, \notin \} \).

A program rule for \( f \in FS^o \) has the form: \( f(t_1, \ldots, t_n) = r \equiv \varphi_1, \ldots, \varphi_m \), where the left-hand side is linear (i.e., without multiple occurrences of variables), \( t_i \in T^o_\Sigma(DV) \), \( 1 \leq i \leq n \), \( r \in E^o_\Sigma(DV) \), all variables occurring in \( r \) occur also in the left-hand side and \( \varphi_j \in R^o_\Sigma(DV) \), \( 1 \leq j \leq m \). Program rules must be well-typed in the usual way; see e.g. [5]. We define programs as pairs \( P = (\Sigma, \mathcal{R}) \), where \( \Sigma \) is a polymorphic signature and \( \mathcal{R} \) is a finite set of program rules for defined function symbols in \( FS \). Programs must be well-typed, i.e., all program rules must be well-typed.

Goals \( G \) have the form: \( \varphi_1, \ldots, \varphi_m \), where \( \varphi_j \in R^o_\Sigma(DV) \), \( 1 \leq j \leq m \), and must be well-typed in the usual way; see e.g. [5].

A solution for a goal is a pair \( \langle SS, \phi \rangle \), where \( SS = \{ X_1 = t_1, \ldots, X_n = t_n \} \) is a system of equations in solved form, i.e., each variable \( X_i \) occurs only once in \( SS \), \( t_i \in T^o_\Sigma(DV) \), \( 1 \leq i \leq n \), and \( \phi \) is a set of constraints in solved form. There are two kinds of constraints in solved form: \( X \not= t \) (with \( t \) different from \( X \)) and \( t \not\in Xs \). SETA's implementation is designed in such a way that \( SS \) is computed by Prolog's unification, whereas \( \phi \) is computed by using a constraint store (which is updated by the proper implementation).

**Example 1.** As an example, consider \( TC = \{ \text{action}^0, \text{bool}^0 \} \), \( DC = \{ \text{sleep, study, work, doShopping} : \rightarrow \text{action, true, false} : \rightarrow \text{bool} \} \) and the function symbols \( FS = \{ \text{asBusy : mset(action), mset(action) : } \rightarrow \text{bool} \} \). A multiset of actions represents the daily tasks of a person. The function \( \text{asBusy}(M, N) \) below checks if \( M \) has to do at least the same tasks as \( N \). Of course, the action of \text{sleeping} is not considered. The code for function \text{asBusy} is the following:

\[
\begin{align*}
(R1) \ & \text{asBusy(Xs, } \{ \} ) = \text{true} \\
(R2) \ & \text{asBusy(Xs, } \{ \text{sleep} | Ys \} ) = \text{asBusy(Xs, } Ys) \\
(R3) \ & \text{asBusy(} \{ X | Xs \}, \{ Y | Ys \} ) = \text{asBusy(Xs, } Ys) \iff X = Y, X \not= \text{sleep}
\end{align*}
\]

A possible goal for this program would be:

\[
Y \notin \{ X | Ys \}, \text{asBusy(} \{ X | Ys \}, \{ \text{sleep, work, doShopping} \} ) = \text{true},
\]

The intended meaning of \( \{ X | Xs \} \) is to add a new copy of the element \( X \) to the multiset \( Xs \).
for which we expect, among others, the solutions:

\[
\begin{align*}
\{ & X = \text{work}, \ Y = \{ \text{doShopping} | \text{Z} \} \}, \ \{ \ Y \neq \text{work}, \ Y \neq \text{doShopping}, \ Y \neq \text{Z} \} \\
\{ & X = \text{doShopping}, \ Y = \{ \text{work} | \text{Z} \} \}, \ \{ \ Y \neq \text{work}, \ Y \neq \text{doShopping}, \ Y \neq \text{Z} \} \\
\{ & Y = \{ \text{work}, \ \text{doShopping} | \text{Z} \} \}, \ \{ \ Y \neq \text{work}, \ Y \neq \text{doShopping}, \ Y \neq \text{Z} \}
\end{align*}
\]

Note that, because of the axiom \((\text{Mset})\), we get different solutions by binding \(X\) and \(Y\) conveniently. Note also that there are more solutions coming from considering \textit{sleep} as a possible element of \(Y\). In fact, there are infinite solutions. 

\[\blacksquare\]

3 A Demand Driven Strategy for \textsc{seta}

This section describes a Demand Driven Strategy \((\text{ddsm}, \text{for short})\) to deal with multiset unification, comparing this strategy with other existing ones. First of all, note that as in AC1-unification (see \cite{14}), the unification between two terms involving multisets may have several incomparable unifiers, even for very simple unification problems, as e.g. \(\{ X, Y \} = \{ a, b \} \) (where \(X, Y\) are different variables and \(a, b\) different constant symbols), which has two different unifiers \(X = a, Y = b\) and \(X = b, Y = a\). Another difficulty is related to \textit{pattern matching}. For instance, matching a given expression \(e\) with a multiset pattern such as \(\{ a | Xs \} \) cannot be solved simply by reducing \(e\) to \textit{head normal form} (i.e., variable or outermost constructor). Imagine that \(e\) happens to reduce to the expression \(\{ b, a | Ys \}\). This shows that lazy evaluation is much harder to implement in presence of multisets. Nevertheless, laziness must be preserved as much as possible.

In \cite{5} the authors present a Prolog-based implementation for \textsc{seta}, based on the idea of translating multisets into Prolog terms which keep information about their cardinalities. Such a representation turns out very useful when solving equality or disequality constraints since allows to detect failures and successes without evaluating. As an example, a disequality of the form \(\{ f, f \} \neq \{ f \} \), where \(f\) is a function symbol defined by the rule \(f = f\) (diverges) succeeds because of discrepancy of cardinalities (\(f\) is not evaluated). However, the proposal of \cite{5} has two important drawbacks. Firstly, there is a disagreement between \textsc{seta}'s syntax and computed answers. The reason is that multiset representation in Prolog, although ingenious, forces to use constraints about cardinalities which are solved by a constraint solver (more concretely, by the solver for real constraints of Sistets Prolog \cite{21}). Hence, computed answers contain cardinality constraints, not allowed in \textsc{seta}'s syntax. As an example, a goal of the form \(\{ X | Xs \} \neq \{ a, b \} \) would compute the solution \(\{ Xs = Ys : L \}, \{ L \neq 1 \}\), expressing that both multisets have different cardinality. As shown by the example, the answer contains multiset variables \(\{ Ys : L \}\) and real constraints \(\{ L \neq 1 \}\) not allowed in \textsc{seta}'s syntax. Secondly, the lazy narrowing strategy they use (i.e., the way of translating program rules to Prolog) is very inefficient, even without using multisets. More concretely, in \cite{5} authors implement the naive lazy narrowing strategy presented firstly in \cite{18}, but changing the process of unification to consider multisets. As a result, any program rule of the form \(f(t_1, \ldots, t_n) = r \iff \varphi\) is translated\(^2\) into the prolog clause:

\[
\begin{align*}
& f(X_1, \ldots, X_n, H) : \neg \text{unify}(X_1, t_1), \ldots, \neg \text{unify}(X_n, t_n), \text{solve}(\varphi), \text{hnf}(r, H).
\end{align*}
\]

\(^2\) Of course, all expressions occurring at the program rule at hand must be translated into their corresponding Prolog terms. We are not considering such a translation because it is irrelevant for the understanding of the problem.
The argument $H$ of $f$ returns the head normal form of the evaluation of the rule, the predicate $solve$ solves the constraint $\varphi$, $hnf$ computes the head normal form of its first argument, returning the result in the second one, and finally, predicate $unify$ unifies an expression with a term, possibly involving multisets.

As an example consider the function $p$ which returns $true$ if a multiset contains either the constant symbol $a$ or both constant symbols $a$ and $b$.

$$p(a[Xs]) = true \quad g1 = b \quad h = \{g1, g2\}$$

$$p(a, b[Xs]) = true \quad g2 = a$$

A goal of the form $p(h) == L$ returns the solution $L = true$ twice, since both rules of $p$ can be applied. Imagine the translation of $p$ into Prolog by following the naive strategy. In order to compute both answers, we would need to call, firstly to $unify(h, \{a[Xs]\})$ and secondly to $unify(h, \{a, b[Xs]\})$. The first call to $unify$ computes the head normal form of $h$ (which is $\{g1, g2\}$) and, afterwards, tries to unify such a head normal form with $\{a[Xs]\}$ by evaluating $g1$ (which is $b$) to head normal form. Since $b$ and $a$ are not syntactically unifiable, the unification process (embedded in predicate $unify$) mutates the multiset $\{g1, g2\}$ returning $\{g2, g1\}$. Then computes the head normal form of $g2$ (which is $a$) and again, checks the syntactic unification, that in this case succeeds getting the first solution. If we would want to compute the second solution, solving the call $unify(h, \{a, b[Xs]\})$, then we would need to reevaluate $h$, $g1$ and $g2$ to head normal form. All head normal forms computed to calculate the first solution are lost!

As an improvement over the naive strategy without multisets, in [18] authors specify a so called demand driven strategy, using definition trees [1] to guide unification with the left-hand sides of rewrite rules. The idea is roughly as follows: given a set of rules $S$ for a function symbol $f \in FS^n$, we partition $S$ according to demanded positions. Each resulting subset $S_i$ will have demanded the position $i$, i.e., in such a position all rules in $S_i$ will have a term with a constructor at head. As a consequence, in order to narrow an expression of the form $f(e_1, \ldots, e_n)$, the head normal form $h_i$ of $e_i$ will be computed only once, and after this, we will apply the associated rule depending on the form of $h_i$. Note that the naive strategy would reevaluate $e_i$ to head normal form as many times as rules in $S_i$ before the first one yielding success. As an example, a function $p$ above would be translated, following the demand driven strategy in [18] (modifying conveniently the unification process, as in the naive strategy), as follows:

$$p0(A, H) := hnf(A, HA), p1(H, A, H)$$

$$p1(A, H) := unify(A, \{Y \mid Ys\}), hnf(Y, HY), p2(\{HY \mid Ys\}, H).$$

$$p2(\{Y \mid Ys\}, H) := unify(Y, a), hnf(HYs, H), p3(\{aHYs\}, H).$$

$$p3(\{a \mid Ys\}, H) := unify(Ys, \{Z \mid Zs\}), hnf(Z, HZ), p4(\{a, HZ \mid Zs\}, H).$$

$$p4(\{a, Y \mid Ys\}, H) := unify(Y, b), hnf(true, H).$$

Note that the first argument of both rules for $p$ is demanded since it has the multiset constructor at head. For such a reason, the clause $p0$ computes the head normal form of the first argument (the evaluation always is required). Similarly, the position occupied by the constant $a$ in rules for $p$ is also demanded. Hence, the clause $p1$ computes the head normal form for such position. The second clause for $p2$ corresponds to the second rule of $p$, that demands the evaluation of the rest of the multiset. After this, such a rule demands also the evaluation of the head of the tail of the multiset, what is reflected by clause $p3$. If we
consider again the goal $p(h) \iff L$, we can observe that the head normal form of $h$ is computed only once in order to capture both solutions. In general, this improves the behaviour of the naive strategy. However, in presence of multisets, this code may re-evaluate head normal forms in the same way that the naive strategy does. The problem relies on the need of mutating. Let us follow the execution of the goal from the call $p1(\{g1, g2\}, H)$. The head normal form of $g1$ (which is $b$) is computed. Since $b$ and $a$ are not unifiable, then the unification process mutates the multiset returning the $\{g1, q\}$ (as in the naive strategy). Then computes the head normal form of $g2$ (which is $a$) and again, checks the syntactic unification, that in this case succeeds, getting the first solution by applying the first clause of $p2$. To compute the second solution, by the second clause of $p2$, we need to evaluate $g1$ (previously computed!) to head normal form. Basically the problem now is that the evaluation to head normal form of the elements of a multiset is not kept!

What we have done in the present paper is to modify the above demand driven strategy in two senses: First, we have separated the unification process from the mutation process, and second, we have mixed the head normal form process with the mutation process. As an introductory example of the new resulting strategy dds, we anticipate the new code for function $p$:

$$p0(A, H) : \leftarrow \text{hnf}(A, HA), p1(HA, H),$$

$$p1(A, H) : \leftarrow \text{mutateHnf}(A, HA), \text{unifyHnf}(HA, \{Y\}Ys), p2(\{Y\}Ys, H).$$

$$p2(\{Y\}Ys, H) : \leftarrow \text{unifyHnf}(Y, a), \text{hnf}(true, H).$$

$$p3(\{a\}Ys, H) : \leftarrow \text{mutateHnf}(Ys, HYs), p3(\{a\}HYs, H).$$

$$p4(\{a, Y\}Ys, H) : \leftarrow \text{unifyHnf}(Y, b), \text{hnf}(true, H).$$

where $\text{unifyHnf}$ does the syntactic unification between its two arguments, not mutating. The need of this predicate is due to the constraints supported by SETA.

We will comment on this later. For the moment we can think only on syntactic unification. Predicate $\text{mutateHnf}$ reorganizes the representation of a multiset in different ways, trying to bring all the elements at the head position, computing their head normal forms. Now, the computation of the goal $p(h) \iff L$ does not re-evaluate anything. By $p0$, the head normal form of $h$ is computed. After this, predicate $\text{mutateHnf}$ returns $\{b, g2\}$. Since the unification between $a$ and $b$ fails, there is a backtracking process over predicate $\text{mutateHnf}$, which returns as new solution $\{a, b\}$ (does not destroy the previously computed head normal form). Finally, both clauses for $p2$ compute the solutions.

### 3.1 Definitional Trees

In order to describe the dds strategy, we rely on the idea of using definitional trees [1], [18] as a tool to generate the Prolog code associated to dds. However, as we will see later, our definitional trees are not identical to the presented ones in the previous works, since the presence of multisets requires new kinds of nodes in order to consider the axiom (MSet). Before presenting formally the algorithm to generate definitional trees, we need some auxiliary concepts similar to that defined in [18].

**Preliminary notions** Let $f \in FS^n$ be a defined function symbol in a program $P = (\Sigma, R)$:

- $R_f$ stands for the set of all rules for $f$ in P.
A call pattern for $f$ is any linear expression of the form $f(t_1, \ldots, t_n)$, where $t_i \in T_{DC}(DV)$, $1 \leq i \leq n$.

A generic call pattern for $f$ is any call pattern for $f$ of the form $f(X_1, \ldots, X_n)$, where $X_i$, $1 \leq i \leq n$, are pairwise distinct variables.

Let $pat_1$ and $pat_2$ be two call patterns. We say that $pat_1$ matches $pat_2$ iff $pat_1$ is an instance of $pat_2$ via some term substitution (a mapping from $DV$ to $T_{DC}(DV)$). Moreover, $pat_2$ is a variant of $pat_1$ iff this term substitution is a variable renaming.

Given a call pattern $pat$, $P(pat)$ denotes the set of all positions in $pat$, and it is defined recursively as follows:

- $P(t) = \{\varepsilon\}$ if $t \in DV \cup DC^0$, where $\varepsilon$ stands for the empty position.
- $P(f(t_1, \ldots, t_n)) = \{1, \ldots, n\} \cup \bigcup_{i=1}^{n} \{i \cdot u | u \in P(t_i)\}$, if $f \in FS^n$.

For instance, if $pat = f(c(X, Y), d(Z, d, s(W)))$, then $P(pat) = \{1, 1 \cdot 1, 1 \cdot 2, 2 \cdot 1, 2 \cdot 2, 3, 3 \cdot 1\}$. $u \in P(pat)$ identifies the subterm $t$ of $pat$ occurring at position $u$. If $t$ has a constructor symbol $c$ at head, then $u$ also identifies $c$. For the example at hand, position $1$ refers to $c(X, Y)$ and also to $c$. Similarly, position $2 \cdot 1$ identifies variable $Z$. $VP(pat) \subseteq P(pat)$ denotes the set of variable positions in $pat$. Following with our example, $VP(pat) = \{1 \cdot 1, 1 \cdot 2, 2 \cdot 1, 3 \cdot 1\}$.

Let $pat$ be a call pattern for a function symbol $f$, which matches the left-hand side of at least one defining rule $Rul \equiv t = r \equiv \varphi$ in $R_f$. Let $u$ be a position in $VP(pat)$. We say that:

- $u$ is a demanded position by $Rul$ iff $l$ has a constructor at position $u$.
- $u$ is a uniformly demanded position by a set of rules $S \subseteq R_f$ iff $u$ is a demanded position by every rule in $S$.

As an example, consider the function $asBusy$ defined in Example 1 together with the generic call pattern $pat = asBusy(X, Y)$. Position $2 \in VP(pat)$ is uniformly demanded by $R_{asBusy}$, since the second argument of all rules in $R_{asBusy}$ has a (multiset) constructor at head. Position $1 \in VP(pat)$ is demanded only by rule $(R3)$ (and also uniformly demanded by the set $\{R3\}$), which has, in its first argument, the multiset constructor at head.

Generation of definitional trees Let $f$ be a defined function symbol. The definitional tree of $f$ is built according to the recursive algorithm $dt$ below. A generic call to $dt$ has the form $dt(pat, oldpat, S)$, where $pat$ is a call pattern, $oldpat$ is (intuitively) the last modified $pat$'s ancestor in the tree, and $S \subseteq R_f$.

An initial call to $dt$ is as follows: $dt(pat, pat, R_f)$, where $pat$ is a generic call pattern (note that in this case oldpat = pat). The algorithm termination can be proved by considering that any call $dt(pat, oldpat, S)$ verifies the following invariant properties:

- $S \subseteq R_f$.
- $pat = oldpat$ or $pat = oldpat[Y/d(Z_1, \ldots, Z_n)[$, where $Y$ is a variable occurring at a uniformly demanded (by $S$) position in $oldpat$. The notation $pat[Y/t]$ stands for the result of replacing in $pat$ the variable $Y$ by the term $t$. In the sequel, we will continue using such notation. Similarly, we will use $\tilde{Z}_i$ to denote $Z_1, \ldots, Z_n$.
- If $VP(pat)$ has no uniformly demanded positions by $S$, then left-hand side of rules in $S$ are variants of $pat$.
\textbf{dt-Algorithm}

Because of item (a.1) below, for the description of the algorithm we will assume that all multisets occurring at left-hand sides of program rules have been reorganized in order to put at the beginning all elements with a constructor at head. For instance, the multiset $\{ X, a, Y, b Z_s \}$, where $a, b$ are constant symbols and $X, Y, Z_s$ are variables would be reorganized as $\{ a, b, X, Y | Z_s \}^3$.

1. $\text{dt}(\text{pat}, \text{oldpat}, S) = \emptyset$ if $S = \emptyset$.
2. Otherwise, compute $\text{VP}(\text{pat})$ and distinguish the following cases:

(a) Some position in $\text{VP}(\text{pat})$ is uniformly demanded by $S$. Let $u$ be the leftmost one (this choice is arbitrary) and let $X$ be the variable at position $u$ in $\text{pat}$. Let $c_1, \ldots, c_k$ be all different constructors occurring at position $u$ in the left-hand sides of rules in $S$ (we assume that these $c_i$ are taken in textual order).

Let $S_u$ be the subset of $S$ composed of those rules containing $c_i$ at position $u$ ($1 \leq i \leq k$), build the new call pattern:

$$\text{pat}_i \equiv \text{pat}[X/c_i(X_1, \ldots, X_{m_i})]$$

where $m_i$ is the arity of $c_i$ and $X_j, 1 \leq j \leq m_i$, are new variables.

(a.1) If $u \equiv p_1 \cdots p_{m-1} \cdot 1$ ($m > 1$) and the symbol at position $p_1 \cdots p_{m-1}$ in $\text{pat}$ is the multiset constructor$^4$, then:

$$\text{dt}(\text{pat}, \text{oldpat}, S) = \text{pat} - \text{mutate}_{\text{match}}$$

$$\alpha : \text{dt}(\text{pat}_1, \text{pat}, S_u^\alpha)$$

$$\alpha : \text{dt}(\text{pat}_2, \text{pat}, S_u^\alpha)$$

$$\alpha : \text{dt}(\text{pat}_k, \text{pat}, S_u^\alpha)$$

(a.2) Otherwise:

(a.2.1) If $\text{pat} = \text{oldpat}[Y/Z|Z_s]$ then the definitional tree has the following form:

$$\text{dt}(\text{pat}, \text{oldpat}, S) = \text{pat} - \text{match}_{\text{case}} X \text{ of}$$

$$\alpha : \text{dt}(\text{pat}_1, \text{pat}, S_u^\alpha)$$

$$\alpha : \text{dt}(\text{pat}_2, \text{pat}, S_u^\alpha)$$

$$\alpha : \text{dt}(\text{pat}_k, \text{pat}, S_u^\alpha)$$

(a.2.2) Otherwise, the definitional tree has the form:

$$\text{dt}(\text{pat}, \text{oldpat}, S) = \text{pat} - \text{case} X \text{ of}$$

$$\alpha : \text{dt}(\text{pat}_1, \text{pat}, S_u^\alpha)$$

$$\alpha : \text{dt}(\text{pat}_2, \text{pat}, S_u^\alpha)$$

$$\alpha : \text{dt}(\text{pat}_k, \text{pat}, S_u^\alpha)$$

(b) Some position in $\text{VP}(\text{pat})$ is demanded, but none of them is uniformly demanded. Let $u_1, \ldots, u_k$ be those positions in $\text{VP}(\text{pat})$ which are demanded, taken in the same order as they occur as demanded positions in the rules.

For each $1 \leq i \leq k$, we define recursively the sets of rules $Q_0 = S$, $Q_i = Q_{i-1} - \{ \alpha \}$, $S_{u_i} = \{ \alpha \}$. Let $S_0$ be

$^3$ Of course, there exists more possible representations. But we need only to ensure that $a, b$ are at the beginning. It does not matter the order between them.

$^4$ As commented at the beginning of this section, all demanded elements in a multiset are at the beginning.
the set of rules of \( S \) which does not demand any position \( (S_0 = Q_k) \). Then
the definitional tree has the following form:

\[
dt(pat, oldpat, S) = \begin{cases} pat \lor \\
dt(pat, oldpat, S_0) \\
dt(pat, oldpat, S_{i_1}) \\
\vdots \\
dt(pat, oldpat, S_{i_m}) \end{cases}
\]

(c) No position in \( VP(pat) \) is demanded. All the left-hand sides of rules in \( S \) are
variants\(^5\) of \( pat \). Let \( pat = r_i \equiv \varphi_i, 1 \leq i \leq m, \) be renamings of rules in \( S \)
with the left-hand side equal to \( pat \), taken in textual order.

(c.1) If \( pat = oldpat[X / \{ Z | Ys \}] \), then the definitional tree has the following
form:

\[
dt(pat, oldpat, S) = \begin{cases} pat - \text{mutate}_{\text{try}} \\
\{r_1 \equiv \varphi_1; \\
r_2 \equiv \varphi_2; \\
\vdots \\
r_m \equiv \varphi_m \} \end{cases}
\]

(c.2) Otherwise, the definitional tree has the following form:

\[
dt(pat, oldpat, S) = \begin{cases} pat - \text{try} \\
\{r_1 \equiv \varphi_1; \\
r_2 \equiv \varphi_2; \\
\vdots \\
r_m \equiv \varphi_m \} \end{cases}
\]

Before explaining briefly the \( dt \)-algorithm, we are going to present an example
showing the structure of the definitional tree associated to the generic call pattern \( asBusy(A, B) \) from Example 1.

In general, definitional trees show how to compute goals by following the
\( ddsn \) strategy. In our example at hand, the evaluation of a call \( asBusy(e_1, e_2) \),
e_1, e_2 \in \mathcal{E}_2(DV) \), requires the evaluation of \( e_2 \) to head normal form \( (h_2) \) in order
to look for either the empty multiset (rule \((R1)\)) or the multiset constructor
(rules \((R2), (R3)\)). In the root, position 2 is uniformly demanded by all rules
of \( asBusy \), thus the root is a case-node. If \( h_2 \) and the empty multiset unify,

\(^5\) Because of the invariant properties.
the solution true is returned (try-node). Otherwise, if \( h_2 \) and \( \{ Y | Y \} \) unify, we have two possible rules ((R2), (R3)) to continue (or-node), which do not demand uniformly any new position. The leftmost subtree of the or-node works only on rule (R2). Since such a rule has the constructor sleep as first element of the multiset occurring at second argument, then a mutate_hnf-node is required. Such a node indicates that the multiset \( \{ Y | Y \} \) must be mutated in order to put as head of the multiset the constant symbol sleep (this process will be done by predicate mutate_hnf described in Sect. 3.2). After this, a recursive call asBusg(A, Y’s) is executed (try-node). The rightmost subtree of the or-node uses rule (R3). This rule demands the evaluation of \( e_1 \) to head normal form \( (h_1) \) without forgetting that we are in presence of a multiset. For such a reason a mutate_case-node is required in order to possibly mutate the multiset \( \{ Y | Y \} \).

If the unification between \( h_1 \) and \( \{ X | Xs \} \) succeeds, we can then apply rule (R3). But since the last unification involved a multiset, now it is needed to use a mutate_jry-node to take into account the possible mutations of \( \{ X | Xs \} \).

As we have just seen in the above example, a definitional tree may have or (item (b)), case (item (a.2.2)) and try (item (c.2)) nodes (similarly to [18]) together with three new types of nodes (to manage multisets): mutate_case (item (a.2.1)), mutate_jry (item (c.1)) and mutate_hnf (item (a.1)). mutate_case and mutate_jry nodes are needed to implement the unification modulo the equational axiom \( \text{MSET} \). A mutate_hnf node, also manages multiset unification, avoiding reevaluations of head normal forms, as shown at the beginning of this section.

mutate_jry nodes are similar to try nodes, i.e., they are used to apply program rules in order to return a solution. The difference between these two nodes relies on the presence of multisets. If the last unification process in the tree involved multisets, then a mutate_jry node is required in order to possibly mutate the multiset before getting a solution. mutate_case and mutate_hnf nodes are similar to case nodes in the sense that there exists a uniformly demanded position. Again, the difference relies on the presence of multisets which requires the mutation of the multiset (mutate_case and mutate_hnf nodes) before evaluating to head normal form the demanded position. However, there are substantial differences between muta case and muta hnf nodes. A mutate_case node is used in presence of multisets not having demanded elements. Hence, there is no evaluation to head normal form of elements! On the contrary, a mutate_hnf node is used in presence of multisets of the form \( \{ t | Xs \} \), where \( t \) has a constructor at head (i.e., the position of the first element is demanded). In such a case we need to mutate the multiset but also to compute the head normal form of the demanded element (trying not to lose it). For such a reason, a mutate_hnf node does a special kind of mutation which, simultaneously, evaluates the element to head normal form and does the mutation. Any other possible mutation of the multiset would not destroy the head normal form previously computed.

3.2 Prolog translation of program rules

From a given definitional tree for a function symbol \( f \), we can generate the Prolog code for \( f \). The generated code makes use of several auxiliary predicates, for which we are going to give a brief description:

\( hnf(E, H) \) computes a head normal form for \( E \) in \( H \). We remark that an expression is in head normal form iff it is either a variable or an expression with a constructor at head. The Prolog clauses for hnf are:

\( \text{Given a multiset } \{ t_1, t_2, \ldots, t_n | X \}, \text{ by elements we mean } t_1, \ldots, t_n. \)
\[ \text{hnf}(f(\bar{E}_n), H) : = !, f(\bar{E}_n, H). \]
\[ \text{hnf}(E, H) : = H = E. \]

Note that the head normal form for a function call is computed by calling to predicate \( f(\bar{E}_n, H) \). This last predicate will be specified below, when giving the algorithm which translates the definitional tree of \( f \) into Prolog clauses (which simulates the \textit{ddsm} strategy).

\> \textit{unifyHnf}(E, T) unifies syntactically \( E \) and \( T \) (which are in head normal form). Its associated Prolog clauses are:

\[ \text{unifyHnf}(E, T) : = \text{var}(E), !, \text{extractAndPropagateCtri}(E), E = T. \]
\[ \text{unifyHnf}(E, T) : = T = E. \]

The first clause of \textit{unifyHnf} unifies a variable with a term. Since \textit{SETA} incorporates constraints, before doing the unification \( E = T \), we need to check that \( E \) is not “constrained” by \( T \). For instance, if the variable \( E \) has an associated constraint \( \not\equiv T \), then the unification between \( E \) and \( T \) is not possible. This is checked by predicate \textit{extractAndPropagateCtri} which imposes on \( T \) all the constraints associated to \( E \). All constraints associated to a variable will be stored in a \textit{constraint store}. Whenever we impose a constraint on a variable \( X \), a constraint will be stored in the constraint store. Similarly, whenever a variable \( X \) is bound to a term \( T \), we will impose on \( T \) all constraints for \( X \). The constraint store is also used to compute the final solution, more concretely, the part referring to constraints in solved form. For such a reason, the constraint store only contains constraints in solved form. Due to the lack of space, we have focused on the \textit{ddsm} strategy, discarding the management of constraints. Interested readers can consult, e.g., [5].

Note that, differently to [5], the predicate \textit{unifyHnf} does syntactic unification.

In [5] the unification is done modulo the equational axiom (\textit{mset}).

\> \textit{mutateHnf}(E, R) recognizes the representation of a multiset trying to bring all the elements (computing their head normal form) at the head position. For instance, consider the program rules \( f = 2 \) and \( g = 3 \), and the multiset \( \{f, g\} \). A call to \textit{mutateHnf}(\{\{f, g\}\}, R) would return \( \{2, g\} \) as first solution and \( \{3, 2\} \) as second solution. The predicate \textit{mutateHnf} is similar to predicate \textit{mutateHnf} but without computing head normal forms for elements. Thus, a call to \textit{mutate}(\{\{f, g\}\}, R) \( R \) would return \( \{f, g\} \) as first solution and \( \{g, f\} \) as second solution. The Prolog code for \textit{mutateHnf} and \textit{mutate} is:

\begin{verbatim}
mutateHnf(\{E|Es\}, R) : = \text{hnf}(E, HE), \text{mutateHnf(\{HE|Es\}, R).}
mutateHnf(\{E|Es\}, R) : = \text{hnf}(E, HE), \text{mutateHnf(\{HE|Es\}, \{Y|Ys\}).
mutateHnf(\{E|Es\}, R) : = \text{hnf}(E, HE), \text{mutateHnf(\{HE|Es\}, \{Y|Ys\}).
mutateHnf(\{E|Es\}, R) : = \text{hnf}(E, HE), \text{mutateHnf(\{HE|Es\}, \{Y|Ys\}).
mutate(\{E|Es\}, R) : = \text{hnf}(E, HE), \text{mutate(\{HE|Es\}, \{Y|Ys\}).
mutate(\{E|Es\}, R) : = \text{hnf}(E, HE), \text{mutate(\{HE|Es\}, \{Y|Ys\}).
\end{verbatim}

The \textit{ge}-algorithm

The \textit{ge}-algorithm generates all Prolog predicates associated to a function \( f \in FS^n \) by means of its associated definitional tree (the first argument of \textit{ge}). The names of such predicates are built from the demanded positions in \( R_f \), hence the algorithm will have a second argument in which to keep the sequence of demanded positions (of the form \( u_1 u_2 \ldots u_n \)). A generic call to \textit{ge} has the form \textit{ge}(\textit{tree}, \textit{positions}), whereas an initial call has the form \textit{ge}(\textit{\{\{f(\bar{X}_n)\}, f(\bar{X}_n)\}, R_f}, \varepsilon).
where $\varepsilon$ is the empty sequence of positions. A call to $gc(tree, \textit{positions})$ generates Prolog clauses as follows:

(1) If $tree = \emptyset$ then $gc(tree, \textit{positions}) = \emptyset$.

(2) If $tree = \textit{pat} \cdot \textit{mutate}\_\textit{inf}\$

\[
\begin{align*}
&c_1 : \text{tree}_1 \\
&c_2 : \text{tree}_2 \\
&\ldots \\
&c_m : \text{tree}_m
\end{align*}
\]

then, let $NY'$ be a new variable. By construction of the definitional tree, $\textit{oldpat} = f(\tilde{t}_j), 1 \leq j \leq n$, and $\textit{pat} = \textit{oldpat}[Y / \{Z|Zs\}]$. Let $u \cdot 1$ be the position of $Z$ and then $u$ the position of the multiset $\{Z|Zs\}$ in $\textit{pat}$. Build $t'_j$ as $t_j[Y / \{Z|Zs\}][X/\textit{HX}]$. Then the following clause is generated:

\[
\textit{f\_positions}(\tilde{t}_j, H) :- \textit{mutate}\_\textit{inf}(Y, NY'), \textit{unify}\_\textit{inf}(NY, \{Z|Zs\}), \\
\textit{hnf}(X, \textit{HX}), \textit{f\_positions}\_\textit{\_u}(t'_j, H).
\]

and the calls $gc(tree_i, \textit{positions}\_\textit{\_u}), 1 \leq i \leq m$, are generated.

(3) If $tree = \textit{pat} \cdot \textit{case} X \textit{ of}$

\[
\begin{align*}
&c_1 : \text{tree}_1 \\
&c_2 : \text{tree}_2 \\
&\ldots \\
&c_m : \text{tree}_m
\end{align*}
\]

then let $NY'$ be a new variable. By construction of the definitional tree, $\textit{oldpat} = f(\tilde{t}_j), 1 \leq j \leq n$, and $\textit{pat} = \textit{oldpat}[Y / \{Z|Zs\}]$. Let $u$ be the position of $X$ in $\textit{pat}$ and $\textit{HX}$ be a new variable. Build $t'_j$ as $t_j[Y / \{Z|Zs\}][X/\textit{HX}]$. Then the following clause is generated:

\[
\textit{f\_positions}(\tilde{t}_j, H) :- \textit{mutate}(Y, NY'), \textit{unify}\_\textit{inf}(NY, \{Z|Zs\}), \\
\textit{hnf}(X, \textit{HX}), \textit{f\_positions}\_\textit{\_u}(t'_j, H).
\]

and the calls $gc(tree_i, \textit{positions}\_\textit{\_u}), 1 \leq i \leq m$, are generated.

(4) If $tree = \textit{pat} \cdot \textit{case} X \textit{ of}$

\[
\begin{align*}
&c_1 : \text{tree}_1 \\
&c_2 : \text{tree}_2 \\
&\ldots \\
&c_m : \text{tree}_m
\end{align*}
\]

then let $u$ be the position of $X$ in $\textit{pat}$ and $\textit{HX}$ be a new variable.

(4.1) If $\textit{oldpat} = \textit{pat} = f(\tilde{t}_j), 1 \leq j \leq n$, then build $t'_j$ as $t_j[X/\textit{HX}]$. The following Prolog clause is generated:

\[
\textit{f\_positions}(\tilde{t}_j, H) :- \textit{hnf}(X, \textit{HX}), \textit{f\_positions}\_\textit{\_u}(t'_j, H).
\]

(4.2) If $\textit{oldpat} = f(\tilde{t}_j), 1 \leq j \leq n$, and $\textit{pat} = \textit{oldpat}[Y / d(\tilde{Z}_k)]$ where $d \in DC^1 - \{\{\cdot, 1\}\}$, then build $t'_j$ as $t_j[Y / d(\tilde{Z}_k)][X/\textit{HX}]$. Then the following Prolog clause is generated:

\[
\textit{f\_position}(\tilde{t}_j, H) :- \textit{unify}\_\textit{inf}(Y, d(\tilde{Z}_k)), \textit{hnf}(X, \textit{HX}), \textit{f\_positions}\_\textit{\_u}(t'_j, H).
\]

In both cases the calls $gc(tree_i, \textit{positions}\_\textit{\_u}), 1 \leq i \leq m$, are generated.

(5) If $tree = \textit{pat} \cdot \textit{or}$

\[
\begin{align*}
&\text{tree}_0 \\
&\text{tree}_1 \\
&\ldots \\
&\text{tree}_m
\end{align*}
\]

then we have the code generated by the calls $gc(tree_i, \textit{positions}), 0 \leq i \leq k$.

(6) If $tree = \textit{pat} \cdot \textit{mutate}\_\textit{try}$

\[
\begin{align*}
&\text{if} \phi_1 \\
&\text{if} \phi_2 \\
&\ldots \\
&\text{if} \phi_m
\end{align*}
\]
then let \( NY \) be a new variable. By construction of the definitional tree, 
\( \text{oldpat} = f(\tilde{t}_i), 1 \leq j \leq n, \) and \( \text{pat} = \text{oldpat}[Y / \{Z | Z \in S\}] \). Build \( r'_j \) as \( r_j[Y / \{Z | Z \in S\}] \) and \( \varphi'_i \) as \( \varphi_i[Y / \{Z | Z \in S\}] \). Then, for each alternative, the following Prolog clause is generated:

\[
\text{f\text{-}positions}(\tilde{t}_i, H) := \text{mutate}(Y, \text{NY}), \text{unifyHnf}(\text{NY}, \{Z | Z \in S\}), \text{solve}(\varphi'_i), \text{hnf}(r'_i, H).
\]

These clauses are generated in the textual order of rules of \( f \) in the program.

(7) If tree = \( \text{pat} \rightleftharpoons \text{true} \)
\[
\begin{align*}
\varphi_1 & \iff \varphi_1 \\
\varphi_2 & \iff \varphi_2 \\
\varphi_n & \iff \varphi_n
\end{align*}
\]

(7.1) If \( \text{oldpat} = \text{pat} \equiv f(\tilde{t}_i), 1 \leq j \leq n, \) then for each alternative, generate the clause:

\[
\text{f\text{-}positions}(\tilde{t}_i, H) := \text{solve}(\varphi_i), \text{hnf}(r_i, H).
\]

(7.2) If \( \text{oldpat} = f(\tilde{t}_i), 1 \leq j \leq n, \) and \( \text{pat} = \text{oldpat}[Y / d(\tilde{Z}_k)] \) where \( d \in DC^k = \{\ldots, \}, \) then build \( r'_i \) as \( r_i[Y / d(\tilde{Z}_k)] \) and \( \varphi'_i \) as \( \varphi_i[Y / d(\tilde{Z}_k)] \). For each alternative, generate the clause:

\[
\text{f\text{-}positions}(\tilde{t}_i, H) := \text{unifyHnf}(Y, d(\tilde{Z}_k)), \text{solve}(\varphi'_i), \text{hnf}(r'_i, H).
\]

In both cases, clauses are generated according to textual order of the rules of \( f \) in the program. On the other hand, the predicate solve solves the condition (constraints) of a rule.

Example 2. The Prolog translation for function \textit{asBusy} in Example 1 generates the following Prolog clauses:

\[
\begin{align*}
\text{asBusy}(A, B, H) & := \text{hnf}(B, HB), \text{asBusy}(A, HB, H).
\text{asBusy}(A, B, H) & := \text{unifyHnf}(B, \emptyset), \text{hnf}(\text{true}, H).
\text{asBusy}(A, B, H) & := \text{mutateHnf}(B, HB), \text{unifyHnf}(HB, \{Y | Y \in S\})
\text{asBusy}(A, B, H) & := \text{asBusy}(A, \{Y | Y \in S\}, H).
\text{asBusy}(A, B, H) & := \text{mutate}(B, HB), \text{unifyHnf}(HB, \{Y | Y \in S\}, H), \text{hnf}(A, HA)
\text{asBusy}(A, B, H) & := \text{asBusy}(HA, \{Y | Y \in S\}, H).
\text{asBusy}(A, Y \in S, H) & := \text{unifyHnf}(Y, \text{sleep}), \text{hnf}(\text{asBusy}(A, Y), H).
\text{asBusy}(A, \emptyset, H) & := \text{mutate}(A, HA), \text{unifyHnf}(HA, \{X | X \in S\})
\text{solve}(\text{asBusy}(X, HA), H).
\end{align*}
\]

\[
\]

4 Implementation

Currently we have implemented \( dtdms \), the naive strategy in [5] and the demand driven strategy (\textit{dds} in the table below) in [18] modified to support multiset management. The three strategies have been incorporated into the system \textit{TCY} [19] (implemented in Sicstus Prolog) as independent modules. For an efficient implementation, several optimizations are required. As an example, following a technique introduced by Cheong [9], we modify the representation of expressions to support sharing. The system \textit{TCY} including the module for \textit{dtdms} can be loaded from http://titan.sip.ucm.es/toy/multisets.tar.gz

Our implementation extends the language \textit{SETA} presented in Sect. 2 with arithmetic constraints and also with cardinality constraints. As commented in Section 3, the incorporation of cardinality constraints to \textit{SETA} close the syntax of the language to computed answers. The management of constraints, as
commented in Section 3.2, is controlled by a constraint store which keeps those constraints associated to a variable. Such a constraint store is updated each time a constraint is imposed on a variable. Also plays an important role when binding variables, since it is necessary to check (by consulting the constraint store) that the binding is possible.

To illustrate the performance of our strategy we use the SETA program presented in Section 3 but adding a new constant symbol \( c \) and modifying the functions \( g_1, g_2 \) as follows:

\[
\begin{align*}
g_1 \iff b &\iff \text{genList}(20000) \iff \text{genList}(0) = [] \iff \text{genList}(N) \iff \epsilon[N] \iff N > 0
\end{align*}
\]

The table below shows the time consumed to obtain either all solutions or the first one (as shown in the table) of the corresponding goals. For the case of all solutions, the time needed to detect that a goal has no more solutions is also included. Time is measured in milliseconds. On the other hand, in order to get times, we have changed the textual order of both rules of \( p \). The goal -which is a scheme goal over the parameter \( N \)- is \( p([c, \ldots, c, g_1, g_2]) \iff L \).

<table>
<thead>
<tr>
<th>( N )</th>
<th>( (\text{all solutions}) )</th>
<th>( (\text{all solutions}) )</th>
<th>( (\text{first solution}) )</th>
<th>( (\text{first solution}) )</th>
<th>( (\text{first solution}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>203800</td>
<td>191400</td>
<td>8130</td>
<td>8130</td>
<td>8130</td>
</tr>
<tr>
<td>1</td>
<td>100000</td>
<td>200000</td>
<td>957300</td>
<td>957300</td>
<td>957300</td>
</tr>
<tr>
<td>5</td>
<td>5670</td>
<td>5730</td>
<td>5730</td>
<td>5730</td>
<td>5730</td>
</tr>
</tbody>
</table>

5 Conclusions

The previous work [5] deals with the problem of implementing lazy narrowing with multiset constraints using the technique of translating into Prolog. More concretely, the authors proposed a naive strategy for translating programs rules into Prolog based on the idea of representing multisets as Prolog terms which keep information about multiset cardinalities. As commented in [18] (but without multisets), the naive strategy performs redundant work because of reevaluations of head normal forms. More efficient strategies, such as the demand driven [18], adapted to manage multiset unification, are not very appropriate because computed head normal forms of multiset elements are, in general, lost. What we have proposed here is a new strategy (\( dbsm \)) for lazy narrowing with multisets which repairs some of the problems presented by the other strategies.

On the other hand, we have extended the syntax of SETA in [5] to support cardinality constraints and arithmetic constraints. This repairs the problem in [5], where computed answers contain information about cardinalities not allowed in the syntax of the language. Due to the lack of space, we have focused on the presentation of \( dbsm \), commenting briefly on the rest of details.

Finally, we have developed a complete Prolog-based implementation for SETA, incorporating it into the system \( TOY \) [19],[4]. Furthermore, we have also implemented the naive strategy [5] and the demand driven strategy [18] (modified to manage multisets), proving that, in general, the behaviour of \( dbsm \) improves the other strategies.
References

Computational properties of term rewriting with strategy annotations

Salvador Lucas
DSIC, Universidad Politécnica de Valencia
Camino de Vera s/n, E-46022 Valencia, Spain
e.mail: slucas@dsic.upv.es

Abstract. Strategy annotations have been used in several programming languages to improve termination and efficiency of computations. Eager (rewriting-based) languages (e.g., Lisp, OBJ, CafeOBJ, or Maude) interpret them as replacement restrictions in order to become 'more lazy', thus (hopefully) avoiding nontermination. In this paper, we investigate some computational properties (termination and completeness) of programs whose execution is controlled by strategy annotations.

Keywords: Declarative programming, replacement restrictions, term rewriting, termination.

1 Introduction

A number of programming languages permit the explicit specification of strategies aimed at controlling the execution of the programs. For instance, strategy annotations have been used in the OBJ family of languages\textsuperscript{1} [OBJ\textregistered{}2 [FGJM85], OBJ\textregistered{}3 [GW MFJ00], CafeOBJ [FN97], or Maude [CELM96]] to introduce replacement restrictions aimed at improving efficiency of computations (by reducing the number of attempted matchings). Their usefulness has been demonstrated in practice: in [FGJM85] the authors remark that, due to their use in OBJ\textregistered{}2 programs, \textit{the ratio between attempted matches and successful matches is usually around 2/3, which is really impressive}. For instance, OBJ\textregistered{}s built-in conditional operator has the following (implicit) strategy annotation [GW MFJ00]

\texttt{op if\_then\_else: Bool Int Int \rightarrow Int [strat (1 0)]}

which says to evaluate the first argument until it is reduced, and then apply rules at the top (indicated by '0'). Intuitively, this strategy annotation improves efficiency, since reducing the second or third argument of \texttt{if} only makes sense after knowing the value of the first argument. The presence of such 'true' replacement restrictions (e.g., by forbidding replacements in the second and third arguments of \texttt{if}) is also invoked to justify that OBJ\textregistered{} programs are able to avoid nontermination ([GW MFJ00], Section 2.4.4; [OF00]).

\textsuperscript{*} This work has been partially supported by Spanish CICYT and Conselleria de Cultura y Educación de la Generalitat Valenciana.

\textsuperscript{1} As in [GW MFJ00], by OBJ we mean OBJ\textregistered{}2, OBJ\textregistered{}3, CafeOBJ, or Maude.
Example 1. The following OBJ3 program (borrowed from [OF97]):

```
obj EXAMPLE is
  sorts Sort .
  op 0 : -> Sort .
  op s : Sort -> Sort .
  op _::_ : Sort Sort -> Sort [strat: (1 0)] .
  op inf : Sort -> Sort .
  op nth : Sort Sort -> Sort .
  var X Y L : Sort .
  eq nth(s(X),Y::L) = nth(X,L) .
  eq nth(0,X::L) = X .
  eq inf(X) = X::inf(s(X)) .
end
```

specifies an explicit strategy annotation for the list constructor `::` which disables replacements on the second argument. In this way, the evaluation of expression nth(s(0),inf(0)) always finishes and produces the term s(0), even if the ‘infinite list’ inf(0) is a part of the expression.

Using restrictions of rewriting may give rise to incomplete computations, i.e., normal forms of some terms could be unreachable by restricted computation. In order to avoid this, negative indices have been introduced aimed at enabling reductions on arguments ‘on-demand’, where a ‘demand’ is an attempt to match a pattern to the term that occurs in such an argument position [OF00].

Example 2. The following CafeOBJ program (borrowed from [NO01]):

```
mod! TEST {
  [T]
  op 0 : -> T
  op s : T -> T {strat: (1)}
  op _::_ : T T -> T {strat: (1 -2)}
  op 2nd : T -> T {strat: (1 0)}
  op from : T -> T {strat: (0)}
  var X Y Z : T
  eq from(X) = X::from(s(X)) .
  eq 2nd(X::(Y::Z)) = Y .
}
```

specifies negative indices in the strategy for ::. A local strategy such as (1) disables the evaluation of 2nd(0::from(s(0))) into s(0), since reducing the second argument of `::` is not allowed. On the other hand, with (1 2), the following infinite reduction sequence is possible:

```
2nd(from(0)) -> 2nd(0::from(s(0))) -> 2nd(0::s(0)::from(s(s(0)))) -> ...
```

Negative indices are aimed at keeping computations terminating, but providing more opportunities for achieving completeness.
In this paper, we investigate termination and completeness of programs whose execution is controlled by strategy annotations. Following a recent classification ([Vis01], Section 5.7), we consider the following kinds of computational strategies associated to strategy annotations:

1. E-strategy [Eke98,GWMFJ00], which permits us to completely avoid the evaluation of some arguments of function symbols (in an ordered way).
2. Just-in-time [Pol01], which is designed to delay the evaluation of arguments as much as possible.
3. On-demand E-strategy [OF00], which is similar to E-strategy, but enables the evaluation of subterms 'on-demand'.
4. Laziness annotations [FKW00], where the arguments of symbols can be annotated as 'lazy'. No reductions should be performed for subterms of these arguments unless needed for matching.

We use term rewriting systems (TRSs [BN98]) as a suitable computational model for programs written in more sophisticated programming languages. We show that context-sensitive rewriting (CSR, a simple restriction of rewriting that forbids reductions on selected arguments of functions [Luc98]) provides a suitable framework for describing and analyzing computations with programs using positive strategy annotations (i.e., only non-negative integers are allowed) as used in E-strategies and just-in-time. Due to lack of space, we discuss in detail only these two computational interpretations of strategy annotations. A preliminary analysis of completeness and termination of on-demand E-strategies and laziness annotations (covering, e.g., termination and completeness of the program in Example 2) are given in [Luc01a,Luc01b].

Section 2 gives some preliminary definitions. Section 3 introduces CSR. Sections 4 and 5 discuss termination and completeness of computations under the E-strategy and just-in-time, respectively. Section 6 concludes.

2 Preliminaries

Given a set $A$, $\mathcal{P}(A)$ denotes the set of all subsets of $A$. Let $R \subseteq A \times A$ be a binary relation on a set $A$. We denote the reflexive closure of $R$ by $R^+=\cup\ R$, its transitive closure by $R^*$, and its reflexive and transitive closure by $R^*$. An element $a \in A$ is an $R$-normal form, if there exists no $b$ such that $a \rightarrow_R b$; NFR is the set of $R$-normal forms. We say that $b$ is an $R$-normal form of $a$ (and that $a$ is $R$-normalizing), if $b$ is an $R$-normal form and $a \rightarrow_R b$. We say that $R$ is terminating iff there is no infinite sequence $a_1 \rightarrow_R a_2 \rightarrow_R a_3 \rightarrow_R \ldots$. Throughout the paper, $A'$ denotes a countable set of variables and $\Sigma$ denotes a set of function symbols \{$f, g, \ldots$\}, each having a fixed arity given by a function $ar: \Sigma \rightarrow \mathbb{N}$. We denote the set of terms built from $\Sigma$ and $A'$ by $T(\Sigma, A')$. A context $C[]$ is a term from $T(\Sigma \cup \{\Box\}, A')$, where $\Box$ is a new constant symbol. A term is said to be linear if it has no multiple occurrences of a single variable. Terms are viewed as labelled trees in the usual way. Positions $p, q, \ldots$ are represented by chains of positive natural numbers used to address subterms of $t$. By $A$, we denote the empty chain. Given positions $p, q$, we denote
its concatenation by p.q. Positions are ordered by the usual prefix ordering \(\leq\). Given a set of positions \(P\), \(\text{minimal}_{\leq}(P)\) (\(\text{maximal}_{\leq}(P)\)) is the set of minimal (resp. maximal) positions of \(P\) w.r.t. \(\leq\). If \(p\) is a position, and \(Q\) is a set of positions, \(p,Q\) is the set \(\{p,q | q \in Q\}\). By \(\text{Pos}(t)\) we denote the set of positions of a term \(t\). Positions of non-variable symbols in \(t\) are denoted as \(\text{Pos}_{\Sigma}(t)\) and \(\text{Pos}_{X}(t)\) are the variable occurrences. The subterm at position \(p\) of \(t\) is denoted as \(t[p]\) and \(t[s]_p\) is the term \(t\) with the subterm at position \(p\) replaced by \(s\). If \(P = \{p_1, \ldots, p_n\}\) is a set of disjoint positions (i.e., incomparable according to \(\leq\)), then we write \(t[s]_p\) instead of \(t[s]_{p_1} \cdots s_{p_n}\). The symbol labelling the root of \(t\) is denoted as \(\text{root}(t)\). A substitution is a mapping \(\sigma: X \rightarrow T(\Sigma, X)\) which homomorphically extends to a mapping \(\sigma: T(\Sigma, X) \rightarrow T(\Sigma, X)\).

A rewrite rule is an ordered pair \((l, r)\), written \(l \rightarrow r\), with \(l, r \in T(\Sigma, X)\), \(l \not\in X\), and \(\text{Var}(r) \subseteq \text{Var}(l)\). The left-hand side (lhs) of the rule is \(l\) and the right-hand side (rhs) is \(r\). A TRS is a pair \(\mathcal{R} = (\Sigma, R)\) where \(R\) is a set of rewrite rules. \(L(R)\) denotes the set of lhs's of \(R\). \(R\) is left-linear if \(L(R)\) is a set of linear terms. An instance \(\sigma(l)\) of a lhs \(l \in L(R)\) is a redex. The set of redex positions in \(t\) is \(\text{Pos}_{\Sigma}(t)\). A term \(t \in T(\Sigma, X)\) rewrites to \(s\) (at position \(p\)), written \(t \xrightarrow{\rho} \_s\) or just \(t \rightarrow s\), if \(t[p] = \sigma[l]\) and \(s = t[\sigma(r)]_p\), for some \(l \rightarrow r \in R\). If \(t \rightarrow s\) and \(p \in \text{maximal}_{\leq}(\text{Pos}_{\Sigma}(t))\), then \(t \rightarrow s\) is terminating. A TRS is terminating if \(\rightarrow\) is terminating. We say that \(t\) innermost rewrites to \(s\), written \(t \xrightarrow{\rho} _{\text{inner}} \_s\), if \(t \rightarrow s\) and \(p \in \text{maximal}_{\leq}(\text{Pos}_{\Sigma}(t))\). A TRS is innermost terminating if \(\rightarrow\) is terminating.

3 Rewriting with syntactic replacement restrictions

Given a signature \(\Sigma\), a mapping \(\mu: \Sigma \rightarrow \mathcal{P}(\mathbb{N})\) is a replacement map (or \(\Sigma\)-map) if for all \(f \in \Sigma\), \(\mu(f) \subseteq \{1, \ldots, \text{ar}(f)\}\) [Luc98]. The inclusion ordering \(\subseteq\) on \(\mathcal{P}(\mathbb{N})\) extends to an ordering \(\sqsubseteq\) on \(M_\Sigma\), the set of all \(\Sigma\)-maps: \(\mu \sqsubseteq \mu'\) if for all \(f \in \Sigma\), \(\mu(f) \subseteq \mu'(f)\). Thus, \(\mu \sqsubseteq \mu'\) means that \(\mu\) considers less positions than \(\mu'\) (for reduction). We also say that \(\mu\) is more restrictive than \(\mu'\).

A replacement map \(\mu\) specifies the argument positions which can be reduced for each symbol in \(\Sigma\). Accordingly, the set of \(\mu\)-replacing positions \(\text{Pos}_{\mu}(t)\) of \(t \in T(\Sigma, X)\) is \(\text{Pos}_{\mu}(t) = \{\mu|t\}\), if \(t \in X\) and \(\text{Pos}_{\mu}(t) = \{\mu|t\} \sqcup \bigcup_{(l, r) \in \text{Pos}(t)} \{\text{Pos}_{\mu}(l)\}\), if \(t \not\in X\). The set of positions of replacing redexes in \(t\) is \(\text{Pos}_{\mu}(t) = \text{Pos}_{\mu}(t) \cap \text{Pos}_{\mu}(l)\). In context-sensitive rewriting (CSR), we (only) rewrite redexes at replacing positions: \(t\) \(\mu\)-rewrites to \(s\), written \(t \Rightarrow_{\mu} s\), if \(t \rightarrow_{\mu} s\) and \(p \in \text{Pos}_{\mu}(t)\).

Example 3. Consider the TRS [NO01]:

\[
\begin{align*}
\text{2nd}(x,y,z) & \rightarrow y \\
\text{from}(x) & \rightarrow x:\text{from}(s(x))
\end{align*}
\]

and \(\mu([]:1) = \mu(\text{2nd}) = \mu(\text{from}) = \mu([s]) = \{1\}\). Then we have:

\[
\begin{align*}
\text{2nd}(\text{from}(0)) & \Rightarrow_{\mu} \text{2nd}(0:\text{from}(s(0)))
\end{align*}
\]

where, since \(1.2 \not\in \text{Pos}_{\mu}(\text{2nd}(0:\text{from}(s(0))))\), \(\mu\)-rewriting stops here.
The $\rightarrow_\mu$-normal forms are called $\mu$-normal forms. Let $\text{NF}_R^{\mu}$ be the set of $\mu$-normal forms of $R$. A term is $\mu$-normalizing if it is $\rightarrow_\mu$-normalizing. A TRS $R$ is $\mu$-terminating if $\rightarrow_\mu$ is terminating (see [Luc96]). With innermost context-sensitive rewriting $\rightarrow_{\mu}$, we only contract maximal positions (w.r.t. $\subseteq$) of replacing redexes: $t \rightarrow_{\mu} s$ if $t \overset{\beta}{\rightarrow}_R s$ and $p \in \text{maximal}_\subseteq (\text{POS}_R(t))$. We say that $R$ is innermost $\mu$-terminating if $\rightarrow_\mu$ is terminating.

A (non-deterministic) rewriting strategy for a TRS $R$ is a function $S$ that assigns a non-empty set of non-empty finite rewrite sequences each beginning with $t$ to every term $t$ which is not a normal form. A (non-deterministic) context-sensitive rewriting strategy (or just $\mu$-strategy) for $R$ is a function $H$ that assigns a non-empty set of non-empty finite $\mu$-rewrite sequences each beginning with $t$ to every term $t$ which is not a $\mu$-normal form [Luc00]. We write $t \rightarrow_{\mu} H$ if $H(t)$ contains a $\mu$-reduction sequence ending with $s$. Given a $\mu$-strategy $H$, a $\mu$-reduction sequence of the form $t_1 \rightarrow_{\mu} t_2 \rightarrow_{\mu} \ldots$ is called a $H$-sequence. An $H$-sequence $t_1 \rightarrow_{\mu} t_2 \rightarrow_{\mu} \ldots \rightarrow_{\mu} t_n$ is maximal if $t_n$ is a $\mu$-normal form. The $\mu$-strategy $H$ is $\mu$-normalizing if, for all $\mu$-normalizing term $t$, there is no infinite $H$-sequence starting from $t$.

The maximal replacing context $MRC^\mu(t)$ of $t$ consists of the maximal part of $t$ whose positions are $\mu$-replacing in $t$, see [Luc00]. Every one-step $\mu$-strategy $H$ can be extended to a one-step strategy $S_H$ as follows:

$$S_H(t) = \begin{cases} H(t) & \text{if } t \notin \text{NF}_R^{\mu} \\ C[S_H(t_1) \ldots, S_H(t_n)] & \text{if } t \in \text{NF}_R^{\mu} - \text{NF}_R, \text{where} \\ \emptyset & \text{otherwise} \end{cases}$$

Here, for a given context $C[]$ and sets of rewrite sequences $S_1, \ldots, S_n$, issued form terms $t_1, \ldots, t_n$, $C[S_1, \ldots, S_n]$ is the set of sequences $C[t_1, \ldots, t_n] \rightarrow^+$ $C[s_1, \ldots, s_n]$, where either $s_i = t_i$ (during the whole sequence) or $s_i$ is the end point of a sequence in $S_i$ (also, at least one of the $s_i$ must be taken in this way).

### 4 $E$-strategy

A positive local strategy (or $E$-strategy) for a $k$-ary symbol $f \in \Sigma$ is a sequence $\varphi(f)$ of integers taken from $\{0, 1, \ldots, k\}$ which are given in parentheses (see Example 1). A mapping $\varphi$ that associates a local strategy $\varphi(f)$ to every $f \in \Sigma$ is called an $E$-strategy map [NO01]. Nagaya describes the operational semantics of term rewriting under $E$-strategy maps as follows [Nag99]: Let $\mathcal{L}$ be the set of all lists consisting of natural numbers. By $\mathcal{L}_n$, we denote the set of all lists of natural numbers not exceeding $n \in \mathbb{N}$. We use the signature $\Sigma_{\mathcal{L}} = \{f_L \mid f \in \Sigma \wedge L \in \mathcal{L}_{\text{ar}(f)}\}$ and labelled variables $\mathcal{X}_{\mathcal{L}} = \{x_{\text{nil}} \mid x \in \mathcal{X}\}$. An $E$-strategy map $\varphi$ for $\Sigma$ is extended to a mapping from $\mathcal{T}(\Sigma, \mathcal{X})$ to $\mathcal{T}(\Sigma_{\mathcal{L}}, \mathcal{X}_{\mathcal{L}})$ as follows:

$$\varphi(t) = \begin{cases} x_{\text{nil}} & \text{if } t = x \in \mathcal{X} \\ f_{\varphi(f)}(\varphi(t_1), \ldots, \varphi(t_k)) & \text{if } t = f(t_1, \ldots, t_k) \end{cases}$$
The mapping $\text{erase} : \mathcal{T}(\Sigma, X) \rightarrow \mathcal{T}(\Sigma, X)$ removes labelings from symbols in the obvious way. The binary relation $\rightarrow_\varphi$ on $\mathcal{T}(\Sigma, X) \times \mathbb{N}_0$ (i.e., pairs $\langle t, p \rangle$ of labelled terms $t$ and positions $p$) is [N001,Nag99]: $\langle t, p \rangle \rightarrow_\varphi \langle s, q \rangle$ if and only if $p \in \mathcal{P}os(t)$ and either

1. $\text{root}(t\|_p) = f_{\text{nil}}$, $s = t$ and $p = q;i$ for some $i$; or
2. $t\|_p = f_{\text{l}}L(t_1, \ldots, t_k)$ with $i > 0$, $s = t[f_{\text{l}}L(t_1, \ldots, t_k)]_p$ and $q = p;i$; or
3. $t\|_p = f_{\text{r}}L(t_1, \ldots, t_k)$, $\text{erase}(t\|_p)$ is not a redex, $s = t[f_{\text{r}}L(t_1, \ldots, t_k)]_p$, $q = p$; or
4. $t\|_p = f_{\text{r}}L(t_1, \ldots, t_k) = \sigma(t\|')$, $\text{erase}(t\|') = \ell, s = t[\sigma(\varphi(r))]_p$ for some $\ell \rightarrow r \in R$ and substitution $\sigma, q = p$.

We write $c \in L$ to denote that item $c$ appears somewhere within the list $L$.

Given an $E$-strategy map $\varphi$ for $\Sigma$, we define $\mu^\varphi \in M_\Sigma$ as follows: for all $f \in \Sigma$, $\mu^\varphi(f) = \{ i > 0 \mid i \in \varphi(f) \}$. We will drop superscript $\varphi$ if no confusion arises.

We have the following.

**Theorem 1.** [Luc01b] Let $\mathcal{R}$ be a TRS and $\varphi$ be a positive $E$-strategy map. Let $t \in \mathcal{T}(\Sigma, X)$, and $p \in \mathcal{P}os^\varphi(\text{erase}(t))$ be s.t. $\text{root}(t\|_p) = f_{\text{l}}L$ for some suffix $L$ of $\varphi(f)$. If $\langle t, p \rangle \rightarrow_\varphi \langle s, q \rangle$, then $q \in \mathcal{P}os^\varphi(\text{erase}(s))$ and $\text{erase}(t) \rightarrow_\varphi^* \text{erase}(s)$.

Algebraic languages $\text{OBJ2}$, $\text{OBJ3}$, CafeOBJ, and Maude admit the specification of $E$-strategies. Symbols without an explicit local strategy are given a default one whose concrete shape depends on the language considered. Semantics of OBJ programs under a given $E$-evaluation map $\varphi$ is usually given by means of a recursive evaluation function $\text{eval}_\varphi$ (from terms to their sets of ‘computed values’) rather than specifying the concrete rewrite steps leading to computed values [Eke98]. Nakamura and Ogata describe the evaluation strategy $\text{eval}_\varphi$ for positive $E$-strategy maps by using the reduction relation $\rightarrow_\varphi$ [N001]: given a TRS $\mathcal{R} = (\Sigma, \mathcal{T})$ and a positive $E$-strategy map $\varphi$ for $\Sigma$, $\text{eval}_\varphi : \mathcal{T}(\Sigma, X) \rightarrow \mathcal{T}(\Sigma, X)$ is defined as $\text{eval}_\varphi(t) = \{ \text{erase}(s) \in \mathcal{T}(\Sigma, X) \mid \langle \varphi(t), A \rangle \rightarrow_\varphi^* \langle s, A \rangle \in \text{NF}_{\rightarrow_\varphi^*} \}$.

**Example 4.** Consider the following TRS $\mathcal{R}$ [Eke98]:

\[
\begin{align*}
\text{f}(b) & \rightarrow c \\
\text{g}(x) & \rightarrow \text{h}(x) \\
\text{g}(x) & \rightarrow \text{f}(\text{a}) \\
\text{a} & \rightarrow \text{b}
\end{align*}
\]

and the $E$-strategy map $\varphi$ given by $\varphi(\text{f}) = \{ 0, 1 \}, \varphi(\text{g}) = \{ 0 \}, \varphi(\text{a}) = \{ 1, 0 \}, \varphi(\text{b}) = \{ 0 \}$, and $\varphi(\text{n}) = \{ \text{n} \}$. Consider the evaluation of $t = \text{g}(\text{f}(\text{a}))$ (more precisely, the evaluation of $\varphi(t) = \{ \text{g}(\text{f}(\text{a})) \}$) using $\rightarrow_\varphi$ (we underline contracted redexes in the ‘term’ component of pairs):

\[
\begin{align*}
\langle \text{g}(\text{f}(\text{a})) \rangle & \rightarrow_\varphi \langle \text{g}(\text{f}(\text{a})) \rangle \\
& \rightarrow_\varphi \langle \text{g}(\text{f}(\text{a})) \rangle \\
& \rightarrow_\varphi \langle \text{g}(\text{f}(\text{a})) \rangle \\
& \rightarrow_\varphi \langle \text{g}(\text{f}(\text{a})) \rangle
\end{align*}
\]

\(^2\) For instance, in Maude, the default local strategy associated to a $k$-ary symbol $f$, is $(1, 2, \ldots, k, 0)$, see [Eke98].
\[ \rightarrow_{\varphi} (h(b_{n|l}(b_{m|l})), 1) \rightarrow_{\varphi} (h(b_{n|l}(b_{m|l})), A) \]

where \( (b_{n|l}(f_{n|l}(b_{m|l}))), A) \) is a \( \rightarrow_{\varphi} \)-normal form. Then, \( h(f(b)) \in \text{eval}_{\varphi}(t) \).

### 4.1 Termination

Given an OBJ program \( P \) with positive \( E \)-strategy map \( \varphi \), we let \( R_{P} \) and \( \mu \) be the TRS and replacement map associated to it. According to the previous definition of \( \text{eval}_{\varphi} \), we can say that

\[ P \text{ (or } R_{P} \text{) is } \varphi \text{-terminating if, for all } t \in T(\Sigma, \Lambda), \text{ there is no infinite } \rightarrow_{\varphi} \text{-rewrite sequence starting from } (\varphi(t), A) \].

Thus, according to Theorem 1, we have the following.

**Theorem 2.** [Luc01b] An OBJ program \( P \) with positive \( E \)-strategy map \( \varphi \) is terminating if \( R_{P} \) is \( \mu \)-terminating.

Theorem 2 connects termination of CSR and termination of OBJ programs with positive evaluation strategies. Termination of CSR has been studied in [GM99,Luc96,Zan97]. For instance, the OBJ program in Example 1 can be proven terminating by using Zanema’s techniques (see Examples 2 and 3 of [Zan97]).

However, termination of CSR only approximates termination of such OBJ programs.

**Example 5.** Consider the TRS [Gra96]:

\[ f(a) \rightarrow f(a) \]
\[ a \rightarrow b \]

and a local strategy \( \varphi \) such that \( \varphi(f) = (1 \ 0) \) and \( \varphi(a) = (0) \). This TRS is \( \varphi \)-terminating, but it is not \( \mu \)-terminating, since we have:

\[ f(a) \leftarrow_{\mu} f(a) \leftarrow_{\mu} \cdots \]

The point here is that OBJ computations are ‘basically’ innermost. Innermost rewriting computations can be terminating even though the TRS is not terminating. This gives rise to the topic of innermost termination of rewriting which has been studied in e.g., [AG97,Gra96].

Given \( R = (\Sigma, R) \), we consider \( \Sigma \) as the disjoint union \( \Sigma = \mathcal{C} \uplus \mathcal{F} \) of symbols \( c \in \mathcal{C} \), called constructors and symbols \( f \in \mathcal{F} \), called defined functions, where \( \mathcal{F} = \{ \text{root}(l) \mid l \rightarrow r \in R \} \) and \( \mathcal{C} = \Sigma - \mathcal{F} \). We say that an \( E \)-strategy map \( \varphi \) is elementary if for all \( f \in \mathcal{F}, \varphi(f) = (i_{1} \cdots i_{n} \ 0) \) and \( i_{j} > 0 \) for \( 1 \leq j \leq n \).

**Remark 1.** Contiguous occurrences of zero can be simplified into a single one (Corollary 3.3 in [Eke98]). Since [Eke98,Nag99] largely motivate the interest in requiring that 0 be the last index of local strategies associated to defined symbols, the only critical requirement which is introduced with elementary strategies is that 0 occur only at the end of the local strategy.
Computations with elementary strategies can be modeled using innermost CSR.

**Theorem 3.** Let $\mathcal{R} = (C \ni F, R)$ be a TRS and $\varphi$ be a positive elementary $E$-strategy map. Let $t \in T(\Sigma, X)$. If $\langle \varphi(t), A \rangle \rightarrow^*_\varphi \langle s, p \rangle$, then $t \rightarrow^*_\mu \text{erase}(s)$.

Without requiring elementarity of $\varphi$, Theorem 3 does not hold.

**Example 6.** Consider $\mathcal{R}$, $\varphi$, and $t$ as in Example 4. According to Theorem 1, the $\mu$-rewriting steps associated to the evaluation of $t$ are:

$$g(f(a)) \rightarrow^* \mu g(f(b)) \rightarrow^* \mu h(f(b))$$

Due to redex $f(b)$, the second $\mu$-rewriting step is not innermost.

**Theorem 4.** An OBJ program $P$ with positive elementary $E$-strategy map $\varphi$ is terminating if $\mathcal{R}_P$ is innermost $\mu$-terminating.

For nonelementary $E$-strategies, Theorem 5 can fail to hold.

**Example 7.** Consider the TRS $\mathcal{R}$ of Example 5 and $\varphi$ given by $\varphi(f) = (0, 1, 0)$, $\varphi(a) = (0)$, and $\varphi(b) = \text{nil}$. Note that $\mathcal{R}$ is innermost $\mu$-terminating. However, $\mathcal{R}$ is not $\varphi$-terminating, since we have:

$$\langle f(0, 1, 0) (a(0)), A \rangle \rightarrow^* \varphi \langle f(0, 1, 0) (a(0)), A \rangle \rightarrow^* \varphi \cdots$$

Since $\mu$-termination implies innermost $\mu$-termination (but not vice versa), analyzing innermost termination of CSR provides a more accurate framework for proving termination of OBJ programs with positive, elementary $E$-strategies.

**Theorem 5.** Let $P$ be an OBJ program with positive elementary $E$-strategy map $\varphi$. If $P$ is terminating, then $\mathcal{R}_P$ is innermost $\mu$-terminating.

Without elementarity, $\varphi$-termination may not imply innermost $\mu$-termination.

**Example 8.** Consider $\mathcal{R}$ and $\varphi$ as in Example 4. Note that $\mathcal{R}$ is not innermost $\mu$-terminating, since we have:

$$h(c) \rightarrow^* \mu g(f(a)) \rightarrow^* \mu g(f(b)) \rightarrow^* \mu g(c) \rightarrow^* \mu h(c) \rightarrow^* \mu \cdots$$

However, $\mathcal{R}$ is $\varphi$-terminating, since, as shown in Example 4, whenever (the labelled version of) the term $g(f(a))$ is reached, the derivation stops in $h(f(b))$ without producing $h(c)$ which is needed to generate the cycle.

**Related work** In [FGK01], an inductive method is proposed to prove (ground) termination of rewriting with positive $E$-strategies. We have checked that the two examples used in [FGK01] to illustrate their technique can be easily proven terminating by using Zantema’s transformation [Zan97] and an automatic tool such as Contejean and Marché’s GiME 2.0 system<sup>3</sup>. Moreover, we note that only Theorem 2 is necessary to deal with these examples. On the other hand, in order to be able to use their methods, it is necessary to use a different technique to

<sup>3</sup>Available at http://www.lri.fr/~demons/cime.html.
ensure that the constant symbols are terminating (w.r.t. computations guided
by the strategies). This is easy if there is no rewrite rule \( c \rightarrow r \) associated to any
constant symbol \( c \). Note that \( \varphi \)-termination of (a TRS containing) the TRS \( \mathcal{R} \),
\[
a \rightarrow \xi(a)
\]
with \( \varphi(\xi) = \text{nil} \), could not be proven in this way. However, \( \varphi \)-termination of
\( \mathcal{R} \) is easily proved by using the \( \mu \)-contractive transformation of [Luc90] (that
essentially removes the non-\( \mu \)-replacing subterms from left and right-hand
sides of rules), since the transformed TRS:
\[
a \rightarrow \xi
\]
is clearly terminating.

4.2 Completeness

Computations with programs that use replacement restrictions may not directly
compute normal forms (i.e., terms without redexes), but rather normal forms
w.r.t. the reduction relation that is used for implementing computations (e.g.,
\( \rightarrow_{\varphi} \)). This is the case with OBJ programs, where terms returned by \( \text{eval}_{\varphi} \) are
called \( E \)-normal forms (ENFs). We have the following.

Theorem 6. [Luc01b] Let \( \mathcal{R} = (\Sigma \cup \mathcal{F}, R) \) be a TRS and \( \varphi \) be a positive E-
strategy map such that for all \( f \in \mathcal{F} \), \( \varphi(f) \) ends in 0. If \( s \in \text{eval}_{\varphi}(t) \), then \( s \) is
a \( \mu \)-normal form.

Correctness of OBJ computations (i.e., if \( s \in \text{eval}_{\varphi}(t) \), then \( s \in \text{NF}_{\mathcal{R}} \)) has been
studied in [Nag99] (Theorem 6.1.12) and [NO01] (Theorem 3.2 and Corollary 3.8). Here we consider completeness (regarding normalizations), i.e., if there
exists \( s \in \text{NF}_{\mathcal{R}} \) such that \( t \rightarrow_{\varphi}^{*} s \), we want that \( s \in \text{eval}_{\varphi}(t) \).

Normal forms of a term \( t \) can be obtained by successively computing its \( \mu \)-
normal forms \( s \), splitting \( s \) as \( s = C[s_1, \ldots, s_n] \) where \( C[ ] = \mathcal{M}_C^n(s) \) and con-
tinuing the evaluation of \( t \) by (recursively) normalizing \( s_1, \ldots, s_n \) (normalization via \( \mu \)-normalization [Luc00]). This works for replacement maps \( \mu \) which are less
restrictive than the canonical replacement map \( \mu_{\mathcal{R}}^{\text{can}} \). Given a TRS \( \mathcal{R} = (\Sigma, R) \),
\( \mu_{\mathcal{R}}^{\text{can}} \) is the most restrictive replacement map (in \( \mathcal{M}_f \)) ensuring that the non-
variable subterms of the left-hand side of the rules of \( \mathcal{R} \) are replacing, i.e., the
minimum \( \Sigma \)-map \( \mu \) such that \( \forall t \in \text{L}(\mathcal{R}), \text{Pos}_{\Sigma}(t) \subseteq \text{Pos}_{\mu}(t) \) [Luc98,Luc00].

Theorem 7 (Normalization via \( \mu \)-normalization [Luc00]). Let \( \mathcal{R} = (\Sigma, R) \)
be a left-linear, confluent TRS and \( \mu \in \mathcal{M}_f \) be such that \( \mu_{\mathcal{R}}^{\text{can}} \subseteq \mu \). If \( \mathcal{H} \) is a
\( \mu \)-normalizing \( \mu \)-strategy, then \( \mathcal{S}_{\mu} \) is normalizing.

If \( \mathcal{R} \) is \( \varphi \)-terminating and ENFs are \( \mu \)-normal forms, then, according to Theorem
1, \( \text{eval}_{\varphi} \) could be seen as the \( \mu \)-strategy that associates to each term \( t \) the set of
all possible \( \mu \)-rewrite sequences obtained by applying \text{erase} to terms in \( \rightarrow_{\varphi} \)-sequences issued from \( \varphi(t) \). Then, we have the following.

Theorem 8 (Normalization via \( \varphi \)-normalization). Let \( \mathcal{R} = (\Sigma, R) \) be a
left-linear, confluent TRS and \( \varphi \) be an E-strategy map such that for all \( f \in \mathcal{F} \),
\( \varphi(f) \) ends in 0 and \( \mu_{\mathcal{R}}^{\text{can}} \subseteq \mu \). If \( \mathcal{R} \) is \( \varphi \)-terminating, then \( \mathcal{S}_{\text{eval}_{\varphi}} \) is normalizing.
Related work With regard to the computation of normal forms by directly using the E-strategy, Nagayama provides conditions (on the TRS and the E-strategy \( \varphi \)) ensuring that \( \varphi \) is normalizing, i.e., it is able to compute a normal form of a term whenever it exists (i.e., he studies completeness of the E-strategy w.r.t. normalization). However, these results concern quite a restricted subclass of orthogonal TRSs. Complementarily, Theorem 8 establishes that completeness is possible for left-linear, confluent and \( \varphi \)-terminating TRSs. However, we need a ‘meta-operation’ (namely, \( S_{\text{eval}_A} \)) that uses \( \text{eval}_{\varphi} \) to obtain partially evaluated results (i.e., \( E \)-normal forms) in order to obtain normal forms.

5 Just-in-time

Let \( \mathcal{R} = (\Sigma, R) \) be a TRS. According to van de Pol [Pol01], a strategy annotation associated to a given symbol \( f \in \Sigma \) is a list \( \zeta(f) \) whose elements can be either

1. a number \( i \) with \( 1 \leq i \leq \text{ar}(f) \); or
2. a rule \( l \rightarrow r \in R \) such that \( \text{root}(l) = f \).

In principle, strategy annotations contain no duplicated items.

Example 9. Consider the TRS [Pol01]:

\[
\begin{align*}
\alpha & : \text{if}(\text{true},x,y) \rightarrow x \\
\beta & : \text{if}(\text{false},x,y) \rightarrow y
\end{align*}
\]

Then, a possible strategy annotation for if is \( \zeta(\text{if}) = [1, \alpha, \beta, 2, 3, \gamma] \).

We say that \( \zeta \) is r-full if for all \( l \rightarrow r \in R, l \rightarrow r \in \zeta(\text{root}(l)) \).

Given a strategy annotation, van de Pol describes the rewriting strategy that it specifies. A strategy is seen as a function that, given a term \( t \), yields either some rewrite of \( t \), i.e., a pair \( (p,s) \) such that \( t \xrightarrow{{\mathcal{R}}} s \), or \( \perp \) if no rewrite step has been selected. Given a term \( t \) and a strategy annotation \( \zeta \), \( \text{rew}_t \) indicates the (unique, if any) rewrite step that can be issued on \( t \).

Definition 1. [Pol01] Let \( \mathcal{R} = (\Sigma, R) \) be a TRS, \( \zeta \) be a strategy annotation, and \( t \in \mathcal{T}(\Sigma, \lambda) \). Then, \( \text{rew}_t(t, \zeta(\text{root}(t))) \), where

\[
\begin{align*}
\text{rew}_t(t, \text{nil}) & = \perp \\
\text{rew}_t(t, (l \rightarrow r : L)) & = \begin{cases} (s, t) & \text{if } t = s(t) \text{ for some } s \\
\text{rew}_t(t, L) & \text{otherwise} \end{cases} \\
\text{rew}_t(t, (i : L)) & = \begin{cases} (i, p, t[x]) & \text{if } \text{rew}_t(t[x]) = (p, s) \text{ for some } p, s \\
\text{rew}_t(t, L) & \text{otherwise} \end{cases}
\end{align*}
\]

We write \( t \xrightarrow{\perp} s \) (or just \( t \rightarrow s \)) if \( (p, s) = \text{rew}_t(t) \neq \perp \). Thus, \( t \) is a \( \rightarrow_{\zeta} \)-normal form (or just a \( \zeta \)-normal form) if and only if \( \text{rew}_t(t) = \perp \). Given a strategy annotation \( \zeta \) for \( \Sigma \), we define \( \mu^{\zeta} \in M_{\Sigma} \) as follows: for all \( f \in \Sigma \), \( \mu^{\zeta}(f) = \{ i \in \mathbb{N}^f | i \in \zeta(f) \} \). We will drop superscript \( \zeta \) if no confusion arises. The following theorem establishes a very close connection between \( \rightarrow_{\zeta} \) and \( \xrightarrow{\mu} \).
Theorem 9. Let \( R \) be a TRS, \( \varsigma \) be a strategy annotation, and \( t, s \in T(\Sigma, \mathcal{X}) \). If \( t \rightarrow_\varsigma s \), then \( t \rightarrow_\mu s \).

According to the definition of \( \rightarrow_\varsigma \), given terms \( t, s \), and \( t' \), it follows that \( t \rightarrow_\varsigma s \) and \( t \rightarrow_\varsigma t' \) imply that \( s = s' \), i.e., each \( \rightarrow_\varsigma \)-reduction step is deterministic.

5.1 Termination

Given a term \( t \) and a strategy annotation, van de Pol defines the reduction sequence associated to \( t \) by concatenating \( \rightarrow_\varsigma \)-steps (see Section 2.3 of [Po01]). Thus, we say that a TRS is \( \varsigma \)-terminating if \( \rightarrow_\varsigma \) is terminating. According to Theorem 9, we have the following immediate consequence.

Theorem 10. Let \( R \) be a TRS and \( \varsigma \) be a strategy annotation. If \( R \) is \( \mu \)-terminating, then it is \( \varsigma \)-terminating.

Termination of \( \rightarrow_\varsigma \) can be more accurately characterized as innermost \( \mu \)-termination. Conditions for doing this are similar to those given in Section 4.1 for \( E \)-strategies.

5.2 Completeness

According to Theorem 9, \( \mu \)-normal forms are always \( \varsigma \)-normal forms. We also have the following.

Theorem 11. Let \( R \) be a TRS, \( \varsigma \) be an \( r \)-full strategy annotation, and \( t \in T(\Sigma, \mathcal{X}) \). If \( t \) is a \( \varsigma \)-normal form, then \( t \) is a \( \mu \)-normal form.

Therefore, for \( r \)-full strategy annotations, \( \varsigma \)-normal forms and \( \mu \)-normal forms coincide. In this case, \( \rightarrow_\varsigma \) can be thought of as the reduction relation \( \rightarrow_\mu \), associated to a (one-step, deterministic) context-sensitive rewriting strategy \( H_\varsigma \), given by \( H_\varsigma(t) = \{ t \rightarrow_\varsigma s \} \).

The main concern of [Po01] is to achieve normalization using \( \rightarrow_\varsigma \). The author introduces the notion of full (and in-time) strategy annotation. A strategy annotation \( \varsigma \) is full if it is \( r \)-full and for all \( f \in \Sigma \) and \( i \in \{1, \ldots, ar(f)\} \), \( i \in \varsigma(f) \).

A strategy annotation \( \varsigma \) is in-time if for all \( f \in \Sigma \), \( \alpha : l \rightarrow r \in R \) such that \( \mathrm{root}(l) = f \), and \( i \in \{1, \ldots, ar(f)\} \), whenever \( \varsigma(f) = a_1 a_2 l_1 a_3 \), then \( i \) is not needed for \( \alpha \). Here, index \( i \) is needed for a rule \( \alpha : l \rightarrow r \) if \( l_1 \not\in \mathcal{X} \) or \( l_1 \not\in \mathcal{X} \) occurs in \( l_1 \) for \( i \neq j \). For example, the strategy of Example 9 is full and in-time\(^4\). There is a trivial connection between needed indices and the canonical replacement map \( \mu^\varsigma \).

Proposition 1. Let \( R = (\Sigma, R) \) be a left-linear TRS and \( \alpha : l \rightarrow r \in R \). If \( i \in \{1, \ldots, ar(\mathrm{root}(l))\} \) is needed for \( \alpha \), then \( i \in \mu^\varsigma(\mathrm{root}(l)) \).

\(^4\) The name just-in-time that entitles this section (and van de Pol’s paper) corresponds to the full and in-time strategy that can be automatically associated to a given TRS (see Section 4 of [Po01]). The just-in-time strategy is aimed at delaying (as much as possible) the evaluation of arguments of function symbols. For instance, strategy \( \varsigma \) in Example 9 is the just-in-time strategy that corresponds to the TRS in the example.
However, there can be indices in $\mu_{R^{\text{rs}}}^\mathcal{R}(f)$ that are not needed for any rule $l \rightarrow r$ such that $\text{root}(l) = f$.

**Example 10.** Consider the TRS $\mathcal{R}$ (based on [Pol01]):

\[
\begin{align*}
\alpha : f(x, a) & \rightarrow f(b, c) \\
\beta : g(f(c, c)) & \rightarrow c
\end{align*}
\]

Index 1 is not needed for $\alpha$ (which is the only rule associated to $f$). However, $1 \notin \mu_{R^{\text{rs}}}^\mathcal{R}(f)$.

Van de Pol also provides the following normalization (partial) function:

**Definition 2.** [Pol01] Let $\mathcal{R} = (\Sigma, \mathcal{R})$ be a TRS, $\varsigma$ be a strategy annotation, and $t \in \mathcal{T}(\Sigma, \mathcal{X})$. Then, $\text{norm}_{\varsigma}(t) = \text{norm}_{\varsigma}(t, \varsigma(\text{root}(t)))$, where

\[
\begin{align*}
\text{norm}_{\varsigma}(t, \text{nil}) &= t \\
\text{norm}_{\varsigma}(t, (l \rightarrow r : L)) &= \begin{cases} 
\text{norm}_{\varsigma}(\mathcal{R}(\varsigma)(r)) & \text{if } t = \mathcal{R}(\varsigma)(l) \text{ for some } \mathcal{R}(\varsigma) \\
\text{norm}_{\varsigma}(t, L) & \text{otherwise}
\end{cases} \\
\text{norm}_{\varsigma}(t, (i : L)) &= \text{norm}_{\varsigma}(t[i]$, $L)
\end{align*}
\]

Then, he proves the following theorems.

**Theorem 12.** [Pol01] If $\varsigma$ is in-time, then $\text{norm}_{\varsigma}(t)$ is the last element of the maximal $\rightarrow_{\varsigma}$-reduction sequence starting from $t$.

**Theorem 13 (Correctness [Pol01]).** Let $\varsigma$ be full and in-time. If $s \in \text{norm}_{\varsigma}(t)$, then $s$ is a normal form.

Unfortunately, no analysis of completeness is given for $\text{norm}_{\varsigma}$. According to Theorem 10 and Theorem 12, $\mu$-termination of $\mathcal{R}$ implies that $\text{norm}_{\varsigma}$ is defined on all terms, i.e., it is actually a mapping. On the other hand, if $\varsigma$ is full (as required in Theorem 13), $\mu$-termination and termination coincide. Thus, no clear (or easily provable) benefit is obtained (regarding termination) from restricting computations. We can improve this situation as follows.

**Theorem 14.** Let $\mathcal{R} = (\Sigma, \mathcal{R})$ be a TRS and $\varsigma$ be $r$-full and in-time. If $s \in \text{norm}_{\varsigma}(t)$, then $s$ is a $\mu$-normal form.

---

In page 40 of [Pol01], van de Pol calls ‘a strategy annotation $\varsigma$ complete if $\text{norm}_{\varsigma}$ is partially correct’, i.e., ‘if $\text{norm}_{\varsigma}(t) = s$, then $s$ is a normal form of $t$’. We think that this terminology is misleading, since it does not correspond to the standard notion of completeness. This can eventually confuse the reader when the author claims that ‘full and in-time are sufficient syntactic criteria for completeness’. Moreover, in page 41, the author refers to completeness of E-strategy annotations which ‘is proved in Theorem 6.1.12 of [Nag99] for full annotations ending in 0’. However, as remarked above, this result concerns correctness; completeness (or normalizability in Nagaya’s terminology) is studied elsewhere in [Nag99], e.g., Theorems 6.2.8 and 6.3.8. On the other hand, Nakamura and Ogata do not study completeness in [NO01].
Now, if $\mathcal{R}$ can be proved $\varsigma$-terminating (e.g., by using Theorem 10), we can use $\text{norm}_\varsigma$ (or $\mathcal{H}_\varsigma$) as a $\mu$-normalizing $\mu$-strategy (where $\text{norm}_\varsigma(t)$ is the singleton containing the unique finite $\to_\varsigma$ sequence starting from $t$ and ending in a $\mu$-normal form of $t$) which can be used within the normalization via $\mu$-normalization process described in Section 4.2.

**Theorem 15 (Normalization via $\varsigma$-normalization).** Let $\mathcal{R} = (\Sigma, \mathcal{R})$ be a left-linear, confluent TRS and $\varsigma$ be an $r$-full and in-time strategy annotation such that $\mu_{\mathcal{R}}^{\text{cont}} \subseteq \mu$. If $\mathcal{R}$ is $\mu$-terminating, then $S_{\text{norm}_\varsigma}$ (and $S_{\text{H}_\varsigma}$) is normalizing.

Unfortunately, $r$-full and in-time alone (i.e., without requiring $\mu_{\mathcal{R}}^{\text{cont}} \subseteq \mu$) do not ensure correctness of the procedure.

**Example 11.** Consider the TRS $\mathcal{R}$ of Example 10 and the $r$-full and in-time strategy annotation $\varsigma$ given by $\varsigma(\text{f}) = [2, a]$, $\varsigma(\text{g}) = [1, a]$, $\varsigma(\text{b}) = [\gamma]$. Hence, $\mu(\text{f}) = \{2\}$, $\mu(\text{g}) = \{1\}$, and $\mu(\text{b}) = \emptyset$. Then we have,

$$\text{norm}_\varsigma(\text{g}(\text{f}(\text{b}, \text{b}))) = \text{g}(\text{f}(\text{b}, \text{c}))$$

which is not a normal form (but it is a $\mu$-normal form). Now, by the normalization via $\varsigma$-normalization process, and since the only non-$\mu$-replacing position of $s = \text{g}(\text{f}(\text{b}, \text{c}))$ is $1$, we recursively normalize $s_{1,1} = \text{b}$ and obtain $s_{\text{norm}_\varsigma(\text{b})}^{[1]} = \text{g}(\text{f}(\text{c}, \text{c}))$ which is a normal form. Note that $\mu_{\mathcal{R}}^{\text{cont}} \not\subseteq \mu$.

The reason for this ‘bad’ behavior of $r$-full and in-time strategy annotations is that 'in-time' does not ensure that computations under (not full, but possibly $r$-full) strategy annotations finally obtain head-normal forms.

**Example 12.** (continuing Example 10) Term $\text{g}(\text{f}(\text{b}, \text{c}))$ obtained as the evaluation of $\text{g}(\text{f}(\text{b}, \text{b}))$ is not a head-normal form, since we have

$$\text{g}(\text{f}(\text{b}, \text{c})) \to \text{g}(\text{f}(\text{c}, \text{c}))$$

and $\text{g}(\text{f}(\text{c}, \text{c}))$ is a redex.

However, we have the following

**Theorem 16.** [Luc98] Let $\mathcal{R} = (\Sigma, \mathcal{R})$ be a left-linear TRS and $\mu \in M_\Sigma$ be such that $\mu_{\mathcal{R}}^{\text{cont}} \subseteq \mu$. Every $\mu$-normal form is a head-normal form.

And hence, according to Theorems 14 and 16, we have:

**Corollary 1.** Let $\mathcal{R} = (\Sigma, \mathcal{R})$ be a left-linear TRS, $\varsigma$ be $r$-full and in-time such that $\mu_{\mathcal{R}}^{\text{cont}} \subseteq \mu$. If $s \in \text{norm}_\varsigma(t)$, then $s$ is a head-normal form.

We note that left-linearity cannot be dismissed in this result.

**Example 13.** Consider the following TRS $\mathcal{R}$ [Luc01b]:

$$\alpha : \text{f}(x, x) \to \text{c}(x)$$

$$\beta : a \to b$$

which is not left-linear, and let $\varsigma(\text{f}) = [1, 2, a]$, $\varsigma(\text{a}) = [\gamma]$, and $\varsigma(\text{b}) = [\varsigma(\text{c}) = [\emptyset]$. Note that $\varsigma$ is in $r$-full and in-time. Term $t = \text{f}(\text{c}(\text{a}), \text{c}(\text{b}))$ is not root-stable:

$$\text{f}(\text{c}(\text{a}), \text{c}(\text{b})) \to \text{f}(\text{c}(\text{b}), \text{c}(\text{b}))$$

and $\text{f}(\text{c}(\text{b}), \text{c}(\text{b}))$ is a redex. However, $\text{norm}_\varsigma(t) = t$. 

References:
Related work We think it fair to point out that ‘in-time’ alone (i.e., without fullness, or even with r-fullness) does not provide better results than obtained using E-strategies (concerning the ability of computing head-normal forms [NO01]). This complements van de Pol’s claim in the introduction of [Pol01] (see page 41): ‘we generalize the mentioned results (by Nakamura and Ogata) by having a more liberal criteria on a larger class of TRSs’, since Nakamura and Ogata’s analysis also concerns strategy annotations which are not full. Moreover, Nagaya’s result on correctness of E-strategies (Theorem 6.1.12 of [Nag99]) is also valid for arbitrary TRSs. However, Nagaya requires that strategy annotations end in 0, at least for defined symbols; this is not necessary in van de Pol’s approach.

6 Conclusions and future work

We have shown that the theory of CSR is useful for analyzing computations with strategy annotations. The analysis of (innermost) termination of CSR provides a characterization (which can even be complete) of termination of OBJ programs with positive local strategies.

Nakamura and Ogata prove that E-strategies \( \varphi_0 \) which compute head-normal forms (i.e., such that if \( s \in \text{eval}_{\varphi_0}(t) \), then \( s \) is a head-normal form of \( t \)) can be completed to E-strategies \( \varphi \) which compute normal forms (i.e., such that if \( s \in \text{eval}_{\varphi}(t) \), then \( s \) is a normal form of \( t \), see Theorem 3.2 in [NO01]). Therefore, this is a result which ensures correctness of \( \text{eval}_{\varphi} \) if constructed from an appropriate \( \varphi_0 \). We conjecture that \( \varphi_0 \)-termination of such E-strategies ensures completeness of \( \varphi \), even without any ‘meta-operation’ (e.g., \( \text{eval}_{\varphi_0} \)). In this way, proofs of termination of CSR would be useful for ensuring completeness (or normalization) of full E-strategies (and also full and in-time strategy annotation). A similar statement would hold for just-in-time style. We plan to address these problems in future work.

References


Computational properties of term rewriting with strategy annotations


A Proof of Correctness for the STG Machine

Alberto de la Encina and Ricardo Peña
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
e-mail: {albertoe,ricardo}@sip.ucm.es

Abstract. J. Launchbury gave an operational semantics for lazy evaluation and showed that it was correct and complete w.r.t. a denotational semantics of the language. P. Sestoft then introduced several abstract machines for lazy evaluation and showed that they were correct and complete w.r.t. Launchbury's operational semantics. We go a step forward and show that the Spineless Tagless G-machine is complete and (almost) correct w.r.t. one of Sestoft's machines. In the way to this goal we also prove some interesting properties about the operational semantics and about Sestoft's machines which clarify some minor points on garbage collection and on closures' local environments. Unboxed values and primitive operators are excluded from the study.

1 Introduction

One of the most successful abstract machines for executing lazy functional languages is the Spineless Tagless G-machine (STG machine) [5] which is at the heart of the Glasgow Haskell Compiler (GHC) [6]. The compiler receives a program written in Haskell [7] and, after some steps and intermediate transformations, produces a program in a very simple functional language called the STG language. This is the input for the STG machine. The back-end then generates imperative code emulating the transitions of the machine.

The STG machine has proved to be efficient compared with some other machines for lazy languages such as the G-machine [2] or the TIM (Three Instructions Machine) [1]. But until recently there has been no formal proof of its correctness. This was provided for the first time by J. Mountjoy in [4]. There, the author starts from Launchbury’s natural semantics for lazy evaluation [3] and transforms it to successive more elaborated semantics. From these semantics he derives a STG-like machine and proves its correctness w.r.t. the semantics. The machine has a single stack (instead of three) and does not treat constructions as normal forms. Additionally, he proves that the more elaborated semantics are in fact equivalent to Launchbury’s.

Launchbury’s semantics is a good starting point because it has been accepted de facto as the reference for defining the meaning of lazy evaluation. It is however debatable whether the right approach to proving the correctness of a machine is to refine the specification so that it gets closer and closer to the desired operational behaviour.

* Work partially supported by the Spanish-British Acción Integrada HB 1999-0102 and Spanish project TIC 2000-0738.
In order to prove the correctness of the STG machine, we have followed a different strategy. We accept Launchbury’s semantics as the specification and we continue with the abstract machines developed by Sestoft in [9] which were shown to be correct and complete w.r.t. Launchbury’s semantics. In order to have a common language, we define a language similar to that of STG which can be considered a subset of the language used by Sestoft’s machines. Then, we define and prove a bisimulation between the Sestoft machine called Mark-2 and the STG. The bisimulation holds for a single-stack STG machine. The one described in [5] and implemented in the first versions of the GHC compiler had three separate stacks. We show that the three stack machine is not correct w.r.t. to the semantics for some ill-typed programs.

Other contributions are: improvements to Sestoft’s semantics in order to solve a small problem related to freshness of variables and to take into account garbage collection. Also, a property about Sestoft’s machines environments is proved.

The plan of the paper is as follows: After this introduction, in Section 2 Launchbury’s semantics is summarized. Then, Sestoft’s and our improvements are presented. Section 3 introduces Sestoft’s Mark-2 machine and presents our proposition about its environments. Section 4 defines the STG-like language, the single-stack STG machine and proves the bisimulation with the Mark-2 machine. Section 5 shows that the three-stack STG machine is complete but not correct w.r.t. the semantics. Finally, Section 6 draws some conclusions.

2 Natural Semantics

2.1 Launchbury’s original proposal

A well-known work from Launchbury [3] defines a big-step operational semantics for lazy evaluation. The only machinery needed is an explicit heap where bindings are kept. A heap is considered to be a finite mapping from variables to expressions, i.e. duplicated bindings to the same variable are disallowed. The language used was a normalized \( \lambda \)-calculus, extended with recursive \textit{let}, (saturated) constructor applications and \textit{case} expressions. To ensure sharing, arguments of applications are forced to be variables. A grammar for this language is given in Figure 1 where the overline notation \( \overline{A} \) denotes a vector \( A_1, \ldots, A_n \) of subscripted entities.

To avoid variable capture, the normalized language has the additional restriction that all bound variables (either lambda, let or case bound) in the initial expression must be distinct. (Weak head) normal forms are either lambda abstractions or constructions. Throughout the paper, the symbol \( w \) will be used to denote normal forms. The semantic rules are reproduced in Figure 2. There, a judgement \( \Gamma : e \Downarrow \Theta : w \) means that expression \( e \), with free variables bound in heap \( \Gamma \), reduces to normal form \( w \) and produces a final heap \( \Theta \). The notation \( \bar{w} \) means expression \( w \) where all bound variables have been replaced by fresh names. We say that \( \Gamma : e \Downarrow \Theta : w \) is a (successful) derivation if it can be proved by using the rules. A derivation can fail to be proved for instance because of entering in a \textit{blackhole}. This would happen in rule \textit{Var} when a reference to variable \( x \) appears while reducing expression \( e \) and before reaching a normal form.

\footnote{Propositions proofs can be found at http://dalila.sip.ucm.es/~albertoe.}
As this is done in a heap $\Gamma$ not containing a binding for $x$, no rule can be applied and the derivation cannot be completed. Other forms of failure are those corresponding to ill-typed programs or infinite loops.

The main theorem in [3] is that the operational semantics is correct and complete with respect to a non-strict denotational semantics for the language. i.e. if $e$ is a closed expression, then $[e]_{\rho_0} = \nu \neq \bot$ if and only if there exist $\Theta$ and $w$ such that $\{ \} : e \triangleright \Theta : w$ and $[w]_{\Theta} = \nu$, being $\rho_0$ the empty environment and $\rho_0 \Theta$ an environment defining the free variables of $w$ and obtained from heap $\Theta$.

2.2 Sestoft's improvements

Sestoft introduces in [9] two main changes to the operational semantics of Figure 2: (1) to move the renaming of variables from the $\text{Var}$ rule to the $\text{Letrec}$ one, and (2) to make in the $\text{Letrec}$ rule the freshness condition locally checkable by extending judgements with a set $A$ of variables under evaluation. The first modification aims at getting the semantics closer to an implementation in terms of abstract machines. In the usual implementations, fresh variables (i.e. pointers) are created when introducing new closures in the heap in the $\text{Letrec}$ rule. This is also more economical than renaming all bound variables in a normal form. The second modification makes more precise the definition of freshness; a variable is fresh in a judgement $\Gamma : e \triangleright_A \Theta : w$ if it does not belong to either dom $\Gamma$ or $A$, and it is not bound either in range $\Gamma$ or $e$. The modified rules can be seen in Figure 3. In the $\text{Letrec}$ rule, the notation $\hat{e}$ means the renaming $e[\overline{p} / \overline{x}]$ where $\overline{p}$ are fresh variables in the judgement $\Gamma : \text{letrec } \overline{x} = \overline{e} \text{ in } e \triangleright_A \Theta : w$.

The difference between the new rules and the ones in Figure 2 is the place where renaming is done. So, the only thing needed to prove the equivalence between the two sets of rules is that there is neither name capture nor duplicated bindings to the same variable.

**Proposition 1 (Sestoft)** Let $e$ be a closed expression and $\{ \} : e \triangleright \{ \} \Theta : w$ a successful derivation. Then, in no instance of rule $\text{App}$ there can be variable capture in $e[\overline{p} / \overline{x}]$, and in no instance of rule $\text{Var}$ is $\overline{p}$ already bound in $\Delta$.

Moreover, Sestoft proves that, in every derivation tree, there is a clean distinction between free variables and bound variables in expressions appearing in judgements and in heaps. The first ones are always pointers (in Figure 3 and in what
follows, they are denoted by $p$, and they are either bound in the corresponding heap, or they are under evaluation and belong to $A$. The second ones are program variables belonging to the original expression (in Figure 3 and in what follows, they are denoted by $x$ or $y$).

Unfortunately, the proof of the theorem was done before introducing case expressions and constructors and, when the latter were introduced, the theorem was not redone. With the current Case rule the freshness property is not locally checkable anymore. While reducing the discriminant in judgement $\Gamma : e \downarrow_A \Delta : C_k \Gamma$, fresh variables may be created with the same names as bound variables in the alternatives, without violating the freshness condition. Then, in the second part of the premise, name capture may occur in expression $e_k [p_j/x_k]$. In the next section we introduce a modification to the rules in order to keep the freshness locally checkable in presence of case expressions.

A problem not addressed by Sestoft is garbage collection. One invariant of the derivation of any expression is that heaps always grow with new bindings, i.e. in every judgement $\Gamma : e \downarrow_A \Delta : w$, it turns out that $\text{dom } \Gamma \subseteq \text{dom } \Delta$. We are interested in having a semantics reflecting that garbage collection may happen at any time without altering the result of the evaluation. To this aim, we develop a set of rules in which all heaps are assumed to contain only live bindings. So, the garbage collector must be explicitly activated in certain rules. This forces us to maintain new sets during a derivation in order to keep all the roots of live bindings. This extension is also done in the next section.

### 2.3 A modified natural semantics

To solve the first problem, i.e. having freshness locally checkable, we introduce a multiset $C$ of continuations associated to every judgement. The alternatives of a case are stored in this multiset during the evaluation of the discriminant. We say then that a variable is fresh in a judgement $\Gamma : e \downarrow_A \Delta : w$ if it does not belong either to $\text{dom } \Gamma$ or to $A$, and it is not a bound variable either in $e$, or in $\text{range } \Gamma$, or in any continuation of $C$.

To provide for garbage collection, we must first decide which are the roots of live closures. Of course, free variables of the current expression belong to the
set of roots. By observing the rules, it is clear that the continuations in set \( C \) should also provide additional roots. Otherwise, a minimal heap during the derivation of the discriminant might not include bindings for the free variables of the alternatives. Symmetrically, during the derivation of the normal form of function \( e \) in rule App, we should include the argument \( p \) of an application in the set of roots. So, we introduce an additional multiset \( B \) in judgements standing for arguments of pending applications. A judgement will have the following form: 
\[ \Gamma : e \Downarrow_{ABC} \Delta : w \] 
where the intention is that \( \Gamma \) be minimal w.r.t. \( e, B \) and \( C \), and \( \Delta \) be minimal w.r.t. \( w, B \) and \( C \).

As the knowledgeable reader may have already noticed, set \( A \) is not an additional source of roots. This set represents bindings currently under evaluation or, in more operational terms, pending updates. If the only reference to a pending update is that of set \( A \), this means that the value of the corresponding free variable will not be used anymore in the future. So, the variable can be safely deleted from \( A \) and the corresponding update avoided\(^2\). Moreover, we want to have also minimal sets of pending updates in our derivations. This means that the set \( A \) associated to the initial expression of a given judgement needs not be the same anymore than the set \( A' \) associated to the final expression. To take this into account, a last modification of judgements is needed. Their final form is the following one:

\[ \Gamma A : e \Downarrow_{BC} \Delta A' : w \]

where \( \Gamma \) and \( A \) are minimal w.r.t. \( e, B \) and \( C \), and \( \Delta \) and \( A' \) are minimal w.r.t. \( w, B \) and \( C \). Its meaning is that expression \( e \) reduces to normal form \( w \) starting with heap \( \Gamma \) and set \( A \), and ending with heap \( \Delta \) and set \( A' \).

That heaps and sets of pending updates are minimal is just a property that must be proved. To preserve this property in derivations, garbage collections and trimming of pending updates must be activated at certain points. The semantic rules exactly clarify which these points are.

**Definition 2** Given a heap \( \Gamma \), an expression \( e \), a multiset \( B \) of variables, and a multiset \( C \) of continuations, we define the set of live variables of \( \Gamma \) w.r.t. \( B, C \) and \( e \), denoted \( \text{live}^{BCe}_{\Gamma} \):

\[
\text{live}^{BCe}_{\Gamma} = \text{fix} (\lambda L. L \cup \text{fv}e \cup B \cup \text{fv}C \cup \bigcup_{p \in L} \{ \text{fv}e' \mid (p \mapsto e') \in \Gamma \})
\]

where \( \text{fv}e \) denotes the set of free variables of expression \( e \), \( \text{fv}C \) is the obvious extension to a continuation and \( \text{fix} \) denotes the least fixed-point.

**Definition 3** Given a heap \( \Gamma \), a set of pending updates \( A \), an expression \( e \), a multiset \( B \) of variables, and a multiset \( C \) of continuations, we define the live heap of \( \Gamma \) w.r.t. \( B, C \) and \( e \), denoted \( \Gamma^{BCe}_{A} \), and the subset of live updates of \( A \) w.r.t. \( \Gamma, B, C \) and \( e \), denoted \( A^{BCe}_{\text{live}^{BCe}_{\Gamma}} \):

\[
\Gamma^{BCe}_{A} = \{ p \mapsto e \mid (p \mapsto e) \in \Gamma \land p \in \text{live}^{BCe}_{\Gamma} \}
\]

\[
A^{BCe}_{\text{live}^{BCe}_{\Gamma}} = A \cap \text{live}^{BCe}_{\Gamma}
\]

\(^2\) This trimming of the set of pending updates is done in the STG machine after each garbage collection. See [5, Section 10.7].
In a judgment $\Gamma A : e \cdot B C \Delta A' : w$, if a minimal heap and update set should be ensured before the derivation starts, we will write $\Gamma_{BC_e} A_{BC_e} : e \cdot B C \Delta A' : w$, meaning that the initial heap and update set should respectively be $\Gamma_{BC_e}$ and $A_{BC_e}$. These $gc$ annotations exactly mark the points in a derivation where a garbage collection or a trimming of the update set may be needed.

The new set of rules is shown in Figure 4. Some explanations follow:

**Maintaining the correct set of roots** When evaluating the discriminant of a case (see rule Case), the pending alternatives must be included in set $C$ in order to avoid losing bindings for the free variables of the alternatives. Also, when evaluating the function of an application (see rules $AppA$ and $AppB$), the argument must be included in set $B$ in order to avoid losing the binding for it.

**Activating garbage collection in the appropriate points** The $gc$ annotation, meaning the trimming of a heap or of an update set, must be written in those points where there may be the danger of dead bindings. These are:

- in rule $AppB$, when the parameter of the function does not appear in the body. There is a danger that the binding for $p$ in $\Delta$ becomes dead.
- in rules $VarA$ and $VarB$, when the reference to $p$ disappear from the current expression. There may be no other reference to $p$ either in $e$, $B$, or $C$.
- in rule $Case$, when a particular alternative is chosen. The discarded alternatives may have free variables that now are dead.

**Avoiding unnecessary updates** This is reflected in rule $VarB$. Assuming that the pair $(\Delta, A')$ is minimal w.r.t. $w$, $B$, and $C$, and knowing that $p \notin A'$, then the update for variable $p$ may be safely discarded (compare with rule $VarA$).

**Assuming no dead code in letrec** Notice in the antecedent of rule Letrec that no garbage collection is launched. So, we are assuming that all the new bindings are live. This is not true if there exists dead code in the $letrec$ expression. It is easy for a compiler to eliminate unreachable bindings in a $letrec$. In what follows we will assume that dead code has been eliminated in our programs.

**New definition of freshness** In the consequent of rule Letrec a set $p$ of fresh variables is created. A variable is fresh in judgment $\Gamma A : e \cdot B C \Delta A' : w$ if it does not belong to either $dom \Gamma$ or $A$, and it is not bound either in $range \Gamma$, $e$ or $C$.

We will see now that the properties desired for our semantics in fact hold.

**Definition 4** Given a judgment $\Gamma A : e \cdot B C \Delta A' : w$, we say that the configuration $\Gamma ; e$ is $ABC$-good, if

1. $A \cap \\text{dom} \ \Gamma = \emptyset$
2. $\text{live}_{B C_e} \Gamma = A \cup \\text{dom} \ \Gamma$
3. $(\text{bv} \Gamma \cup \text{bv} e \cup \text{bv} C) \cap (A \cup \text{dom} \ \Gamma) = \emptyset$

where $\text{bv} e$ denotes the bound variables of expression $e$, $\text{bv} \Gamma$ its extension to all expressions in $range \ \Gamma$, and $\text{bv} C$ its extension to a continuation.

The first property has a similar counterpart in Sestoft’s semantics and it asserts that variables under evaluation are not at the same time defined in the heap. The second one asserts that every free variable is either defined in the heap or is under evaluation and also that the pair $(\Gamma, A)$ is minimal w.r.t. $B, C$ and $e$. The third one asserts that free variables are different from bound ones.
This definition ensures that the starting point of a derivation already meets the requirements we want for the whole derivation. The following proposition and corollary establish that the desired properties in fact hold.

**Proposition 6** Let $\Gamma : e : \|_{BC} \Delta' : w$ be a derivation using the rules of the semantics. If it is a promising judgement, then

1. The configuration $\Delta : w$ is $A'BC$-good.
2. $A' \subseteq A$
3. Every judgement in the derivation is a promising one.

**Corollary 7** Let $e$ be a closed expression and $\Gamma \{e\} : e : \|_{BC} \Delta : w$ be a derivation. Then,

1. In no instance of rules AppA and Case there can be variable capture in substitutions of the form $e[p/x]$.
2. In no instance of rule VarA is $p$ already bound in $\Delta$.

The differences between our semantics and Sestoft’s are two:

1. Sestoft’s rules App and Var have been split into two in our semantics. In the first case, the distinction is due to the desire of not launching garbage collection when it is not needed, but in fact both rules could be combined in the following single one:

   $$\Gamma : e : \|_{BC[D]} \Delta' : \lambda x e' \Delta x', A'_e : e'[p/x] : \|_{BC} \Theta A'' : w$$

   $$\Gamma : e : \|_{BC} \Theta A'' : w$$

   In the second case, our rule VarB does not add to the heap a binding $[p \mapsto w]$ that is known to be dead.

2. Our heaps and update sets are minimal in the corresponding judgements.

Otherwise, the semantic rules are the same. Once we have proved that free variables in judgements are either defined in the heap, or they belong to the pending updates set, both semantics produce exactly the same derivations.

---

**Fig. 4.** A natural semantics with minimal heaps and minimal update sets

**Definition 5** A judgement $\Gamma : e : \|_{BC} \Delta' : w$ is promising if the configuration $\Gamma' : e$ is $ABC$-good.

This definition ensures that the starting point of a derivation already meets the requirements we want for the whole derivation. The following proposition and corollary establish that the desired properties in fact hold.

**Proposition 6** Let $\Gamma : e : \|_{BC} \Delta' : w$ be a derivation using the rules of the semantics. If it is a promising judgement, then

1. The configuration $\Delta : w$ is $A'BC$-good.
2. $A' \subseteq A$
3. Every judgement in the derivation is a promising one.

**Corollary 7** Let $e$ be a closed expression and $\Gamma \{e\} : e : \|_{BC} \Delta : w$ be a derivation. Then,

1. In no instance of rules AppA and Case there can be variable capture in substitutions of the form $e[p/x]$.
2. In no instance of rule VarA is $p$ already bound in $\Delta$.

The differences between our semantics and Sestoft’s are two:

1. Sestoft’s rules App and Var have been split into two in our semantics. In the first case, the distinction is due to the desire of not launching garbage collection when it is not needed, but in fact both rules could be combined in the following single one:

   $$\Gamma : e : \|_{BC[D]} \Delta' : \lambda x e' \Delta x', A'_e : e'[p/x] : \|_{BC} \Theta A'' : w$$

   $$\Gamma : e : \|_{BC} \Theta A'' : w$$

   In the second case, our rule VarB does not add to the heap a binding $[p \mapsto w]$ that is known to be dead.

2. Our heaps and update sets are minimal in the corresponding judgements.

Otherwise, the semantic rules are the same. Once we have proved that free variables in judgements are either defined in the heap, or they belong to the pending updates set, both semantics produce exactly the same derivations.
After revising Launchbury’s semantics, Sestoft introduces in [9] several abstract machines in sequence, respectively called Mark-1, Mark-2 and Mark-3. The one we will use for deriving an STG-like machine is Mark-2. There, a configuration consists of a heap $\Gamma$, a control expression $e$ possibly having free variables, an environment $E$ mapping free variables to pointers in the heap, and a stack $S$. The heap $\Gamma$ is a function from pointers to closures, each one $(e,E)$ consisting of an expression $e$ and an environment $E$ binding its free variables to pointers. The stack contains three kinds of objects: (1) arguments of pending applications, represented by pointers; (2) continuations of pending pattern matchings, each one consisting of a pair $(alts,E)$ where $alts$ is a vector of case alternatives and $E$ is an environment binding its free variables; and (3) update markers of the form $\#p$, where $p$ is a pointer.

The reader may have already recognized that stack $S$ represents in fact the union of sets $B$, $C$ and $A$ we introduced in the revised semantics of Section 2.3. The main difference now is that these entities form a list instead of a set or a multiset, and that they appear ordered from more recent to older ones. In Figure 5 the operational rules of Mark-2 machine are shown.

We have followed Sestoft’s convention that program variables are denoted by $x$ or $y$, and pointers by $p$. The machine never makes explicit substitutions of pointers for program (free) variables as the semantics does. Instead, it maintains environments mapping program variables to pointers. If $e$ is a closed expression, the initial configuration is $((\{\}, e, \{\}, [])$. The machine stops when no rule can be applied. If the final configuration has the form $(\Gamma, w, E, [])$, then $w$ has been successfully derived from $e$ and we write $((\{\}, e, \{\}, []) \Rightarrow* (\Gamma, w, E, [])$.

The main theorem proved by Sestoft, is that successful derivations of the machine are exactly the same as those of the semantics.

---

**Table 1:** Abstract machine Mark-2

<table>
<thead>
<tr>
<th>Heap $\Gamma$</th>
<th>Control $e$</th>
<th>Environment $E$</th>
<th>Stack $S$</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma$</td>
<td>$(e, x)$</td>
<td>$E \cup [x \mapsto p]$</td>
<td>$S$</td>
<td>appl1</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma$</td>
<td>$e$</td>
<td>$E \cup [x \mapsto p]$</td>
<td>$p : S$</td>
<td>app2</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma$</td>
<td>$\lambda y.e$</td>
<td>$E$</td>
<td>$p : S$</td>
<td>app2</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma \cup [p \mapsto (e', E')]$</td>
<td>$x$</td>
<td>$E \cup [x \mapsto p]$</td>
<td>$S$</td>
<td>var1</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma$</td>
<td>$e'$</td>
<td>$E'$</td>
<td>$#p : S$</td>
<td>var2</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma \cup [p \mapsto (\lambda y.e, E)]$</td>
<td>$\lambda y.e$</td>
<td>$E$</td>
<td>$#p : S$</td>
<td>var2</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma \cup [p \mapsto (\lambda y.e, E)]$</td>
<td>$\lambda y.e$</td>
<td>$E$</td>
<td>$S$</td>
<td>var2</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma \cup [p \mapsto (\text{letrec } (x_1 = e_1) \text{ in } e, E)]$</td>
<td>$e$</td>
<td>$E'$</td>
<td>$S$</td>
<td>letrec $(\ast)$</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma$</td>
<td>$\text{case } e \text{ of } alts$</td>
<td>$E$</td>
<td>$S$</td>
<td>case1</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma$</td>
<td>$e$</td>
<td>$E \cup [\text{alts}, E] : S$</td>
<td>$S$</td>
<td>case2 $(\ast\ast)$</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma$</td>
<td>$C_1 \varphi_1$</td>
<td>$E \cup [\varphi_1 \mapsto p_1]$</td>
<td>$(\text{alts}, E') : S$</td>
<td>case2 $(\ast\ast)$</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma$</td>
<td>$e_k$</td>
<td>$E'$ \cup $[\varphi_k \mapsto E']$</td>
<td>$S$</td>
<td>case2 $(\ast\ast)$</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma$</td>
<td>$C_k \varphi_k$</td>
<td>$E$</td>
<td>$#p : S$</td>
<td>var3</td>
</tr>
<tr>
<td>$\Rightarrow \Gamma \cup [p \mapsto (C_k \varphi_k, E)]$</td>
<td>$C_k \varphi_k$</td>
<td>$E$</td>
<td>$S$</td>
<td>var3</td>
</tr>
</tbody>
</table>

$(\ast)$ $p_1$ are distinct and fresh w.r.t. $\Gamma$, letrec $(x_1 = e_1) \text{ in } e$, and $S, E' = E \cup [x_1 \mapsto p_1]$

$(\ast\ast)$ Expression $e_k$ corresponds to alternative $C_k \varphi_k \mapsto e_k$ in $alts$

**Figure 5.** Abstract machine Mark-2
Proposition 8 (Sestoft) For any closed expression $e$, then
\[(\{\}, e, \{\}, []) \Rightarrow^* (\Gamma, w, E, []) \text{ if and only if } \{\} e : \downarrow_1 \Gamma : w\]

It is worth to note that the correctness and completeness of machine Mark-2 w.r.t. the operational semantics (by transitivity, w.r.t. the denotational semantics), does not rely on programs being well typed. If a program is ill-typed both the machine and the semantics will be unable to derive a normal form for it.

3.1 Some properties of environments

Mark-2 environments have a complex evolution: they grow with lambda applications, pattern matching and \texttt{letrec} execution; they are stored either in closures or in the stack in some transitions, and they are retrieved from there in some other transitions. It is natural to wonder about how much can they grow.

Definition 9 A closure $(e, E)$, is consistent if
1. \(fv e \cap bv e = \emptyset\) and all variables in $bv e$ are distinct.
2. \(fv e \subseteq \text{dom} E\).
3. \(bv e \cap \text{dom} E = \emptyset\)

This definition can be easily extended to a continuation of the form $(\text{alts}, E)$ and to a heap $\Gamma$ consisting of a set of closures.

Definition 10 A configuration $(\Gamma, e, E, S)$ of machine Mark-2 is consistent if
1. $\Gamma$ is consistent.
2. The pair $(e, E)$ is consistent.
3. All continuations $(\text{alts}, E) \in S$ are consistent.

These definitions only take care of program variables being well defined in environments. That pointers are well defined in the heap (or they belong to stack $S$ as update markers) was already proved by Sestoft for all his machines.

Proposition 11 Let $e$ be a closed expression in which all bound variables are distinct, and $(\{\}, e, \{\}, []) \Rightarrow^* (\Gamma', e', E, S)$ any (possibly partial) derivation of machine Mark-2. Then,
1. $(\Gamma', e', E, S)$ is consistent.
2. $E$ exactly binds all variables in scope in expression $e'$.
3. In any closure $(e_i, E_i) \in \Gamma$, $E_i$ exactly binds all variables in scope in $e_i$.
4. In any pair $(\text{alts}, E) \in S$, $E$ exactly binds all variables in scope in $\text{alts}$.

4 The abstract machine STG-1S

4.1 The common language

In order to get closer to the STG machine, firstly we define a common $\lambda$-calculus for both machines, Mark-2 and STG. This is presented in Figure 6 and we call it FUN. It is equivalent to the STG language (STGL) excluding primitive values as expressed in a more traditional $\lambda$-calculus syntax.

Compared to the original Mark-2 $\lambda$-calculus (see Figure 1), it is clear that FUN is just a subset of it, having the restrictions: that lambda abstractions may only appear in bindings and that applications have the form $x \overline{x_n}$ understood by Mark-2 as $(\ldots (x \overline{x_1}) \ldots) \overline{x_n}$. 
e → x \overline{x^{\downarrow}} -- n ≥ 0, application/variable
  | letrec bind e in e -- recursive let
  | C \overline{x^{\downarrow}} -- constructor application
  | case e of alt t  -- case expression
bind → x = if \{\}
if → λ \overline{x^{\downarrow}}.e -- n ≥ 0, lambda form
alt → C \overline{x^{\downarrow}} → e

Fig. 6. Definition of FUN

The notation λ \overline{x^{\downarrow}}.e is an abbreviation of λx_1.⋯.λx_n.e, where the arguments have been numbered downwards for convenience. In this way, λ \overline{x^{\downarrow^{n-1}}}.e means λx_{n−1}.⋯.λx_1.e and x \overline{x^{\downarrow^1}} means x x_1 ⋯ x_n. When n = 0 we will simply write e instead of λ \overline{x^{\downarrow}}.e and x instead of x \overline{x^{\downarrow}}.

A last feature added to FUN is trimmers. The notation if \{\} means that a lambda form is annotated at compile time with the set \(t\) of its free variables. This set \(t\) was called a trimmer in [9]. It will be used when constructing a closure in the heap for the lambda form. The environment stored in the closure will only bind the variables contained in the trimmer. This implies a small penalty in terms of execution time but a lot of space saving. Analogously, alt \{\} means the annotation of a set of case alternatives with the trimmer \(t\) of its free variables. When alt \{\} is pushed in to the stack, its associated environment will be trimmed according to \(t\). Both optimizations are done in the STG machine, even though the second one is not reflected in the rules given in [5].

4.2 Mark-2 machine for language FUN

In Figure 7, the transition rules of Mark-2 for language FUN are shown. Let us note that the control expression is in general a lambda form if. In particular, it can also be an expression e, if if = λ \overline{x^{\downarrow}}.e. Also, all occurrences of superscripts \(n\) are assumed to be \(n > 0\). Note that, this is not a different machine, but just the same machine Mark-2 executed with a restricted input language. Additionally, there are some optimizations which do not essentially affect the original behavior:

- Original rule var1 has been split into three: the one corresponding to the original var1 is now called var1c; the two other rules are just special cases in which the expression referenced by pointer \(p\) is a normal form. The original Mark-2 machine will execute in sequence either rule var1 followed by rule var2, or rule var1 followed by rule var3. These sequences have been respectively subsumed in the new rules var1a and var1b.
- Trimmer sets have been added to lambda forms and to continuations. Environments are trimmed to the set of free variables when new closures are created in the heap in rules letrec and var3, and also when continuations are stored in the stack in rule case1. This modification only affects to the set of live closures in the heap which now is smaller. Otherwise, the machine behavior is the same.

In Figure 7, a new column Last has been added recording the last rule executed by the machine. This field is important to define stable configurations, which will be used to compare the evolution of Mark-2 and STG-1S (see Section 4.3).
In the STG all these intermediate states do not exist. If the last rule applied is in which a FUN expression appears in the control, all these states must be pattern matching has not been done yet. As we want to compare configurations in which a FUN expression appears in the control, all these states must be regarded as ‘internal’. The second possibility is just a termination state.

Definition 12 A configuration \((Γ, l, E, S, I)\) of machine Mark-2 is stable if one of these two conditions hold:

1. \(l = e \land l \notin \{\text{app}1, \text{var}3\}\), or
2. \(S = \emptyset \land (l = \lambda e \Rightarrow e \land n > 0) \lor l = C \varnothing\)

In the STG machine, lambda abstractions never appear in the control expression, so it seems natural to exclude lambda abstractions from stable configurations. If the last rule executed is \(\text{app}1\), then Mark-2 is still pushing arguments in the stack and it has not yet evaluated the variable \(x\) corresponding to the function to be applied. In the STG all these intermediate states do not exist. If the last rule applied is \(\text{var}\), then the STG is probably still doing updates and, in any case, pattern matching has not been done yet. As we want to compare configurations in which a FUN expression appears in the control, all these states must be regarded as ‘internal’. The second possibility is just a termination state.

Definition 13 Let us assume that \(m\) and \(m'\) are stable configurations of Mark-2 machine, and \(m \Rightarrow^{+} m'\), (i.e. there must be at least one transition) and there is no other stable configuration between \(m\) and \(m'\). We will say that \(m\) evolves to \(m'\) and will denote it by \(m \Rightarrow^{+} m'\).

### 4.3 The machine STG-1S

In this section we define an abstract machine very close to the STG [5] and show that it is correct and complete w.r.t. Mark-2 of Figure 7. We call it STG-1S.
because the main difference with the actual STG is that it has one stack instead of three. The single stack of STG-1S, contains the three usual kind of objects: arguments of applications, continuations and update markers. Being faithful to STGL, the control expression of STG-1S may have three different forms:

- **Eval e E**, where e is a FUN expression (we recall that this excludes lambda forms) and E is an environment mapping e’s free variables to heap pointers.
- **Enter p**, where p is a heap pointer. Notice that there is no environment.
- **ReturnCon C \( \pi \)**, where C is a data constructor and \( \pi \) are its arguments given as a vector of heap pointers. Also, there is no environment here.

We will call each of these expressions an **instruction**, and use the symbol i to denote them. In order to better compare it with the **Mark-2** machine, we will consider a configuration of the STG-1S to be a 4-tuple \((\Gamma, i, E, S)\), where \(\Gamma\) is a heap mapping pointers to closures, i is the control instruction, E is the environment associated to instruction i in case the instruction is of the form **Eval e**, and the empty environment \(\{\}\) otherwise, and S is the stack. In Figure 8, the transition rules of STG-1S are shown.

We have numbered the rules with the same numbers used in [5] for easy reference. As there is no explicit flag \(\forall\pi\) in FUN lambda forms in order to avoid unnecessary updates, rules 2 and 2' reflect that no update frame is pushed in the stack when explicit normal forms in the heap are referenced. Rule 2’ does not appear in [5], but it is implicit in rule 2.
Now we proceed with the comparison. As in Mark-2, we first define stable configurations in STG-1S. A stable configuration corresponds either to the evaluation of a FUN expression or to a termination state.

**Definition 14** A configuration \( s = (\Gamma, i, E, S) \) of machine STG-1S is stable if
1. \( i = \text{Eval } e \) for some \( e \), or
2. \( s = (\Gamma, \text{ReturnCon } C \ \overline{\pi}_x, \{\}, \{\}) \), or
3. \( s = (\Gamma \cup \{ p \mapsto (\lambda x^{(m+1)}, E') \}, \text{Enter } p, \{\}, \{p_1, \ldots, p_k\} \land n > k \geq 0 \).

Configurations 2 and 3 correspond to termination states. Notice in 3 that the STG-1S may successfully stop with a non-empty stack. This would happen when the initial expression evaluates to a lambda abstraction. As in Mark-2 machine, we will use \( s \xrightarrow{+} s' \) to denote the evolution between two stable configurations in STG-1S with no intermediate stable ones, and say that \( s \) evolves to \( s' \). The notion of consistent configuration for the STG-1S machine is the same given in Definition 10 for machine Mark-2.

We will now compare two evolutions, one in each machine starting from equivalent states, and show that they exactly pass through the same number of stable configurations and that the corresponding configurations are equivalent. This amounts to saying that there exists a bisimulation between the machines. To simplify to notion of configuration equivalence, we will assume that both machines use exactly the same fresh name generator in rule \( \text{letrec}. \) So, if the generator is given the same inputs (i.e. the same control expression, heap and stack), it will generate the same set of fresh variables.

**Definition 15** A configuration \( m = (\Gamma, i, E, S, l) \) in a stable state of machine Mark-2 and a configuration \( s = (\Gamma', i, E', S') \) in a stable state of machine STG-1S are said to be equivalent, written \( m \equiv s \), if

- \( \Gamma = \Gamma' \), and
- one of the following possibilities holds:
  1. \( i = \text{Eval } e \land i' = e \land E = E' \land S = S' \)
  2. \( i = \text{ReturnCon } C \ \overline{\pi}_x \land i' = C \ \overline{\pi}_x \land \overline{\pi}_x = E \ \overline{\pi}_x \land S = S' = \{\} \)
  3. \( i = \text{Enter } p \land i' = (\lambda x^{(m+1)}, E') \land S' = \{p_1, \ldots, p_k\} \land n > k \geq 0 \land i' = \lambda x^{(m+1)}e \land E [x_n, \ldots, x_{n-k+1}] = [p_1, \ldots, p_k] \)

The following proposition and corollary establish that STG-1S and Mark-2 machines bisimulate each other. By transitivity this shows that STG-1S is correct and complete w.r.t. Lauchbury’s natural semantics.

**Proposition 16** Given two stable and consistent configurations \( m \) and \( s \) in respectively Mark-2 and STG-1S machines such that \( m \equiv s \),

1. If \( m \xrightarrow{+} m' \), then there exists a stable and consistent configuration \( s' \) such that \( s \xrightarrow{+} s' \) and \( m' \equiv s' \).
2. If \( s \xrightarrow{+} s' \), then there exists a stable and consistent configuration \( m' \) such that \( m \xrightarrow{+} m' \) and \( m' \equiv s' \).

**Corollary 17** If \( e \) is a closed FUN expression, then \( (\{\}, e, \{\}, [], \perp) \Rightarrow^* m_f \) in Mark-2 machine with \( m_f = (\Delta, w, E, [\]) \) if and only if there exists a stable configuration \( s_f \) such that \( (\{\}, \text{Eval } e, \{\}, [], \perp) \Rightarrow^* s_f \) in STG-1S and \( m_f \equiv s_f \).
5 The abstract machine STG

It has three stacks: the argument stack as containing arguments for pending applications; the return stack rs containing pairs (alts, E); and the update stack us containing update frames. An update frame is a triple (as, rs, p) consisting of an argument stack, a return stack and a pointer p to the closure to be updated. We do not show the STG rules as they can be easily derived from those of STG-1S. A configuration will be a 6-tuple (Γ, i, E, as, rs, us).

The two differences with the STG-1S machine of previous section are:

- Pushing and popping is done in the appropriate stack according to the rule.
- Instead of pushing update markers, the STG machine pushes update frames and leaves empty argument and return stacks in the configuration. When a normal form is reached with empty stacks, an update is triggered.

Apparently, these differences are not essential and one may think that the behaviours of both machines are the same. This is not the case as we will see in a moment. The splitting of the single stack into three has the unfortunate consequence of losing the temporal order of events between stacks as and rs. Then, a continuation pushed into rs before an argument is pushed into as can be retrieved also before the argument is retrieved from as instead of after, as it would be the case in the STG-1S machine. Consider the following ill-typed program:

\[ e = \text{letrec } y_1 = \text{Nil}; \text{id} = \lambda x. x \text{ in case } y_1 y_1 \text{ of Nil \to id} \]

which has no semantics. The STG machine reduces it as follows:

\[
(\{\}, e, \{\}, [], [], [\]) \\
\Rightarrow (\Gamma_1, \text{Eval (case } y_1 y_1 \text{ of Nil \to id), } E_1, [\], [\], [\]) \\
\Rightarrow (\Gamma_1, \text{Eval (y_1 y_1), } E_1, [\], [(\text{Nil \to id, } E_1)], [\]) \\
\Rightarrow (\Gamma_1, \text{Enter p_1, } [\], [p_1], [(\text{Nil \to id, } E_1)], [\]) \\
\Rightarrow (\Gamma_1, \text{Eval Nil, } [\], [p_1], [(\text{Nil \to id, } E_1)], [\]) \\
\Rightarrow (\Gamma_1, \text{ReturnCon Nil }\{\}, [p_1], [(\text{Nil \to id, } E_1)], [\]) \\
\Rightarrow (\Gamma_1, \text{Eval id, } E_1, [p_1], [\], [\]) \\
\Rightarrow (\Gamma_1, \text{Enter p_2, } [\], [p_1], [\], [\]) \\
\Rightarrow (\Gamma_1, \text{Eval x, } [x \to p_1], [\], [\], [\]) \\
\Rightarrow (\Gamma_1, \text{Eval Nil, } [\], [], [\], [\]) \\
\Rightarrow (\Gamma_1, \text{ReturnCon Nil }\{\}, [\], [\], [\])
\]

where \(\Gamma_1 = [p_1 \mapsto [\text{Nil}, \{\}], p_2 \mapsto (\lambda x. x, \{\})]\) and \(E_1 = [y_1 \mapsto p_1, \text{id} \mapsto p_2]\).

So, the correctness of the STG machine with three stacks relies on programs being well-typed. This was not the case with the STG-1S machine: if a program is ill-typed, both Launchbury's semantics and STG-1S will be unable to derive a normal form for it.

However, the STG is complete in the sense that every successful derivation done by the STG-1S can obviously be done by the STG. For every configuration of the STG-1S we can exactly compute a single equivalent configuration in the STG machine. The opposite is not true, i.e. given the three stacks as, rs and us of STG, many different stacks for the STG-1S can be constructed by interleaving the contents of the corresponding sections of stacks as and rs.
6 Conclusions

The paper has followed all the way from an abstract operational semantics for lazy evaluation such as Launchbury's, to a very concrete and efficient abstract machine such as the STG. Part of that way had already been followed by Sestoft in [9]. We have started at one of his machines, the Mark-2, and have shown that a STG machine with one stack can be derived from it, and that a bisimulation can be defined between both.

We have solved a small problem of Sestoft's semantics regarding freshness of variables and also added some garbage collection considerations to his semantics. As a result, the stack of Sestoft's machines appears very naturally as a transformation of some sets $A$, $B$, and $C$ needed by the semantics in order to have a complete control over freshness and over live closures. It is interesting to note that the optimization of not using the set of update markers as roots for the garbage collector can be easily understood at the semantic level.

We have also shown that the correctness of the three stacks STG machine as described in [5] relies on program being well-typed. This was an underlying assumption which was not explicitly stated in that description. The obvious solution to this 'problem' is to come back to a single stack machine, and this seems to be the option recently chosen by GHC's implementors [although probably due to different reasons] [8]. Having only one stack complicates the garbage collector task because pointers and non-pointers must be clearly distinguished. The presence of unboxed primitive values in the stack makes the problem even worse. In compensation, update markers are smaller than update frames and, most important of all, the temporal order of events is preserved.

References

Improving the Efficiency of Non-Deterministic Computations

Sergio Antoy\textsuperscript{1}  Pascual Julián Iranzo\textsuperscript{2,3}  Bart Massey\textsuperscript{1}

Computer Science Department
Portland State University
Portland, Oregon
{antoy, pjulian, bart}@cs.pdx.edu

Abstract. Non-deterministic computations greatly enhance the expressive power of functional logic programs, but are often computationally expensive. We analyze two programming techniques that improve the time and memory efficiency of some non-deterministic computations. These techniques rely on the introduction of a new symbol into the signature of a program. In one technique, this symbol is a polymorphic defined operation, in the other an overloaded constructor. Our programming techniques may save execution time by reducing the number of steps of a computation, as well as memory occupation, by reducing the number of terms constructed by a computation. We show how to apply our techniques using some examples, and informally reason about their effects.

1 Introduction

Functional logic programming studies the design and implementation of programming languages that integrate both functional programming and logic programming into a homogeneous paradigm. In recent years, it has become increasingly evident that non-determinism is an essential feature of these integrated languages. Non-determinism is a cornerstone of logic programming. It allows problem solving using programs that are textually shorter, easier to understand and maintain, and more declarative than their deterministic counterparts.

In a functional logic programming language, non-deterministic computations are modeled by the defined operations of a constructor-based left linear conditional rewrite system. With respect to logic computations, which are based on resolution, functional logic computations are nested and therefore can be lazily executed. The combination of these features makes functional logic languages both more expressive than functional languages and more efficient than traditional logic languages.

\textsuperscript{1} Supported in part by the NSF grant INT-9981317.
\textsuperscript{2} Supported by the Spanish Knowledge Society Foundation.
\textsuperscript{3} Permanent Address: Univ. Castilla La-Mancha, Spain, pjulian@inf-cr.uclm.es.
A typical approach to the definition of non-deterministic computations is by means of the defined operations of a constructor based non-confluent rewrite system. The following emblematic example [8, Ex. 2] defines an operation, coin, that non-deterministically returns either zero or one. Natural numbers, represented in Peano notation, are defined by the datatype (or sort) nat.

\[
\text{datatype nat} = 0 \mid s \text{ nat}
\]

\[
\text{coin} = 0
\]

\[
\text{coin} = s \text{ 0}
\]

Rewrite systems with operations such as coin are non-confluent. A computation in these rewrite systems may have distinct normal forms and/or non terminate. To understand non-determinism in the context of a computation, consider the following operations:

\[
\text{add 0 Y = Y}
\]

\[
\text{add (s X) Y = s (add X Y)}
\]

\[
\text{positive 0 = false}
\]

\[
\text{positive (s _) = true}
\]

The evaluation of a term such as positive (add coin 0) requires the evaluation of subterm coin. This subterm has two replacements, i.e., 0 and s 0. Each replacement leads to a different final result. The choice between these two replacements is non-deterministic. Assuming that non-determinism is appropriately used in the program where the evaluation occurs, there is no feasible means of deciding which replacement should be chosen at the time coin is evaluated. Therefore, evaluation under both replacements must be considered.

To ensure operational completeness, all the possible replacements of a non-deterministic computation must be executed fairly. In fact, if one replacement is executed only after the computation of another replacement is completed, the second replacement will never be executed if the computation of the first replacement does not terminate. Thus, continuing with our example, to compute positive (add coin 0) one must compute fairly and independently both positive (add 0 0) and positive (add (s 0) 0).

This approach, which we refer to as fair independent computations, captures the intended semantics, but clearly it is computationally costly. In some situations the cost of fair independent computations might be avoided. For example, define a “bigger” variant of coin:

\[
\text{bigger} = s 0
\]

\[
\text{bigger} = s (s 0)
\]

and consider again the previous example, but invoking bigger instead of coin. The evaluation of positive (add bigger 0), which will be shown in its entirety later, may be carried on, as in the previous example, using fair independent computations. However, this is not necessary. The computation has a single result that may be obtained using only deterministic choices. Avoiding fair independent computations saves execution time, memory occupation, and the duplication of the result.

In this paper, we discuss two programming techniques that have been considered within a project aiming at the implementation of a back-end for a wide class of functional logic languages [7]. In some cases, these techniques have the po-
tential to offer substantial improvements. In other cases, they tend to consume slightly more memory, but without a substantial slowdown. We are currently working to assess whether either or both techniques should be deployed in the back-end: this document is a report of our preliminary findings.

Section 2 discusses the usefulness of non-deterministic computations in functional logic programs and how they are related to our work. Section 3 justifies our overall approach to measuring the efficiency of a computation. Section 4 presents the programming techniques that are the focus of our work. In some cases, these techniques reduce the computing time and/or the memory consumption attributed to non-deterministic computations. Section 5 discusses, both theoretically and experimentally, the effects of our techniques on some examples. Section 6 contains our conclusions.

2 Non-Determinism

Non-determinism is an essential feature of logic programming, perhaps the single most important reason for its acceptance and success. Some early proposals of functional logic programming languages neglected this aspect. Programs in these early languages were modeled by weakly orthogonal rewrite systems. In these languages, the results of non-deterministic computations are obtained by instantiating the arguments of a predicate. A serious drawback of this situation is that a non-deterministic computation cannot be functionally nested in another computation. The lazy evaluation of non-deterministic computations becomes impossible and the efficiency of a program may incur severe losses.

More recently [4,8], non-determinism in functional logic programming has been described using the operations of a non-confluent Term Rewriting System (TRS). These operations are quite expressive, in that they allow a programmer to translate problems into programs with a minimal effort. For example, the following operation computes a non-empty regular expression over an alphabet of symbols. Each non-empty regular expression is obtained by appropriate non-deterministic choices of a computation.

\[
\begin{align*}
\text{regexp} \, X & = X \\
\text{regexp} \, X & = "(" \, \text{ regexp } \, X \, \text{ ++ } \, "\text{)"} \\
\text{regexp} \, X & = \text{ regexp } \, X \, \text{ ++ } \, \text{ regexp } \, X \\
\text{regexp} \, X & = \text{ regexp } \, X \, \text{ ++ } \, "*" \\
\text{regexp} \, X & = \text{ regexp } \, X \, \text{ ++ } \, "[" \, \text{ regexp } \, X \, \text{ ++ } \, "]" \, \text{ regexp } \, X
\end{align*}
\]

The definition of operation \text{regexp} closely resembles the formal definition of regular expression, e.g., as found in [1, p. 94]. This transparency in semantics can be very convenient for the programmer. For example, to recognize whether a string \( s \) denotes a well-formed non-empty regular expression over some alphabet \( a \), it suffices to evaluate \( \text{regexp} \, a = s \).

Non-deterministic operations support a terse programming style, but may impose a stiff penalty on execution performance. In practice, several computations originating from a non-deterministic choice may have to be executed fairly. Therefore, techniques to improve the efficiency of non-deterministic computations, in particular to limit the number of fair independent computations that
originate from a non-deterministic choice, are quite useful. The overall goal of this paper is the study of two techniques for this purpose.

3 Cost Analysis

The most common approach to analyzing the efficiency of a program is measuring its execution time and memory occupation. We measure the execution time of benchmark programs by means of primitives available in our run-time environment. In addition to measuring the amount of memory used during the execution of a program by means of primitives, we compute the amount of memory used by simple benchmarks using a theoretical technique. In this section, we discuss this theoretical approach to memory usage measurement.

Our starting point is the number of applications cost criterion defined in earlier work on partial evaluation [2, Def. 2]. This criterion intends to measure the storage that must be allocated for executing a computation. We adapt the criterion to the behavior of our run-time environment. We also address the problems of non-deterministic steps. We show that non-determinism, which is not considered in the earlier definition, adds an interesting twist to the situation.

The following definitions formalize our adaptation of the cost criterion “number of applications.”

Definition 1 (number of applications). We denote by $\mathcal{A}$ an overloaded function, called the number of applications, as follows:

- If $t$ is a term, $\mathcal{A}(t) = \sum_{p \in \mathcal{P}(t)} (\text{arity}(\text{root}(t_p)) + 1)$, where $\mathcal{P}(u)$ is the set of positions of non-variable symbols of arity greater than zero in any term $u$, root($u$) is the root symbol of any term $u$, and arity($f$) is the arity of any symbol $f$.

- If $R \equiv l \rightarrow r$ is a rewrite rule, we define $\mathcal{A}(R) = \mathcal{A}(r)$.

- If $C \equiv t \rightarrow R_1 \rightarrow R_2 \cdots \rightarrow R_n$ is a computation of a term $t$ to a constructor term $t_n$, we define $\mathcal{A}(C) = \mathcal{A}(t_n) + \sum_{i=1}^{n} \mathcal{A}(R_i)$.

The number of applications of a term $t$ is the total number of occurrences of $n$-ary symbols, with $n > 0$, in $t$, plus their arities. In our run-time environment (and, we believe, in many lazy language implementations) it is appropriate to consider both defined operation and constructor symbols occurring in the term. The number of applications of a computation accounts for the number of applications of each step and the number of applications of the result. In a run-time environment that supports in-place updates, it would not be necessary to account for the number of applications of the result. We use the implementation of narrowing in Prolog described in [6]. We have verified on several simple programs that this implementation allocates memory in accordance to our definition.

Earlier work [2] shows that the number of reduction steps of a computation is weakly correlated to its execution time. Nevertheless, we count the number of steps [2, Def. 1] of a computation, since the computation of all cost criteria in this work is based on steps.

---

1 Without loss of generality, we consider only unconditional rules [5].
Most cost analysis techniques in the literature are proposed for deterministic computations. Non-deterministic computations in functional logic programming are a relatively newer concept, and introduce significant theoretical and practical complications.

To ensure operational completeness, non-deterministic computations must be executed fairly. A consequence of this condition is that when a program outputs a result (derived, for example, by using the first alternative in a non-deterministic computation) the time and space resources consumed from the beginning of the execution to the time of the output may not be a correct indication of the cost of computing that result. The reason is that the measured values may include resources spent to partially compute other results that have not yet been output. The extent of these computations, and consequently a quantification of the resources spent by these computations, are generally difficult to estimate.

A better approach would be to measure the resources needed to compute all the results of a non-deterministic computation, but this is impossible in practice for computations over an infinite search space, such as the computation of the \texttt{regexp} operation presented earlier.

To deal with these difficulties, which to date have no universally accepted solution, we consider only simple examples. In particular, we reason with natural numbers in Peano notation. This decision is quite convenient for explanation purposes. In practice, one technique that we will discuss in the next section may not be well suited for built-in types, such as binary integers.

We informally reason about the number of steps of a computation and the memory occupied to represent terms. In typical implementations of rewrite systems and functional logic programs, terms are represented by dynamic (linked) data structures. In these structures, each occurrence of a symbol of arity \( n \) greater than zero takes \( n + 1 \) units of dynamic (heap) memory. Nullary symbols are allocated in global (static) memory. Variables are local to rules or clauses and are allocated in local (stack) memory. The following picture informally shows the 5 units of memory allocated to represent the term \texttt{positive (add coin 0)}. The symbols in the right column are allocated in global memory. They are not not specifically allocated to represent any term, but are shared by all terms. The number of arguments of an occurrence of a symbol is not a part of the term because in our run-time environment, Prolog symbols are fully applied.

The previous analysis [2] and the adaptation introduced in this section are limited to deterministic computations. The extension to non-deterministic computations would be a non-trivial task. We believe that our less formal discussion is appropriate for our goals and easier to grasp than a more rigorous approach.

To understand why the treatment of non-deterministic computations is more complicated, consider the evaluation of \( t = s \text{ coin} \). This term has two normal
forms, \s 0 and \s (s 0). The root symbol of each normal form can be traced back to the root symbol \t. This shows that fair independent computations may have to duplicate the portion of a term above a reduct with distinct reducts. Hence, even in run-time environments that support in-place updates, the cost of a step may depend on its context. This consideration further supports the appropriateness of including the number of applications of the result of a computation in the number of applications of the computation itself.

4 Programming Techniques

We attempt to improve the efficiency of non-deterministic computations by avoiding the duplication of both reduction steps and term representations that occur within fair independent computations. We use two programming techniques that achieve some improvements in some cases. In other words, our approach is a guideline for the programmer, i.e., a suggestion on how to code certain problems into programs. However, we envision that an optimizing compiler or a similar specialized tool could automatically transform a program in the same way. In fact, several experimental variations of the second technique have been automatically implemented in our current system [7].

Both of our programming techniques are based on the introduction of a new symbol into the signature of the TRS modeling a functional logic program. One technique regards the new symbol as a polymorphic defined operation, the other as an overloaded constructor. The first approach is not new [8]; our contribution in this case is limited to recognition that there are potential benefits of this technique in the context of modern FLP implementation, and the quantification of these benefits. The new symbol that we introduce, is denoted by the infix operator “!”, and read as alternative.

4.1 The Alternative Operator

In the first programming technique, the alternative operation is defined by the rules:

\[
X \parallel Y = X \\
X \parallel Y = Y
\]

An operation with these rules is called \(alt\) and denoted by “/” in the work of González-Moreno et. al. [8,9]. We could regard this symbol as left associative or overload it for arbitrarily long argument lists: in our examples the symbol is always binary, so the difference is irrelevant.

The alternative operation allows us to give a different, though equivalent, definition of the operation bigger presented earlier.

\[
\text{bigger} = s (0 \parallel s 0)
\]

The significant difference is that a common portion of the right-hand sides of the two rewrite rules of the original definition of \text{bigger} has been “factored”. This new definition can be directly coded by the programmer or it could be automatically obtained from the original definition by an optimizing compiler or other specialized tool.
The advantage of this definition of \texttt{bigger} with respect to the original one is that if only the factored portion of the two alternative right-hand sides of the rewrite rules of \texttt{bigger} is needed by a context, no fair independent computations are created by a \textit{needed} strategy \cite{4}. A single deterministic computation suffices in this case. This is exactly what the composition of \texttt{positive} and \texttt{add} requires, as shown by the following derivation:
\[
\text{positive (add bigger 0)} \\
\quad \rightarrow \text{positive (add (s (0 ! s 0)) 0)} \\
\quad \rightarrow \text{positive (s (add (0 ! s 0)) 0)} \\
\quad \rightarrow \text{true}
\]
Two computations have been replaced by a single computation of the same length. In cases where factoring the right-hand sides of two rewrite rules does not eliminate the need of fair independent computations, the run-time cost of the factorization is a single additional rewrite step. For realistic programs, this cost is negligible. Hence, the factorization of right-hand sides is a worthwhile potential improvement. In the best case, it saves computing time and/or storage for representing terms. In the worst case, it costs one extra step and very little additional memory.

4.2 The Alternative Constructor

Our second approach is to consider the \textit{alternative} symbol as a constructor. Since functional logic programs are generally strongly typed, they are modeled by many‐sorted rewrite systems. This condition requires overloading the symbol “!” for each sort in which it is introduced.

The consequences of introducing such an overloaded constructor are interesting. For example, the new definition of \texttt{bigger} is like the previous one
\[
\texttt{bigger = s (0 ! s 0)}
\]
except that the right-hand side is an irreducible (constructor) term. In this example, new constructor terms should be interpreted as non-deterministic choices in sets of terms. The right-hand side of the definition of \texttt{bigger} is interpreted as an element in the set \{\texttt{s 0, s (s 0)}\}. In general, we think that extending builtin types, (such as the integers or booleans) or well-known types (such as the naturals) is inappropriate. Extending a sort with new constructor symbols radically changes the nature of that sort. The interpretation of the new terms of an extended sort may be difficult. Nevertheless, we do it here for its immediateness and to ease the comparison with the examples presented for the first technique.

The introduction of a new constructor symbol makes some formerly well-defined operations incompletely defined. It is relatively easy to correct this problem in the class of the overlapping inductively sequential TRSs \cite{4}. Every operation in this class has a definitional tree \cite{3}. The necessary additional rules may be determined from this tree. For example, consider the operation that halves a natural:
\[
\text{half 0 = 0} \\
\text{half (s 0) = 0}
\]
\[ \text{half } (s \ (s \ X)) = s \ (\text{half } X) \]

If the type natural is extended by an alternative constructor, the following additional rewrite rules complete the definition of \( \text{half} \):

\[ \text{half } (X \ ! \ Y) = (\text{half } X) \ ! \ (\text{half } Y) \]
\[ \text{half } (s \ (X \ ! \ Y)) = (\text{half } (s \ X)) \ ! \ (\text{half } (s \ Y)) \]

In general, a new rewrite rule is needed for each branch of the tree. If \( \pi \) is the pattern of a branch and \( p \) is the inductive position of \( \pi \), then the required rewrite rule is:

\[\pi[X \ ! \ Y]_p \rightarrow \pi[X]_p \ ! \pi[Y]_p\]

The advantages of factoring right-hand sides when the alternative symbol is an operation are preserved by additional rewrite rules of this kind when the alternative symbol is a constructor as well. However, when one of the new rewrite rules is applied, additional storage is required for the representation of terms. Referring to the example under discussion, the representation of \( \text{half } (s \ X) ! \text{half } (s \ Y) \) takes more storage—exactly three units for the top occurrence of the alternative constructor—than the representations of \( \text{half } (s \ X) \) and \( \text{half } (s \ Y) \) combined.

In general, it is not possible to say whether defining the alternative symbol as a constructor will increase or decrease the storage used to represent the terms of a computation. In some case, the alternative symbol allows a more compact representation of some results of a computation. For example, consider the evaluation of:

\[\text{add } (s \ 0) \ \text{coin} \]
\[\rightarrow s \ (\text{add } 0 \ \text{coin}) \]
\[\rightarrow s \ (\text{coin}) \]
\[\rightarrow s \ (0 \ ! \ s \ 0)\]

If the alternative symbol were not a constructor, the last term of the above computation would create two independent computations. To complete these computations both additional steps would be executed and additional storage would be needed for the execution of these steps.

A consequence of defining the alternative symbol as a constructor is that several alternative normal forms are represented by a single term. Therefore, it is likely inappropriate to adopt this programming technique to code problems where only a small fraction of the potentially computed values of a computation are actually needed.

## 5 Examples

In order to reason about the advantages and disadvantages of our techniques, we analyze a few computations using the cost criterion discussed in Section 3. As noted there, the theory that we use has previously been studied only for deterministic computations. In our simple examples, where the computation space is finite, we adapt it to non-deterministic computations as follows.

Consider a complete independent computation for each non-deterministic step, and two independent computations that differ for a non-deterministic replacement. In our implementation [7], some fair independent computations may
share steps and terms. In these cases, our theory would predict that the storage
allocated for all the computations of a term is higher than it is actually is.

We consider the computations of positive (add bigger 0) with and without
using our first technique. In the following tables, each line represents a step
of a computation. We measure both the number of steps and the number of
applications of a computation. The columns of a table respectively show the step
counter, the rewrite rule applied in the step, and the number of applications of
the step. The result does not contribute to the number of applications of the
computation because it is a constant (a millary symbol).

Tables 1–3 show that when bigger is defined by two rewrite rules, the res-
sources spent to compute positive (add bigger 0) are 6 steps and 16 units of
memory. By contrast, our first technique cuts the number of steps in half and
reduces the memory consumption by 25%. These exact savings are also obtained
with our second technique.

<table>
<thead>
<tr>
<th>Step</th>
<th>Rule</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bigger → s 0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>add</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>positive</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1. Computation when bigger non-deterministically rewrites to s 0.
Total resources: steps 3, memory units 7.

<table>
<thead>
<tr>
<th>Step</th>
<th>Rule</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bigger → s (s 0)</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>add</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>positive</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2. Computation when bigger non-deterministically rewrites to s (s 0).
Total resources: steps 3, memory units 9.

<table>
<thead>
<tr>
<th>Step</th>
<th>Rule</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bigger → s (0 ! s 0)</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>add</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>positive</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3. Computation when bigger rewrites to s (0 ! s 0) and "!" is an op-
eration.
Total resources: steps 3, memory units 12.

A similar analysis for the computation of half bigger shows that when
bigger is defined by two rules, the resources spent by all the computations are 5
steps and 12 units of memory. By contrast, using our second technique (i.e., when
bigger rewrites to s (0 ! s 0) and "!" is a constructor) the resources used are 5
steps and 27 units of memory; there is a 108% increase in memory consumption.
The first technique uses 6 steps and 19 units of memory, an increase of 58%.

On these examples, the implementation of [6] allocates memory according to
our theoretical model. However, the above examples are too small and artificial
for understanding the effects of our programming techniques in practice. Also,
our theoretical analysis is difficult to apply to programs that make more than a
few steps. For this reason, we benchmark both the memory consumption and the
execution times of larger programs. Our programming language is Curry [10].
The compiler is PAKCS, which transforms Curry source code into Prolog for
execution.

The first program that we benchmark is an implementation of the game of 24.
Some of us first encountered this problem at a meeting of the Portland Extreme
Programming User Group on June 5, 2001: it is inspired by a commercial game intended to develop mathematical skills in middle school students. The game is played as follows: given four 1-digit positive integers find an arithmetic expression in which each digit occurs exactly once and that evaluates to 24. A number can be divided only by its factors. For example, a solution for the instance \([2, 3, 6, 8]\) is \((2 + 8) \times 3 - 6\). There are 25 other distinct solutions of this instance (including commutatively and associatively equivalent solutions), including \(3 \times (2 + 8) - 6\) and \(6 \times 3 + (8 - 2)\).

The program for this problem, shown in its entirety in the Appendix, proceeds via straightforward generate-and-test. Table 4 shows the CPU time (on a Sun SPARCStation running Solaris) spent for computing all the solutions of a few problems, and the global and local stack allocations for the computation reported by the Curry primitive \texttt{evalSpace}. The first group of data is for a version of the program that does not use our techniques. The second group is for a program that uses our first technique, i.e., the alternative symbol is a defined operation. The second technique is not appropriate for this problem. The runtime measures are nearly identical over several executions. The memory measures are constant for every execution.

The data shows that our technique consumes slightly more memory, but speeds the execution of the program by 44\% on average. The speedups for various problems range from 27\% to 58\%. (The speedup achieved by our technique is computed by \(t_1/t_2\)/\(t_1\) where \(t_1\) and \(t_2\) are the averages of the execution times of the programs not using and using the technique respectively. This speedup indicates the percentage of execution time saved by the technique.)

### Table 4. Runtime (msec.) and memory usage (bytes) for "24" instances.

<table>
<thead>
<tr>
<th>problem</th>
<th>\text{Runtime}</th>
<th>\text{G. stack}</th>
<th>\text{L. stack}</th>
<th>\text{Runtime}</th>
<th>\text{G. stack}</th>
<th>\text{L. stack}</th>
<th>\text{Speedup}</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2, 3, 6, 8]</td>
<td>66</td>
<td>2596</td>
<td>932</td>
<td>48</td>
<td>2800</td>
<td>1100</td>
<td>27%</td>
</tr>
<tr>
<td>[2, 3, 4, 9]</td>
<td>94</td>
<td>2632</td>
<td>860</td>
<td>52</td>
<td>2860</td>
<td>972</td>
<td>45%</td>
</tr>
<tr>
<td>[3, 4, 5, 8]</td>
<td>65</td>
<td>2176</td>
<td>868</td>
<td>27</td>
<td>2680</td>
<td>1036</td>
<td>58%</td>
</tr>
<tr>
<td>[1, 2, 6, 8]</td>
<td>64</td>
<td>2812</td>
<td>868</td>
<td>36</td>
<td>2816</td>
<td>980</td>
<td>44%</td>
</tr>
<tr>
<td>[4, 6, 8, 9]</td>
<td>38</td>
<td>2416</td>
<td>832</td>
<td>19</td>
<td>2620</td>
<td>1000</td>
<td>50%</td>
</tr>
<tr>
<td>\text{Average}</td>
<td>65</td>
<td>2586</td>
<td>872</td>
<td>36</td>
<td>2750</td>
<td>1017</td>
<td>44%</td>
</tr>
</tbody>
</table>

Our second example is a parser that takes a string representing a parenthesized arithmetic expression and returns a parse tree of the input. Our implementation is simplified to the extreme and serves only as a proof of concept. The abstract syntax generated by the parser is defined by the type:

\[
\text{data \ AST = Num \ String \ | \ Bin \ Char \ AST \ AST}
\]

For example, on input "1+(2-3)" the parser generates

\[
\text{Bin'}+\text{'} (\text{Num'}\text{'}1') (\text{Bin'}-\text{'} (\text{Num'}\text{'}2')(\text{Num'}\text{'}3')).
\]
Replacing the argument of *Num* with an integer and the *Bin Char* combination with a token would be more appropriate, but it would add to the program details that are irrelevant to our analysis. The language recognized by the parser is generated by the following grammar:

\[
expression ::= \text{term \ '++' \ expression} \\
\ | \text{term \ '-' \ expression} \\
\ | \text{term} \\
\text{term} ::= \text{'} \ expression \text{'} \\
\ | \text{digits}
\]

Sequences of *digits* are recognized by a scanner.

The parser is implemented using two defined operations: *expression* and *term*. The type of both operations is \([\text{Char}] \rightarrow [\text{Char}] \rightarrow \text{AST}\). For all strings *s* and *r*, \(expression \ s \ r\) evaluates to *a* if and only if there exists a string *u* such that \(s = u \ r\) and *a* is the parse tree of *u*. Operation *term* is analogous. For example, \(term \ "1*(2-3)" \ "+(2-3)"\) evaluates to *Num*"1". To parse an entire string, operation *expression* is initially called with its second argument equal to the empty string. In recursive calls, the second argument is a free variable.

Table 5 shows execution time and memory usage on a 233MHz Pentium PC running Linux. In this program, too, the data show that our first technique consumes more memory, but substantially cuts the execution time to parse certain strings. The speedup is highly dependent on the structure of the input string.

<table>
<thead>
<tr>
<th>input</th>
<th>regular program</th>
<th>first technique</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Runtm</td>
<td>G. stick</td>
<td>L. stick</td>
</tr>
<tr>
<td>&quot;1+1+1+1+1+1+1+1+1+1+1&quot;</td>
<td>10</td>
<td>12528</td>
<td>736</td>
</tr>
<tr>
<td>&quot;((((()))))&quot;</td>
<td>1440</td>
<td>2676</td>
<td>8</td>
</tr>
<tr>
<td>&quot;5-(2+1)+3+(5-4)&quot;</td>
<td>60</td>
<td>10468</td>
<td>528</td>
</tr>
<tr>
<td>Average</td>
<td>500</td>
<td>8557</td>
<td>424</td>
</tr>
</tbody>
</table>

6 Conclusions

Non-deterministic computations are an essential feature of functional logic programming languages. Often, a non-deterministic computation is implemented as a set of fair independent computations whose results are used, and possibly discarded, by a context. A non-deterministic computation can be costly to execute: any reasonable attempt to improve its efficiency is worthwhile.

In this paper, we have proposed two simple programming techniques intended to improve the efficiency of certain non-deterministic computations. Both techniques are based on the introduction of a new symbol, called *alternative*, into the signature of a program. In one technique, the *alternative* symbol is a polymorphic defined operation. In the other technique, the *alternative* symbol is
an overloaded constructor. This symbol allows a program to factor a common
portion of the non-deterministic replacements of a redex.

Either technique may improve the efficiency of a computation by reducing
the number of computation steps or the memory used in representing terms. These
savings are obtained in two situations. For both techniques, savings are obtained
when fair independent computations are avoided because only the factored portion
of non-deterministic replacements is needed. For the second technique, savings are
obtained when distinct non-deterministic results are more compactly
represented by sharing a common factor. In some cases, the improvements offered
by these techniques are substantial. In all cases, the cost of applying the
first technique is small. There are cases in which the application of the second
technique may actually result in computations that consume more memory.

We have discussed how to apply our techniques, and we have quantified the
effects of the application of these techniques in simple examples. Our techniques
are applicable to programs coded in many existing or proposed functional logic
programming languages. Our techniques can be directly adopted by programmers or can be introduced into a program automatically at compile time.

References

Addison-Wesley, 1985.

2. E. Albert, S. Antoy, and G. Vidal. Measuring the effectiveness of partial eval-
uation in functional logic languages. In *Proc. of 10th Intl Workshop on Logic-based

Algebraic and Logic Programming (ALP’92)*, pages 143–157. Springer LNCS 632,


5. S. Antoy. Constructor-based conditional narrowing. In *Proc. of 3rd Intl Conf. on
Principles and Practice of Declarative Programming (PPDP’01)*. Springer LNCS,

6. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into pro-
log. In *Proc. of the 3rd International Workshop on Frontiers of Combining Systems

7. S. Antoy, M. Hanus, B. Massey, and F. Steimer. An implementation of narrowing
strategies. In *Proc. of 3rd Intl Conf. on Principles and Practice of Declarative

8. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and
M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc.

9. J.C. González-Moreno, F.J. López-Fraguas, M.T. Hortalá-González, and
M. Rodríguez-Artalejo. An approach to declarative programming based on a re-

0.71).* Web document http://www.informatik.uni-kiel.de/~mh/curry/report.
Appendix

The Game of 24

This program solves the game of 24.

```haskell
infixr 5 +
(+++) eval flex
[] ++ x = x
(x:xs) ++ y = x:xs ++ y

permute [] = []
permute (x:xs) | u+++v =: permute xs = u++[x]+v where u,v free

data exp = num Int
  | add exp exp
  | mul exp exp
  | sub exp exp
  | dvr exp exp

generate [y] = num y
generate (y:y1:ys)
  | (y:y1:ys) := u:us+++v:vs
  = add (generate (u:us)) (generate (v:vs)) where u,us,v,vs free
generate (y:y1:ys)
  | (y:y1:ys) := u:us+++v:vs
  = mul (generate (u:us)) (generate (v:vs)) where u,us,v,vs free
generate (y:y1:ys)
  | (y:y1:ys) := u:us+++v:vs
  = sub (generate (u:us)) (generate (v:vs)) where u,us,v,vs free
generate (y:y1:ys)
  | (y:y1:ys) := u:us+++v:vs
  = dvr (generate (u:us)) (generate (v:vs)) where u,us,v,vs free

test (num y) = y
test (add x y) = test x + test y
test (mul x y) = test x * test y
test (sub x y) = test x - test y
test (dvr x y) = opdvr (test x) (test y)
  where opdvr x y = if y == 0 || not (x `mod` y == 0)
    then failed else x `dvr` y

solve p | test x == 24 = x where x = generate (permute p)

-- example: solve [2,3,6,8]

The application of our technique calls for the definition of the alternative function and the replacement of operation generate.

infixl 0 !
x ! - = x
- ! y = y
```
generate \([y] = \text{num } y\)
generate \((y:y_1:ys)\)

\[
(y:y_1:ys) ::= \text{u:us++v:vs} = \text{add }! \text{mul }! \text{sub }! \text{div} \times \text{generate } \text{u:us}) \times \text{generate } \text{v:vs})
\]
where \(\text{u,us,v,vs free}\)

The Parser

This is a parser for parenthesized arithmetic expressions.

```haskell
--import Char
data AST = Num String | Bin Char AST AST

expression X0 X3
  | A1 ::= term X0 X1 &>
  | '(':X2 ::= X1 &>
  | A2 ::= expression X2 X3
  = Bin 'a' A1 A2 where X1, X2, A1, A2 free

expression X0 X3
  | A1 ::= term X0 X1 &>
  | '-':X2 ::= X1 &>
  | A2 ::= expression X2 X3
  = Bin '-' A1 A2 where X1, X2, A1, A2 free
term X0 X2
  | X:X1 ::= takeWhile isDigit X0 &> X0 ::= X:X1 ++ X2
  = Num (X:X1)
  | X0 ::= '(' Y0
  = expression Y0 (')':X2)
  where Y0, X, X1 free

-- example expression "1+(2-3)"

The application of our technique calls for the definition of the alternative function, as in the previous program, and the replacement of operation expression with the following code.

expression X0 X3
  | A1 ::= term X0 X1 &>
    ( 0P:X2 ::= X1 &>
      0P ::= ('+'|'-') &>
      A2 ::= expression X2 X3 &>
      TREE ::= Bin 0P A1 A2 )
  ! (X3 ::= X1 &> TREE ::= A1)

  = TREE where 0P, X1, X2, A1, A2, TREE free
Abstract. The last years have seen a renewal of interest in applying dynamic programming to natural language processing. The main advantage is the compactness of the representations, which is turning this paradigm into a common way of dealing with highly redundant computations related to phenomena such as non-determinism.

Natural language parsing adds another challenge, since grammatical information is often insufficient. We describe an extension of parsing techniques for partial parsing in dynamic programming. Our aim is to obtain as much information as possible, that is incomplete parses, while preserving compactness of the representations.

Keywords: Partial parsing, dynamic programming, deductive parsing scheme.

1 Introduction

Highly redundant computations are usual when we deal with complex grammar formalisms. This claim has been used to motivate parsing techniques that encode trees and computations in some kind of shared structure. A major area of application is natural language processing (NLP), where dynamic programming has been known for a long time [3]. In particular, natural language parsing comes across the problem of partial information. This lack of information is due to the errors in former stages of analysis and the fact that practical grammars and lexicons are incomplete and even incorrect.

We refer to standard parsing as complete parsing, reserving the term partial parsing for all the possible subcomputations of a complete parsing. Our aim is to obtain every correct partial parse even when there is no complete parse.

Previous studies have illustrated the practical suitability of dynamic programming for dealing with context-free grammars (CFGs) [11], middle-sensitive grammars [1] and definite clause grammars [10]. Our goal is to show the validity of these results for partial parsing. We approach the problem by extending models from complete to partial parsing while preserving the related benefits of dynamic programming.

Often, these techniques improve performance by pruning the search space by means of the inclusion of static control. Unfortunately, while dealing with partial parses, the
static control will prune some analysis branches that are useless in a complete parse, but which are necessary in some partial parses. To overcome this problem we deal with the starting and finishing points of a partial parse like any other source of non-determinism. The static control is modified to take them into account.

In contrast to previous works in the domain of partial parsing, our proposal introduces a parsing framework based on the notion of deduction scheme [7]. This clearly differentiates our work from approaches oriented to particular parsing architectures [5, 9, 6], and provides a uniform description and operational formalism to validate the performance in each case.

In Section 2 of this paper we introduce a uniform parsing framework, describing different schema for both complete and partial parsing. These schema include classic top-down and bottom-up approaches, but also mixed strategies with dynamic and static control. Section 3 gives a survey of the dynamic interpretation process. In Section 4 we compare in practice the schema introduced, with preliminary experimental results. Finally, Section 5 is a conclusion about the work presented.

\begin{figure}[h]
\centering
\includegraphics[width=0.2\textwidth]{aababab.png}
\caption{Partial parses of aababab}
\end{figure}

2 On partial parsing

In order to obtain a practical definition of partial parsing, we must relax the conditions we apply to the standard concept. The notion of grammar includes an initial symbol or axiom, \( S \), leading to the parse of all sentences of the language generated by the grammar. Instead of this, we use a set of initial symbols, \( S \), following the entry point concept [4], a classic auxiliary structure in the abstract syntax that enables the parsing of program fragments. We introduce a CFG as a 4-tuple \( G = (N, \Sigma, P, S) \), where \( N \) is a finite set of non-terminal categories, \( \Sigma \) is a finite alphabet and \( P \) is a finite set of context-free rules. As mentioned, \( S \) is the set of initial symbols leading to complete or partial parses. We assume \( \mathcal{L}(G) \) to be the language generated by \( G \), and we try to determine the partial parses of an input string \( w_{1..n} \), of length \( n \).

In particular, we discuss the extension to the partial case of classical context-free parsing methods including pure top-down and pure bottom-up architectures, Earley’s algorithm [3] as representative of a mixed-strategy with dynamic prediction and a LR(1) proposal as representative of a mixed-strategy with static prediction.

For the sake of a better exposition we have chosen a common descriptive framework, the deductive parsing scheme [7], close to the parsing schemata proposal [8].
The deduction system consists of a set of items representing parsing states and a set of deduction steps performing over those items.

As our running example we shall consider the language, \( P \), of palindromes over the alphabet \( \Sigma = \{a, b\} \), generated by the grammar that follows:

\[
\begin{align*}
\text{Palin} & \rightarrow a \\
\text{Palin} & \rightarrow b \\
\text{Palin} & \rightarrow a \text{Palin} a \\
\text{Palin} & \rightarrow b \text{Palin} b
\end{align*}
\]

Observe, for example, that although the input string is \( aababab \not\in P \), it contains sub-strings that do belong to the language, as is the case of the trees in Fig. 1.

### 2.1 A top-down scheme

The scheme for top-down complete parsing is shown in Fig. 2. We have a single axiom that predicts the analysis of the initial symbol, and a single goal that represents the parse of the complete input sentence. Now we will make a short description of the scheme. Each item has the form: \([\bullet \beta, j, (T)]\), stating that we have constructed the derivation \( S \Rightarrow w_1 \cdots w_j \beta \), and \( T \) is the parse tree. The dot is a reference to position \( j \) in the input string. The parse has reached this position and has to continue from there. Soundness and correctness are proved in [7].

<table>
<thead>
<tr>
<th>Item form</th>
<th>([\bullet \beta, j, T])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invariant</td>
<td>( S \Rightarrow w_1 \cdots w_j \beta )</td>
</tr>
<tr>
<td>Axioms</td>
<td>([\bullet S, 0, ()])</td>
</tr>
<tr>
<td>Goals</td>
<td>([\bullet, n, T])</td>
</tr>
<tr>
<td>Scanning</td>
<td>([\bullet w_{j+1} \beta, j, T_1 \text{tree}(B, \alpha \bullet w_{j+1} \beta)T_2])</td>
</tr>
<tr>
<td></td>
<td>([\bullet, j + 1, T_1 \text{tree}(B, \alpha w_{j+1} \bullet \beta)T_2])</td>
</tr>
<tr>
<td>Prediction</td>
<td>([\bullet B \beta, j, T_1 \text{tree}(A, \alpha \bullet B \beta T_2)])</td>
</tr>
<tr>
<td></td>
<td>([\bullet \gamma \beta, j, T_1 \text{tree}(A, \alpha \text{tree}(B, \bullet \gamma) \bullet \beta)T_2]) ( (r_k : B \rightarrow \gamma \in R) )</td>
</tr>
</tbody>
</table>

Fig. 2. Top-down schema

The parse starts at position 0 and symbol \( S \), yielding axiom item \([\bullet S, 0, ()]\). We then apply the following deduction steps:

**Scanning:** It moves the point one position forward. This rule is obtained after observing that items \([\bullet w_{j+1} \beta, j, (T)]\) and \([\bullet \beta, j + 1, (Tw_{j+1})]\) represent the same state in the derivation \( S \Rightarrow w_1 \cdots w_{j+1} \beta \).

**Prediction:** It takes the next non-terminal symbol to parse \( (B) \) and rewrites it as the right hand side of a matching rule \( (r_k : B \rightarrow \gamma) \). It predicts the use of the rule \( r_k \).
Finally, \( w \in \mathcal{L}(G) \) iff the goal item \([\bullet, n, (T)]\) is generated. This means that \( S \Rightarrow w_1 \cdots w_n \), and \( T \) is the parse tree.

In order to adapt this scheme for partial parsing, we consider a modified item form, adding a reference to the starting position of the potentially partial parse. A partial parse covers any piece of the input string. So, instead of an axiom item starting at position 0, we have now the following set of axioms:

\[
\text{Axioms} = \{ [\bullet, A, i, i, ()], \ A \in S, \ 0 \leq i < n \}
\]

The deductive steps remain as before, but keeping the starting point. As a partial parse may cover any piece of the input string, it may start at any position, and it may finish at any position after the starting point. As consequence, we have that:

\[
\text{Goals} = \{ [\bullet, i, j, T], \ A \in S, \ 0 \leq i \leq j \leq n \}
\]

We shall generate \( m \times n \) axioms, where \( m = ||S|| \) and \( n \) is the input length. For each of those axioms we will generate a new analysis branch, and, consequently, new items.

In our running grammar, for an input string \( aaba \) and a complete top-down parse, we need to create 30 items, instead of 82 in the partial case.

### 2.2 A bottom-up scheme

We include in Fig. 3 the scheme for bottom-up complete parsing. Items are now of the form \([\alpha \bullet, j, (T)]\), stating that \( \alpha w_{j+1} \cdots w_n \Rightarrow w_1 \cdots w_n \), \( T \) being the parse tree. The dot indicates that \( \alpha \) reduces the input substring till position \( j \).

Bottom-up parsing starts at position 0 before reducing any piece of the input string. Therefore the axiom is \([\bullet, 0, ()]\). The deduction steps are:

**Scanning:** It shifts the next terminal and moves the dot one position forward.

**Completion:** It reduces the \( k_n \) symbols immediately after the dot. Those symbols must match the right hand side of rule \( r_k \).
As usual, $w \in \mathcal{L}(G)$ iff the goal item $[S \bullet n, \{T\}]$ is generated. This means that $S \Rightarrow w_1 \cdots w_n$, where $T$ is the parse tree.

In order to deal with partial parsing, items are extended with a reference to the starting position. Regarding axioms, a partial parse may start at any position in the input string. So, we have that:

$$\text{Axioms} = \{[\bullet, i, i], 0 \leq i < n\}$$

In deduction steps we keep the starting position. Again, the goal item is replaced by a set of items. To construct this set we must have taken into account that a partial parse may finish at any symbol of $S$, at any input position. So, we have that:

$$\text{Goals} = \{[A \bullet, i, j, T], A \in S, 0 \leq i \leq j \leq n\}$$

As far as efficiency issues are concerned, the main difference with complete parsing is the set of axioms. The size of this set is $n$, $n$ being the length of the input. Retaking the former example, this yields 29 items for complete parsing and 54 for partial parsing.

<table>
<thead>
<tr>
<th>Item form</th>
<th>$[A \rightarrow \alpha \bullet \beta, i, j, T], A \rightarrow \alpha \beta \in R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invariant</td>
<td>$S \Rightarrow w_1 \cdots w_j\beta$</td>
</tr>
<tr>
<td></td>
<td>$\alpha w_{j+1} \cdots w_n \Rightarrow w_1 \cdots w_n$.</td>
</tr>
<tr>
<td>Axioms</td>
<td>$[S' \rightarrow \bullet S, 0, 0, ()]$</td>
</tr>
<tr>
<td>Goals</td>
<td>$[S' \rightarrow S \bullet, 0, n, T]$</td>
</tr>
<tr>
<td>Scanning</td>
<td>$[A \rightarrow \alpha \bullet w_{j+1}\beta, i, j, T]$</td>
</tr>
<tr>
<td></td>
<td>$[A \rightarrow \alpha w_{j+1} \bullet \beta, i, j + 1, T w_{j+1}]$</td>
</tr>
<tr>
<td>Prediction</td>
<td>$[A \rightarrow \alpha \bullet B\beta, i, j, T]$</td>
</tr>
<tr>
<td></td>
<td>$[B \rightarrow \bullet \gamma, j, j, ()]$</td>
</tr>
<tr>
<td></td>
<td>$\langle r_k : B \rightarrow \gamma \in R \rangle$</td>
</tr>
<tr>
<td>Completion</td>
<td>$[A \rightarrow \alpha \bullet B\beta, i, k, T_1][B \rightarrow \gamma \bullet, k, j, T_2]$</td>
</tr>
<tr>
<td></td>
<td>$[A \rightarrow \alpha B \bullet \beta, i, j, T, \text{tree}(B, T_2)]$</td>
</tr>
</tbody>
</table>

**Fig. 4. Earley’s scheme**

### 2.3 Earley’s scheme

Earley’s algorithm [3] will illustrate the extension from complete to partial parsing using a mixed-strategy with dynamic prediction. The complete parsing scheme is shown in Fig. 4.

We have now items of the form $[A \rightarrow \alpha \bullet \beta, i, j, T]$. The dot is still a reference to position $j$, but now the item represents the state in recognizing the rule $A \rightarrow \alpha \beta$, where $\alpha$ reduces some of the substring just before the dot and $\beta$ remains to be parsed. In relation to $i$, it points to the beginning of the substring parsed by $\alpha$. Consequently
items represent a local state of parsing process instead of a global one. Having local information enclosed makes it easier to share computations among items.

The grammar is augmented with an artificial rule \( S' \to S \), where \( S' \) is a distinct symbol. This facilitates the definition for axioms and goals. The parse starts with the axiom \([S' \to \bullet S, 0, 0, ()]\), at position 0 we want to reduce the initial symbol S. The deduction steps are as follows:

**Scanning:** After recognition of terminal \( w_{j+1} \), it moves the pointer from position \( j \) to \( j + 1 \).

**Prediction:** It predicts all the rules \( B \to \gamma \), because they may reduce \( B \) from position \( j \).

**Completion:** After we finish the parse of a rule \( B \to \gamma \), from position \( k \) to \( j \), it searches for items whose next symbol to analyze is \( B \) at position \( k \). For those items it generates a new one by moving the dot to just after \( B \), position \( j \).

Once we generate the goal item \([S' \to A \bullet, i, j, T]\), we know \( w_0...n \) reduces to the initial symbol \( S \), and \( w \in \mathcal{L}(G) \).

We need to modify axioms and goal items in order to deal with partial parses. Axioms start with any initial symbol in \( S \) at any position:

\[
\text{Axioms} = \{[S' \to \bullet A, i, i, ()], A \in S, 0 \leq i < n\}
\]

and goals finish at any position after the starting point:

\[
\text{Goals} = \{[S' \to A \bullet, i, j, T], A \in S, 0 \leq i \leq j \leq n\}
\]

The comparison between complete and partial parsing is similar to that we made for the top-down scheme, the number of axioms growing from one to \( n \times m \), where \( n \) is the input string length and \( m = ||S|| \). So, returning to our running grammar, this increment means that we need 35 items for a complete parse and 52 for a partial one.

### 2.4 A mixed-strategy with static control

We introduce now the complete parsing scheme for an LR(0) based parse. A preliminary description of the deductive interpretation, where we have omitted the finite state control in order to make the differences with Earley’s scheme clear, is shown in Fig. 5. This difference is rooted in the meaning of the items. In Earley’s case, the sequence of symbols \( \alpha \), immediately to the left of the dot, reduce the substring \( w_i...j \). In LR(0), only the symbol \( X (\alpha = \alpha'X) \) immediately on the left of the dot reduces the substring. As a consequence, we need different deduction steps to manage different items:

**Shift:** It is similar to scanning in Earley’s case. The starting position reflects the last symbol analyzed, \( w_j \).

**Reduce:** It replaces the completion step of Earley. In Earley’s scheme, we need an item \([B \to X_1 \cdots X_m \bullet, j_0, j_m, T_m]\) that reflects the recognition of \( B \to X_1 \cdots X_m \) between positions \( j_0 \) and \( j_m \). In LR(0) we need \( m \) items of the form \([B \to X_1 \cdots X_i \cdots X_m, j_{i-1}, j, T_i]\). Each item reflects the recognition of the \( m \) symbols of the right hand side of the rule at adjacent positions between \( j_0 \) and \( j_m \).
<table>
<thead>
<tr>
<th>Item form</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[A \to \alpha \beta, i, j, T]$</td>
<td>$A \to \alpha \beta \in R, 0 \leq i \leq j \leq n$</td>
</tr>
<tr>
<td>$S \Rightarrow w_1 \cdots w_i X\beta$</td>
<td>$X \Rightarrow w_{i+1} \cdots w_j$</td>
</tr>
<tr>
<td>$\alpha = \alpha', \exists k, k \leq i, \alpha' \Rightarrow w_{k+1} \cdots w_j$</td>
<td></td>
</tr>
</tbody>
</table>

| Axiom | $[S \to \bullet \alpha, 0, 0, ()]$ |
| Goal | $[S \to \alpha, 0, n, T]$ |

| Shift | $[A \to \alpha \beta, i, j, T]$ |
| Prediction | $[A \to \alpha \beta, i, j, T]$ |

| Reduce | $[B \to \bullet \gamma, j, j, ()]$ |

$[B \to X_1 X_2 \cdots X_m, j_m-1, j_m, T_m]$, ..., $[B \to \bullet X_1 X_2 \cdots X_m, j_0, j_1, ()]$, $[A \to \alpha \beta, j_0, j_1, T_0]$, $[A \to \alpha \beta, j_m, \text{tree}(B, T_1, \ldots, T_m)]$

**Fig. 5.** LR(0) scheme, omitting finite control

**Prediction:** The same as Earley’s scheme.

Next, we will get a more efficient scheme adding a finite state control. This implies replacing dotted rules $A \rightarrow \alpha \bullet \beta$ by an state, $st$, representative of its equivalence class. In order to build the finite state control, we init $st_0 = [S \rightarrow \bullet \alpha]$. Then we build the other states with the closure of their items $[A \rightarrow \alpha \bullet B\beta]$. More precisely, the closure operation adds items $[B \rightarrow \bullet \gamma]$ for each rule $B \rightarrow \gamma$. The closure is in fact equivalent to the prediction step. So, it will be removed from the LR parsing scheme with finite state control, as shown in Fig. 6. Here, $action(state, token)$ denotes a shift or reduce action in the automaton for a given state and token. In the same way, $goto(state, variable)$ looks for a goto action. Finally, $reveals(state)$ refers to all those states with a shift or goto action over state.

To extend the scheme to partial parsing, it must be allowed to start the parsing at any point of the input string, with any symbol of $S$:

$$Axioms = \{[st_0, i, i, ()], 0 \leq i \leq n\}$$

and, to finish at any position after the starting point:

$$Goals = \{[st_f, i, j, T], 0 \leq i \leq j \leq n\}$$

We need to change the initialization step when building the finite state control. So, instead of $st_0 = [S \rightarrow \bullet \alpha]$, the first state must be $st_0 = \{[A \rightarrow \bullet \alpha] \mid A \in S\}$.

We can improve the LR(0) parsers with a better finite state control. Now, when building the states, we must add information about which lookahead symbols are compatible with their actions. This control could result in an LR(1) or LALR(1) parsing.
algorithm, depending on the computation of the lookahead symbols. Next, in order to adapt the LR(0) scheme to use the LALR(1) control, we will add preconditions to deductive steps. The preconditions will check that the lookahead is compatible with the operation. The resulting scheme is shown in Fig. 7.

Actually action and goto are the core the table of the LALR(1) automaton. This table changes for an LR(1) automaton, but its interpretation remains the same. As a consequence, we can use the same parsing schema for LALR(1) and LR(1), provinding we change the finite state control.

Once again, the extension to partial parsing implies adding axioms and goals, but with one new consideration. Because a partial parse may finish at any input position, the finishing operation is compatible with any terminal, and not only with the end of the input terminal. So, we have that the set of axioms is

\[ \text{Axioms} = \{ [st_0, \$, i, ()), 0 \leq i \leq n, A \in S \} \]
and the set of goals is

\[
\text{Goals} = \{[s_t, i, j, T], 0 \leq i \leq j \leq n, A \in S\}
\]

To build now the finite state control, we need the concept of a \textit{variable} terminal that matches any terminal of the grammar. This may produce an exponential growth of the number of states. To mimic the intended behavior without modifying the set of states we make the input string ambiguous. At each position, we have both the original terminal, \(w_i\), and the end of the input. The former is compatible with any operation that follows the parsing process and the latter is compatible with finishing a parse which is probably partial.

In our running example, the complete LR(0) scheme needs 35 items, while the partial one needs 52. The LALR(1) schema need, respectively, 32 and 42 items.

<table>
<thead>
<tr>
<th>Item form</th>
<th>( [A, st, i, j, T] \cup [\nabla_{r,s}, st, i, j, T] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axiom</td>
<td>(-, st_0, 0, 0, ())</td>
</tr>
<tr>
<td>Goal</td>
<td>([S', st_f, 0, n, T])</td>
</tr>
<tr>
<td>InitShift</td>
<td>( [A, st, i, j, T] )</td>
</tr>
<tr>
<td></td>
<td>( A_{r,1} = w_j \land )</td>
</tr>
<tr>
<td></td>
<td>shift_{st'} \in \text{action}(st, w_j) )</td>
</tr>
<tr>
<td>Shift</td>
<td>( [A_{r,s}, st, i, j, T] )</td>
</tr>
<tr>
<td></td>
<td>( A_{r,s+1} = w_j \land )</td>
</tr>
<tr>
<td></td>
<td>shift_{st'} \in \text{action}(st, w_j) )</td>
</tr>
<tr>
<td>Sel</td>
<td>( [\nabla_{r,n}, st, i, j, T] )</td>
</tr>
<tr>
<td></td>
<td>(\text{reduce}, \in \text{action}(st, w_j))</td>
</tr>
<tr>
<td>Red</td>
<td>( [\nabla_{r,s}, st, k, j, T_1] )</td>
</tr>
<tr>
<td></td>
<td>( A_{r,s}, st, i, k, T_2 )</td>
</tr>
<tr>
<td></td>
<td>( \nabla_{r,s-1}, st', i, j, T_1 T_2 )</td>
</tr>
<tr>
<td></td>
<td>(\text{st'} \in \text{reveals(st)})</td>
</tr>
<tr>
<td>Head</td>
<td>( [\nabla_{r,0}, st, i, j, T] )</td>
</tr>
<tr>
<td></td>
<td>( A_{r,0}, st', i, j, \text{tree}(A_{r,0}, T) )</td>
</tr>
<tr>
<td></td>
<td>(\text{st'} \in \text{goto}(st, A_{r,0}))</td>
</tr>
</tbody>
</table>

Fig. 8. LALR(1) scheme with implicit binarization

### 3 The dynamic interpretation

Given that actions on the automaton depend on the first, and possibly the second, elements in the stack, we implicitly consider a grammar which is a 2-form one. As a consequence, we obtain two interesting features that are not usual in other context-free parsing algorithms:

- Time complexity for the parser is \(O(n^3)\), where \(n\) is the length of the sentence. This result is achieved without the need for the language grammar to be in Chomsky Normal Form.
Sharing of computations on the parsing of a tail of sons in a node is possible. More exactly, bottom-up parsing may share only the rightmost constituents, while top-down parsing may only share the leftmost ones. The reason is simple and relies to the type of search used to build the forest. Breadth-first search results in bottom-up constructions and depth-first search results in top-down ones, as is shown in figure 9.

Fig. 9. Sharing of a tail of sons in a node

In order to obtain $O(n^3)$ complexity in the general case, we can use an implicit binarization of rules. We do this by splitting each reduction involving $m$ elements in the reduction of $m+1$ rules with at most 2 elements on their right-hand side. Thus, the reduction of a rule $A_{r,0} \rightarrow A_{r,1} \cdots A_{r,n_r}$ is equivalently performed as the reduction of the following $n_r + 1$ rules:

$$
\begin{align*}
A_{r,0} & \rightarrow \nabla_{r,0} \\
\vdots & \\
\nabla_{r,n_r-1} & \rightarrow A_{r,n_r} \nabla_{r,n_r} \\
\nabla_{r,n_r} & \rightarrow \epsilon
\end{align*}
$$

This treatment of reductions involves a change in the form of the items. We add a new element, representing a symbol in a rule or a $\nabla_{r,j}$ meaning that elements $A_{r,j+1} \cdots A_{r,n_r}$ have been reduced$^1$.

With respect to deduction steps, we must now differentiate between whether we make the shift of the first symbol in the right hand side of a rule (InitShift) or the shift of other symbols (Shift).

The Reduce step has also been refined into three steps. The selection of the rule to be reduced (Sel), the reduction of the implicit binary rules (Red), and the recognizing of the left-hand symbol of the rule to be reduced (Head). The resulting scheme is shown in Fig. 8. This scheme corresponds to a dynamic interpretation of LALR(1) parsing algorithms using an inference system based on $S^1$ items [2].

$^1$ $\nabla_{r,i}$ is equivalent to the dotted rule $A_{r,0} \rightarrow \alpha \beta$ where $\alpha = A_{r,1} \cdots A_{r,i}$ and $\beta = A_{r,i+1} \cdots A_{r,n_r}$.
The extension for partial parsing is analogous to previous schema, following an identical approach for table construction, and adding new axioms and goals. So, the set of axioms is given by:

$$\text{Axioms} = \{[st_0, \omega i, i, ()], 0 \leq i \leq n, A \in \mathcal{S}\}$$

and the set of goals by:

$$\text{Goals} = \{[st_j, \omega i, j, T], 0 \leq i \leq j \leq n, A \in \mathcal{S}\}$$

Fig. 10. Num. of items in complete parsing

4 Experimental results

To illustrate the practical aspects of our proposal, we provide now some preliminary experimental results. We have parsed several input strings with lengths varying from 1 to 20, considering our running example, the language $P$. The number of items is different even for input strings of the same length. So, for each length we have parsed several input strings, computing the average number of items generated, both for complete and partial parsing. The grammar we are using seems to be well-suited for top-down parsing, as shown in Fig. 10. Bottom-up parsing only performs well for short length inputs. On the other hand, dynamic programming approaches, both Earley parsing and dynamic interpretation of LALR(1), perform as well as top-down parsing.
Regarding partial parsing, Fig. 11, top-down parsing suffers a drop in performance, while bottom-up still performs well for short length inputs. Dynamic programming approaches continue to show a good behavior.

In Fig. 12 we illustrate the relation between complete and partial parsing, synthesizing the last two figures. We have replaced the number of generated items by the increment from the number of items in complete parsing to the number of items in partial parsing. As is shown, top-down partial parsing is not as advantageous as it was for complete parsing. It suffers from exponential growth. On the other hand, the parsing schema with some kind of bottom-up strategy scale well.

5 Conclusions

We have described a practical approach to partial parsing in the domain of CFGs. In comparison with previous works, our proposal is based on a deductive parsing scheme, which provides a uniform framework to compare performances between different parsing strategies for both complete and partial cases.

From a theoretical point of view, we have graduated the introduction of each parsing scheme in order to make clear the existing relationships with previous strategies. This leads to a better understanding of the mechanisms regulating the definition of the deduction rules and even of the structures manipulated by these. The evolution from complete parses to partial ones is also justified in each case.
Fig. 12. Ratio partial/complete parsing

6 Acknowledgments

This work has been partially supported by the European Union, Government of Spain and Autonomous Government of Galicia under projects 1FD97-0047-C04-02, TIC2000-0370-C02-01 and PGIDT99X110502B, respectively.

The source code of the deduction scheme interpreter comes from the original one of [7], and has been adapted by V.J. Diaz Madrigal.

References


Applying Frameworks and Object-Oriented Techniques for developing Language Processors Tools

M. Cándida Luengo, Juan M. Cueva, Jose E. Labra, Néstor García, David Basanta

Languages and Computer Systems Area
Department of Computer Science
University of Oviedo
Calvo Sotelo s/n
33007-OVIEDO (SPAIN)
{candi,labra,nestor,cueva}@lsi.uniovi.es

Abstract. While the most traditional applications for language processors are the development of compilers and interpreters, they can also be found in the XML parsers of most e-commerce systems, editors with syntax highlighting like those used by programmers, analyzers of complex configuration files like those of Sendmail or Apache, grammar checkers like those present in word processing applications or in component-based development environments, where small scripting languages are used to glue components together. Compiler construction tools have proven to be very useful for developers. In contrast to their importance, traditional tools have failed to take advantage of the object-oriented techniques in the field of computer science. This is particularly unfortunate because compilers being very complex systems would definitely benefit from the modularity and code reuse that object orientation allows. In this paper, we present a tool called O2C2 that brings into the world of compiler construction tools some of the improvements that have taken place in computer science in the last two decades. This tool has been used in several projects, to generate a newer version of O2C2, to implement a XML non-validating parser and to build a parser for the graphical user interface of O2C2. This tool is well suited for projects either large or small.

1 Introduction

Since the 1970s, compilers are developed with the help of scanner and parser generators like Lex[1] and YACC [2] and several versions of this tools like Jlex[3], CUP[4], JavaCC [5] or ANTLR [6]. These tools vastly enhance the ability of individuals to develop language processors and take benefit from a large user base. The problem with these tools is that they have some drawbacks in the way they operate and in the way that the analyzers they generate are designed.
They read a specification that contains both the grammar of the language and the actions associated with each alternative of a production of an element in the grammar. This is a big software engineering problem and a fundamental flaw due to its lack of modularity.

The specifications tend to become huge and debugging requires the following steps: writing action code, compiling the specification file, compiling the resulting code, executing the resulting program to find errors, locating the errors in the program and looking back in the specification file for the related erroneous action code.

The responsibility of keeping the specification and the code synchronized is left to the programmer.

Syntax predicates and semantic predicates are an integral part of the resulting grammar in ANTLR. This makes the grammar obscure.

In this paper, we present a tool called O2C2 that brings into the world of compiler construction tools some of the advancements that have appeared in computer science in the last two decades [7, 8].

This paper is organized as follows. In section 2 we will study how object oriented techniques could help easing those problems with traditional compiler construction tools. In section 3 we describe the graphical user interface. In section 4 we use O2C2 to build an interpreter for a simple calculator. In section 5 O2C2 final remarks are presented. In section 6 we discuss related work. In section 7 we describe future work and in section 8 some conclusions are presented.

2 Object-Oriented Techniques and Frameworks in O2C2

Changing the way compilers are developed is no easy task. Compilers are expected to work with a source text through a series of stages and that makes it easy to think about procedural techniques.

We will define a framework as a reusable design of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate [9]. What this means is that we can develop a set of classes designed to work together and with other external classes and that those classes will facilitate the development of a certain kind of software systems that will work in a related field.

A framework can reduce the cost of developing this type of systems by making it easy to reuse the design and the code in a well-defined domain. This means that people with a basic knowledge of compiler theory can easily develop variants of language processors. A well-defined set of classes is necessary in a framework in order to achieve a consistency between the different software applications that could be using that framework. However, a sufficient level of flexibility is also a must because otherwise the compiler would not adapt to the language it is designed for.

The aim of O2C2 is to help developers create language processors by generating a set of classes from a given syntactic specification of the language. O2C2 generated classes are organized around a framework that represents an AST (Abstract Syntax Tree). This AST’s nodes will be strictly typed so the data cannot be corrupted which helps reducing debugging time.
2.1 The Main Framework

Each time O2C2 creates a new framework, it doesn't need to begin from scratch, O2C2 relays on the O2C2-RT (O2C2 Run Time). O2C2-RT is the core of the framework (as shown in figure 1) and remains the same no matter what language we are working for, and allows a level of consistency between the language processors of different languages.

Node is the root of every framework generated by O2C2. When an O2C2-based analyzer is used, an AST is generated so any token found in the input automatically becomes an instance of a class that inherits from Node. Node also has methods that are common in every node in the AST.

The rest of the classes in this hierarchy are designed to work with the different elements that could appear in a grammar expressed in EBNF notation. For instance, a subclass of NodeChoice is created to treat each of the productions that a non-terminal element in the grammar could have.

![Core Framework in UML notation](image)

**Fig. 1.** Core Framework in UML notation

NodeOptional is the superclass of all the classes that treat the elements of the grammar whose appearance is not mandatory. This represents that some elements may appear in a source text and may not appear in another one and both can be following the rules as given by a grammar.

NodeSeq’s subclasses represent a sequence of elements of the grammar; let them be terminals, non-terminals or EBNF elements that will be discussed later.

2.2 The Grammar and Specific Classes

A grammar in O2C2 is a set of rules that defines the syntax of a programming language. As an example, consider the following grammar:
Non-terminals are represented by words between “<” and “>”, and terminals are represented by capitalized words. With this tool, the user has to add links, quoted words (in the example, "operation", "addition", ...) in the sections of the grammar where she would like her listener classes to be called. Then, the listener classes (in section 2.3) only have to subscribe to classes whose name is that of the link.

With that input, O2C2 would complete the core framework with a group of classes as illustrated by the diagram of figure 2.

The diagram shows that for each element and each production of the grammar a class is generated. For instance, each non-terminal generates a class whose name is the same as the non-terminal. Of course, all these classes are descendants from the interfaces and classes that compose the core framework.

For instance, S is a subclass of NodeChoice. On the other hand operation is a subclass of NodeSeq, which means that to parse operation we first have to parse num (two times as seen in the grammar definition) and op.

numberEBNF0 is a special case. It is a subclass of NodeSeqOptional and its name comes from the fact that there is a EBNF type repetition that occupies the first position (hence the 0 in the name) in the production. This class will do when invoked is parse while there is a DIGIT to be read at the input.

This diagram also represents the calling dependencies between the different classes generated from the grammar. The process of parsing a source text whose structure follows a grammar previously defined means calling the parse() method of the first class in the object diagram, in the previous example this was class S. From this class, the parser generated by O2C2 continues calling the parse() method of class operation all the way down to a class that calls a NodeToken.
O2C2 is not a framework: it is a framework generator. With these classes and the framework, the user already can obtain a working parser.

2.3 Listener Classes

When O2C2 finishes its work, the user gets a working analyzer ready to be compiled and used. O2C2 has already taken care of the analysis of the source but is up to the user to do something with it.

When a parser generated by O2C2 reduces an expression, it calls an object provided by the user if there is one. So what the user has to do in order to provide functionality is to create objects that can be called by the classes in the framework.

This way of adding code to the framework is also known as the listener pattern, and it is already popular among Java programmers because it is present in the JFC [10] framework. By using it, O2C2 allows the co-existence and communication of classes created by the user and its own classes.

Basically, a parser generated by O2C2 analyzes a source text trying to match the input with an element of the grammar. When that element of the grammar corresponds to a section attached to a link, an event is generated. This event may or may not be interesting to the user. If it is, the user should create a class that registers to this class of event. In the example of the calculator we can create a class that will subscribe to events generated by number. This means that every time the parser finds a group of tokens organized as described by a grammar section which is in the left side of the number link, then a special method of the user’s class, called performAction(), will be called. In this method the user can do what she needs, and when it finishes the parsing is resumed.

![Fig. 3. Traditional versus modern compiler construction tools](image)

As opposed to other tools, the user doesn’t need to create classes for all the events that might appear in an analysis. When an event hasn’t got a listener subscribed then a default action is performed by O2C2RT so the values of the event are preserved for...
later use in other stages of the analysis. This feature frees the user from having to create action code that doesn’t provide anything meaningful to the parsing process.

The flexibility of the listener pattern approach has many advantages. An object can register or de-register at run-time depending on the circumstances of the source that has already been parsed. A grammar can be used with two or more sets of listener classes. Compilers that work in multiple passes can be developed easily just registering and de-registering some classes. A group of user classes that deal with the same syntactic structure can be reused effortlessly. That means that if one or more classes are in charge of an “if-then-else”, these classes could be reused completely in almost any language with such a syntactic expression.

The clear separation of the grammar specification from the action code associated to that grammar means that a change in one side won’t affect the other. For example, whenever a change is made to the action code, the specification has to be run through the YACC program and then through the C compiler. In O2C2, when a change is made in the action code, only one class file has to be recompiled. This has the potential to vastly shorten the edit-compile-debug cycle as shown by the diagram of figure 3.

2.3 Error Handling and Recovery

Error handling and recovery uses the same listener pattern. By default, when an O2C2 generated parser finds an error, it displays a meaningful error message and exits, but it is very easy to add extra code that will make the parser fault-tolerant.

3 The Graphical User Interface

The O2C2 system also includes a graphical user interface called O2C2ui (O2C2 user interface). The aim of this user interface is threefold:

• A usable and intuitive user interface can ease things for the novice user and guide him through the different stages of the development of a language processor.

• Unlike similar tools, O2C2 doesn’t generate lexical analysers. This has some advantages because it gives more choice to the user and makes O2C2 syntactic specifications simpler. This is, an O2C2 specification file not contain the lexical definitions. Nonetheless, it is not easy for novices to create lexical analysers so O2C2ui helps the user create lexical specifications with jflex [11]. We plan to make O2C2ui able to work with other tools.

• O2C2ui also includes support for the semantic analysis. This means that with O2C2ui it is possible to integrate the three stages of a language processor in just one place, as shown in figure 4.

To give a semantic meaning to the language processor, the user has to include action code in the classes generated whose name is that of the link in the grammar. The skeleton of these classes appears in the section Actions with syntax highlighting, as shown in figure 4.
O2C2ui allows the creation of XML files with the information of the scanner, the parser and the semantic specifications for a entire project. It makes easy the user to work with his project in other environments, come back to O2C2ui to read the XML file and to generate the parser if the user has all the specification lexical, specification syntactic and actions code necessary.

Fig. 4. A screenshot taken from O2C2ui

4 General steps to built a language processors using O2C2

We present all the steps of building a language processor (interpreter) using O2C2 for a simple calculator.

1. Creating a specification file for the grammar. Lets call it syntactical.o2c2:

   ```
   <S> ::= <num> OP <num> "operation" ;
   <num> java.lang.Integer::= DIGIT (DIGIT) "number" ;
   ```

   The words in uppercase are the tokens of the grammar and for the non-terminal element `num` we only accept values of type `Integer`.  

2. Creating a lexical analyzer that recognizes digits and operators. For this we will use the jflex tool [11] and a specification file named `lexical.flex`. Here is the code.

   ```
   %
   class lexer
   extends symb
   implements o2c2rt.Lexer
   type stnode.Symbol
   %
   ```
We use some directives of the Jflex specification to indicate that the class name that are going to generate is lexer. This class implement the o2c2rt.Lexer class and it descends from the Symb class. The Symb class is generated by O2C2 and it contains the constant associates to each token. Also we define that the class that we are going to return is o2c2rt.Symbol and it is the type to return from yylex. Now, we generate the syntactic analyzer and lexical analyzer.

```java
% java o2c2.Main sintactical.o2c2
% jflex lexical.flex
% javac parser/*.java
% javac *.java
```

To test the parser generated we can create one file named input with only one line, for example: 56+42

```java
% java myParser input
```

In case of errors, the error handling and recovery system that incorporates O2C2 will output a message indicating where it has found it and that it was expected.

3. Creating the listeners.

If we have already created the specification and O2C2 has generated the framework and some auxiliary classes, then we can begin the process of creating the
listeners. In the grammar of the example there are only two productions and we are interested in creating listeners for them. The first alternative is:

\[ \text{num} \ ::= \text{DIGIT} \ (\text{DIGIT}) \ "number" ; \]

The class associated with the alternative \text{num} is called \text{number}. This means that the class that we are going to create, will subscribe to it. This example shows how to implement the interface \text{Listener}.

```java
import parser.*;
import o2c2rt.*;
import java.util.Vector;
public class Anumber implements Listener {
    public Anumber () {
        number.addListener (this); }
    public void performAction (Node o) {
        number num = (number) o;
        String s1 = (String) num.getValue(0);
        String s2 = "";
        Vector v1 = (Vector) num.getValue(1);
        if (v1!=null) {
            for (int contador=0;contador<v1.size(); contador++) {
                String aux = (String) v1.elementAt(contador);
                s2 = s2+aux;
            }
        }
        num.setActionResult (new Integer(s1+s2));
    } catch (ParserException e) {
        e.printStackTrace();
    }
}
```

The constructor adds the class to the set of listeners that have requested subscription to any instance of \text{number}.

The functionality provided by this method is the following: collect the digits found at the input and put them in a \text{String} forming a number.

As we have defined the non-terminal \text{num} of type \text{Integer}, when we call the method \text{setValue()} it is necessary to use an \text{Integer} as the parameter.

In the method \text{performAction()}, the variable \text{getValue(0)} will be a digit, but we first translate it to a string so we can concatenate it with the rest of digits

On the other hand, \text{getValue(1)}, which is the variable that corresponds to \{\text{DIGIT}\}, is of type EBNF, that is, while there are \text{DIGIT}s to be read at the input, the class will continue its work of making the analysis. As we don’t have any listener subscribed to this class, the semantic value of the EBNF class is the default value, in this case, a list (\text{java.util.Vector}) of the tokens that have been read by the class.

There is no need to create a listener for each elements of the grammar that has a corresponding class in the framework generated by O2C2. If a class of the framework has no listeners, then the values obtained are preserved for later use in the analysis.

Obviously this is not the easiest way to obtain a number. This task is in fact more likely to be delegated to the lexical analyzer, not to the syntactic analyzer, but we have done things this way for pedagogic reasons.

The other production we have to deal with is:

\[ \text{S} \ ::= \text{num} \ \text{OP} \ \text{num} \ "operation" ; \]
The class to which we will subscribe our listener will be called *operation* and the listener is:

```java
import o2c2rt.*;
import parser.*;
public class AOperation implements Listener {
    public AOperation ()
    { operation.addListener (this); }
    public void performAction (Node o)
    {
        int result= 0;
        int num1, num2;
        String op;
        try {
            operation s = (operation) o;
            num1 = ((Integer)n.getValue(0)).intValue();
            num2 = ((Integer)n.getValue(2)).intValue();
            op= (String) n.getValue(1);
            if (op.compareTo("-")==0) result=num1-num2;
            else if (op.compareTo("+")==0) result=num1+num2;
            else if (op.compareTo("*")==0) result=num1*num2;
            else if (op.compareTo("/")==0) { 
                if (num2==0) {
                    throw new SemanticException
                    ("Error, division by zero");
                } else result = num1/num2; }
            System.out.println ("\nResult: \n"+result);
        } catch (ParserException pe) {
            pe.printStackTrace();
        }
    }
}
```

The listener transforms *getValue(0)* and *getValue(2)*, the two *<num>* of the alternative, to two integers and *getValue(1)*, which corresponds to *<op>*; is transformed to a string. Depending of the content of *getValue(1)*, the listener will perform an operation or another and present the result to the user.

So far, the only thing we have got are two classes with their instances subscribed to classes generated by O2C2 from a grammar specification. For the whole system to work someone has to create instances of those classes and so far the only way to do this is to make the modifications in the file *myParser.java*.

Those modifications consist in adding somewhere the code that is needed to create instances of the listener classes. Class *myParser* is:

```java
import o2c2rt.*;
import parser.*;
import java.io.*;
public class myParser extends Parser implements Serializable {
    Node first;
    AOperation s;
    ANumber n;
    public myParser (String in)
    { super();
        try {
            1 = new lexer (new FileReader(in));
            catch (FileNotFoundException e) {
                System.out.println(e);
            }
            first = new StS(this);
            n = new ANumber();
```
The modifications are written in bold letters. Compiling again the class and the listeners will result 98.

5 O2C2 and usability

The main purpose of O2C2 is to improve some features of language processors development like extensibility, modularity, reusability and the length of the development cycle using object-oriented techniques.

Now, we will see how the improvement of these features translates into an improvement in usability [12]. Usability is a concept that many people apply only to applications whose user interface (UI) plays an important role. This does not need to be true so restricted and we can see that the advantages of considering usability in language processors tools have the potential to increase the quality of software [13]. There is some agreement that these six attributes comprise the idea of usability:

- Utility

  The first mission of a system is to be useful for some purpose. The utility of O2C2 is similar to that of older tools but if previously, these tools were more compiler-oriented, nowadays the scope of language processors is much broader. O2C2 generates frameworks for object-oriented language processors in Java. This makes it very well suited for new tasks like XML parsers, editors with syntax highlighting, or syntax checking in word processors. Also, it has shown to be particularly well suited when working in large and complex software systems.

- Learnability

  For tools to be used, they have to be easy to learn. For compiler tools to be easy to learn they must be based on a small set of concepts. All language processors generators expect their users to have a minimum knowledge of compiler theory. Nonetheless there are some differences between O2C2 and other tools. The specification files in other tools are so complex that it is difficult for a user without previous experience to begin using the program. On the other hand, O2C2’s specification file is simple because it contains only the grammar in EBNF notation.
Furthermore, the O2C2 system includes a graphical user interface that makes it easy for new users to begin use it.

- **Efficiency**

  If we want users to use one application, that application should be efficient; otherwise they will resort to traditional methods. The importance of this attribute is easy to see in the case of web sites. Regardless of the usefulness of the site, if it’s too slow to load on the user’s browser it will never be used. This will mean that tools should be efficient but also the analyzers generated by them should be efficient in both, resource consumption and speed.

  Nonetheless, efficiency in these tools can be seen from various points of view. If we are mainly interested in speed of execution or resource consumption then, older tools can be considered more efficient than O2C2. Anyway, we believe that an efficient tool is the one that makes the user more efficient. O2C2 separates the grammar from the action code, so it is not necessary to use the tool constantly. Instead, the tool will be used only when there is a change in the grammar, something that seldom happens. We think that by shortening the development cycle, O2C2 is a more efficient tool.

- **Retainability**

  When an application forces the user to deal with lots of data simultaneously and it is easy to get lost under the sheer number of details, then that application cannot be called usable. In the case of language processors generator tools this means complex specifications for the grammar or lots of different concepts needed to understand the analyzer.

  In traditional tools, this problem has special relevance because the specification file includes both the grammar and the action code. This means that even for simple languages, specifications easily become huge and complex so the user can get lost between so many details.

  As we have said, O2C2 separates the grammar specification and the user-provided action code. The tool tries to be simple and this simplicity means that with little interaction with the user, O2C2 can generate a working analyzer.

  Also affecting the retainability of a tool is the number of concepts that must be understood by a potential user before using it. The less the number of concepts involved, the better. The number of concepts needed to begin using O2C2 is minimal and chances are that the user already knows them, like the listener pattern.

- **Errors**

  Applications should be ready to work under unexpected conditions and in the case of compiler generators, this means providing help to the user when there is something wrong in his grammatical specification but also generating analyzers that are able to work when they find some problems.

  Error management has not received a lot of attention by the designers of traditional compiler tools, or at least that is what is seems. This is very unfortunate because language designers and implementers are as prone to commit errors as any other kind of user.
As we have seen in other attributes, there are two users we have to take in mind when designing compiler generators. On one side we have the users of the tools. These users have to deal with complex grammatical specifications, which means that errors will happen. A compiler construction tool should not only detect errors in the specification but also provide some guidance to the user, helping him to locate and correct the error. Many compiler tools detect errors in the specification but the only give obscure error messages.

The other user of a compiler generator is the user of the language processors generated by it. We will call him end-user. If the tool creates analyzers in which it is difficult to incorporate code for error management, this code will probably be missing in the application. So far, compiler tools only provide the means to catch errors in end-user input to the analyzer, but it is not easy to treat those error where they have taken place. This makes it very difficult for an analyzer to recover from a end-user error and continue working.

The error handling and recovery in O2C2 uses the same listener pattern already described for the action code. By default, when an O2C2-generated parser finds an error, it displays a meaningful error message and exits, but it is very easy to add extra code that will make the parser fault-tolerant.

For the first kind of users, O2C2 detects errors at their inputs and explains where they have taken place and some times how to solve it. This is not as hard as it seems because most of user’s errors are always the same and happen for the same reasons.

Also, O2C2 take advantage of Java’s implementation of exception management to allow users treat errors committed by end-users when feeding an input to an analyzer generated by O2C2.

- Satisfaction

An application with an intuitive and reliable behavior is likely to be satisfactory for most users. What is more important in compiler tools is not to cause frustration to the user. Most of the frustration that these tools can cause in the user is due to unnecessary complexity, unexpected behavior, lack of documentation and examples, or lack of power.

YACC antiquity and popularity means that there are lots of examples and documentation. As for power, YACC’s table-based analyzers are very powerful. On the other hand, the complexity of its specifications and of the analyzers it generates is undoubtedly a source of frustration for the user.

O2C2’s documentation tries to be comprehensive and pedagogic. The specification of a grammar couldn’t be simpler, the analyzers it generates are predictive and it is behavior easier to understand.

6 Related works

Recently, there have been some attempts to introduce object-oriented techniques in the field of compiler construction. One of the most interesting examples of this is SableCC [14]. O2C2 and SableCC have some things in common, but there are also some important differences.
O2C2 has a core framework that acts as a run-time and that is present across all parsers. SableCC’s grammatical specification is bigger because it includes lexical and the syntactical specifications whereas O2C2 specifications contain only in the syntactical specification. The reason behind this decision is that we wanted to make O2C2 more modular. SableCC generates a framework that represents an AST and the user has to build a tree-walker class for it. On the other hand, the framework generated by O2C2 parses a source-text and when it finds something of interest it calls the class or classes generated by the user that have shown interest in that kind of event. We feel that O2C2’s approach is more modular, dynamic and flexible.

The event delegation mechanism of O2C2 is present in the JFC [10] and SaX API [15]. This means that those people, who already know how to use a SaX parser, also know the basic principles that rule the behaviour of an O2C2-generated parser and therefore should be able to develop their own custom-made parsers in almost no time.

7 Future Work

O2C2 is an unfinished work and while we are getting near a stable release some areas of improvement can still be identified:

- Make O2C2’s architecture flexible and language-independent. Right now, the O2C2 run-time uses Java and the code it generates is in Java too. We plan to make a flexible integration.
- To incorporate an editor that understood both rules and translation, and can be programmed to edit both using different conventions.

There has been an increasing interest in modular development of programming languages with the aim of rapidly prototyping domain specific languages. This modularity can be achieved separating the language semantic through the notion of monads [16]. The connection between our framework and those works seems a promising area for future research.

8 Conclusions

Language processors generators are prone to be counter-intuitive, difficult to learn and complex to operate. The fact is that designing compiler construction tools for usability has benefits not only for the users of these tools but also for the users of the language processors generated by them.

Traditional compiler tools have brought lots of advantages for programmers and language designers as they shortened the development time and lowered the barrier needed to build a language processor, but they felt short in other areas like usability.

On the other hand, compiler construction tools have in most cases failed to take advantage of object oriented technologies. The use of such techniques to develop compilers has the potential to ease the modularity and to improve code reusability. It could also reduce the length of the code written by the programmer, shorten the development time and make the code easier to read and maintain.
The introduction of frameworks in the field of language processors construction, while not an obvious task, will undoubtedly foster the use of object oriented techniques in the same way that frameworks fostered the use of object oriented techniques in the field of graphical user interfaces.

We feel that O2C2 has the potential to increase the productivity of programmers and the quality of their products as compared with traditional compiler generators because it was designed with usability in mind and object-oriented techniques. We think it is easy to learn and the concepts it uses are known for most programmers. For all those reasons, we believe that O2C2 could be successfully used with new projects either large or small. We have developed a prototype implementation in Java. It is available in [http://www.di.uniovi.es/~candi/o2c2/o2c2.htm](http://www.di.uniovi.es/~candi/o2c2/o2c2.htm).

References

Language Prototyping in the Maude Metalanguage

Manuel Clavel\textsuperscript{1}, Francisco Durán\textsuperscript{2}, Steven Eker\textsuperscript{3}, Patrick Lincoln\textsuperscript{3}, Narciso Mart-Oliet\textsuperscript{4}, and José Meseguer\textsuperscript{5}

\textsuperscript{1} Departamento de Filosofía, Universidad de Navarra, Spain, clavel@unav.es.
\textsuperscript{2} E.T.S.I. Informática, Universidad de Málaga, Spain, duran@lcc.uma.es.
\textsuperscript{3} SRI International, Menlo Park, CA 94025, USA, \{eker,lincoln,meseguer\}@sri.arl.com.
\textsuperscript{4} Facultad de Matemáticas, Universidad Complutense, Madrid, Spain, narciso@csip.ucm.es.

Abstract. This paper presents the principles and techniques supported by the rewriting logic language Maude to prototype a wide range of languages. Maude should be viewed as a metalanguage in which the syntax and semantics of all these languages can be formally defined, and in which entire environments for such languages can be built. An important advantage is extending the logical framework in which the languages are prototyped from equational logic to rewriting logic. Equational logic is quite adequate for prototyping deterministic languages, whose semantics can be naturally understood in functional terms and whose execution can be adequately captured by Church-Rosser equations. In general, however, the operational semantics of a language can be highly nondeterministic. Furthermore, issues of strategy become essential in the very definition of a nondeterministic language. Rewriting logic, while containing equational logic as a sublogic, can deal very naturally with nondeterministic languages and with strategy issues. A further advantage offered by Maude is the systematic use of reflection to achieve more powerful metalanguage functionalities, including a new style of language definition that “compiles” language modules into rewrite theories instead of defining an interpreter, the capacity to define and prototype module operations, the capacity to deal with languages that can be extended by user-definable syntax, and the internalization of strategies within the logic.

1 Introduction

This paper presents the principles and techniques supported by the rewriting logic language Maude \cite{23,4} to prototype a wide range of languages, including formal specification languages, logics, theorem proving systems, architectural description languages, programming languages, and domain-specific languages. Maude should be viewed as a metalanguage in which the syntax and semantics of all these languages can be formally defined, and in which entire environments for such languages, including parsers, module operations, execution environments, debugging tools, pretty printing, and input/output, can be built.
This work is part of a rich language prototyping tradition based on algebraic techniques, and should be seen as a further step in the context of previous efforts to use algebraic languages to give executable semantic definitions of other languages, including OBJ [18], ASF+SDF [11], action semantics [25], and ELAN [20]. There are, however, a number of new contributions that we think Maude can make in this area. The first, shared with ELAN, is extending the logical framework in which the languages are prototyped from equational logic to rewriting logic. Equational logic is quite adequate for prototyping deterministic languages, whose semantics can be naturally understood in functional terms and whose execution can be adequately captured by Church-Rosser equations. In general, however, the operational semantics of a language can be highly nondeterministic either because of its style—for example a logical language—or because of its concurrent character. Furthermore, issues of strategy become typically very important in the very definition of a nondeterministic language and do not have an adequate treatment within a purely equational context. Rewriting logic, while containing equational logic as a sublogic, can deal very naturally with nondeterministic languages and with strategy issues.

A further new contribution of Maude is the systematic use of reflection to achieve more powerful metalanguage functionalities, including a new style of language definition that “compiles” language modules into rewrite theories instead of defining an interpreter, the capacity to define and prototype module operations, the capacity to deal with languages that can be extended by user-definable syntax, and the internalization of strategies within the logic. In particular, when these capabilities are applied to Maude itself, they allow new very flexible ways of extending the language and its module operations, and of changing its syntax and its basic commands. In the rest of this introduction we explain these ideas in more detail and we summarize the contents of the paper.

1.1 Prototyping Languages in the Rewriting Logic Framework

One of the key goals of rewriting logic from its beginning [23] has been to provide a semantic framework in which many models of computation—particularly concurrent and distributed ones—and languages can be naturally represented. Because of the intrinsic duality between logic and computation that rewriting logic supports, the very same reasons making rewriting logic a suitable semantic framework, make it also an attractive logical framework [21] to represent many different logics. Thanks to the sustained efforts of many researchers, there is by now very extensive evidence supporting the claim that rewriting logic is indeed a very flexible and simple semantic and logical framework. Moreover, object-oriented design languages, architectural description languages, and languages for distributed components also have a natural semantics in rewriting logic. See [22] for references.

What is common to all these applications is that the models of computation, logics, or languages are represented in rewriting logic by mappings of the form

\[(\dag) \\Phi : \mathcal{L} \rightarrow RWLogic.\]
The representations are typically very simple and natural. A rewrite theory is a pair formed by an equational specification and a set of labeled rewrite rules of the form \( r : t \rightarrow t' \) expressing that the state \( t \) can change to the state \( t' \). The syntax and functional aspects of a language or language module can then be axiomatized as the initial algebra associated to the equational specification, whereas the dynamics or operational semantics of the language or module can be axiomatized by the rewrite rules, that can be highly nondeterministic and that need not terminate.

1.2 A Reflective Approach to Language Prototyping

For language prototyping purposes, the obvious question to ask is: How can a rewriting logic language best support representation maps of the form \( \{ \} \), so that it becomes a *metalanguage* in which a very wide variety of programming, specification, and design languages, and of computational and logical systems can be both *semantically defined*, and *implemented* in it? Our answer is: By being reflective. Maude’s language design and implementation make systematic use of the fact that rewriting logic is reflective \([8,2]\) in the specific sense that its metalevel can be represented at the object level by a *universal theory* that can represent all other theories, including itself. This makes the metatheory of rewriting logic accessible to the user in a clear and principled way, and makes possible many advanced metaprogramming applications, including user-definable strategy languages, language extensions by new module composition operations, development of theorem proving tools, and reifications of other languages and logics within rewriting logic.

Specifically, we can reify a representation map \( \Phi \) of the form \( \{ \} \) by defining an abstract data type \texttt{Module}_{\mathcal{L}} \) representing modules in the logic or language \( \mathcal{L} \). Since in the universal theory we also have a data type \texttt{Module} whose terms represent rewrite theories (see Section 2.2), we can then *internalize* the representation map \( \Phi \) as an equationally defined function

\[
\overline{\Phi} : \texttt{Module}_{\mathcal{L}} \rightarrow \texttt{Module}.
\]

In fact, thanks to the general metasresult of Bergstra and Tucker \([1]\), any computable representation map \( \Phi \) can be specified in this way by a finite number of Church-Rosser and terminating equations.

Since in Maude key functionality of the universal theory is efficiently supported by the predefined module META-LEVEL (see Section 2), using this metalevel gateway, we can then execute in Maude the rewrite theory \( \overline{\Phi}(M) \) associated to a module \( M \) in \( \mathcal{L} \). This has been done, for example, for linear logic in \([21,2]\), for structured Maude modules in \([12]\), and could be done for a very wide range of other languages and logics using the same method.

Notice that this style of representing languages is more modular and flexible than the traditional method of defining an *interpreter*. This could also be done, but using a representation map \( \overline{\Phi} \), we can instead “compile” each module \( M \) in our language \( \mathcal{L} \) into a separate rewrite theory that can be much simpler and compact than a whole interpreter.
Another important advantage of our reflective approach is that we can deal naturally with modular languages. Indeed, since we have reified the modules in $\mathcal{L}$ as elements of a data type $\text{Module}_\mathcal{L}$, we can then define and execute a wide range of module operations as functions on such a data type. As we further explain in the paper, this has already been done for modules in Maude itself [12], but the method is fully general.

Many modular languages have user-definable syntax. By using reflective functions in its META-LEVEL module, Maude can naturally represent and execute such languages within its own logic, something impossible or requiring an ad-hoc treatment in previous approaches.

The topic of strategies that, as we have mentioned, is of paramount importance for nondeterministic languages has also a very simple treatment within the logic. The idea is that, thanks to reflection, strategies can be internalized as theories at the metalevel that guide the execution of the rules at the object level [2].

Besides its reflective capabilities, Maude has a number of additional features that seem particularly useful for language prototyping purposes. As other algebraic languages, it supports mixfix syntax, which is extremely useful to preserve the original syntax of the language being prototyped and to ease readability. Furthermore, rewriting modulo a wide range of combinations of equational axioms such as associativity, commutativity, and identity, is efficiently supported. Finally, thanks to its advanced semicompilation techniques, Maude’s rewrite engine has a high performance allowing some language prototypes to be in fact used as very acceptable language implementations. In particular, this holds true of language extensions for Maude itself [12], and of theorem proving tools for Maude [6, 7], which have been implemented according to the methods presented in this paper and have fully acceptable performance.

1.3 Summary of the paper

We begin the paper with a brief explanation of the reflective capabilities supported by the module META-LEVEL. For creating an environment for a language $\mathcal{L}$ using Maude, we need generic syntax definition, meta-parsing, and meta-pretty printing capabilities that can deal with expressions in any language, including languages like Maude itself whose modules have user-definable syntax. And we need a general facility for input/output that can be customized for each language of interest. Section 3 explains how all this can be done in Maude thanks to its reflective design. Experience and applications are discussed in Section 4. We conclude the paper with some remarks about future research plans.

2 Maude’s Metalevel

Rewriting logic is reflective [8, 2], that is, there is a finitely presented rewrite theory $\mathcal{U}$ that is universal in the sense that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) and any terms $t, t'$ in $\mathcal{R}$ as terms $\overline{t}, \overline{t'}$
and $\overline{t}, \overline{t'}$ in $\mathcal{U}$, and we then have the following equivalence

$$\mathcal{R} \vdash t \rightarrow t' \Leftrightarrow \mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \rightarrow (\overline{\mathcal{R}}, \overline{t'}).$$

Since $\mathcal{U}$ is representable in itself, we can achieve a “reflective tower” with an arbitrary number of levels of reflection, because we have

$$\mathcal{R} \vdash t \rightarrow t' \Leftrightarrow \mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \rightarrow (\overline{\mathcal{R}}, \overline{t'}) \Leftrightarrow \mathcal{U} \vdash (\mathcal{U}, (\overline{\mathcal{R}}, \overline{t})) \rightarrow (\mathcal{U}, (\overline{\mathcal{R}}, \overline{t'})) \ldots$$

In this chain of equivalences we say that the first rewriting computation takes place at level 0, the second at level 1, and so on. In a naive implementation, each step up the reflective tower comes at considerable computational cost, because simulating a single step of rewriting at one level involves many rewriting steps one level up. It is therefore important to have systematic ways of lowering the levels of reflective computations as much as possible—so that a rewriting subcomputation happens at a higher level in the tower only when this is strictly necessary.

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module $\text{META-LEVEL}$. Furthermore, several other useful functions are also built-in for efficiency reasons. What follows describes the module $\text{META-LEVEL}$.

We recall that Maude’s functional modules are equational theories that are assumed to be Church-Rosser and terminating modulo some axioms for which matching algorithms are available in the implementation, and that system modules are rewrite theories whose equational part satisfies the same requirements as a functional module, and where the equations and the rules are assumed to be weakly coherent [32] modulo the axioms. In $\text{META-LEVEL}$

- Maude terms are reified as elements of a data type $\text{Term}$ of terms;
- Maude modules are reified as terms in a data type $\text{Module}$ of modules;
- the processes of reducing a term to normal form in a functional module and of finding whether such a normal form has a given sort are reified by a descent function $\text{meta-reduce}$;
- the process of applying a rule of a system module to a subject term is reified by a descent function $\text{meta-apply}$;
- the process of rewriting a term in a system module using Maude’s default interpreter is reified by a descent function $\text{meta-rewrite}$; and
- parsing and pretty printing of a term in a signature, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also reified by corresponding metalevel functions.

We first introduce the syntax used in $\text{META-LEVEL}$ for representing terms; then we explain how modules are represented; and finally we discuss the different built-in functions, namely, the descent functions, and the parsing, pretty printing, and sort functions.

### 2.1 Representing Terms

Terms are reified as elements of the data type $\text{Term}$ of terms, including the following operations:
subsort Qid < Term .
subsort Term < TermList .
op {...} : Qid Qid -> Term .
op {_[_]} : Qid TermList -> Term .
op {...} : TermList TermList -> TermList {assoc} .
op error* : -> Term .

The first declaration, making Qid a subsort of Term, is used to represent variables by the corresponding quoted identifiers. Thus, the variable N is represented by 'N. The operator {...} is used for representing constants as pairs, with the first argument the constant in quoted form, and the second argument the sort of the constant, also in quoted form. For example, the constant 0 in the module NAT in Section 2.2 is represented as {'0}'Nat. The operator {_[_]} corresponds to the recursive construction of terms out of subterms, with the first argument the top operator in quoted form, and the second argument the list of its subterms, where list concatenation is denoted {...}. For example, the term s(s(0)) + s(0) of sort Nat in module NAT is metarepresented as

'+'{_[_]}{s['0]{Nat}}, {s['0]{Nat}} .

As already mentioned when discussing the universal theory, since terms in the module META-LEVEL can be metarepresented just as terms in any other module, the representation of terms can be iterated. For example, the metaterepresentation s(0) of the term s(0) in NAT is the term

'-'{_[_]}{s['0]{Nat}}, {'0}'{Nat}{Nat}'{Nat}{Qid} .

The last declaration above for the data type of terms is a constant error* to be used as an error element.

2.2 Representing Modules

Functional and system modules are metaterepresented in a syntax very similar to their original user syntax. The main differences are that: (1) terms in equations, membership axioms, and rules are now metaterepresented as we have already explained; (2) in the metaterepresentation of modules we follow a fixed order in introducing the different kinds of declarations for sorts, subsorts, variables, equations, etc., whereas in the user syntax there is considerable flexibility for introducing such different declarations in an interleaved and piecemeal way; and (3) sets of identifiers—used in declarations of sorts—are represented as sets of quoted identifiers built with an associative and commutative operator {...}.

The module META-LEVEL provides sorts and constructors for each of the declarations that can appear in modules. For example, we have sorts OpDecl and OpDeclSet to represent declarations of operations and sets of declarations of operations, respectively. If there are no operation declarations in a particular module, then this argument will be none. Otherwise, the set is given by the associative and commutative constructor {...}, which is declared with identity element none. Each of the operation declarations is then given by a term of sort OpDecl using the constructor...
op _._:->[_]. : Qid QidList Qid AttrSet -> OpDecl .

The last argument of this constructor corresponds to the set of attributes of an operator. For this, sorts Attr and AttrSet are used. Such attributes may be equational axioms to rewrite modulo, syntactic attributes for parsing purposes, and so on. For example, there are declarations

\[\text{ops assoc comm idem : } \rightarrow \text{Attr .} \]
\[\text{ops id left-id right-id : Term } \rightarrow \text{Attr .} \]

The \textit{META-LEVEL} syntax for the top-level operator representing functional modules is as follows:

\[\text{op fmod.is} \_\_\_\_\_\_\_\_\_\_\_\_\text{endfm : Qid ImportList SortDeci SubsortDeciSet}
\text{UpDeciSet VarDeciSet MemAxSet EquationSet } \rightarrow \text{Module .} \]

To motivate the general syntax for representing modules, we illustrate it with a simple example—namely, a module \texttt{NAT} for natural numbers with zero, successor, and commutative addition. The metarepresentation of the module \texttt{NAT} on the left is the term of sort \texttt{Module} displayed on the right, so that the reader can appreciate the similarity between both notations:

\[\text{fmod NAT is nnil .} \]
\[\text{sorts Zero Nat .} \]
\[\text{subsort Zero < Nat .} \]
\[\text{op 0 : } \rightarrow \text{Zero .} \]
\[\text{op } s : \text{Nat } \rightarrow \text{Nat .} \]
\[\text{op } _+_{ } : \text{Nat Nat } \rightarrow \text{Nat [comm].} \]
\[\text{vars } N \text{ M : Nat .} \]
\[\text{eq 0 } + \text{ N } = \text{ N .} \]
\[\text{eq } s(N) + M = s(N + M) . \]
\[\text{endfm} \]

\[\text{fmod } \texttt{'NAT is nnil .} \]
\[\text{sorts 'Zero ; 'Nat .} \]
\[\text{subsort 'Zero < 'Nat .} \]
\[\text{op '0 : nil } \rightarrow \text{'Zero [none] .} \]
\[\text{op 's : 'Nat } \rightarrow \text{'Nat [none] .} \]
\[\text{op '}_{+}_{ } : \text{'Nat 'Nat } \rightarrow \text{'Nat [comm].} \]
\[\text{var 'N ; 'Nat .} \]
\[\text{eq '0 } + \text{'N } = \text{'N .} \]
\[\text{eq 's(N) + N = 's(N + N) .} \]
\[\text{endfm} \]

Since \texttt{NAT} has no list of imported submodules and no membership axioms, those fields are filled by the constant \texttt{nil} of sort \texttt{ImportList}, and the constant \texttt{none} of sort \texttt{MemAxSet}. Similarly, since the zero and successor operators have no attributes, they have the \texttt{none} set of attributes.

Note that—just as in the case of terms—terms of sort \texttt{Module} can be metarepresented again, yielding then a term of sort \texttt{Term}, and this can be iterated an arbitrary number of times. This is in fact necessary when a metalevel computation has to operate at higher levels. A good example is the inductive theorem prover described in [6], where modules are metarepresented as terms in the inference rules for induction, but they have to be meta-metarepresented as terms of sort \texttt{Term} when used in strategies that control the application of the inductive inference rules.
2.3 Descent Functions

The module META-LEVEL has three built-in descent functions: meta-reduce, meta-rewrite, and meta-apply [3]. They provide three useful and efficient ways of reducing metalevel computations to object-level ones.

The operation meta-reduce, with syntax

\[ \text{op meta-reduce : Module Term } \rightarrow \text{ Term} \]

takes as arguments the representations of a module \( \mathcal{R} \) and of a term \( t \) in that module, and returns the representation of the fully reduced form of the term \( t \) using the equations in \( \mathcal{R} \), e.g.

Maude> red meta-reduce(NAT, s(0) + 0).
result Term: s(0)

The operation meta-rewrite has syntax

\[ \text{op meta-rewrite : Module Term MachineInt } \rightarrow \text{ Term} \]

It is entirely analogous to meta-reduce, but instead of using only the equational part of a module it now uses both the equations and the rules to rewrite the term using Maude’s default strategy. Its first two arguments are the representations of a module \( \mathcal{R} \) and of a term \( t \), and its third argument is a positive machine integer \( n \). Its result is the representation of the term obtained from \( t \) after at most \( n \) applications of the rules in \( \mathcal{R} \) using the strategy of Maude’s default interpreter, which applies the rules in a fair, top-down fashion. When the value 0 is given as the third argument, no bound is given to the number of rewrites, and rewriting proceeds to the bitter end.

The operation meta-apply has syntax

\[ \text{op meta-apply : Module Term Qid Substitution MachineInt } \rightarrow \text{ ResultPair} \]

The first four arguments are representations in META-LEVEL of a module \( \mathcal{R} \), a term \( t \) in \( \mathcal{R} \), a label \( l \) of some rules in \( \mathcal{R} \), and a set of assignments (possibly empty) defining a partial substitution \( \sigma \) for the variables in those rules. The last argument is a natural number \( n \). meta-apply then returns a pair of sort ResultPair consisting of a term and a substitution. The operation meta-apply is evaluated as follows:

1. the term \( t \) is first fully reduced using the equations in \( \mathcal{R} \);
2. the resulting term is matched against all rules with label \( l \) partially instantiated with \( \sigma \), with matches that fail to satisfy the condition of their rule discarded;
3. the first \( n \) successful matches are discarded; if there is an \((n + 1)\)th match, its rule is applied using that match and the steps 4 and 5 below are taken; otherwise the pair \{error*, none\} is returned, where none denotes the identity substitution;
4. the term resulting from applying the given rule with the \((n + 1)\)th match is fully reduced using the equations in \( \mathcal{R} \).
5. the pair formed using the constructor \{ \_, \_ \} whose first element is the representation of the resulting fully reduced term and whose second element is the representation of the match used in the reduction is returned.

We finish this section with an example illustrating the support for a reflective tower in \texttt{META-LEVEL}, plus the fact that in the same computation we can descend to different theories at different levels. We “meta-reduce” the pair consisting of the metarepresentation of \texttt{META-LEVEL} and the metarepresentation of the term “meta-reducing” an expression in \texttt{NAT}, to get the meta-metarepresentation of the result.

\begin{verbatim}
Maude> red meta-reduce(META-LEVEL, meta-reduce(NAT, s(0) + 0)).
result Term: s(0).
\end{verbatim}

\section*{2.4 Internal Strategy Languages}

By using the functions \texttt{meta-reduce}, \texttt{meta-rewrite}, and \texttt{meta-apply} as basic building blocks to define more complex strategies, one can easily use Maude’s metalevel to program execution strategies with equations and rewrite rules. In this way, strategies become internalized within rewriting logic—so that they have a logical semantics and can be reasoned about as other rewrite theories—and strategy languages become easily extensible \cite{9, 2}. As a very simple example, one can encode the most naive description of sorting as a single rewrite rule as follows:

\begin{verbatim}
crl [sort]: sort([X, E1, Y, E2, Z]) => sort([X, E2, Y, E1, Z]) if E2 < E1.
\end{verbatim}

Then, various sorting algorithms can be encoded at the metalevel in very straightforward ways. In particular, bubble sort can be captured by searching for matches of the above rewrite rule where \texttt{Y} is empty. Other sorting algorithms can be encoded with slightly more elaborate strategies \cite{5}. Internal strategies can be used to specify algorithms, to control the execution of interpreters and theorem provers, to encode winning strategies for games, to emulate a model checker, and for many other purposes \cite{2, 6, 4}.

\section*{2.5 Parsing, Pretty Printing, and Sort Functions}

Besides the descent functions already discussed, \texttt{META-LEVEL} provides several other functions that naturally belong to the universal theory and could have been equationally axiomatized in such a theory. However, for efficiency reasons they are provided as built-in functions. These functions allow parsing and pretty printing a term in a module at the metalevel (see Section 3.2), and performing efficiently a number of useful operations on the sorts declared in a module’s signature. The operations on sorts include \texttt{sortLeq}, \texttt{leastSort}, \texttt{lessersorts}, and \texttt{globalsorts}. They provide commonly needed functions on the poset of sorts of a module in a built-in way at the metalevel, and have been explained in detail in \cite{4}.

3 Language Prototyping in Maude

As we have already explained in the introduction, models of computation, logics, and languages are represented in rewriting logic by maps of the form

$$\Phi : \mathcal{L} \rightarrow RWLogic.$$

A map of this kind can be internalized by means of an abstract data type $\text{Module}_C$ representing modules in the logic or language $\mathcal{L}$, and an equationally defined function

$$\overline{\Phi} : \text{Module}_C \rightarrow \text{Module},$$

where $\text{Module}$ is the data type representing finitely presentable rewrite theories in the language (see Section 2.2). Using this metalevel gateway, we can then execute in Maude the rewrite theory $\overline{\Phi}(M)$ associated to a module $M$ in $\mathcal{L}$.

In practice, however, we would like to be able not only to represent and transform the modules of a language $\mathcal{L}$ as terms of a data type $\text{Module}_C$ within Maude. We would like to use Maude to generate a whole environment for $\mathcal{L}$, including a facility for defining and modifying the language’s syntax, an input/output facility, a parser, a pretty printer, and an execution environment for it. Furthermore, we would like to be able to do this for languages that—like Maude itself, and many other formal specification languages—have modules with user-definable syntax, so that expressions in those modules cannot be parsed with a fixed syntax for the language, but need to be parsed with the particular syntax introduced in the module.

3.1 Syntax Definition

In order to generate in Maude a whole environment for a language $\mathcal{L}$, the first thing we need to do is to define the syntax for $\mathcal{L}$-modules. This can be done by extending the module $\text{META-LEVEL}$ with a data type $\text{Module}_C$ for $\mathcal{L}$-modules, and with other auxiliary data types for commands and other constructs. This can be easily and naturally achieved using the mixfix frontend, and the built-in data types $\text{Token}$ (any identifier) and $\text{Bubble}$ (any string of identifiers). The intuition behind these types is that they correspond to pieces of a module in a language that can only be parsed once the grammar introduced by the signature of the module is available.

The idea is that for a language that allows modules with user-definable syntax—as it is the case of Maude—it is natural to see its syntax as a combined syntax, at two different levels: what we may call the top level syntax of the language, and the user-definable syntax introduced in each module. The data types $\text{Token}$ and $\text{Bubble}$ allow us to reflect this duality of levels in the syntax definition.

To illustrate these concepts, suppose that we want to define the syntax of Maude in Maude. Consider the following Maude module:
Language Prototyping in the Maude Metalanguage

```plaintext
fmod NAT3 is
  sort Nat3
  op 0 : -> Nat3 .
  op s_ : Nat3 -> Nat3 .
  eq \| s s s 0 \| = 0 .
endfm
```

Notice that the strings of characters inside the boxes are not part of the top level syntax of Maude. In fact, they can only be parsed with the grammar associated to the signature of the module NAT3. A correct and useful (in terms of defining afterwards a meaningful parser) definition of the syntax of Maude in Maude must reflect this duality of syntax levels. We show below parts of the module MAUDE that defines the syntax for functional modules in Maude.

```plaintext
fmod MAUDE is
  sort PreModule PreCommand .
  subsort Decl < DecList.
  op eq_= : Bubble Bubble -> Decl .
  op fmod_is_endfm : Token DecList -> PreModule .
  op red in_- : Token Bubble -> PreCommand .
  ...
```

Notice how we explicitly declare operators that correspond to the top level syntax of Maude, and represent as terms of sort Bubble those pieces of the module that can only be parsed with the user-defined syntax, like, for example, the left and right hand sides of an equation.

Then, the functional module NAT3 above can be parsed as a term of sort PreModule in MAUDE. The name of this sort reflects the fact that not all terms of PreModule actually represent Maude modules. In particular, for a term of sort PreModule to represent a Maude module all the ‘bubbles’ must be correctly parsed in the user-defined syntax. We will come back to this important point in the next section. Finally, notice that we have also defined the syntax for commands like `reduce` in Maude. We shall see in Section 3.4 how the user can define the semantics of commands of any kind for his language of interest.

### 3.2 Parsing and Pretty Printing

A built-in function `meta-parse` is declared in META-LEVEL with syntax

```plaintext
op meta-parse : Module Bubble -> Term .
```

The function `meta-parse(M, i_1 ... i_n)` checks whether the string of identifiers `i_1 ... i_n` is a well-formed term in the module represented by `M`: if this is the case, `meta-parse` returns the representation of `i_1 ... i_n` as a term of sort `Term` in META-LEVEL; otherwise, it returns `error`. This built-in function can be very useful to specify in Maude an efficient parser for a language `L` that allows modules with user-definable syntax. For example, a parser for pre-modules in Maude essentially consists of a function `parse-premodule` with syntax
Given a `PreModule` term $M$, `parse-premodule` generates, first, a term of sort `Module` $M'$ that represents a module in Maude with the exact same signature as $M$ but without equations, rules, or any other declaration containing bubbles; then, `parse-premodule` attempts to transform the `PreModule` $M$ into a `Module` $M''$ by reducing each `Bubble` $b$ in $M$ to a `Term` $t$, where $t$ is the successful result of `meta-parse($M'$, $b$)`. In case of failure, an error term in the supersort `Module` of `Module` is returned.

The inverse function to `meta-parse` is also included as a built-in function in the module `META-LEVEL`, with syntax

\[
\text{op meta-pretty-print : Module Term -> Bubble .}
\]

It takes as arguments the representation of a module $M$ and the representation of a term $t$, and it returns a string of identifiers produced by pretty printing $t$ in the syntax given by $M$.

### 3.3 Module Algebra and Execution

We are interested in using Maude to build environments for languages $\mathcal{L}$ such as formal specification languages and modular programming languages whose modules can have user-definable syntax. In addition, such modules can be highly structured and parameterized. That is, there can be a rich collection of module composition operations endowing $\mathcal{L}$ with a module algebra. In such cases we typically have two data types of modules, a data type `Module_\mathcal{L}` of flat or unstructured modules, and a more general data type `StrModule_\mathcal{L}` of structured modules.

The point is that both data types and all the module algebra operations for $\mathcal{L}$ can be defined within Maude as an extension of the module `META-LEVEL`. In this way, the environment that we can build for $\mathcal{L}$ using Maude can also support all the module composition operations of $\mathcal{L}$. Among those module operations, a common and important one is flattening, that is, the process of passing from a structured module to its unstructured flat form. This can be understood as a function

\[
\text{StrModule}_\mathcal{L} \xrightarrow{\text{flatten}} \text{Module}_\mathcal{L}.
\]

Since modularity constructs can change from language to language, it may be simpler to represent $\mathcal{L}$ in rewriting logic by representing only its flat modules, that is, by a function $\mathcal{F} : \text{Module}_\mathcal{L} \rightarrow \text{Module}$ which makes the language $\mathcal{L}$ executable on top of Maude thanks to the descent functions `meta-reduce`, `meta-rewrite`, and `meta-apply`, as well as other more complex evaluation strategies that can also be defined at the metalevel to execute the rewrite theory $\mathcal{F}(M)$ representing a given module in $\mathcal{L}$. Using the function $(\cdot)^{\mathcal{L}}$ we can also make structured modules in $\mathcal{L}$ executable by means of the function composition

\[
\text{StrModule}_\mathcal{L} \xrightarrow{(\cdot)^{\mathcal{L}}} \text{Module}_\mathcal{L} \xrightarrow{\mathcal{F}} \text{Module}.
\]
The first most obvious language $\mathcal{L}$ to which we can apply these ideas is Maude itself. The Core Maude sublanguage has flat modules represented by the data type Module. But general Maude modules can be structured and parameterized, and can contain very complex module expressions that instantiate and rename several, possibly parameterized, modules; and all this can also happen for object-oriented modules. A module algebra for Maude written in Maude is presented in [12]. In particular, such an algebra contains as well a flattening function making structured Maude modules executable in the Maude engine.

### 3.4 Input/Output

Using object-oriented concepts, we can specify in Maude a general input/output facility provided by a module LOOP that extends the module META-LEVEL into a generic read-eval-print loop. This facility can then be specialized for each language $\mathcal{L}$. There is a class I/O of input/output objects acting on behalf of users, with two attributes: an input and an output buffer storing Bubbles. There is also a class System with an input buffer of sort Input, which is defined as a supersort of PreModule and PreCommand, an output buffer of sort Bubble, and an attribute of sort Database. The sort Database is left completely unspecified, so that, depending on the language $\mathcal{L}$ of interest it can be specialized accordingly.

For example, for Maude it will contain the current database of modules already entered in the system.

- subsorts PreCommand PreModule < Input .
- class I/O | input : Bubble, output : Bubble .
- class System | db : Database, input : Input, output : Bubble .

The module LOOP provides a persistent pair of objects—one of class I/O called user and one of class System called system—that interact with each other by exchanging data in and out between their buffers.

For each particular language both the sort Database for the system object and the additional rewrite rules defining the system behavior for different language commands are specified according to the specific details of the language in question. We illustrate below the case of Maude. Processing of a PreModule once it has been entered into the system is done by the rule

```
var DB : Database . var PM : PreModule .
var i : identifier . var B : Bubble .

rl [premodule] :
  < system : System | db : DB, input : PM >
  => < system : System | db : processPreModule(PM, DB),
        input : empty > .
```

The function processPreModule attempts to parse the PreModule using the function parse-premodule, and, if it succeeds, it introduces the resulting Module into the database.

Then, for user-defined commands as `red in _:_`, we can define rules of the form
where \texttt{getModule} is a function that extracts from the database the flat module whose name is given in its first argument.

## 4 Experience and Applications

We summarize here our experience in using Maude to prototype other languages, as well as different application areas for these techniques, including extending Maude itself, representing languages, and building theorem proving tools.

### 4.1 Extending Maude

As already mentioned in Section 3, the first most obvious area where Maude can be used as a metalanguage is in building language extensions for Maude itself. Our experience in this regard, reported in [12], is very encouraging. We have been able to define in Core Maude a language extension with parameterized modules, views (for module instantiation) and module expressions together with all the additional functionality required for parsing, evaluating, and pretty printing modules in the extended language. Furthermore, thanks to the efficient implementation of the rewrite engine, the parser, and the module \textsc{Meta-Level}, such a language extension executes with very reasonable efficiency [4].

### 4.2 Reifying Languages and Logics

Rewriting logic has great potential as a logical framework that can represent many different languages and logics. As we have already explained, by reflection, such representation maps can be \textit{reified} and executed within rewriting logic. The key idea is to define a data type \texttt{Module}_\mathcal{L} for the modules of the language or logic \mathcal{L} in question, and then reify the representation map at the metalevel as an equationally-defined function mapping terms in \texttt{Module}_\mathcal{L} to terms in Module. More generally, one can in a similar way reify translations between two different languages or logics \mathcal{L} and \mathcal{L}' as functions from \texttt{Module}_\mathcal{L} to \texttt{Module}_\mathcal{L}'; in this way, one can execute such translations in Maude. This can for example be used to connect different formal tools such as theorem provers to interoperate them in a heterogeneous environment.

Rewriting logic definitions of several languages have been given by different authors, including, among others, the lambda calculus and mini-ML [21, 27], Prolog and narrowing languages like BABEL [33], CCS [21, 30], the \pi-calculus [31, 27], the reflective concurrent logic programming language GAEA [19], the
active network programming language PLAN [24], the UML metamodel [29, 17],
and the cryptographic protocol specification language CAPSL [10].

Similarly, many different logics have been shown to have a very natural con-
servative representation in rewriting logic [21], including equational logic, Horn
logic with equality, linear logic, logics with quantifiers, and any sequent calculus
presentation of a logic for a very general notion of “sequent”. The point is that
all such representation maps can be reified using the reflective method described
above. For the case of the representation map from linear logic into rewriting
logic this was defined in [21] and implemented in Maude in [2]. Finally, rewriting
logic may also be used to reason about the models of computation expressed
in different languages, providing a semantic foundation for building connections
between diverse models of computation. See [22] for a detailed list of references.

4.3 Developing Theorem Proving Tools

A particularly useful area of metalevel applications is representing in Maude an
inference system for a logic of interest, to get in this way a theorem prover for
it. In our experience this makes developing a theorem prover almost as easy as
writing down its inference system in a scientific paper [7, 6]. Furthermore, tools
designed in this way are very easily modifiable and extensible, and, thanks to
Maude’s high performance implementation, they can often compete in efficiency
with tools developed by conventional means. Reflection plays here a crucial role
for two reasons. Firstly, the inference system itself may perform theory trans-
formations, so that the theories themselves must be treated as data; therefore,
the rules of our inference system may already be at the metalevel to begin
with. Secondly, we need strategies to guide the application of the inference rules.
Therefore, if the inference rules themselves are already at the metalevel, the
strategies are then at the meta-metalevel. For tool development, this distinction
of levels is of great practical importance, because it allows a very modular mod-
ification of the proof tactics with no change whatsoever to the inference system.
In conventional implementations both aspects are sometimes so intertwined that
modifications can be very difficult.

Maude has been used in exactly this way [7, 6] to build tools such as an
inductive theorem prover; tools to check the Church-Rosser property, coherence,
and termination, and to perform Knuth-Bendix and coherence completion [16,
14, 13]; and a tool to specify, analyze and model check real-time specifications
[26].

Another area of current research is using Maude’s metalanguage facilities
to define translations between different theorem provers. For example, the map
HOL → Nuprl between the logics of the HOL and Nuprl theorem provers has
been specified in Maude by Stehr, Naumov, and Meseguer [28].

This method of building formal tools is not restricted to Maude-related tools.
One can generate tools from their rewriting logic specifications for any finitary
logic.
5 Future Directions: Mobile Maude

As another example of a language that is being prototyped using our approach, we are currently advancing the design of Mobile Maude [15]. This is an extension of Maude supporting mobile computation that uses reflection in a systematic way to obtain a simple and general declarative mobile language design. The two key notions are processes and mobile objects. Processes are located computational environments where mobile objects can reside. Mobile objects can move between different processes in different locations, and can communicate asynchronously with each other by means of messages. Each mobile object contains its own code—that is a rewrite theory $\mathcal{R}$—metarepresented as a term $\mathcal{R}$. In this way, reflection endows mobile objects with powerful “higher-order” capabilities within a simple first-order framework.

We expect that Mobile Maude will have good support for secure mobile computation for two reasons. Firstly, mobile objects will communicate with each other and will move from one location to another using state-of-the-art encryption mechanisms. Secondly, because of the logical basis of Mobile Maude, we expect to be able to prove critical properties of applications developed in it with much less effort than what it would be required if the same applications were developed in a conventional language such as Java.

References


In this talk, we present a calculus of channels that we call Rew. The purpose of Rew is to allow compositional development of the “glue code” for connecting the components that comprise a dynamically evolving software system.

Rew provides a paradigm for composition of software components based on the notion of mobile channels. Both components and channels are mobile in Rew, in the sense that (1) components can move at any time from one location to another, retaining their existing channel links, and (2) the same channels can be disconnected and reconnected to other components, thus, dynamically changing the topology of inter-component communication.

The component composition paradigm of Rew is in the style of the IWIM coordination model, and is an extension of our earlier work on a formal-logic-based component interface description language to convey the observable semantics of components. The main focus of attention in Rew is the channels and operations on them, not the processes that operate on them or the components that they are connected to. The composition operations in Rew combine various channel types to produce complex dynamic topologies of “connectors” to which processes or components can be attached.
INDALOG: A Declarative Deductive Database Language

Jesús M. Almendros-Jiménez* and Antonio Becerra-Terón
Dpto. de Lenguajes y Computación, Universidad de Almería.
email: {jalmen, abecerra}@ual.es

Abstract. In this paper we present the main features of a deductive database language named INDALOG based on the integration of functional and logic paradigms. As most deductive database systems, INDALOG allows the handling of negation, higher-order functions, grouping operators, support for aggregation, handling of non-ground facts, and support for indexing structures on both extensional and intensional relations of a database. Moreover, we present the semantic foundations of this language.

1 Introduction
Deductive database systems [16] are database management systems whose query language and storage structure are designed around a logic data model. Deductive database systems offer a rich query language which extends the relational model in many directions (for instance, support for non-first normal form and recursive relations) and they are suited for application in which a large number of data must be accessed and complex queries must be supported (see [16] for applications of deductive systems). With respect to the information management, the deductive database systems split their information into two categories:

- **Facts** represented by **literals** consisting of a predicate applied to **terms**, which are constants, variables, or functors applied to terms. For instance, the fact parent(mary,peter) sets that Peter is parent of Mary. This predicate is represented extensionally, that is, all tuples for this predicate are stored in a database relation.

- **Rules**, which are written in Prolog-style notation as follows $p : -q_1, ... , q_n$, where $p$ (the **head**) and $q_i$’s (the **subgoals**) are literals.

The rules are referred as the intensional database (IDB) and the facts as extensional database (EDB). The intensional database plays a role similar to views in conventional database systems, although there will be large numbers of intensional predicates w.r.t. the numbers of views defined in typical database applications. A typical example about the expressivity power of deductive databases is as follows:

**Extensional Database**

\[
\begin{align*}
\text{parent}\text{(john, mary)} & ; \\
\text{parent}\text{(mary, thomas)} & ; \\
\end{align*}
\]

**Intensional Database**

\[
\begin{align*}
\text{anc(X,Y)} : & -\text{parent}(X,Y) ; \\
\text{anc}(X,Y) : & -\text{parent}(X,Z), \text{anc}(Z,Y) ; \\
\end{align*}
\]

\[
\begin{align*}
\text{Query} & : -\text{anc}(\text{john,Y}) ;
\end{align*}
\]

* The author has been partially supported by the Spanish CICYT (project TIC 98-0445-C03-02 TREND)
Here, there exist two relations, one extensional relation defined by four rules and called parent, and another intensional one, named anc and defined by a recursive rule. The proposed query will compute all John's ancestors, that is $Y = m\text{ary}$, $Y = f\text{rank}$, $Y = t\text{homas}$, and $Y = m\text{iichael}$.

The relations of a deductive database can include functors to store the so-called complex terms. In a logic programming context, the complex terms are called functions because of their syntactic appearance, consisting of a function name [named the functor] followed by a list of arguments. In a deductive language, these complex terms are not used to represent functions; rather, they are used as variable-length subrecords. For instance, suppose the following complex term address(Name, City) which can be used in the base relation person(DNI, address (Name, City), Job) in order to structure the stored information.

Most deductive database systems (for instance, DATALOG [20], CORAL [14], ADITI [21], LOLA [23]) allow to handle negative literals, increasing their expressive power as query languages. The introduction of negation in logic programming (see [4] for a survey), and thus the study of semantic models for logic programs have been widely studied in the past being most relevant one the well-founded semantics [22]. Some deductive database systems have adopted such semantics and operational models [17, 9] based on this semantics. For instance, in the deductive system DATALOG, we can write the following database:

**Extensional Database**

- student(peter, senior).
- student(jim, junior).
- student(mary, junior).
- took(peter, cs1, 3.0).
- took(jim, cs2, 3.3).
- took(jim, cs3, 3.0).
- took(mary, cs1, 2.0).

**Intensional Database**

- lacks cs3(Name) :- student(Name, junior), -hastaken(Name, cs3).

**Query**

- lacks cs3(John).

Here, the extensional relations student and took set the name and year of a student as well as the courses taken by the student with the corresponding grade. The intensional rules are meant to compute junior students who did not take course cs3. The computed answer will be $N = m\text{ary}$.

In addition, most deductive database systems allow grouping and aggregation operators to be used with groups of tuples. The idea of grouping operator is to collect the set (multiset) of values for a given attribute and, in the case of aggregation operators, to manage the collected values. For instance, the deductive database system CORAL [14, 13] allows to collect values in the form of sets and multisets by means of set {()} and multiset {<>()} grouping constructors. The following rule illustrates the use of the multiset construct:

\[
\text{set_of_grades(Class, < Grade >)} : = \text{student(Name, Class)},
\]
\[
\text{took(Name, Course, Grade)}.
\]

Wherein for each Class value, a multiset containing all corresponding values for Grade is created. Similarly with the set-grouping construct. The use of these grouping primitives can be combined with aggregation operators like count, min, max, sum, product, average, any and so on. In CORAL we can write the following rule for computing the average grade:

\[
\text{max_grade(Class, average(< Grade >))} : = \text{student(Name, Class)},
\]
\[
\text{took(Name, Course, Grade)}.
\]
Some of the deductive database systems (for instance, XSB [18]) enrich their expressivity power by adding higher-order features. The deductive database system XSB adopts HiLog [6], a foundation for higher order logic programming, by providing an elegant way to construct and manipulate sets. For instance, the following XSB database:

**Extensible Database**

- `package1(health_ins, required)`
- `package1(life_ins, optional)`
- `package2(health_ins, required)`
- `package2(life_ins, required)`
- `package2(free_car, optional)`

**Query**

\[ \text{benefits(john, P), P(X, Y), benefits(bob, package2).} \]

defines the extensional relations `package1` and `package2` used to denote the set of john’s and bob’s benefits, respectively. Benefits are a set of facts indicating the type of benefit and whether it is optional or required. The query binds `P` to the name of the set of john’s benefits (\(P = \text{package1}\)), and then retrieves the facts that describe his benefits explicitly, that is \(X = \text{health_ins} Y = \text{required}\) and \(X = \text{life_ins} Y = \text{optional}\). Also, this representation can be extended to include set operations. For instance, the following rule computes the intersection of two sets and the query allows to obtain both john’s and bob’s benefits.

**Intensional Database**

\[ \text{intersect(S1, S2)(X, Y) : S1(X, Y), S2(X, Y).} \]

**Query**

\[ \text{benefits(john, P1), benefits(bob, P2), intersect(P1, P2)(X, Y).} \]

Following with the features of deductive database systems, there exist systems which allow to use non-ground facts. The use of non-ground facts (i.e. facts containing universally quantified variables) is useful in the deductive database context. The deductive system CORAL [19] supports efficiently the handling of non-ground facts. For instance, suppose a database about employee salary:

**Extensible Database**

\[ \text{base_salary(john, 1500), base_salary(marc, 2500).} \]
\[ \text{base_salary(peter, 3500), complement_salary(x, 1000).} \]

wherethere exists one non-ground fact, `complement_salary(X, 1000)`, expressing that all employees will have a salary complement (1000 euros), and thus avoiding the specification of one fact for each employee. Now, we can compute the total salary by means of the following rule:

**Intensional Database**

\[ \text{total_salary(X, Y) : base_salary(X, T), complement_salary(X, Z), Y = T + Z.} \]

With respect to the implementation, the deductive database systems provide a representation of indices. Most deductive database systems use the typical indexing structures used in database applications, like hash and B-trees indices on the extensional and intensional database relations. By means of adding indices, some database operations, like `join`, can be efficiently made for disk-resident data. For instance, XSB [18] supports different kinds of indexing. The default is hashing on the first argument of a relation. However, declarations can be used to indicate the desired options. Indices can be constructed on any attribute or set of attributes (multi-attribute indexing). For example, a predicate `p` with arity 5 could have the following index declaration:

\[ \text{index(p/5, [1, 2, 3 + 5]).} \]

which will cause indices on `p/5` in such a way that a retrieval will use the index
on the first argument, if *ground*, otherwise on the second, if *ground*, and otherwise on the third and fifth combined.

Other example of indexing is the deductive language CORAL [15] which supports hash-based indices for memory relations and B-trees indices for disk relations. It allows two types of hash-based indices, (1) *argument indices* with the traditional multi-attribute hash index and (2) *pattern indices* which is an index structure more complicated since that such indices are used with complex terms and non-ground facts. For instance, given a relation *employee* with two arguments, *Name* and a complex term *address*(Street, City) in CORAL, then the following declaration creates a pattern index to retrieve employees without knowing their street.

```
@make_index_employee(Name, address(Street, City)|(Name, City)).
```

Here, we are interested in the study of a deductive database system based on the integration of the functional and logic paradigms. The integration of functional and logic programming has been widely investigated during the last years. It has led to the design of modern programming languages such as *CURRY* [8] and *TOY* [10]. The aim of such integration is to include features from functional (cfr. determinism, higher order functions, partial, non-strict and lazy functions, possibly infinite data structures) and logic (cfr. logic variables, function inversion, non-determinism, built-in search) languages.

Our idea is to develop a deductive database query language, named *IN/DALOG*, which includes these advantages and also allows the efficient management of data like in deductive databases based on logic programming.

The main difference with respect to a “pure” functional-logic language, such as *CURRY* or *TOY*, is that *IN/DALOG* is thought for working with a large volume of data which usually will be stored in secondary memory. Like Prolog, functional-logic languages work efficiently with main-memory resident data, but inefficiently with respect to the disk accesses when data are stored in secondary memory. The main reason for this drawback is that the operational mechanism of such languages works with a tuple (fact) at time, whereas the deductive database systems allow a set of tuples at time.

The basic idea of the operational mechanism of deductive database languages is to use a *bottom-up* evaluation for query solving. This bottom-up mechanism is based on the application of the *immediate consequence operator* defined for logic programs which allows to compute the *Herbrand model* of a logic program. The use of this operator for query solving is very inefficient and thus a program transformation techniques, called Magic Sets-based transformations [5, 12], have been studied. The aim of this process is to transform the original program w.r.t. the proposed query in such a way that the application of the immediate consequence operator on the transformed program and query is *goal-oriented* and can be used for query solving. Moreover, this evaluation mechanism is more set-of-tuples-oriented than top-down evaluation mechanism supported by logic languages and therefore adequate for indexing of tuples. In our case, we have proposed a bottom-up evaluation mechanism for functional-logic programming in [1], which has been extended in [2] for the handling of negation in the line of [11], to be considered as the computational model for *IN/DALOG*.

The aim of this paper is to present the main features already developed and to be developed in our deductive database language *IN/DALOG*. As most deductive database systems, this language will allow the handling of negation, higher-order
functions, grouping operators, support for aggregation, handling of non-ground facts, and support for indexing structures on both extensional and intensional relations of a database. Moreover, we will present the semantic foundations of INDALOG.

2 INDALOG Features

In this section, we present the main features in our deductive language. An INDALOG deductive database \( DS \) will be constituted by the following set of modules:

- **type definition** which includes the definition of type symbols.
- **base relations** which includes the definition of each extensional relation of the database. Also called **base facts**, typically they will be indexed and stored in secondary memory (see subsection 2.4 for more details).
- **derived relations** which includes the definition of each intensional relation of the database. Facts obtained from derived relations will be also indexed and stored at run-time in secondary memory.
- **function definition** which is a set of manipulating functions to be used in an INDALOG database which they are neither base nor derived relations. Typically, they represent the functional part of the language which manages data types such as list, trees, etc. It is assumed that indices are not used for the functional part.
- **query definition** which is a set of conditions to be solved.

\[
\begin{array}{l}
DS \coloneqq \text{Types} \mid \text{Rules} \mid \text{Query}.\\
\text{Types} \coloneqq \text{Type definition}, \ldots, \text{Type definition}, \text{endType}.\\
\text{Type definition} \coloneqq \text{Data declaration} \mid \text{Extension declaration} \mid \text{Intension declaration} \mid \text{Function declaration}.\\
\text{Data declaration} \coloneqq \text{Data constrID Type variID} \mid \ldots \mid \text{Type variID} \mid \text{Const constrID} \mid \ldots \mid \text{Const constrID}.\\
\text{Const constrID} \coloneqq \text{Data constrID Simple type expr} \mid \ldots \mid \text{Simple type expr}.\\
\text{Simple type expr} \coloneqq \text{Type variID} \mid \text{Simple type argument}.\\
\text{Simple type argument} \coloneqq \text{Simple constrID Simple type expr} \mid \ldots \mid \text{Simple type expr}.\\
\text{Extension declaration} \coloneqq \text{Ext relationID} :: \text{Type expr}.\\
\text{Intension declaration} \coloneqq \text{Int relationID} :: \text{Type expr}.\\
\text{Function declaration} \coloneqq \text{Function ID} :: \text{Type expr}.\\
\text{Type expr} \coloneqq \text{Type variID} \mid \text{Type argument}.\\
\text{Type argument} \coloneqq \text{Type constrID Type expr} \mid \ldots \mid \text{Type expr} \mid \{\text{Type expr} \rightarrow \text{Type expr}\} \mid \{\text{Type expr} \rightarrow \text{Type expr}\}.
\end{array}
\]

In the previous table, we have shown the type syntax which is similar to other languages, such as **CURRY** and **TOY**. In the type declaration, two different notations are used for functional types: \( a \rightarrow b \) and \( a \Rightarrow b \). The first case is the usual functional type, and the second one is a special functional type denoting that this argument is flexible, which means that the actual parameters can be only rigid expressions (i.e. left-most symbol cannot be a variable). Our idea is to avoid the extensively search of functional values allowed in some higher order languages, and thus the programmer can note his wishes about the using mode of this functions.

The syntax of the rest of modules is shown in the following table. The base and derived relations can use complex terms in its definition in the form of **patterns**. A pattern is an expression without total applications of extensional, intensional and functional symbols. Type constrID, Type variID, Data constrID, Data variID, Ext relationID, Int relationID and Function ID are identifiers (starting with upper letters in the case of variables), and the symbols in bold-font constitute the set of the **keywords**.
Next, we show the typical example of the “ancestor” and, as you can see, 
\(I^N\) has a functional syntax and thus base and derived relations can be 
represented by means of functions. This functions can have conditions in the form 
of \emph{joinability} equalities \cite{7} whose semantics is to represent the same 
constructor term. Moreover, the joinability equality can be also used in the queries like in this example.

type
\begin{verbatim}
type person = frank | john | mary | michael | thomas

endtype.

code

base_relations
\begin{verbatim}
parent :: person \to person.

endbase_relations.

derived_relations
\begin{verbatim}
apc :: person \to person.

endderived_relations.

query
\begin{verbatim}
apc

endquery.
\end{verbatim}
\end{verbatim}

In the rest of sections, we will explain more in detail the \(I^N\) features, 
but now we want to explain briefly why three kind of declarations are considered: 
base, derived relations and functions. It is assumed that for a given query, 
the language will generate, at compile-time and automatically, indices in order 
to obtain a more efficient evaluation. These indices will be only generated for 
the base and derived relations. The base relations will be stored at compile-time 
and the derived relations at run-time, both in secondary memory. The indexing 
criteria will be based on the query and the form of the rules to be used in the 
query solving. In the previous example, we requested the john’s ancestors, 
and thus the relation \(\text{apc}\) will be indexed by its unique argument in order to improve 
the retrieval of such values. Moreover, the function \(\text{apc}\) uses the function \(\text{parent}\) 
in its definition and thus \(\text{parent}\) will be also indexed by its (only) argument. 
Intuitively, for each call to the function \(\text{apc}\), then the corresponding value for 
\(\text{parent}\) and the ancestors of the value obtained from \(\text{parent}\) must be accessed.

2.1 Handling of Negation in \(I^N\)

The incorporation of negation supposes to study its semantics foundations. In 
\cite{11}, a framework called Constructor Based ReWriting Logic with Failure (\textit{CRWLF}) has been presented, extending the \textit{CRWL} semantics \cite{7}, and allowing to 
handle negative information in functional-logic programming. In this framework, 
the negation is intended as ‘finite failure’ of reduction. \textit{CRWLF} provides four kinds of 
operators: (a) \(\bowtie\) (joinability constraint) and (a’) \(\not\bowtie\) (its logical negation), and 
(b) \(\Leftrightarrow\) (divergence constraint) and (b’) \(\nleftrightarrow\) (its logical negation). As semantics,
we adopt CRWLF and thus these constraints can be included in any database. An operational semantics based on bottom-up evaluation extended for the handling of negation has been studied in [2], according to [11].

Next, we show an example of a database which includes researching heads, boss, and departments, dept, as an extensional database, and a recursive rule superboss as an intensional one which allows to compute each hierarchy line for a given researcher.

type
  person := john | mary | peter | thomas.
  depart := cs | elect.
  boss :: person -> person.
  dept := person -> depart.
  superboss :: person -> person.
endtype.

base_relations
  boss john := mary.
  boss mary := peter.
  dept john := cs.
  dept mary := elect.
end_base_relations.

derived_relations
  superboss P := boss P.
end_derived_relations.

query
  superboss X [\| X := John, dept X := cs].
end_query.

In the query, the obtained values for superboss are compared with john and one value X=mary (not belonging to the department cs) satisfying the query is found.

2.2 Higher Order

Other feature of the language INDALOG is the use of higher-order. Higher order programming has been widely studied in the functional logic paradigm offering a rich expressivity power by allowing, among others, more abstraction and reuse in the code. The higher-order example for XSB presented in the introduction can be also written in our language, given that we can use higher-order patterns in the head and body of both intensional and extensional part:

type
  benefit_name := free | health_ins | life_ins | long_a_cations.
  benefit_type := optional | required.
  person := bob | john.
  package1 :: benefit_name -> benefit_type.
  package2 :: benefit_name -> benefit_type.
  benefits :: person -> (benefit_name -> benefit_type).
endtype.

base_relations
  package1 health_ins := required.
  package1 life_ins := optional.
  package2 health_ins := required.
  package2 free | optional.
  package2 long_a_cations := optional.
  benefits john := package1.
  benefits bob := package2.
end_base_relations.

query
  benefits john := F, F X := Y.
end_query.

The query will obtain the following answers F = package1, X = health_ins, Y = required and F = package1, X = life_ins, Y = optional.

2.3 Grouping Primitives

Our idea is to include primitives in INDALOG for collecting answers for a given query which can be either a set or a multiset. For this reason we consider two kinds of primitives: one considering the answers as a set and the other one as a multiset. The set primitives are set and set_at_least? and the syntax is as follows:

set n var query    set_at_least? n var query

wherein n is a natural number, var is a variable, and query is a set of conditions. The primitive set is a function which obtains the n-th answer (as set) for the
variable var of query. There is a restriction: the variable specified by the argument var can only appear in query. Otherwise a renaming of variables will be accomplished. Moreover, this argument cannot be instantiated, only indicates the variable for which values must be computed. This primitive is similar to most Prolog system's primitive setof(var,goal,list), but with a relevant difference. In order to leave to the programmer the decision about what kind of data structure is used to store the answers of the query (list, tree, etc), the \texttt{INTER} primitive set returns the solutions once at time. By iterating the counter \( n \) of type \texttt{nat}, we can obtain the (partial or complete) set of solutions for the query to be stored in the wished structure. This structure can represent a possibly infinite data in the case of infinite solutions, which can nicely be managed in functional logic programming.

For instance, w.r.t. the ancestor database, the following primitive call \texttt{set 2 X (anc john \&\& X)} will compute the second answer for the variable \( X \) in the query \( \text{anc john} \Rightarrow \text{X} \). In this case, the answer will be thomas or michael depending on the order which the answer is computed in. In addition, we will need other set primitive, named \texttt{set_at_least?}, which is a boolean function and returns \texttt{true} whenever there exist at least \( n \) solutions for \( \text{var} \) in the query. For instance, the primitive call \texttt{set at least? 2 X (anc john \&\& X)} will compute \texttt{true}.

From a theoretical point of view (implementation details apart), \texttt{set at least?} will return \texttt{true} whether there exists the \( n \)-th answer, or fail whenever there not exist more than \( n \) answers, and \texttt{set} will return the \( n \)-th answer or will be undefined, respectively. In order to know if a query has no more than \( n \) answers, we have to proceed as follows: we have to find all the solutions of the query (which must be less than \( n \)) and to refute the query for the rest of values. A query is refuted whether the complementarity of any constraint can be proved. For instance, \texttt{set 4 (anc john \&\& X)} fails because \texttt{anc john \&\& mary, anc john \&\& peter, anc john \&\& thomas, anc john \&\& john}. In some cases, there can be neither proved nor refuted the satisfiability of the query for some value, and thus \texttt{set} and \texttt{set at least?} can remain undefined. In the case of removing \texttt{anc X := parent X} of the ancestor example, neither \texttt{anc john \&\& mary} nor \texttt{anc john \&\& mary} can be proved and the same for the rest of values, due to \texttt{anc} is undefined, and thus \texttt{set 1 (anc john \&\& X)}, \texttt{set at least? 1 (anc john \&\& X)} are also undefined.

As a nice example of its use, in the above ancestor database is the definition of the function ancestors which allows to collect all ancestors of a person in a possibly infinite list:

\begin{verbatim}
  type
    list(A) := [] | [A] | list(A).
  collect_list : person -> set -> list(person).
  ancestors : person -> list(person).
  
  endtype.

  derived relations
  collect_list : person -> set -> list(person)
  ancestors : person -> list(person).

  endderived relations.

  query
    ancestors john \&\& L.

  endquery.

  The control established by the primitive \texttt{set at least?} in the definition rule of \texttt{collect_list} will allow us to control the number of solutions of the query and therefore the end of the list to be build. In such a way that all computed answers are collected in the list and the query will obtain all john's ancestors, that is \( L = \{\text{mary, frank, thomas, michael}\} \) (or even \( L = \{\text{frank, mary, michael, thomas}\} \).
With respect to the multiset primitives, we will include two primitives, called
bag and bag_at_least?, with the syntax: bag n var query and bag_at_least? n var query wherein n is natural number, var is a variable, and query is a set of
conditions. The meaning of these primitives is similar to the primitives set and
set_at_least? but considering the solutions as a multiset.

Finally, we will show the use of these four primitives together with the higher-
order features to generate aggregation operations. For instance, suppose the follow-
ing IN_DALOG database:

type
person :: frank | john | marya | michael | thomas.
list(A) :: [ | | A | list(A)].
parent :: person -> person.
salary :: person -> nat.
asc :: person -> person.
collect_salary_list :: person -> nat -> list(nat).
salary_sum :: person -> nat.
fold :: [A -> B -> A] -> A -> list(B) -> A.
sum_list :: list(nat) -> nat.

endtype.
base_relations
parent john := marya. parent marya := thomas.
person john := frank. person frank := michael.
salary john := 2500. salary marya := 3000.
salary thomas := 1200. salary franks := 2500.

endbase_relations.
derived_relations
anc X := parent X. anc X := anc(parent X).
collect_salary_list X Y :=
   bag [ Y | salary (anc X) = Y | collect_salary_list (X + 1)]
   bag_at_least? Y [ Y (salary (anc X) = Y) := true.
collect_salary_list X Y :=
   bag [ Y | salary (anc X) = Y] := true.
salary_sum Y := sum_list [collect_salary_list Y].

endderived_relations.
functions
fold F X Z := Z. fold F X [E] := fold F (F X) E.
sum_list L := fold X 0 L.

endfunctions.
query
salary_sum john := Y.

endquery.

This example allows us to compute the sum of all salaries of john’s ancestors.
Firstly, we define, in a recursive way, a collector, named collect_salary_list Z N,
which collects all salaries in a list. Secondly, we use a manipulating function,
named sum_list, defined by using the higher-order function fold. Finally, in
order to obtain the accumulated sum, we define the function salary_sum Y by
applying sum_list to the list obtained from collect_salary_list Y 1. The pro-
posed query salary_sum john := Y will obtain as answer Y = 7900 euros.

2.4 Indexing

With respect to the indexing process, our idea is, given an IN_DALOG database,
to generate automatically the set of indices necessary for the query solving. These
indices will be created on base and derived relations of the database taking into
account the query. With this aim, the indexing process will detect the set of ‘join’
operations necessary for the query solving. Next, consider a database as follows:

type
person :: john | peter | rose.
job_name :: computer | ten_cser.
age :: nat -> person -> nat.
job :: nat -> job_name.
eq 32 := nat = person = job_name.

endtype.
base_relations
age 22 peter := 25. job 10 := computer.

endbase_relations.
Taking into account this query, the indexing process will start analyzing the definition rules for the function eq.23, and it will detect a 'join' operation in the conditions due to the variable ID shared by the functions age and job. Therefore, the indexing process will generate the following compilation directives 
\@index\_age\(\text{ID, Name, Result}\)(\text{ID}) and \@index\_job\(\text{ID, Result}\)(\text{ID}) where Result denotes the result of the relation. Here, two indices have been created, one for the function age indexing by the argument ID and other one for the function job with the argument ID.

Now, suppose the query \(\text{eq.23 } X \text{ john } \bowtie \text{ Z}\) wherein the second argument of the function eq.23 is instantiated by the value \(\text{john}\). In this case, the indexing process will generate the compilation directives \@index\_age\(\text{ID, Name, Result}\)(\text{Name, ID}) and \@index\_job\(\text{ID, Result}\)(\text{ID}). A multi-attribute (i.e. Name and ID) index is generated for the function age due to the instantiated variable in the query, and therefore also instantiated in the function age, as well as the 'join' operation occurring in the conditions of the rule for the function eq.23. The index for the function job is the same than the previous one.

By considering the query \(\text{eq.23 } X \text{ Y } \bowtie \text{ teacher}\), the instantiated value is referred to the result from the function eq.23. Then, there exist two 'join' operations: (1) variable Job shared by the functions eq.23 and job and (2) variable ID shared by the function age and job. Therefore, two indices will be generated by means of the directives \@index\_age\(\text{ID, Name, Result}\)(\text{ID}) and \@index\_job\(\text{ID, Result}\)(\text{ID}). Here a multi-attribute index (primary key is Result and secondary one is ID) will be generated for the function job in order to retrieve each ID for every teacher.

Finally and with respect to the complex terms, the indexing process will allow to generate indices on the arguments of a complex term but never on non basic types. For instance, given the complex term address in the following function person\(\text{(ID, address\{Name, City, Age\})}\), then the indexing process could generate compilation directives for indices on Name or City, or create a multi-attribute index on both Name and City.

To put an end, remark that in order to implement the indexing process, we will use the typical indexing structures implemented in the database context, that is hash index for main memory relations and B-trees for relations stored in secondary memory.

3 Foundations of INDALOG

In this section we will present the semantic foundations of our language. We have adopted the Constructor-based Rewriting Logic with Failure (CRWLF) presented in [11] as semantic framework of INDALOG given that the cited semantics allows to provide meaning to functional logic programming with negative constraints. CRWLF extends the Constructor Based Rewriting Logic presented in [7], wherein the negation is treated as 'finite failure' of reduction. The conditions that are provable in CRWL can also be proved in CRWLF but, in addition, CRWLF provides 'proofs of unprovability' within CRWL. In general, the unprovability is not computable which means that CRWLF can only give an approximation to failure in CRWL that corresponds to the cases in which unprovability refers to 'finite
failure' of reduction. Here, we are interested in the presentation of an alternative characterization of the CRWLF semantics (and therefore equivalent) by using CRWLF Herbrand algebras and models. From now on we restrict our presentation to the first-order fragment of the language.

3.1 Basis
We assume a signature $\Sigma = DC \cup DS$ where $DC = \bigcup_{c \in C} DC^n$ is a set of constructor symbols $c, d, \ldots$ and $DS = \bigcup_{d \in D} DS^n$ is a set of defined symbols $f, g, \ldots$, all of them with associated arity and such that $DC \cap DS = \emptyset$. Defined symbols consist of three sets $ES$ of extensional symbols, $IS$ of intensional symbols and $FS$ of function symbols: $DS = ES \cup IS \cup FS$. We also assume a countable set $\mathcal{V}$ of variable symbols $X, Y, \ldots$. We write $\text{Term}$ for the set of (total) terms $e, e'$ (also called expressions) built up with $\Sigma$ and $\mathcal{V}$ in the usual way, and we distinguish the subset $\text{CTerm}$ of (total) constructor terms or (total) c-terms $t, s, \ldots$, built up only with symbols of $DC$ and $\mathcal{V}$. Terms intend to represent possibly reducible expressions, whereas c-terms represent data values, not further reducible. Terms correspond with first-order expressions and CTerms with first-order patterns in the first-order fragment of the language. We extend the signature $\Sigma$ by adding two new constants: the constant $\bot$ that plays the role of undefined value and the new constant symbol $\top$ that will be used as an explicit representation of failure of reduction. The set $\text{Term}_\bot$ of partial terms and the set $\text{CTerm}_\bot$ of partial c-terms are defined in a natural way. Partial c-terms represent the result of partially evaluated expressions, and thus they can be considered as approximations to the value of expressions. Moreover, we will consider the corresponding sets $\text{Term}_\bot \top$ and $\text{CTerm}_\bot \top$. A natural approximation ordering $\leq$ over $\text{CTerm}_\bot \top$ can be defined as the least partial ordering satisfying: $\bot \leq t, X \leq X$ and $h(t_1, \ldots, t_n) \leq h(t'_1, \ldots, t'_n)$, if $t_i \leq t'_i$ for all $i \in \{1, \ldots, n\}$, $h \in DC \cup DS$. The intended meaning of $t \leq t'$ is that $t$ is less defined or has less information than $t'$. Note that the only relations satisfied by $\top$ are $\bot \leq \top$ and $t \leq \top$. In particular, $\top$ is maximal. This is reasonable, since $\top$ represents 'failure of reduction' and this gives a no further refinable information about the result of the evaluation of an expression.

3.2 INDIALOG Semantics
A conditional rewrite rule for a defined symbol $f \in DS^n$ is of the form:

$$f(t_1, \ldots, t_n) := \begin{cases} r & \text{condition} \\ \text{body} & \text{condition} \end{cases}$$

where $(t_1, \ldots, t_n)$ is a linear tuple (each variable in it occurs only once) with $t_1, \ldots, t_n \in \text{CTerm}; C$ is a set of constraints of the form $e \not\rightarrow e'$ (joinability), $e' \not\rightarrow e''$ (divergence), $e' \not\Rightarrow e''$ (failure of joinability) or $e' \not\Rightarrow e''$ (failure of divergence); $r, e$ and $e'$ belong to the set $\mathcal{T}$ which consists of terms built from $\Sigma, \mathcal{V}, \text{set}(n, X, C), \text{setAtLeast}(n, X, C), \text{bag}(n, X, C)$ and $\text{bagAtLeast}(n, X, C)$ where $n = 0, \text{sort}(X, 0), X \in \mathcal{V}$. The reading of the rule is: $f(t_1, \ldots, t_n)$ reduces to $r$ if the condition $C$ is satisfied.

The meaning of the conditions is as follows: $e \not\rightarrow e'$ (joinability); $e$ and $e'$ can be both reduced to some $t \in \text{CTerm}_\bot \top$ $e' \not\rightarrow e''$ (divergence); $e$ and $e'$ can be reduced to some (possibly partial) c-terms $t$ and $t'$ having a DC-clash; $e \not\Rightarrow e'$; failure of $e \not\Rightarrow e'$ and $\not\Rightarrow e'$; failure of $e \not\Rightarrow e'$, where given set of constructor symbols $S$. 

INDIALOG: A Declarative Deductive Database Language
we say that the c-terms \( t \) and \( t' \) have a \( S \)-clash if they have different constructor symbols of \( S \) at the same position.

We will use the symbol \( \dagger \) to refer to any of the constraints \( \rhd, \llhd, \rhd', \llhd' \). The constraints \( \rhd' \) and \( \rhd' \) are called the complementary of each other; the same holds for \( \llhd' \) and \( \llhd' \), and we write \( \dagger' \) for the complementary of \( \dagger \). The meaning of the constraint \( c \rhd \ r' \) depends on certain syntactic (hence decidable) relations between the corresponding approximations for \( c \) and \( r' \) and is defined as follows.

**Definition 1 (Relations over \( CTerm_{\perp, \rhd} \) [11]).**

- \( t \downarrow t' \iff \exists_f t = t' \in CTerm \)
- \( t \uparrow t' \iff \exists_f t \text{ or } t' \text{ contain } t \text{ as subterm, or they have a DC-clash} \)
- \( \dagger \) is defined as the least symmetric relation over \( CTerm_{\perp, \rhd} \) satisfying:
  1. \( X \not\ni X \) for all \( X \in \mathcal{V} \)
  2. \( t \not\ni t \), for all \( t \in CTerm_{\perp, \rhd} \)
  3. if \( t_1 \ldots t_n \not\ni t' \), then \( c(t_1, \ldots, t_n) \not\ni c(t'_1, \ldots, t'_n) \), for \( c \in \text{DC} \)

When using rules to derive statements, we will need to use what are called \( c \)-instances of such rules. The set of \( c \)-instances of a rule \( R \) is defined as \( [R]_{\rhd} = \{ R \theta : \theta \in CSubst_{\perp, \rhd} \} \), where we use substitutions \( CSubst_{\perp, \rhd} = \{ \theta : \mathcal{V} \rightarrow CTerm_{\perp, \rhd} \} \).

Now, we present \( CRWLF-Herbrand \) algebras and models. We assume the reader has familiarity with basic concepts of model theory on logic programming and functional-logic programming (see [3, 7] for more details). Now, we point out some of the notions used in this paper.

Given \( S \), a partially ordered set (in short, poset) with bottom \( \bot \) (equipped with a partial order \( \leq \) and a least element \( \bot \)), the set of all totally defined elements of \( S \) will be noted \( \text{Def}(S) \). We write \( C(S), I(S) \) for the sets of cones and ideals of \( S \), respectively. The set \( \hat{S} = \text{Def}(S) \) denotes the ideal completion of \( S \), which is also a poset under the set-inclusion ordering \( \subseteq \) and there is a natural mapping for each \( x \in S \) into the principal ideal generated by \( x \), \( \langle y \in S : y \leq x \rangle \in \hat{S} \). Furthermore, \( \hat{S} \) is a ecp (i.e. every directed set \( D \subseteq \hat{S} \) has a least upper bound) whose finite elements are precisely the principal ideals \( \langle x \rangle \), \( x \in \hat{S} \).

**Definition 2 (Herbrand Algebras).** For any given signature \( \Sigma \), a Herbrand algebra \( \mathcal{H} \) is an algebraic structure of the form \( \mathcal{H} = (CTerm_{\perp, \rhd}, \{ f^\mathcal{H} \}_{f \in D}) \) where \( CTerm_{\perp, \rhd} \) is a poset with the approximation ordering \( \leq \) and \( f^\mathcal{H} \in CTerm_{\perp, \rhd} \) for \( f \in \mathcal{D} \), where \( [D \rightarrow E] = \text{def} \{ f : D \rightarrow C(E) : \forall u, u' \in D : (u \leq u' \Rightarrow f(u) \subseteq f(u')) \} \). From the set \( \{ f^\mathcal{H} \}_{f \in \mathcal{D}} \), we can distinguish the deterministic functions \( f \in \mathcal{D} \), holding that \( f^\mathcal{H} \in CTerm_{\perp, \rhd} \) for \( \lambda \in \mathcal{D} \) \( CTerm_{\perp, \rhd} \) when \( [D \rightarrow E] = \text{def} \{ f \in [D \rightarrow E] : \forall u \in D : f(u) \in I(E) \} \). The elements of \( \text{Def}(\mathcal{H}) \) are the elements of \( CTerm_{\perp} \).

Given a Herbrand algebra \( \mathcal{H} \), a valuation over \( \mathcal{H} \) is any mapping \( \eta : \mathcal{V} \rightarrow CTerm_{\perp, \rhd} \), and we say that \( \eta \) is totally defined iff \( \eta(X) \in \text{Def}(\mathcal{H}) \) for all \( X \in \mathcal{V} \). We denote by \( \text{Val}(\mathcal{H}) \) the set of all valuations, and by \( \text{DefVal}(\mathcal{H}) \) the set of all totally defined valuations.

**Definition 3 (Satisficability).** Let \( \mathcal{H} \) be a Herbrand algebra, we say that:

1. \( \mathcal{H} \) satisfies a joinability \( e \rhd \ r' \) under a valuation \( \eta : \mathcal{V} \rightarrow CTerm_{\perp, \rhd} \) and \( \eta \in \text{Val}(\mathcal{H}) \) if and only if there exist \( t \in [e]_{\rhd} \cap CTerm_{\perp, \rhd} \) and \( t' \in [r']_{\rhd} \cap CTerm_{\perp, \rhd} \) such that \( t \downarrow t' \).
2. \( H \) satisfies a divergence \( e \vartriangleleft e' \) under a valuation \( \eta \) (in symbols, \( H, \eta \models e \vartriangleleft e' \)) iff there exist \( t \in \| t \|_\eta \cap \text{Tpm}_\perp \) and \( e' \in \| e' \|_\eta \cap \text{Tpm}_\perp \) such that \( t \uparrow e' \).

3. \( H \) satisfies a failure of joinability \( e \not\triangleright e' \) under a valuation \( \eta \) (in symbols, \( H, \eta \models e \not\triangleright e' \)) iff for every \( t \in \| t \|_\eta \cap \text{Tpm}_\perp \) and \( e' \in \| e' \|_\eta \cap \text{Tpm}_\perp \), then \( t \not\uparrow e' \) holds.

4. \( H \) satisfies a failure of divergence \( e \not\triangleright e' \) under a valuation \( \eta \) (in symbols, \( H, \eta \models e \not\triangleright e' \)) iff for every \( t \in \| t \|_\eta \cap \text{Tpm}_\perp \) and \( e' \in \| e' \|_\eta \cap \text{Tpm}_\perp \), then \( t \not\uparrow e' \) holds.

5. \( H \) satisfies a condition \( C \) under a valuation \( \eta \) (in symbols, \( H, \eta \models C \)) if \( (H, \eta) \models e \hat{\otimes} e' \) for every \( e \hat{\otimes} e' \in C \).

6. \( H \) satisfies the failure of a condition \( C \) under a valuation \( \eta \) (in symbols, \( H, \eta \not\models C \)) if \((H, \eta) \models e \hat{\otimes} e' \) for every \( e \hat{\otimes} e' \in C \).

Satisfiability of constraints (cases from (1) to (4)) is expressed by means of the relations \( \text{Tpm}_\perp \) terms defined in definition 1. Cases (5) and (6) express satisfiability of a condition \( C \) and the failure of a condition \( C \), respectively, which take into account that \( -C \equiv e_1 \hat{\otimes} e_1' \lor \ldots \lor e_n \hat{\otimes} e_n' \), whenever \( C \equiv e_1 \hat{\otimes} e_1' \land \ldots \land e_n \hat{\otimes} e_n' \).

**Definition 4 (Herbrand Denotation).** The evaluation of an \( e \in \text{Tpm}_\perp \) in \( H \) under \( \eta \) yields \( \| e \|_\eta \in C(\text{Tpm}_\perp) \) which is defined recursively as follows:

1. \( \| t \|_\eta = \text{def} < \perp > \), \( \| \text{id} \|_\eta = \text{def} \eta(t) > \) and \( \| X \|_\eta = \text{def} \eta(X) > \), for \( X \in \mathcal{V} \).
2. \( \| e_1, \ldots, e_n \|_\eta = \text{def} \eta(e_1) < \ldots < \eta(e_n) > \) for all \( e \in \mathcal{D}^\mathcal{N} \).
3. \( \| f(e_1, \ldots, e_n) \|_\eta = \text{def} \eta(f)_\eta(e_1, \ldots, e_n) > \) for all \( f \in \mathcal{D}^\mathcal{N} \).
4. \( \| \text{set}(n, X, C) \|_\eta = \text{def} \eta(X) > \) if \( \text{card}(\text{Ans}) \geq [n] \), otherwise \( \text{set}(n, X, C) \|_\eta = \text{def} \eta(X) < \) if \( \text{card}(\text{Ans}) < [n] \), otherwise \( \| \text{bag}(n, X, C) \|_\eta = \text{def} \eta(X) > \) if \( \text{card}(\text{Ans}) \geq [n] \), otherwise \( \| \text{bag}(n, X, C) \|_\eta = \text{def} \eta(X) < \) if \( \text{card}(\text{Ans}) < [n] \), otherwise \( \| \text{set}_{\text{at} \text{least}}(n, X, C) \|_\eta = \text{def} \eta(X) > \) if \( \text{card}(\text{Ans}) \geq [n] \) and for every valuation \( \theta \) either holds \( (H, \eta \cdot \theta) \models C \) or \((H, \eta \cdot \theta) \not\models -C \), otherwise \( \| \text{set}_{\text{at} \text{least}}(n, X, C) \|_\eta = \text{def} \eta(X) < \) if \( \text{card}(\text{Ans}) < [n] \) and for every valuation \( \theta \) either holds \( (H, \eta \cdot \theta) \models C \) or \((H, \eta \cdot \theta) \not\models -C \), otherwise \( \| \text{bag}_{\text{at} \text{least}}(n, X, C) \|_\eta = \text{def} \eta(X) > \) if \( \text{card}(\text{Ans}) \geq [n] \) and for every valuation \( \theta \) either holds \( (H, \eta \cdot \theta) \models C \) or \((H, \eta \cdot \theta) \not\models -C \), otherwise \( \| \text{bag}_{\text{at} \text{least}}(n, X, C) \|_\eta = \text{def} \eta(X) < \) if \( \text{card}(\text{Ans}) < [n] \) and for every valuation \( \theta \) either holds \( (H, \eta \cdot \theta) \models C \) or \((H, \eta \cdot \theta) \not\models -C \), otherwise \( \{f\} \) denotes a multi set of elements, \( \text{card} \) the cardinal of a set (multi set), \([n]\) the value of \( n \) as natural number.

Due to non-determinism the evaluation of an expression yields a cone rather than an element. It can be proved that given a Herbrand algebra \( H \), for any \( e \in \text{Tpm}_\perp \) and \( \eta \in \text{Val}(H) \), then \( \| e \|_\eta \in I(\text{Tpm}_\perp) \) if \( \text{f}^H \) is deterministic for every defined function symbol \( f \) occurring in \( e \), and \( \| e \|_\eta \in I(\text{Tpm}_\perp) \) if \( e \in \text{Tpm}_\perp \) and \( \eta \in \text{DefVal}(H) \).

Set and bag primitives denote the \( n \)th answer of the condition \( C \) if there exists, otherwise are undefined. \( \text{set}_{\text{at} \text{least}}? \) and \( \text{bag}_{\text{at} \text{least}}? \) denote true whenever there are at least \( n \) answers of the condition \( C \), and failure whether there are less than \( n \) answers which can only be ensured (cases (6) and (7)) whenever every valuation either satisfies the condition or the failure of the condition.
Definition 5 (Poset of Herbrand Algebras). We can define a poset with bottom over the Herbrand algebras as follows: given \( A \) and \( B \), \( A \leq B \) iff \( f^A(t_1, \ldots, t_n) \subseteq f^B(t_1, \ldots, t_n) \) for every \( f \in DS^0 \) and \( t_i \in CTerm_{\leq f}, 1 \leq i \leq n \).

It can be proved that the ideal completion of this poset is a cpo, called \( \mathcal{HAC}_G \), and \( | \cdot | \) is continuous w.r.t. \( \mathcal{HAC}_G \).

Definition 6 (Herbrand Models). Let \( \mathcal{H} \) be a Herbrand algebra:
- \( \mathcal{H} \) satisfies a rule \( f(\overline{t}) := r \Leftarrow C \) if:
  1. every valuation \( \eta \) such that \( \langle \mathcal{H}, \eta \rangle \models C \) verifies \( \models \overline{f}(\overline{t})^{\mathcal{H}} \eta \supseteq \models \overline{f}^{\mathcal{H}} \eta \)
  2. every valuation \( \eta \) such that, for some \( i \in \{1, \ldots, n\} \), \( t_i \) and \( t_i \) have a \( DC \cup \{ f \} \) clash, where \( t_i \in |s_i|^{\mathcal{H}} \eta \), verifies \( \models \overline{f}(\overline{t})^{\mathcal{H}} \eta \),
  3. every valuation \( \eta \) such that \( \langle \mathcal{H}, \eta \rangle \models \neg C \) verifies \( \models \overline{f}(\overline{t})^{\mathcal{H}} \eta \).
- \( \mathcal{H} \) is a model of a set of rules \( R_1, \ldots, R_n \) (in symbols, \( \mathcal{H} \models R_1, \ldots, R_n \)) if \( \mathcal{H} \) satisfies every \( R_i \).

Rules can either provide approximations to the value of a function (case (1)) or fail values (cases (2) and (3)). A rule provides fail values whenever either a unification (case (2)) or a condition (case (3)) failure occurs.

Definition 7 (Fix Point Operator). Given a Herbrand algebra \( A \), and \( f \in DS \), we define the fix point operator as:
\[
T_P(\overline{A}, f)(x_1, \ldots, x_n) =_{def} \{ \models x_i \wedge |f(\overline{t}) := r \Leftarrow C, \eta \in Val(\overline{A}) \text{ such that } s_i \in |f(\overline{t})|^{\overline{A}} \text{ and } \langle \overline{A}, \eta \rangle \models C \} \\
\cup \{ \models \overline{f} \wedge \text{if exists } f(\overline{t}) := r \Leftarrow C, \text{ such that for some } i \in \{1, \ldots, n\}, \langle \overline{A}, \eta \rangle \models \overline{f}(\overline{t}) \text{ have a } DC \cup \{ f \} \text{ clash} \} \\
\cup \{ \models \overline{f} \wedge \text{if exists } f(\overline{t}) := r \Leftarrow C, \text{ such that } s_i \in |f(\overline{t})|^{\overline{A}} \text{ and } \langle \overline{A}, \eta \rangle \models \neg C \} \\
\cup \{ \bot \wedge \text{otherwise} \}
\]

In each step of the fix point operator application a set of approximation values (due to the non-determinism) is computed: including \( \bot \) when the rule cannot be used and \( f \) when unification or condition failures occur.

Given \( \overline{A} \in \mathcal{HAC}_G \), there exists a unique \( \overline{B} \in \mathcal{HAC}_G \) denoted by \( T_P(\overline{A}) \) such that \( f^B(t_1, \ldots, t_n) = T_P(\overline{A}, f)(t_1, \ldots, t_n) \) for every \( f \in DS^0 \) and \( t_i \in CTerm_{\leq f}, 1 \leq i \leq n \). With these definitions, we can ensure the following result which characterizes the least Herbrand model of a set of rules.

Theorem 1. The fix point operator \( T_P \) is continuous and satisfies:
1. For every \( \overline{A} \in \mathcal{HAC}_G : \overline{A} \models R_1, \ldots, R_n \) iff \( T_P(\overline{A}) \leq \overline{A} \).
2. \( T_P \) has a least fix point \( \mathcal{M} = \mathcal{H}^0 \) where \( \mathcal{H}^0 \) is the bottom in \( \mathcal{HAC}_G \) and \( \mathcal{H}^{k+1} = T_P(\mathcal{H}^k) \)
3. \( \mathcal{M} \) is the least Herbrand model of \( R_1, \ldots, R_n \).

4 Conclusions and Future Work

In this paper we have presented the main features of a declarative deductive database language based on the integration of functional and logic paradigms. This language includes most relevant features considered in deductive logic languages. In addition, we have shown that this language extends this class of languages by adding the typical advantages of functional logic languages like laziness and possibly infinite data. Finally, we have shown how to use primitives in this language for collecting answers and how indexing is used for the management of large volume of data. As future work, on one hand, we will go towards the implementation of the language, and on the other hand, to the study of an extension of the relational algebra for expressing the semantics of this language as a database query language.
References

A Declarative Debugger of Wrong Answers for Lazy Functional Logic Programs

Rafael Caballero and Mario Rodríguez-Artalejo *

E-mail: {rafamario}@sip.ucm.es
Departamento de Sistemas Informáticos y Programación,
Universidad Complutense de Madrid

Abstract. We present a declarative debugger for lazy functional logic programs with polymorphic type discipline. Debugging is performed by searching in a computation tree which is a logical representation of the computation. Following a known technique, our tool is based on a program transformation: transformed programs return computation trees along with the results expected by source programs. Our translation is provably correct w.r.t. well-typed and program semantics. As additional improvements w.r.t. related approaches, we solve a previously open problem concerning the use of curried functions, and we provide a correct method for avoiding redundant questions to the user during debugging. A prototype implementation of the debugger is available. Case studies and extensions are planned as future work.

1 Introduction

The impact of declarative languages on practical applications is inhibited by many known factors, including lack of debugging tools, whose construction is recognized as difficult for lazy functional languages. As argued in [20], such debuggers are needed, and much of interest can be still learned from their construction and use. Debugging tools for lazy functional logic languages [6] are even harder to construct.

A promising approach is declarative debugging, which starts from a computation considered incorrect by the user (error symptom) and locates a program fragment responsible for the error. In the case of (constraint) logic programs, error symptoms can be either wrong or missing computed answers [18,8,4,10,19]. Declarative debugging has been also adapted to lazy functional programming [14–16, 11, 13, 17] and combined functional logic programming [12]. All these approaches use a computation tree (CT) [12] as logical representation of the computation. Each node in a CT represents the result of a computation step, which must follow from the results of its children nodes by some logical inference. Diagnosis proceeds by traversing the CT, asking questions to an external oracle (generally the user) until a so-called buggy node [12] is found, whose result is erroneous, but whose children have all correct results. The user does not need to understand the computation operationally. Any buggy node represents an erroneous computation step, and the debugger can display the program fragment responsible for it. From an explanatory point of view, declarative debugging can be described as consisting of two stages, namely CT generation and CT navigation [15].

* Work partially supported by the Spanish CICYT (project CICYT-TIC98-0445-C03-02/97 "TREND")
We present a declarative debugger of wrong answers in lazy functional logic programs with polymorphic type discipline. Following a known idea [15, 13, 17], we use a program transformation for CT generation. We give a careful specification of the transformation, we show its advantages w.r.t. previous related ones, and we describe some new techniques which allow to avoid redundant questions to the oracle during the navigation phase. The debugger has been implemented as part of the TOY system [9]; a prototype version can be downloaded from http://titan.sip.ucm.es/toy/debug.tar.gz. Case studies and extensions of the debugger are planned as future work.

A known extension of declarative debugging is abstract diagnosis [3, 1], leading to equivalent bottom-up and top-down diagnosis methods which do not require error symptoms to be given in advance. In order to be effectively implemented, abstract diagnosis uses abstract interpretation techniques to build a finite abstraction of the intended program semantics. These methods are outside the scope of this paper.

The rest of the paper is organized as follows: Sect. 2 collects some preliminaries, and Sect. 3 summarizes the contributions of our work w.r.t. previous related papers. Our approaches to CT generation and navigation, with detailed explanations of the new contributions, are presented in Sect. 4 and 5, respectively. Conclusions and plans for future work are summarized in Sect. 6. Due to lack of space, proofs have been limited to brief sketches. Detailed proofs will be given in a full version of the paper.

2 Preliminaries

Functional Logic Programming (FLP for short) aims at the integration of the best features of current functional and logic languages; see [5] for a survey. This paper deals with declarative debugging for lazy FLP languages such as Curry or TOY [7, 9], which includes pure LP and lazy FP programs as particular cases. In this section we recall the basic facts about syntax, type discipline and declarative semantics for lazy FLP programs. We follow the formalization given in [5], but we use the concrete syntax of TOY for program examples.

2.1 Types, Expressions and Substitutions

We assume a countable set TVar of type variables α, β, . . . and a countable ranked alphabet TC = \( \bigcup_{n \in \mathbb{N}} TC^n \) of type constructors C. The set Type of valid types \( \tau \in Type \) is built as \( \tau ::= \alpha \quad (\alpha \in TVar) \quad (C \ \tau_1 \ldots \tau_n) \quad (C \in TC^n) \quad (\tau \rightarrow \tau') \). By convention, \( C \ \tau_n \) abbreviates \( (C \ \tau_1 \ldots \tau_n) \), \( \rightarrow \) associates to the right, \( \tau_n \rightarrow \tau \) abbreviates \( \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau \), and the set of type variables occurring in \( \tau \) is written \( tvar(\tau) \). A type \( \tau \) is called monomorphic iff \( tvar(\tau) = \emptyset \), and polymorphic otherwise. A type without any occurrence of "\( \rightarrow \)" is called a datalype.

A polymorphic signature over TC is a triple \( \Sigma = \langle TC, DC, FS \rangle \), where \( DC = \bigcup_{n \in \mathbb{N}} DC^n \) and \( FS = \bigcup_{n \in \mathbb{N}} FS^n \) are ranked sets of data constructors resp. defined function symbols. Each n-ary \( c \in DC^n \) comes with a principal type declaration \( c :: \pi_n \rightarrow C \ \pi_k \), where \( n, k \geq 0, \alpha_1, \ldots, \alpha_k \) are pairwise different, \( \pi_i \) are...
datatypes, and $\text{tvar}(\tau_i) \subseteq \{ a_1, \ldots, a_k \}$ for all $1 \leq i \leq n$ (so-called transparency property). Also, every $n$-ary $f \in \mathcal{FS}^n$ comes with a principal type declaration $f :: \tau_1 \cdots \tau_n \rightarrow \tau$, where $\tau_1, \tau$ are arbitrary types. In practice, each FLP program $P$ has a signature which corresponds to the type declarations occurring in $P$. In the rest of this section we assume some fixed signature $\Sigma$, not made explicit in the notation.

Assuming a countable set $\text{Var}$ of variables, disjoint from $\Sigma$, partial expressions $e \in \mathcal{Exp}_\bot$ have the syntax $e ::= \bot \mid X \mid h[ee']$ where $X \in \text{Var}$, $h \in DC \cup FS$. Expressions of the form $(ee')$ stand for the application of $e$ (acting as a function) to $e'$ (acting as an argument). As usual, we assume that application associates to the left and thus $(e_0 e_1 \ldots e_n)$ abbreviates $(\ldots (e_0 e_1) \ldots e_n)$. The special symbol $\bot$ (read bottom) represents an undefined value. The set of data variables occurring in $e$ is written $\text{var}(e)$. An expression $e$ is called closed iff $\text{var}(e) = \emptyset$, and open otherwise. Moreover, $e$ is called linear iff every $X \in \text{var}(e)$ has one single occurrence in $e$. Partial patterns $t \in \mathcal{Pat}_\bot \subseteq \mathcal{Exp}_\bot$ are built as $t ::= \bot \mid X \mid c t_1 \ldots t_n \mid ft_1 \ldots t_m$ where $c \in DC^*, 0 \leq m \leq n$ and $f \in \mathcal{FS}^n$, $0 \leq m < n$. Expressions and patterns without any occurrence of $\bot$ are called total. We write $\mathcal{Exp}$ and $\mathcal{Pat}$ for the sets of total expressions and patterns, respectively. Actually, the symbol $\bot$ never occurs in a program’s text; but it may occur during debugging, as we will see.

An expression $e \in \mathcal{Exp}_\bot$ is called well-typed iff there exist some type environment $T$ (a set of type assumptions $X :: \tau$ for the variables occurring in $e$) and some type $\tau$, such that the type judgement $T \vdash_{WT} e :: \tau$ can be derived by means of the type inference rules from Milner’s type system. A well-typed expression always admits a so-called principal type (PT) that is more general than any other. A pattern whose PT determines the PTs of its subpatterns is called transparent. See [5] for more details.

Total substitutions are mappings $\theta : \text{Var} \rightarrow \mathcal{Pat}$ with a unique extension $\tilde{\theta} : \mathcal{Exp} \rightarrow \mathcal{Exp}$, which will be noted also as $\theta$. The set of all substitutions is denoted as $\text{Subst}$. The set $\text{Subst}_\bot$ of all the partial substitutions $\theta : \mathcal{V} \rightarrow \mathcal{Pat}_\bot$ is defined analogously. We write $e\theta$ for the result of applying the substitution $\theta$ to the expression $e$. As usual, $\theta = \{ X_1 \mapsto t_1, \ldots, X_n \mapsto t_n \}$ stands for the substitution that satisfies $X_i\theta \equiv t_i$ with $1 \leq i \leq n$ and $Y\theta \equiv Y$ for all $Y \in \mathcal{V} \setminus \{X_1, \ldots, X_n\}$.

Type substitutions, mapping type variables to types, can be defined similarly.

### 2.2 Programs and Goals

A well-typed program $P$ is a set of defining rules for the function symbols in its signature. Defining rules for $f \in \mathcal{FS}^n$ have the form

\[
(R) \quad \frac{t_1 \cdots t_n \rightarrow r}{C \quad \text{LD}}
\]

satisfying the following requirements:

(i) $t_1 \ldots t_n$ is a linear sequence of transparent patterns and $r$ any expression.

(ii) $C$ is a sequence of conditions $e_1 \Rightarrow e'_1, \ldots, e_k \Rightarrow e'_k$, where $e_i, e'_i$ are expressions.
(iii) \(LD\) is a sequence of statements \(s_1 \leftarrow d_1, \ldots, s_m \leftarrow d_m\), where \(d_i\) are expressions and \(s_i\) are transparent linear patterns. This is intended as a local, non-recursive definition of values for new variables occurring in \(\varpi_m\). Therefore, we require that any variable in \(s_i\) must not occur in \(t_1, \ldots, t_n, s_1, \ldots, s_{j-1}, s_{j+1}, \ldots, s_m\), and can occur in \(d_j\) only if \(j > i\).

(iv) There is some type environment \(T\) which well-types the left-hand and right-hand sides of the rule according to the principal type of \(f\). Moreover, \(T\) must well-type each statement occurring in \(C\) or \(LD\), by deriving a common type for its two sides.

Informally, the intended meaning of a rule like \((R)\) is that a call to function \(f\) can be reduced to \(r\) whenever the actual parameters match the patterns \(t_i\), and the conditions and local definitions are satisfied. A condition \(e \equiv e'\) is satisfied by evaluating \(e\) and \(e'\) to some common total pattern. A local definition \(s \leftarrow d\) is satisfied by evaluating \(d\) to some possibly partial pattern which matches \(s\).

In the concrete syntax of \(\mathcal{T}OY\), the symbol \(\equiv\) is used in place of \(\rightarrow\) and \("\Leftarrow\) within defining rules. Fig. 2.2 shows a small \(\mathcal{T}OY\) program. In addition to \(\text{z}\) and \(\text{Suc}\), the signature includes also the predefined list constructors \(\emptyset\) and \((::)\) (used in infix notation). The defining rules, having neither conditions nor local definitions, are hopefully self-explanatory. The arity of each function equals the number of formal parameters in its rules. In particular, \(\text{drop}\) (a function which eliminates the first four elements of a given list) has arity 0, in spite of its type. The second rule for \(\text{times}\) is incorrect w.r.t. the intended meaning of \(\text{times}\) as the multiplication operation.

Formally, the semantics of any program can be specified by means of the rewriting calculus GORC from [5]. There, conditions \(e \equiv e'\) are written as \(e \bowtie e'\) and called \(joinability\) \(\text{statements}\), while local definitions \(s \leftarrow d\) are written as \(d \rightarrow s\) and called \(approximation\) \(\text{statements}\). In our previous paper [2] we have presented a simple variant of GORC (the \textit{semantic calculus SC}) which can be
A declarative debugger of wrong answers for lazy functional logic programs

used to prove statements by building Abbreviated Proof Trees (APTs). Results in [2] ensure the correctness of declarative debugging using APTs as CTs. Nodes in APTs include basic facts of the form \( f(t_1, \ldots, t_n) \rightarrow t \) (with \( f \in F^{S^0} \), \( t_1, \ldots, t_n \) patterns), which become the questions asked to the oracle during the navigation phase of debugging. In this context, \( f(t_1, \ldots, t_n) \rightarrow t \) is not interpreted as a local definition. Rather, it claims that \( t \) approximates the result returned by the function call \( f(t_1, \ldots, t_n) \). In the sequel, we write \( P \vdash \varphi \) to indicate that the statement \( \varphi \) can be proved from program \( P \) in the semantic calculus SC. As in [2], we also assume an intended model \( I \) which represents the intended program semantics, given as the set of all the basic facts which are viewed as valid by the oracle.

We consider initial goals of the form \( G = e_1 \equiv e'_1, \ldots, e_k \equiv e'_k \). As in logic programming, goals can include logic variables that may become bound to patterns when the goal is solved. A solution for \( G \) is any total substitution \( \theta \) such that \( P \vdash G \theta \). A solution \( \theta \) is valid iff \( G \theta \) is valid in the intended model, and erroneous otherwise; see [2] for a formal definition. Considering the program in Fig. 2.2, the goal head (tail (map (times N) (from X))) \( \equiv \) Y asks for the second element of the infinite list that contains the product of \( N \) by the consecutive natural numbers starting at \( X \). The first two solutions computed by the \( TQY \) system are \( \theta = \{ N \mapsto z, Y \mapsto z \} \) (valid) and \( \theta' = \{ N \mapsto suc z, Y \mapsto z \} \) (erroneous). The valid solution \( \{ N \mapsto suc z, Y \mapsto suc X \} \) expected by the user is in fact a missing answer. Diagnosing missing answers is beyond the scope of this paper.

3 Problems and Contributions

In this section we summarize the main contributions of this paper to the two stages of declarative debugging, namely CT generation and CT navigation.

3.1 CT Generation

In the context of lazy FP and FLP, two main ways of constructing CTs have been proposed. The program transformation approach [15,13,17] gives rise to transformed programs whose functions return CTs along with the originally expected results. The abstract machine approach [14–16] requires low level modifications of the language implementation. Although the second approach can result in a better performance, we have adopted the first one because we find it more portable and better suited to a formal correctness analysis. With respect to other papers based in the transformational approach, we present two main contributions, described below.

Curried Functions: Roughly, all transformational approaches transform the functions defined in the source program to return pairs of type \((r,cTree)\), \( r \) being the type of the originally expected result and \( cTree \) a datatype for representing CTs. Moreover, the types of functions acting as parameters of higher-order (shortly, HO) functions must be transformed accordingly. From the viewpoint of types, a \( n \)-ary curried function \( f \) is transformed into \( f' \) as follows:

\[ \text{In } [2], \text{ joinability statements were not considered for the sake of simplicity, and all approximation statements were not used to represent local definitions in defining rules.} \]
\[ f :: \tau_1 \to \cdots \to \tau_n \to \tau \Rightarrow f' :: \tau'_1 \to \cdots \to \tau'_n \to (\tau', \text{cTree}) \]

As explained in Sect. 2, \( n \) corresponds to the number of parameters expected by the rewrite rules in \( f \)'s definition, but \( \tau \) can be also a HO type or a variable. For instance, the types of the functions \texttt{plus}, \texttt{drop4} and \texttt{map} from Fig. 2.2 (with respective arities 2, 0 and 2) are translated as follows:

\[
\begin{align*}
\text{\texttt{plus} :: & \text{nat} \to \text{nat} \to \text{nat} \Rightarrow \text{\texttt{plus'}} :: \text{nat} \to \text{nat} \to (\text{nat, cTree})} \\
\text{\texttt{drop4} :: & (\text{[A]} \to \text{[A]}) \Rightarrow \text{\texttt{drop4'} :: (\text{[A]} \to (\text{[A], cTree}), \text{cTree})} \\
\text{\texttt{map} :: & (\text{A} \to \text{B}) \to \text{[A]} \to \text{[B]} \Rightarrow \text{\texttt{map'} :: (\text{A} \to (\text{B, cTree})) \to \text{[A]} \to (\text{[B], cTree})}
\end{align*}
\]

As pointed out in [13,17], the previous approach can lead to type errors when curried functions are used to compute results which are taken as parameters by other functions. For instance, \( (\text{\texttt{map drop4}}) \) is well-typed, but the naïve translation (\( \text{\texttt{map'} drop4'} \)) is ill-typed, because the type of \( \text{\texttt{drop4'}} \) does not match the type expected by \( \text{\texttt{map'}} \) for its first parameter. More generally, the type of the result returned by \( f' \) when applied to \( m \) arguments depends on the relation between \( m \) and \( f \)'s arity \( n \). For example, \( (\text{\texttt{map (plus z)})} \) and \( (\text{\texttt{map plus})} \) are both well-typed; when translating naïvely, \( (\text{\texttt{map'} (plus' z)}) \) remains well-typed, but \( (\text{\texttt{map'} plus')} \) becomes ill-typed.

As a possible solution to this problem, the authors of [13] suggest to modify the translation in such a way that a curried function of arity \( n > 0 \) always returns a result of type \( (\tau, \text{cTree}) \) when applied to its first parameter. According to this idea, \( \text{\texttt{plus}} \) would become \( \text{\texttt{plus'}} :: \text{nat} \to (\text{nat} \to (\text{nat, cTree}), \text{cTree}) \).

However, as noted in [13], such a transformation would cause transformed programs to compute inefficiently, producing CTs with many useless nodes. Therefore, the authors of [13] wrote: "An intermediate transformation which only handles currying when necessary is desirable. Whatever this can be done without detailed analysis of the program is under investigation". Our program transformation solves this problem by translating a curried function \( f \) of arity \( n \), into \( n \) curried functions \( f'_0, \ldots, f'_{n-2}, f' \) with respective arities \( 1, 2, \ldots, n-1, n \), and proper types. Function \( f'_m (0 \leq m \leq n-2) \) is used to translate occurrences of \( f \) applied to \( m \) parameters, while \( f' \) translates occurrences of \( f \) applied to \( n-1 \) or more parameters. For instance, \( (\text{\texttt{map plus})} \) translates into \( (\text{\texttt{map'} plus'}) \), using the auxiliary function \( \text{\texttt{plus'}} :: \text{nat} \to (\text{nat} \to (\text{nat, cTree}), \text{cTree}) \).

We provide a similar solution to deal with partial application of curried data constructors, which can also cause type errors in the naïve approach (think of \( \text{\texttt{twice'} suc}) \), as an example). As far as we know, the difficulties with curried constructors have not been addressed previously. Our approach certainly increases the number of functions in transformed programs, but the extra functions are used only when needed, and inefficient CTs with useless nodes can be avoided.

A detailed specification of the translation is presented in Sect. 4.

**Correctness:** Our program transformation is provably correct: it preserves polymorphic well-typing (modulo a type transformation) and program semantics (as formalized in [5,2]). As we will see in Sect. 4, this correctness result holds independently of the narrowing strategy chosen as goal solving mechanism. To the best of our knowledge, previous related papers [15,13,17] give no correctness
proof for the program transformation. The author of [17], who is aware of the
problem, just relies on intuition for the semantic correctness. He mentions the
need of a formalized semantics for a rigorous proof. As for type correctness, it
is closely related to the treatment of curried functions, which was deficient in
previous approaches.

CT Navigation: In order to be a really practical tool, a declarative debugger
should keep the number of questions asked to the oracle as small as possible. Our
debugger uses a decidable and semantically correct entailment between basic
facts to maintain a consistent and non-redundant store of facts known from
previously answered questions. Redundant questions whose answer is entailed
by stored facts are avoided. Details are explained in Sect. 5.

4 Generation of CTs by Program Transformation

In this section we present the program transformation used by our debugger
and discuss its correctness. Roughly, a program $P$ is converted into a new
program $P'$, where function calls return the same results $P$ would return, but
paired with CTs. Formally, $P'$ is obtained by transforming the signature $\Sigma$ of $P$
into a new signature $\Sigma'$, introducing definitions for certain auxiliary functions,
and transforming the function definitions included in $P$. Let us consider these
issues one by one.

4.1 Representing Computation Trees

A transformed program always includes the constructors of the datatype cTree,
used to represent CTs and defined as follows:

\[
\begin{align*}
\text{data cTree} & = \text{void} \mid \text{Node funId [arg] res rule [cTree]} \\
\text{type arg, res} & = pVal \\
\text{type funId, pVal, rule} & = \text{string}
\end{align*}
\]

A CT of the form (cNode f ts t r1 cts) corresponds to a call to the function
f with arguments ts and result t, where r1 indicates the function rule used to
evaluate the call, and the list cts consists of the children CTs corresponding to
all the function calls (in the local definitions, right-hand side and conditions of
r1) whose activation was needed in order to obtain t. Due to lazy evaluation,
the main computation may demand only partial approximations of the results of
intermediate computations. Therefore, ts and t stand for possibly partial values,
represented as partial patterns; and (f ts t r1 cts) represents the basic fact whose
validity will be asked to the oracle during debugging, as explained in Sect. 2.

As for void, it represents an empty CT, returned by calls to functions which are
trusted to be correct (in particular, data constructors and the auxiliary functions
introduced by the translation). Finally, the definition of arg, res, funId, pVal and
rule as synonyms of the type of character strings is just a simple representation;
other choices are possible.\(^2\)

\(^2\) Actually, our prototype debugger uses a more structured representation for arg and
res, which is helpful to implement the entailment relation described in Sect. 5.
4.2 Transforming Program Signatures

For every n-ary function \( f :: \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau \) occurring in \( P \), \( P' \) must include an \((m+1)\)-ary auxiliary function \( f^m \) for each \( 0 \leq m < n-1 \), as well as an n-ary function \( f' \), with principal types:

\[
\begin{aligned}
   f^m &:: \tau_1^m \rightarrow \ldots \rightarrow \tau_{m+1} \rightarrow (\tau_{m+2} \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau', \text{cTree}) \quad \text{for } 0 \leq m < n-1 \\
   f' &:: \tau_1' \rightarrow \ldots \rightarrow \tau_m' \rightarrow (\tau_{m+2}' \rightarrow \ldots \rightarrow \tau_n' \rightarrow \tau', \text{cTree})
\end{aligned}
\]

Similarly, for each n-ary constructor \( c :: \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau \) occurring in \( P \), \( c' \) must keep \( c \) with the same principal type, and include new \((m+1)\)-ary auxiliary functions \( c_m \) \((0 \leq m < n)\), with principal types:

\[
   c^m :: \tau_1^m \rightarrow \ldots \rightarrow \tau_{m+1} \rightarrow (\tau_{m+2} \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau', \text{cTree}) \quad \text{for } 0 \leq m < n
\]

The type declarations above depend on a type transformation which converts any type \( \tau \) in \( P \)'s signature into a transformed type \( \tau' \). This is defined recursively:

- \( \alpha' = \alpha \) \((\alpha \in \text{TVar})\)
- \( (C \tau_n)^' = C \tau_n' \) \((C \in \text{TC}^n)\)
- \( \mu \rightarrow \nu' = \mu' \rightarrow (\nu', \text{cTree}) \)

Finally, \( P' \) always includes the auxiliary functions \( \text{dVal} \) and \( \text{clean} \), whose types and definitions will be described below.

4.3 Defining Auxiliary Functions

Each auxiliary function \( f^m \) expects \( m+1 \) arguments and returns a partial application of \( f \) paired with a trivial CT. Exceptionally, \( f^0_{n+2} \) returns a partial application of \( f \). The auxiliary functions \( c_m \) are defined similarly, except that \( c^m_{n+1} \) returns a value built with the data constructor \( c \).

\[
\begin{aligned}
   f^0_X : (f, X_1, \text{void}) &:: f^0_{X_1} X_1 = (f(X_1), \text{void}) \\
   f^m_X : (f, X_1, X_2) \ldots (f, X_m, \text{void}) &:: f^m_{X_{m+1}} X_{m+1} = (f(X_{m+1}), \text{void}) \\
   c^0_X : (c, X_1, \text{void}) &:: c^0_{X_1} X_1 = (c(X_1), \text{void}) \\
   c^m_X : (c, X_1, X_2) \ldots (c, X_m, \text{void}) &:: c^m_{X_{m+1}} X_{m+1} = (c(X_{m+1}), \text{void})
\end{aligned}
\]

4.4 Transforming Function Definitions

Each defining rule \( f t_1 \ldots t_n \rightarrow r \Leftarrow C \) where \( LD \) occurring in \( P \) is transformed into a corresponding defining rule for \( f' \) in \( P' \). Assuming that \( LD \) consists of local definitions \( s_j \leftarrow d_j \) and \( C \) consists of conditions \( l_i \Leftarrow r_i \), the transformed defining rule is constructed as

\[
\begin{aligned}
   f' t_1' \ldots t_n' \rightarrow (R, T) \Leftarrow \ldots \leftarrow CL_\ast \Leftarrow CR \Leftarrow \ldots \\
   \left\{
   \begin{aligned}
      s_1' &\Leftarrow d_1' \\
      \ldots \leftarrow \ldots \\
      CL_\ast &\Leftarrow l_i' \\
      CR &\Leftarrow r_i' \\
   \end{aligned}
\end{aligned}
\]

Some additional explanations are needed at this point:

- \( t_1', s_1', d_1', l_1', r_1' \) and \( r' \) refer to an expression transformation (defined below) which converts any \( e :: \tau \) of signature \( \Sigma \) into \( e' :: \tau' \) of signature \( \Sigma' \).
- \( R, T, CL_\ast, CR \) are new fresh variables, and \( j \) is the number of the rule.
- The notation \( \{ \ldots \} \) refers to a transformation of the local definitions explained.
\(<\text{dVal} : A \to \text{pVal}\) is an auxiliary impure function without declarative meaning, very similar to \<\text{dirt}\> in [13,17]. Any call \(<\text{dVal} \ a\) (read: “demanded value of \(a'\)”) returns a representation of the partial approximation of \(a\)'s value which was needed to complete the top level computation. The debugger’s implementation can compute this from the internal structure representing \(a\) at the end of the main computation, replacing all occurrences of suspended function calls by \(n\). \(\_\) which represents the undefined value \(\bot\). \(^3\) Moreover, \(<\text{dVal}\) also renames all the identifiers of auxiliary functions \(f'_m\) resp. \(c'_m\) into \(f\) resp. \(c\). In this way, the results computed by the transformed program are translated back to the original signature.

The expression transformation \(e \mapsto e'\) is defined by recursion on \(e\)'s syntactic structure. The idea is to transform the (possibly partial) applications of functions and constructors within \(e\), using functions from the transformed signature. In order to ensure \(e' \equiv \tau^l\) whenever \(e \equiv \tau\), we use two auxiliary application operators:

\[
\begin{align*}
\emptyset_0 & : (\beta, \text{cTree}) \to \beta & (\emptyset) & : (\alpha \to (\beta, \text{cTree})) \to \alpha \to \beta \\
\emptyset_0 \to R & \text{ where } \{ (R,T) \leftarrow F \} & F \to X & \text{ where } \{ (R,T) \leftarrow F \ X \}
\end{align*}
\]

These are used within \(e'\) at those points where the application of a function from the translated signature is expected to return a value paired with a CT. Applications of higher-order variables are treated in a similar way. Formally:

\[
\begin{align*}
[X \ a_1 \ldots a_k]' &= (\ldots (X \ @ \ a_1') \ @ \ldots) \ @ \ a_k' & (X \in \text{Var}, k \geq 0) \\
(c \ e_1 \ldots e_m)' &= c \ e_1' \ldots e_m' & (c \in DC^n, m < n, n > 0) \\
(c \ e_1 \ldots e_m)' &= c \ e_1' \ldots e_m' & (c \in DC^n, n \geq 0) \\
[f \ a_1 \ldots a_k]' &= (\ldots (f \ @ a_1') \ @ \ldots) \ @ a_k' & (f \in FS^0, k \geq 0) \\
(f \ e_1 \ldots e_m)' &= f \ e_1' \ldots e_m' & (f \in FS^n, n > 0, m < n - 1) \\
(f \ e_1 \ldots e_m, a_1 \ldots a_k)' &= (\ldots (f \ e_1', \ldots e_m') \ @ a_1') \ @ \ldots) \ @ a_k' & (f \in FS^n, n > 0, k \geq 0)
\end{align*}
\]

Looking back to the construction of translated defining rules, we see that the translated expressions \(\ell'_k, \ s'_j, d'_j, l'_k, r'_j, r'\) and \(\tau'\) are intended to ensure well-typing, but ignore CTS. In particular, the local definition of \(T\) renders a CT whose root has the proper form, but whose children are not yet defined. In order to complete the translation, the translated local definitions \(\{ \ldots \}\) are transformed into \(\{ \ldots \} \downarrow\), which means the normal form obtained by applying the transformation rules \(AP_0\) and \(AP_1\) defined below, with a leftmost-innermost strategy.

- \(AP_0\):
  \[
  \begin{align*}
  \{ \ldots; t \leftarrow e[@[0 \ g]]; \ldots \leftarrow c\text{Node} \ldots \text{clean} \ l p \} \quad & \mapsto \\
  \{ \ldots; [R', T'] \leftarrow \emptyset; \ t \leftarrow e[R'] ; \ldots \leftarrow c\text{Node} \ldots \text{clean} \ l p \text{ ++ \{ \text{dVal} \ R', T' \}} \}
  \end{align*}
  \]

- \(AP_1\):
  \[
  \begin{align*}
  \{ \ldots; t \leftarrow e[a \ @ s]; \ldots \leftarrow c\text{Node} \ldots \text{clean} \ l p \} \quad & \mapsto \\
  \{ \ldots; [R', T'] \leftarrow a \ s; \ t \leftarrow e[R'] ; \ldots \leftarrow c\text{Node} \ldots \text{clean} \ l p \text{ ++ \{ \text{dVal} \ R', T' \}} \}
  \end{align*}
  \]

\(^3\) Because of this replacement of \(\bot\) in place of unknown values, the basic facts occurring in proof trees must be understood as approximation statements rather than equalities.
In both transformations, ‘++’ stands for the list concatenation function. \( R' \) and \( T' \) must be chosen as new fresh variables, and \( t \) is the pattern in the lefthand side of a local definition whose righthand side includes a leftmost-innermost occurrence of an application operator \((@_0 \ g)\) or \((a @ s)\) in some context. Because of the innermost strategy, we can claim:

- \( AP_0 \) always finds a nullary function symbol in place of \( g \).
- \( AP_1 \) always finds a pattern in place of \( s \), and either a variable or a pattern of the form \( g t_1 \ldots t_{n-1} \) (with \( g \in FS^n, n > 0 \)) in place of \( a \).

Each application of the \( AP \) transformations eliminates the currently leftmost-innermost occurrence of an application operator, while introducing a new local definition for the result \( R' \) and CT \( T' \) coming from that application, and adding the pair \((dVal \ R', T')\) to the list of children of \( T \). The innermost strategy ensures that no application operators occur in the new local definition. Since the initial number of application operators is finite, the process is terminating and the normal form always exists. When the \( AP \) transformations terminate, no application operators remain. Therefore, \( @_0 \) and \( @ \) do not occur in transformed programs. All the occurrences of ‘++’ within the righthand side of \( T \)'s local definition can be removed, according to the usual definition of list concatenation.

This leads to a list \( lp :: [(pVal, cTree)] \) including as many CTs as application operators did occur in the local definitions, each of them paired with a partial result. Finally, (clean \( lp \)) builds the ultimate list of children CTs, by ignoring those pairs \((pv, ct)\) in \( lp \) such that \( ct \) is void or \( pv \) represents \( \bot \), thus removing all the useless nodes from the CT. The code corresponding to the auxiliary function \( \text{clean} :: [(pVal, cTree)] \rightarrow cTree \) (whose simple definition is omitted here) must be included in any transformed program.

### 4.5 An Example

Below we show part of the functions produced by transforming the \( I/OY \) program from Fig. 2.2. The concrete syntax of \( I/OY \) is used here.

\[
\text{twice'} \quad : \quad (A \rightarrow (A, cTree)) \rightarrow A \rightarrow (A, cTree)
\]
\[
\text{twice'} \quad F \quad X \quad = \quad (R, T)
\]
\[
\begin{align*}
&\quad \quad \text{where} \quad ((R1, T1)) \quad = \quad F \quad X; \\
&\quad \quad (R2, T2) \quad = \quad F \quad R1; \\
&\quad \quad R \quad = \quad R2; \\
&\quad \quad T \quad = \quad cNode \quad \text{"twice"} \quad [dVal \quad F, \quad dVal \quad X] \quad (dVal \quad R1) \quad \text{"twice.1"} \\
&\quad \quad (\text{clean} \quad [(dVal \quad R1, T1), \quad (dVal \quad R2, T2)])
\end{align*}
\]

\[
\text{drop4'} \quad : \quad ([\text{nat}] \rightarrow ([\text{nat}], cTree), cTree)
\]
\[
\begin{align*}
&\quad \quad \text{drop4'} \quad = \quad (R, T) \\
&\quad \quad \text{where} \quad ((R1, T1)) \quad = \quad \text{twice'} \quad \text{twice}_0' \quad \text{tail'}; \\
&\quad \quad R \quad = \quad R1; \\
&\quad \quad T \quad = \quad cNode \quad \text{"drop4"} \quad [] \quad (dVal \quad R1) \quad \text{"drop4.1"} \\
&\quad \quad (\text{clean} \quad [(dVal \quad R1, T1)])
\end{align*}
\]

### 4.6 Transforming Goals

The debugging process can be started whenever some answer answer \( \theta \) computed for the goal \( G \) is considered erroneous by the user. In order to build a suitable CT.
for the navigation phase, an auxiliary function definition \( \text{solution} \, X_n = \text{true} \) \( \Leftrightarrow G \) is considered, whose translation is automatically added to the transformed program. Here, \( X_n \) are the variables occurring in \( G \). Since \( \theta \) is a solution for \( G \), the goal \( \text{solution} \, X_n(\theta) \equiv \langle \text{true}, \, \text{Tree} \rangle \) can be solved by the translated program, without instantiating any free variable in \( X_n \theta \), and instantiating \( \text{Tree} \) to a \( \text{CT} \) with erroneous root. The navigation phase of the debugger proceeds with this \( \text{CT} \).

4.7 Correctness Results

We can prove three main results about the correctness of our program transformation. The first result concerns the type discipline. Thanks to it, the debugger does not need to perform any type checking/inference before entering the \( \text{CT} \) generation phase.

**Theorem 1** The translation \( P' \) of a well-typed program \( P \) is always well-typed.

**Proof Idea.** Assuming an expression \( e \) in \( P \)'s signature and a type environment \( T \) such that \( T \vdash_{WT} e : \tau \), one can prove \( T' \vdash_{WT} e' : \tau' \) for \( T' =_{\theta} \{ X_i : \tau_i' \mid (X_i : \tau_i) \in T \} \). Using this result, and reasoning by induction on the number of \( \text{AP} \) steps involved in the transformation of defining rules, it can be proved that any well-typed defining rule for a function \( f \) in \( P \) is transformed into a well-typed defining rule for \( f' \) in \( P' \). On the other hand, the defining rules for auxiliary functions occurring in \( P' \) are obviously well-typed. \( \Box \)

The second result says that the translation preserves the semantics of source programs, enhanced by the additional computation of \( \text{CTs} \). Recall the notation \( P \vdash f \, \mathbf{r}_n \rightarrow t \) explained in Sect. 2, which can be used also for transformed programs. Intuitively, a basic fact \( f \, \mathbf{r}_n \rightarrow t \) asserts that the function call \( f \, \mathbf{r}_n \) can return a partial result \( t \) according to the semantics of program \( P \). Note also that the translation of a pattern \( t \), following the definition from Subsection 4.4, is always a pattern \( t' \) from which \( t \) can be univocally recovered.

**Theorem 2** For any \( n \)-ary function \( f \) and arbitrary partial patterns \( \mathbf{r}_n, \, t \) in the signature of a program \( P \), it holds:

1. If \( P' \vdash f \, \mathbf{r}_n \rightarrow (t', \, ct) \) then \( P' \vdash f \, \mathbf{r}_n \rightarrow t \).
2. If \( P \vdash f \, \mathbf{r}_n \rightarrow t \) then there is some pattern \( ct \) in \( P \)'s signature, which represents an abbreviated proof tree proving \( P \vdash f \, \mathbf{r}_n \rightarrow t \) in the sense of [2], and such that \( P' \vdash f' \, \mathbf{r}_n \rightarrow (t', \, ct) \).

**Proof Idea.** Abbreviated Proof Trees (\( \text{APTs} \)) were introduced in [2] to formalize proofs of basic facts in a semantic calculus \( \text{SC} \). Due to the form of the inference rules of \( \text{SC} \), it can be checked that an \( \text{APT} \) \( T \) which proves \( P \vdash f \, \mathbf{r}_n \rightarrow t \) must be built by using some particular instance of defining rule for \( f \) of the form \( f \, \mathbf{r}_n \rightarrow r \leftarrow C \) where \( LD \). Moreover, the root of \( T \) must contain \( f \, \mathbf{r}_n \rightarrow t \), and the children must be a list of \( \text{APTs} \) corresponding to \( \text{SC} \)'s proofs which justify the local definitions in \( LD \), the conditions in \( C \) and the statement \( r \rightarrow t \). Moreover, a careful analysis of \( \text{SC} \)'s inference rules shows that these children \( \text{CTs} \) must correspond to function applications which return a partial result different from \( \bot \), taken in leftmost-innermost order. The definition of transformed defining rules in \( P' \) has been designed to build precisely such \( \text{CTs} \). \( \Box \)
Finally, as a consequence of the previous result, the generated CT provides a correct basis for the navigation phase:

**Theorem 3** Assume a program $P$, a wrong answer $\theta$ for a goal $G$, and a CT $T$ obtained by running a transformed program, as explained in Sect. 4.6. Then $T$ has at least one buggy node, which includes an indication of a semantically incorrect instance of a defining rule from $P$.

**Proof.** Because of Theorem 2, $T$ is an APT. The Correctness Theorem from [2] ensures the following: any APT whose root is erroneous always includes some buggy node whose associated program rule instance is not valid in the intended model.

We would like to stress the fact that Theorem 3 holds independently of the narrowing strategy implemented by the system used to run the transformed program. Although a particular narrowing calculus was proposed in [2] to formalize the construction of APTs, all APTs (in particular, those computed by transformed programs) are correct CTs for debugging. Of course, the narrowing strategy affects the order in which eventual wrong answers are computed.

## 5 Navigating the CTs by Oracle Querying

Once the CT associated to a wrong answer has been built (as described in Subsection 4.6), navigation performs a top-down traversal, asking the oracle about the validity of the basic facts associated to the visited nodes (except the root, which is known to be erroneous in advance). For the sake of practical usefulness, it is important to ensure that questions asked to the oracle are as few and as simple as possible. The second condition - simplicity - comes along with our choice of APTs as CTs, since basic facts are the minimal pieces of information needed to characterize the intended model of a program (see [2]).

To reduce the number of questions, the only possibility considered in related papers is to avoid asking repeated questions. As an improvement, we present an entailment relation between basic facts, and we show that it can be used to avoid redundant questions which can be deduced from previous answers.

Our notion of entailment is based on the approximation ordering $\sqsubseteq$, defined as the least partial order over the set $Pat_\bot$ of partial patterns which satisfies:

- $\bot \sqsubseteq t$, for any partial pattern $t$.
- $X \sqsubseteq X$, for any variable $X$.
- $h \tau_n \sqsubseteq h \tau_n$, if $t_1 \sqsubseteq s_1 \ldots t_n \sqsubseteq s_n$, with $h \in DC^n$, $n \leq m$ or $h \in FS^n$, $n < m$.

This ordering has a natural semantic interpretation: $t \sqsubseteq t'$ means that $t'$ has at least as much information as $t$. For instance: $(z : \bot) \sqsubseteq (z : \mu c z : \bot)$. Using $\sqsubseteq$, we define: A basic fact $f \tau_n \rightarrow t$ entails another basic fact $f \tau_m \rightarrow s$ (written as $f \tau_n \rightarrow t \sqsupseteq f \tau_m \rightarrow s$) iff there is some total substitution $\theta \in Subst$ such that $t_1 \theta \sqsubseteq s_1, \ldots, t_n \theta \sqsubseteq s_n, s \sqsubseteq t\theta$.

The following result justifies the interest of entailment, since the answers given by any oracle in declarative debugging are assumed to be valid in the intended model of the program.
Theorem 4 Entailment between basic facts is a decidable preorder (i.e., reflexive and transitive relation). Moreover, for any set \( \mathcal{I} \) of basic statements which represents an intended model: if \( f \mathcal{T}_n \rightarrow t \preceq f \mathcal{T}_n \rightarrow s \) and \( f \mathcal{T}_n \rightarrow t \in \mathcal{I} \) then \( f \mathcal{T}_n \rightarrow s \in \mathcal{I} \).

Proof Idea To prove decidability of entailment, we have defined a system of transformations, somewhat similar to those used in Martelli and Montanari’s unification algorithm. These transformations compute the needed substitution \( \theta \) whenever entailment holds, and they finitely fail otherwise. The fact that entailment is a preorder is easy to check. The last claim of the Theorem is true because the formal definitions given in [2] ensure that intended models are closed under entailment.

Thanks to Theorem 4, a question \( Q \) entailed by a fact already known to be valid because of some previous answer, must be valid. For instance, if we already know that \( \text{from} \ X \rightarrow X \cdot (\text{Suc} \ X) : \bot \) is valid then other basic facts entailed by this one, such as \( \text{from} \ Y \rightarrow \bot \) and \( \text{from} \ (\text{Suc} \ Y) \rightarrow (\text{Suc} \ Y) : (\text{Suc} (\text{Suc} Y)) : \bot \), must also be valid. Dually, a question \( Q \) which entails a fact known to be invalid because of some previous answer, must be invalid. For instance, if we know from a previous answer that \( \text{from} \ Y \rightarrow (\text{Suc} \ Z) : \bot \) is not valid, then other basic facts that entail this one, such as \( \text{from} \ X \rightarrow (\text{Suc} \ X) : (\text{Suc} \ (\text{Suc} X)) : \bot \), must be also invalid. In both cases, a question to the oracle can be avoided.

Our debugger has been implemented as part of the \( TCOY \) system. A prototype version can be downloaded from http://titam.sip.ucm.es/toy/debug.tar.gz.

Here we show a debugging session for our example program of Fig. 2.2. The user activates the debugger because the incorrect answer \( \{N \rightarrow \text{Suc} \ Z, Y \rightarrow \text{Z} \} \) has been computed for the goal \( \text{head} (\text{tail} (\text{map} (\text{times} \ N) (\text{from} X))) = Y \):

Consider the following facts:
1: \( \text{from} \ X \rightarrow (X : (\text{Suc} X) : \bot) \)
2: \( \text{map} (\text{times} \ (\text{Suc} \ Z)) (X : (\text{Suc} X) : \bot) \rightarrow (\bot : z : \bot) \)
3: \( \text{tail} (\bot : z : \bot) \rightarrow (z : \bot) \)
4: \( \text{head} (z : \bot) \rightarrow z \)

Are all of them valid? ([y]es / [n]o) / [a]bort n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: \( \text{map} (\text{times} \ (\text{Suc} \ Z)) (\text{Suc} X : \bot) \rightarrow (z : \bot) \)
Are all of them valid? ([y]es / [n]o) / [a]bort n

Consider the following facts:
1: \( \text{times} \ (\text{Suc} \ Z) \rightarrow z \)
Are all of them valid? ([y]es / [n]o) / [a]bort n

Consider the following facts:
1: \( \text{times} \ (\text{Suc} X) \rightarrow z \)
2: \( \text{plus} z z \rightarrow z \)
Are all of them valid? ([y]es / [n]o) / [a]bort y

Rule number 2 of the function \( \text{times} \) is wrong.
Wrong instance: \( \text{times} \ (\text{Suc} \ Z) \ (\text{Suc} X) \rightarrow (\text{plus} z (\text{times} \ (\text{Suc} X))) \)
As shown by this example, our current prototype debugger searches the CT top-down, using a strategy whose aim is to avoid redundant questions and to give freedom to the oracle. At any point during the search, the current node contains an invalid statement (initially, this is true because the root of the CT corresponds to an error symptom detected by the user). The debugger builds the list L of the basic facts attached to the children of the current node. If some member of L entails a fact known to be invalid from some previous oracle answer, the debugger moves to the corresponding child and continues with the same strategy. Otherwise, the debugger displays the list L for the oracle’s consideration. If the oracle regards all the facts in L as valid, then the current node is buggy, and the debugger shows its associated program rule instance (which can be computed from the CT) as responsible for the bug. Otherwise, the oracle must choose some erroneous fact in the list. The debugger adds this fact to its store of invalid facts, moves to the corresponding child node, and continues with the same strategy.

In the simple example shown above, the entailment relation is not helpful, but in more involved cases it can reduce the number of questions asked to the oracle. Note that the particular search strategy we have described is such that all the answers provided by the oracle are negative, except for the last question. This might not be the case in other alternative strategies, which we have not yet investigated. Our implementation also avoids to ask questions about predefined functions (e.g., arithmetic operations), since they are trusted to be correct. Allowing the user to annotate certain functions to be trusted as correct is a simple albeit useful extension, not yet implemented.

6 Conclusions and Future Work

Program transformation is a known approach to the implementation of declarative debugging of wrong answers in lazy FLP languages [15, 13, 17]. We have given a new, more formal specification of this technique, which avoids type errors related to the use of curried functions and preserves both well-typing and program semantics (as formalized in [5, 2]), independently of the narrowing strategy chosen as goal solving mechanism. A prototype implementation of our debugger for the functional logic language TMY [9] is available. Our implementation uses a semantically correct algorithm to detect and avoid redundant questions to the oracle, thus reducing the complexity of debugging.

In order to improve the practical usefulness of our results, we have started a cooperation with Herbert Kuchen and Wolfgang Lux, to include a similar debugger as a tool within the Curry [7] implementation developed at Münster University. Hopefully, this will eventually allow to evaluate the debugger on practical applications. We also plan to implement and evaluate alternative search strategies for the navigation phase. As more substantial research work, we plan to investigate and implement extensions of the debugger, to support constraint-based computations as well as the diagnosis of missing answers.

Acknowledgements We are grateful to Herbert Kuchen, Paco López and Wolfgang Lux, who have followed our work with interest and provided useful advice.
We are also very indebted to Mercedes Abengózar for her great help with implementation work.

References

Transforming Layered Specifications into Operational Specifications

F. J. Galán and J. M. Cañete

Dept. of Languages and Computer Systems. Faculty of Computer Science of Seville, Av. Reina Mercedes 41012, Seville, Spain. E-mail: galanm@lsi.us.es

Abstract. A specification can describe a set of programs which are difficult to discover. The systematic construction of programs from specifications is known as program synthesis. The use of two different languages (i.e., specification and programming languages) adds new difficulties in order to decide about correctness criteria. A classification of the specifications based on the existence (operational specification) or not (descriptive specification) of a clear trace between specifications and programs is proposed. In this way, a synthesis process may be reconsidered as a sequential composition of two activities: to transform specifications into operational specifications and then to adapt operational specifications to implementation resources. In our work, we treat only the first activity. The language of the layered contexts is proposed to describe functional properties of the software and establish a method which transforms descriptive specifications into operational ones automatically.

1 Introduction

Abstraction and expressiveness are two key properties associated to specification languages. The following (first-order logic) specification $D_{\text{nocc}}$ defines the relation symbol $\text{nocc}$:

$$D_{\text{nocc}} : \text{nocc}(e, \text{empty}, z) \Leftrightarrow z = 0$$
$$\text{nocc}(e, \text{conc}(x, Y), z) \Leftrightarrow x = e \land \text{nocc}(e, Y, z)$$
$$\text{nocc}(e, \text{conc}(x, Y), z) \Leftrightarrow \neg x = e \land \text{nocc}(e, Y, z)$$

From a programming point of view, $D_{\text{nocc}}$ is an abstract description of a set of (possible) programs which compute occurrences of natural elements in a sequence. Its form is closed to the structure of a program and then we consider $D_{\text{nocc}}$ as an operational specification. For example, a Modula-2 programmer would be able to propose the following (correct) implementation for $D_{\text{nocc}}$:

PROCEDURE Nocc(e: CARDINAL, s: SeqNat, VAR z: CARDINAL);
VAR w: CARDINAL;
BEGIN
  IF (s=NIL) THEN z:=0
  ELSIF s^.info = e THEN Nocc(e, s^.next, w); z:=w+1
  ELSE Nocc(e, s^.next, z);
On the other hand, specification languages are usually considered as **expressive** languages because they have description resources which are natural to human thinking. The following specification defines the symbol `perm` in a natural way.

\[
D_{perm} : \text{perm}(L, S) \iff (\text{nocc}(a, L, z) \iff \text{nocc}(a, S, z))
\]

From a programming point of view, \(D_{perm}\) represents not only an abstract but an expressive description of a set of programs which compute sequence permutations. However, the form of \(D_{perm}\) is not closed to the structure of a program (i.e. there is not explicit recursion) and then we tend to consider \(D_{perm}\) as a **descriptive specification**.

Searching for new programming languages encounters an important obstacle if we are interested in abstract and expressive proposals. Program synthesis refers to the elaboration of a program in some systematic manner starting from a specification \([WBK92]\), \([Fle95]\), \([BiD96]\), \([AFM99]\). Among several approaches \([DeK94]\), the **deductive approach** applies transformation rules directly to the specification until a program is obtained \([Par90]\), \([SaT84]\). However, the use of two different languages (i.e. specification and programming languages) adds new difficulties in order to decide about correctness criteria. In this way, a synthesis process can be redefined as a sequential composition of two activities: to redefine specifications into operational terms and then to adapt operational specifications to implementation resources. We think that the synthesis core resides in the first activity, therefore, in this work, we are interested in **transforming specifications which admit equivalent operational redefinitions**.

Layered contexts constitute reasonable starting points to describe functional properties of software in an abstract and expressive way and develop automatic transformations preserving correctness. Basically, a layered context is a first-order theory divided into three parts or layers: the data layer (or layer 0) which contains information for data, the operational layer (or layer 1) which contains a set of operational specifications and the descriptive layer (or layer 2) which contains a set of descriptive specifications. Each element in a layer is defined from elements in the same or lower layers. In example 1, we show the layered context \(S\). Its data layer defines `Nat` (natural numbers) and `Seq(Nat)` (sequences of natural elements) as sorts and relations which decide identities between `Nat`-terms \((D_{Nat,-})\) and `Seq(Nat)`-terms \((D_{Seq(Nat),-})\) respectively. The operational layer defines `nocc` symbol \((D_{nocc})\). The descriptive layer defines the relations `perm` \((D_{perm})\) and `eq` \((D_{eq})\) as a relation which decides a permutation-based sequence equality.

Following, previous works \([LaO94]\), \([LaO95]\), \([GaT98]\), \([GaC01]\) we proposed a method based on a transformation calculus which includes specialized forms for fold, unfold and substitution rules. The constructive definition of the calculus, the automatic generation of transformation plans, and the finite transformations within layered contexts represent key properties in order to consider our method amenable to be compiled and to be executed in practice.
Example 1.

Context $S$

Data layer

- sort $\text{Nat}$ generated by 0, $s$; sort $\text{Seq(Nat)}$ generated by $\text{empty, conc}$;
- $D_0 : 0 = 0 \Leftrightarrow \text{true}$
- $D_\text{empty} : \text{empty} = \text{empty} \Leftrightarrow \text{true}$
- $s(x) = 0 \Leftrightarrow \text{false}$
- $\text{conc}(x, X) = \text{empty} \Leftrightarrow \text{false}$
- $s(x) = s(y) \Leftrightarrow x = y$
- $\text{conc}(x, X) = \text{conc}(y, Y) \Leftrightarrow x = y \land X = Y$

Operational layer

- $D_{\text{nocc}} : \text{nocc}(e, \text{empty}, z) \Leftrightarrow z = 0$
- $\text{nocc}(e, \text{conc}(x, Y), s(z)) \Leftrightarrow x = e \land \text{nocc}(e, Y, z)$
- $\text{nocc}(e, \text{conc}(x, Y), 0) \Leftrightarrow \neg x = e \land \text{nocc}(e, Y, 0)$
- $\text{nocc}(e, \text{conc}(x, Y), s(k)) \Leftrightarrow \neg x = e \land \text{nocc}(e, Y, s(k))$

Descriptive layer

- $D_{\text{perm}} : \text{perm}(L, S) \Leftrightarrow (\text{nocc}(a, L, z) \Leftrightarrow \text{nocc}(a, S, z))$
- $D_{\text{eq}} : \text{eq}(\text{empty, empty}) \Leftrightarrow \text{true}$
- $\text{eq}(\text{conc}(x, X), \text{conc}(y, Y)) \Leftrightarrow \exists Z ((\text{perm}(nL, Z) \land nL = \text{conc}(x, X)) \Leftrightarrow (\text{perm}(Z, nS) \land nS = \text{conc}(y, Y)))$

Operational layer

Our work is explained in the following manner. Section 2 establishes preliminary definitions. Section 3 defines layered contexts as an interesting class of formal theories in order to apply automatic transformations. Section 4 describes the method which transforms descriptive specifications into operational ones and section 5 establishes conclusions.

2 Preliminary Definitions

In this section, we introduce the language for writing contexts. Basically, a context is a theory which defines the specification and transformation frameworks. Then, a classification of the specifications is introduced in order to characterize contexts. We borrow from [BMM83], [LaO94] a class of models, called isoinitial models, as interpretation model for our contexts. The reader is assumed to be familiar with the basic notations of the first-order theory as presented in [Llo87] and of the many-sorted logic as presented in [HL94].

Definition 1 (Syntax of a Formula). A many-sorted (typed) first-order language is assumed to write our formulas. A formula $Q_\tau_1 x_1 \ldots Q_\tau_n x_n F$ where $Q_\tau$, $x_i$ is a universal or existential quantifier defined on a type $\tau_i$, $x_i$ is different from $x_j$ for $i \neq j$, and $F$ contains no quantifier, is said to be in prenex normal form. We consider that, when possible, all quantifiers in a formula are ordered following a lexicographic order defined on the names of their respective types.

Example 2. $\forall_{\text{Nat}} a \forall_{\text{Nat}} z \forall_{\text{Seq(Nat)}} L \forall_{\text{Seq(Nat)}} S (\text{nocc}(a, L, z) \Leftrightarrow \text{nocc}(a, S, z))$. In the following, we assume that all our formulas are in prenex normal form. For legibility reasons, we omit type subscripts when the type can be induced clearly and expressions such as $Q_\tau x_1 \ldots Q_\tau z F$ can be collapsed to $Q_\tau x_1 \ldots, z F$. In addition, when possible, universal quantifiers are omitted in the front of a formula.
Definition 2 (Substitution). A (ground) variable substitution is the pair \((v, t)\) where \(v\) is a variable and \(t\) is a (ground) term. A substitution \(\sigma\) is a set of variable substitutions. Let \(X = \{x_1, \ldots, x_n\}\) and \(Y = \{y_1, \ldots, y_n\}\) be two sets of variables where \(x_i\) is different from \(x_j\) and \(y_i\) is different from \(y_j\) for \(i \neq j\). Let \(Q_X = \{Q_{\tau_1}, \ldots, Q_{\tau_m}\}\) and \(Q_Y = \{Q_{\tau_1}, \ldots, Q_{\tau_m}\}\) be two sets of quantifiers for variables in \(X\) and \(Y\) respectively. We say that \(\sigma = \{(x_1, y_1), \ldots, (x_n, y_n)\}\) is a renaming substitution which agrees w.r.t. quantifications iff \(X \cap Y = \emptyset\) and \(Q_{\tau_i} = Q_{\tau_i}\) with \(i = 1..n\).

Definition 3 (Iff-axioms). An iff-axiom is a formula of the form \(r(X) \Leftrightarrow R(Y)\) where \(R(Y)\) is any formula in the language of \(C\) and \(X\) and \(Y\) are two sets of terms. \(r\) is called the defined symbols. \(r(X)\) is called the left-hand side of the axiom (lhs) and \(R(Y)\) is called the right-hand side of the axiom (rhs). A recursive iff-axiom for a relation \(r\) is an iff-axiom where some literals in \(R(Y)\) are defined on the symbol \(r\) (e.g. axioms for \(\text{nooc}\) in \(S\)). An explicit iff-axiom is a non-recursive iff-axiom (e.g. axioms for \(\text{perm}\) in \(S\)). A descriptive iff-axiom is an iff-axiom where there exists some unbounded variable \(v\) such that \(v \in Y\) and \(v \notin X\) (e.g. axioms for \(\text{perm}\) in \(S\)). An operational iff-axiom is an iff-axiom where \(Y \subseteq X\) (e.g. axioms for \(\text{nooc}\) in \(S\)).

Definition 4 (Specifications and Contexts). A context \(C\) is a first-order theory including a set of type and relation specifications. Types are constructed from function symbols appearing in \(C\). A specification \(D_r\) is the set of all axioms in \(C\) with the same defined symbol \(r\). In the following, we use \(D_{r,n}\) to identify the \(n\)th axiom in \(D_r\). If each axiom in \(D_r\) is operational then \(D_r\) is called operational specification. If some axiom in \(D_r\) is descriptive then \(D_r\) is called descriptive specification. We assume that every recursive specification is well founded. If each axiom in \(D_r\) is an explicit axiom then \(D_r\) is called explicit specification. If some instances of \(r\) are not defined in \(D_r\) then \(D_r\) is called partial specification (e.g. \(D_{eq}\) in \(S\)) else \(D_r\) is called total specification (e.g. \(D_{Nat=}, D_{\text{Seq(Nat)=}}, D_{\text{nooc}}\) and \(D_{\text{perm}}\) in \(S\)). A context \(C\) is called operational iff each specification in \(C\) is operational. \(C\) is called descriptive iff there exists some descriptive specification in \(C\) (e.g. \(S\)). \(C\) is atomically complete if, for every ground atom \(r(t_1, \ldots, t_n)\), either \(C \vdash r(t_1, \ldots, t_n)\) or \(C \vdash \neg r(t_1, \ldots, t_n)\). \(C\) has isoinitial model \(M\) iff for every ground literal \(l\), \(M \models l\) iff \(C \vdash l\). Therefore, the meaning of a relation \(r\) in \(C\) is the set of all ground literals \(l\) defined on \(r\) such that \(C \vdash l\).

Some authors have studied the problem of the existence of isoinitial models for theories in general [BMM83] and some effective criteria have been proposed to construct consistent theories. Following [LaO94], a specification context \(C\) admits an isoinitial model if and only if it is atomically complete. By hypothesis, we assume that our contexts are consistent in this way.

3 Layered Specifications

Some preliminary definitions about term patterns and literal patterns are needed to introduce layered contexts.
Definition 5 (Term Pattern). A term pattern $P(t)$ for a term $t$ in the language of a context $C$ is defined inductively as follows: $P(v)$, for a variable $v$, is the set of all possible ground variable substitutions for $v$. $P(c)$, for a constant $c$, is the set \{c\}. $P(f(t_1, \ldots, t_n))$, for a term $f(t_1, \ldots, t_n)$, is the set of ground terms $\{f(m_1, \ldots, m_n)\}$ with $m_i \in P(t_i)$ ($i = 1..n$).

Example 3. $P(s(x))$ for $s(x)$ in the language of $S$ describes the set of ground terms $\{s(0), s(s(0)), s(s(s(0))), \ldots\}$.

Definition 6 (Relating Term Patterns). Let $t$ and $s$ be two terms in the language of $C$. We say that $P(t)$ and $P(s)$ are variant term patterns, denoted by $P(t) \sim P(s)$, if there exist substitutions $\theta$ and $\sigma$ such that $t = s\theta$ and $s = t\sigma$. We say $P(t)$ is more general than $P(s)$, denoted by $P(t) > P(s)$, if there exists a substitution $\sigma$ such that $P(t\sigma) \sim P(s)$ and there is no substitution $\theta$ such that $P(s\theta) \sim P(t)$. From a syntactical point of view, $P(t) > P(s)$ if either $s$ is a non-variable term and $t$ is a variable not included in $s$, or $t = f(t_1, \ldots, t_n)$ and $s = f(s_1, \ldots, s_n)$ and there exists a set of argument positions in $t$ whose term patterns are more general than the respective term patterns for arguments in $s$ and the rest of arguments represent similar term patterns.

Example 4. $P(\text{conc}(x, X)) > P(\text{conc}(y, \text{conc}(v, V)))$ because $t_1 = x, s_1 = y$ (i.e. $P(t_1) \sim P(s_1)$) and $t_2 = X, s_2 = \text{conc}(v, V)$ (i.e. $P(t_2) > P(s_2)$).

Definition 7 (Literal Pattern). A literal pattern $P(l)$ for a literal $l$ in the language of $C$ is defined as the set of all ground literals $l\sigma$ where $\sigma$ is a ground substitution.

Example 5. $P(\text{noocc}(x, \text{empty}, 0))$ for the literal $\text{noocc}(x, \text{empty}, 0)$ in the language of $S$ describes the set of ground literals $\{\text{noocc}(0, \text{empty}, 0), \text{noocc}(s(0), \text{empty}, 0), \text{noocc}(s(s(0)), \text{empty}, 0) \ldots\}$.

Definition 8 (Relating Literal Patterns). Let $l$ and $k$ be two literals in $C$. We say $P(l)$ and $P(k)$ are variant literal patterns, denoted by $P(l) \sim P(k)$, if there exist substitutions $\theta$ and $\sigma$ such that $l = k\theta$ and $k = l\sigma$. We say that $P(l)$ is a literal pattern more general than $P(k)$, denoted by $P(l) > P(k)$, if $l = r(t_1, \ldots, t_n)$ and $k = r(s_1, \ldots, s_n)$ and there exists a set of argument positions in $l$ whose term patterns are more general than the respective term patterns for arguments in $k$ and the rest of arguments represent similar term patterns. We say $P(l)$ is a literal pattern more general than $P(k)$ with respect to $\text{Pos}$ argument positions, denoted by $P(l) >_{\text{pos}} P(k)$, if and only if $P(t_i) > P(s_i)$ for each $i \in \text{Pos}$.

Example 6. $P(\text{noocc}(x, \text{empty}, 0)) \sim P(\text{noocc}(y, \text{empty}, 0))$ because $l\theta = k, k\sigma = l$ with $\theta = \{x, y\}$ and $\sigma = \{y, x\}$. $P(\text{noocc}(x, L, 0)) > P(\text{noocc}(y, \text{empty}, 0))$ because $P(x) > P(y), P(0) \sim P(0)$ and $P(L) > P(\text{empty}). P(\text{noocc}(0, L, 0)) >_{\{2\}} P(\text{noocc}(y, \text{empty}, 0))$ because $P(L) > P(\text{empty})$.

Definition 9 (Layered Contexts). Let $\text{Rel} = \{r_i\}$ ($i = 1..n$) be the set of all relation symbols declared in a context $C$. Let $D_{r_i} = \{r_i(X_j) \Leftrightarrow R_{i,j}(Y_j)\}$ ($j = 1..d$) be the specification of any $r_i \in \text{Rel}$. We say that $C$ is layered if...
1. \( C \) consists of three parts or layers. The data layer which contains sort declarations and function symbols. The operational layer which contains a set of relations defined by operational specifications. The descriptive layer which contains a set of relations defined by descriptive and explicit specifications.

2. For each pair of axioms in a specification, their lhs's must be either variant or disjoint literal patterns. Each atom defined on recursive symbol \( r \) and located in the rhs of an axiom not in \( D_r \) must be a variant literal pattern of some recursive atom in the rhs of some axiom in \( D_r \). Each atom, defined on descriptive symbol \( r \) and located in the rhs of an axiom not in \( D_r \) must be a variant literal pattern of some atom in the lhs of some axiom in \( D_r \).

3. There exists a mapping \( h \) from \( \text{Rel} \) to the set of natural numbers such that for each \( r_i \in \text{Rel}, h(r_i) \geq h(s_{i,j}), \) for every relation symbol \( s_{i,j} \) in \( R_{i,j}(Y_j) \)

A graphical interpretation of the mapping \( h \) is done by means of a dependency graph \( D(C) \). We say \( r_1 \) depends upon \( r_2 \) if and only if a literal defined on \( r_2 \) symbol appears in the rhs of some axiom in \( D_{r_1} \). A dependency graph for a relation \( r \), denoted by \( D(C, r) \) is a subgraph of \( D(C) \) which describes only dependencies for \( r \). In figure 1, we show the dependency graph for \( S \) and the dependency graph for \( \text{nocc} \) relation symbol in \( S \).

![Dependency graphs](image)

**Fig. 1.** Dependency graphs \( D(S) \) and \( D(S, \text{nocc}) \).

4 Transformation Method

This section describes a method which constructs operational contexts \( C^o \) from layered and descriptive contexts \( C \) preserving correctness. Sorts in \( C^o \) coincide with sorts in \( C \) and for each explicit and descriptive specification \( D_r \) an operational specification \( D^o_r \) is obtained such that \( C^o \models D_r \iff D^o_r \). A totalization process decides interpretations for partial relations. The operational context \( C^o \) is obtained by a succession of steps \( C_0 = C, \ldots, C_k = C^o \). Each step replaces
an explicit and descriptive \( D_\tau \) by an operational \( D_\tau^2 \). Each \( D_\tau^2 \) is obtained in an iterative and incremental way where the transformation is represented by a succession of steps (called transformation steps) \( D_0 = D_\tau, \ldots, D_n = D_\tau^2 \). Each step is a composition of a preprocessing step which constructs a transformation plan from the information in \( C_i \) and a processing step which develops this plan applying basic transformation rules. At each step, probably, new relations are added.

4.1 Totalizing Partial Specifications

Technically, a partial specification \( D_\tau \) does not define the semantics for some instances of \( r \). Our method handles only total specifications, therefore, the specifier must decide an interpretation for ambiguous literals. Let \( Mod \) be the set of all (isoinitial) models for \( C \) (i.e. the set of all models which decide interpretations for ambiguous instances of \( r \)). Let \( C |_{D_\tau} \) be the context obtained from \( C \) replacing \( D_\tau \) by \( D_{\tau^2} \) where \( D_{\tau^2} \) is a new specification which decides the semantics for undefined instances of \( r \). Let \( M^T \) be a model for \( C |_{D_\tau^2} \). We say \( D_{\tau^2} \) is a valid totalization for \( D_\tau \) in \( C \) iff \( M^T \in Mod \). Probably, a closed world assumption could be a reasonable (a priori) completion for partial relations but, technically, there exist many other ways to complete partial relations [LaO96]. We left open the mechanisms to do it.

**Example 7.** \( D_{eq} \) is a partial specification in \( S \). A (valid) totalization for \( D_{eq} \):

\[
D_{eq^T} = D_{eq} \cup \{ eq(\emptyset, \text{conc}(y,Y)) \Rightarrow \text{false}, \quad eq(\text{conc}(x,X), \emptyset) \Rightarrow \text{false}\}
\]

4.2 Basic Transformation Rules

The transformation of a descriptive specification into an operational specifications is obtained by the application of a set of basic transformation rules. These rules are defined in a constructive way. A detailed description, including a proof of the validity of the rules, can be found in [GaC01].

**Definition 10 (Unfolding Rule).** Let \( S \) be a formula in the language of \( C \). Let \( a \) be an atom in \( S \) defined on symbol \( r \) and let \( \{ r(X_j) \Leftrightarrow R_j(Y_j) \} \) (\( j = 1, \ldots, h \)) be a (sub)set of axioms in \( D_\tau \) such that the variables appearing only in \( R_j(Y_j) \) but not in \( r(X_j) \) do not appear in \( S \) and there exists a \( \sigma_j \) with \( r(x_j)\sigma_j = a \). We say that \( S_j \) is obtained from \( S \) unfolding \( a \) with respect to \( r(X_j) \Leftrightarrow R_j(Y_j) \) if and only if \( S_j = S \upharpoonright_{R_j(Y_j)\sigma_j} \) where \( S \upharpoonright_{R_j(Y_j)\sigma_j} \) represents the textual replacement of \( a \) by \( R_j(Y_j)\sigma_j \) in \( S \).

**Example 8.** Let \( S = \text{nooc}(a, \text{conc}(v,V), s(k)) \Leftrightarrow \text{nooc}(a, \text{conc}(w,W), s(k)) \) be a formula in the language of \( S \). Let \( \text{nooc}(a, \text{conc}(v,V), s(k)) \) be an atom in \( S \). We consider the subset of axioms for \( \text{nooc} \) \( \{ D_\text{nooc,2}, D_\text{nooc,3} \} \) with \( \text{lhs}(D_\text{nooc,2})\sigma_1 = \)
\[\text{nocc}(a, \text{conc}(v, V), s(k)) \text{ and } \text{lhs}(D_{\text{nocc}, 3}) \sigma_2 = \text{nocc}(a, \text{conc}(v, V), s(k)) \text{ and } \sigma_1 = \{(e, a), (x, v), (Y, V)(z, k)\}, \sigma_2 = \{(e, a), (x, v), (Y, V)(z, s(k))\}. \]

Then
\[S_1 = (v = a \land \text{noce}(a, V, k)) \iff \text{noce}(a, \text{conc}(w, W), s(m))\]
\[S_2 = (\neg v = a \land \text{noce}(a, V, s(k)) \iff \text{noce}(a, \text{conc}(w, W), s(m))\]
are obtained from \(S\) unfolding a w.r.t. \(D_{\text{noce}, 2}\) and \(D_{\text{noce}, 3}\) respectively.

The following definitions describe an automatic method to introduce recursion by a folding step. It is based on a special notion of similarity between formulas (see [GaC01] for a detailed explanation).

**Definition 11 (Parsing Tree).** Let \(S\) be a formula in prenex normal form. We say that \(\text{Parse}(S)\) (graphical example in figure 2) is the parsing tree for \(S\) if it is a tree representation of \(S\) where (a) each leaf node in \(\text{Parse}(S)\) represents a literal in \(S\), (b) each non-leaf node in \(\text{Parse}(S)\) represents either a quantified set of variables \(Q, x, \ldots, z\) or a logical connective \((\land, \lor, \Rightarrow, \Leftrightarrow)\) in \(S\) and (c) each node in \(\text{Parse}(S)\) has unique identification by means of a number with format \(lx \cdots xp\). The digit \(l\) represents the level where a node is located in \(\text{Parse}(S)\). The digit \(p\) decides if the node is located either at the left-hand side \((p = 1)\) or at the right-hand side \((p = 2)\) of its parent \((\text{if it exists})\). By default, nodes without brother nodes have \(p = 1\). The digits \(x \cdots x\) represent the identification of the parent node. The root node is an exception; it has not any parent therefore we consider a fixed identification for it equal to 1. In this way, a node identification determines univocally the position of a node in a parsing tree. We say that a preterminal node in \(\text{Parse}(S)\) is any non-leaf node in \(\text{Parse}(S)\) with at least one leaf node as child.

![Fig. 2. Parse(∀Nat\(a, z\)∀\(\text{Rel}(Nat)\)L, S(\(\text{noce}(a, L, z) \iff \text{noce}(a, S, z))\).](image)

Two formulas can be compared by the structure of their quantifiers and logical connectives. These measures are called similarity with respect to quantification and similarity with respect to logical connectives respectively. In the
following definitions, we consider that $S_1$ and $S_2$ are two formulas in prenex normal form.

**Definition 12 (Similarity Function).** We say that $f$ is a similarity function from the node identification domain of Parse($S_1$) to the node identification domain of Parse($S_2$) iff each non-leaf $n_1 \in$ Parse($S_1$) is mapped to a non-leaf node $n_2 = f(n_1) \in$ Parse($S_2$) where quantifier/connector in $n_1$ coincides with quantifier/connector in $n_2$ and the level of $n_2$ is greater than or equal to the level of $n_1$.

**Definition 13 (Similarity w.r.t. Quantification).** We say that $S_2$ is similar to $S_1$ w.r.t. quantification iff for each non-leaf node $n_1 \in$ Parse($S_1$) containing the quantified set of variables $Q_r x_1, ..., x_n$ there exists a non-leaf node $n_2 = f(n_1) \in$ Parse($S_2$) containing the quantified set of variables $Q_r y_1, ..., y_m$ such that (a) $m \geq n$ and (b) there exist two sequences of nodes, $M_1$ from Parse($S_1$) and $M_2$ from Parse($S_2$), with $M_1 = M_2^{-1}$ where $M_1$ contains $n_1$ and its predecessors (from bottom to up) and $M_2$ contains $n_2$ and its predecessors (from bottom to up). $M_1^{-1}$ is obtained by applying $f^{-1}$, when defined, to elements in $M_2$. If $S_2$ is similar to $S_1$ w.r.t. quantification then $f$ induces a set of possible renaming substitutions for variables in $S_1$ (from variables in $S_2$) which agree w.r.t. quantification. If $Q_r X$ is the set of quantified variables in $n_1$ and $Q_r Y$ is the set of quantified variables in $n_2 = f(n_1)$ then $f$ induces substitutions of the form $\{ (x_j, y_k) \}$ with $x_j \in X$ and $y_k \in Y$.

**Example 9.** In figure 3, $S_2$ is similar to $S_1$ w.r.t. quantification:

$$Q = \forall_{Nat} X = \{ a, z \} Y = \{ b, k, v, w \}$$

$$Q = \forall_{Seq(\text{Nat})} X = \{ L, S \} Y = \{ V, W \}$$

and some examples of substitutions induced by $f$ are:

$$\{ (a, b), (z, k), (L, V), (S, W) \} \{ (a, k), (z, v), (L, W), (S, V) \} ...$$

**Definition 14 (Similarity w.r.t. Logical Connectives).** We say that $S_2$ is in-depth similar to $S_1$ iff for each non-leaf node $n_1 \in$ Parse($S_1$) containing a logical connective there exists a non-leaf node $n_2 = f(n_1) \in$ Parse($S_2$) and there exist two sequences of nodes, $M_1$ from Parse($S_1$) and $M_2$ from Parse($S_2$), with $M_1 = M_2^{-1}$ where $M_1$ contains $n_1$ and its predecessors (from bottom to up) and $M_2$ contains $n_2$ and its predecessors (from bottom to up). We say that $S_2$ is in-breadth similar to $S_1$ iff for each level $l > 1$ of Parse($S_1$) with $N_{1,l} = \{ x_1 p_1, ..., x_k p_k \}$ as the set of all nodes in $l$ containing logical connectives, there exists a set of nodes in Parse($S_2$), possibly from several levels, say $l_1, ..., l_j$, of the form $N_{2,l_1, ..., l_j} = \{ \pi_1 f(x_1) p_{S_1}, ..., \pi_k f(x_k) p_{S_1} \}$ where $\pi_i$ and $\zeta_i (i = 1, k)$ are (sub)sequences of numbers. If the node with identification 1 (level $l = 1$) of Parse($S_1$) contains a logical connective then there exists a node identification in Parse($S_2$) of the form $\pi f(1) \zeta$ in Parse($S_2$) where $\pi$ and $\zeta$ are (sub)sequences of numbers. We say that $S_2$ is similar to $S_1$ w.r.t. logical connectives iff $S_2$ is in-depth similar and in-breadth similar to $S_1$. 

Transforming Layered Specifications into Operational Specifications 153
Definition 15 (Similarity). Let $S_2$ be similar to $S_1$ w.r.t. quantification and logical connectives by a function $f$. Let $L$ be the set of all literals in $S_1$. Let $NLeaf_1$ be the set of all preterminal nodes in $Parse(S_1)$. Let $NLeaf_2$ be the set of nodes $n_2 \in Parse(S_2)$ with $n_2 = f(n_1)$ and $n_1 \in NLeaf_1$. Let $Leaf_2$ be the set of leaf nodes in subtrees of $Parse(S_2)$ with root node $n_2 \in NLeaf_2$. We say that $S_2$ is similar to $S_1$ if there exist a $SLeaf_2 \subseteq Leaf_2$, with $K$ as the set of literals in nodes of $SLeaf_2$, and a substitution $\sigma$ induced by $f$ such that $L \sigma = K$.

![Diagram](image)

Fig. 3. An example of similarity.

Example 10. In figure 3, we show that $S_2 = (v = b \land nocc(b, V, k)) \Leftrightarrow (w = b \land nocc(b, W, k))$ is similar to $S_1 = nocc(a, L, z) \leftrightarrow nocc(a, S, z)$.

Similarity w.r.t to quantification:

$$f(1) = 1, M_1 = M_2^{-1} = \{1\}, f(211) = 211, M_1 = M_2^{-1} = \{211, 1\}$$

Similarity w.r.t. logical connectives (In-depth similarity).

$$f(32111) = 32111, M_1 = M_2^{-1} = \{32111, 211, 1\}$$

Similarity w.r.t. logical connectives (In-breadth similarity).

$$l = 3, f(211) = 211, N_{1,3} = \{32111\} \ N_{2,3} = \{32111\}$$

$$NLeaf_1 = \{32111\}, \ NLeaf_2 = \{32111\}$$

$$Leaf_2 = \{5432111111, 5432111112, 5432111121, 5432111122\}$$

$$SLeaf_2 = \{5432111112, 5432111122\}$$
Substitution $\sigma$ induced by $f$: $\{ (a,b),(z,k),(L,V),(S,W) \}$

$L = \{ \text{nocc}(a,L,z), \text{nocc}(a,S,z) \}, K = \{ \text{nocc}(b,V,k), \text{nocc}(b,W,k) \}$, $L\sigma = K$

**Definition 16 (Evaluation Rule).** Let $S(l_1, \ldots, l_p, l_{p+1}, \ldots, l_n)$ be a formula in the language of $C$ constructed from literals $l_1, \ldots, l_p, l_{p+1}, \ldots, l_n$.

We say that $S_{\text{eval}}(\{l_1, \ldots, l_p\})$ is obtained from $S$ evaluating the set of literals $\{l_1, \ldots, l_p\}$ iff $S_{\text{eval}}(\{l_1, \ldots, l_p\})$ is of the following form:

$$S_{\text{eval}}(\{l_1, \ldots, l_p\}) =$$

\[
(S(\text{true}, \text{true}, \ldots, \text{true}, l_{p+1}, \ldots, l_n) \land l_1 \land l_2 \land \ldots \land l_p) \lor \\
(S(\text{false}, \text{true}, \ldots, \text{true}, l_{p+1}, \ldots, l_n) \land \neg l_1 \land l_2 \land \ldots \land l_p) \lor \\
(S(\text{true}, \text{false}, \ldots, \text{true}, l_{p+1}, \ldots, l_n) \land l_1 \land \neg l_2 \land \ldots \land l_p) \lor \\
\vdots \\
(S(\text{false}, \text{false}, \ldots, \text{false}, l_{p+1}, \ldots, l_n) \land \neg l_1 \land \neg l_2 \land \ldots \land \neg l_p)
\]

**Definition 17 (Simplification Rules).** In order to simplify specifications, we consider simplification rules in presence of negations and false and true propositions. A formula $S(\text{true}, false, l_{p+1}, \ldots, l_n)$ constructed from literals $l_{p+1}, \ldots, l_n$ and propositions true, false is transformed into the formula $S_{\text{simpl}}$ by application of simplification rules repeatedly (the set of rules can be found in [GaCU]).

**Example 11.** Let $S = (false \land nocc(b,V,k)) \iff (true \land nocc(b,W,k))$ be a formula. $S_{\text{simpl}} = \neg nocc(b,W,k)$ represents the simplified form of $S$.

**Definition 18 (Similarity Based Folding Rule).** Let $S$ be a formula in the language of $C$. Let $r$ be a relation in $C$. Let $D_{t,i} = r(X_t)$ be the $j$th axiom in $D_r$. The variables appearing only in $R_j(Y_j)$ but not in $r(X_t)$ do not appear in $S$. Let $S$ be similar to $R_j(Y_j)$ w.r.t. quantification and logical connectives. Let $L_t$ be the set of all literals in $R_j(Y_j)$ and $K_t$ the set of all literals in $S$. Let $\{K_{k,t}\}$ be the set of disjoint (sub)sets of literals in $S$ and $\theta_{j,k}$, substitutions such that $L_t\theta_{j,k} = K_{j,k}$ ($k = 1 \ldots a$).

We say that $S_{\text{folding}}$ is obtained from $S$ folding by similarity with respect to $D_{t,i}$ iff $S_{\text{folding}} = (S_{\text{eval}}(\{K_{k,t}\}))_{\text{simpl}}\theta_{j,k}$

where $S_{\text{eval}}(\{K_{k,t}\})_{\text{simpl}}$ is obtained from $S$ evaluating (definition 16) literals not in $\bigcup_{k=1..a} K_{j,k}$ and then applying simplification rules (definition 17).

**Example 12.** Let $S = (v = b \land nocc(b,V,k)) \iff (w = b \land nocc(b,W,k))$ be a formula in the language of $S$. Let $K = \{v = b, nocc(b,V,k), w = b, nocc(b,W,k)\}$ be the set of all literals in $S$. S is similar to $\text{rhs}(D_{\text{perm},1})$ w.r.t. quantification and logical connectives. Let $L_1 = \{nocc(a,L,z), nocc(a,S,z)\}$ be the set of all literals in the $\text{rhs}(D_{\text{perm},1})$ and let $\{K_{1,1}\}$ be the set of disjoint (sub)sets of literals in $S$ such that $K_{1,1} = \{nocc(b,V,k), nocc(b,W,k)\}$ and $L_1\theta_{1,1} = K_{1,1}$ with $\theta_{1,1} = \{(L,V),(S,W),(z,k),(a,b)\}$. Then...
\[ S_{\text{eval}}(K - \{K_1, 1\}) = \]
\[
\{ \text{true} \land \text{nooc}(b, V, k) \iff \text{true} \land \text{nooc}(b, W, k) \} \land v = b \land w = b \lor
\{ \text{false} \land \text{nooc}(b, V, k) \iff \text{true} \land \text{nooc}(b, W, k) \} \land \neg v = b \land w = b \lor
\{ \text{true} \land \text{nooc}(b, V, k) \iff \text{false} \land \text{nooc}(b, W, k) \} \land v = b \land \neg w = b \lor
\{ \text{false} \land \text{nooc}(b, V, k) \iff \text{false} \land \text{nooc}(b, W, k) \} \land \neg v = b \land \neg w = b
\]

is obtained from \( S \) evaluating literals not in \( \{K_1, 1\} \) (i.e. \( v = b \) and \( w = b \)). Applying (repeatedly) simplification rules:

\[
S_{\text{eval}}(K - \{K_1, 1\})_{\text{simp}} = (\text{nooc}(b, V, k) \iff \text{nooc}(b, W, k)) \land v = b \land w = b \lor
\neg \text{nooc}(b, W, k) \land \neg v = b \land w = b \lor
\neg \text{nooc}(b, V, k) \land v = b \land \neg w = b \lor
\neg v = b \land \neg w = b
\]

Considering \( (\text{nooc}(a, L, z) \iff \text{nooc}(a, S, z)) \theta_{1,1} = \text{nooc}(b, V, k) \iff \text{nooc}(b, W, k) \) and \( \text{perm}(L, S) \theta_{1,1} = \text{perm}(V, W) \) then

\[
(S_{\text{eval}}(K - \{K_1, 1\})_{\text{simp}} |_{\text{perm}(V, W)}) =
\begin{align*}
\text{perm}(V, W) & \land v = b \land w = b \lor \\
\neg \text{nooc}(b, W, k) & \land \neg v = b \land w = b \lor \\
\neg \text{nooc}(b, V, k) & \land v = b \land \neg w = b \lor \\
\neg v = b \land \neg w = b
\end{align*}
\]

is obtained from \( S \) folding by similarity w.r.t. \( D_{\text{perm}, 1} \).

**Definition 19 (Substitution Rules).** Let \( S \) be a formula in the language of \( C \). Let \( A \) be the set of all atoms in \( S \) with arguments containing a variable \( x \) of type \( \tau \). Let \( \sigma_j = (x, t_j(y)) \) be a variable substitution where \( t_j \) is a function symbol which generates \( \tau \). For each atom \( a \in A \) defined on symbol \( r \) there exists at least an axiom \( r(X_j) \iff R_j(Y_j) \) in \( D_r \) with \( P(a) \succ Pos(x) \) \( P(r(X_j)) \) where Pos(\( x \)) denotes the set of argument positions in a containing \( x \).

We say \( S_j \) is obtained from \( S \) by universal substitution of \( x \) iff \( S_j = S \sigma_j \). We say \( S_j \) is obtained from \( S \) by existential substitution of \( x \) iff \( S_j = \exists y(S \sigma_j) \).

**Example 13.** Let \( S = \text{nooc}(a, V, k) \iff \text{nooc}(a, W, k) \) be a formula in the language of \( S \) and \( W \) be a variable of type Seq(Nat) in \( S \). Let \( A = \{ \text{nooc}(a, W, k) \} \) be the set of all atoms in \( S \) with arguments containing \( W \). Let \( \sigma_1 = (W, \text{empty}) \) and \( \sigma_2 = (W, \text{conc}(x, Y)) \) be the substitutions for \( W \). Considering

\[
P(\text{nooc}(a, W, k), \text{nooc}(a, W, k)) >_{(2)} P(\text{lh}(D_{\text{nooc},1}), \text{lh}(D_{\text{nooc},2}))
\]

\[
P(\text{nooc}(a, W, k), \text{nooc}(a, W, k)) >_{(2)} P(\text{lh}(D_{\text{nooc},3}))
\]

\[
S_1 = S \sigma_1 = \text{nooc}(a, V, k) \iff \text{nooc}(a, \text{empty, k})
\]

\[
S_2 = S \sigma_2 = \text{nooc}(a, V, k) \iff \text{nooc}(a, \text{conc}(x, Y), k)
\]

are obtained from \( S \) by universal substitution of \( W \).
4.3 Transformation Plan

The transformation of a descriptive specification into an operational specification is represented by means of a set of transformation trees. Each transformation tree is constructed by the application of basic transformation rules. However, an application without any control may cause the construction of infinite trees. A transformation plan is needed to guide transformations. The plan consists of a decomposition plan which guides the use of unfolding and substitution rules followed by a composition plan which guides the use of the similarity based folding rule.

**Definition 20 (Transformation Tree).** In a transformation tree, Tree($r$), the root node contains the literal $r(X)$ where each argument in $X$ is a variable and the rest of nodes are obtained by applying transformation rules. Each arc in Tree($r$) connects a node in the level $l$ with a node in the level $l+1$ and it is annotated with the name of a basic transformation rule and substitutions. A node consists of an identification id (Definition 11), a formula $F_{id}$ and a meta-variable result$_id$ needed to register the form of the resulting operational specification. Possible values for result$_id$ are true, false, a literal, a disjunction/conjunction of other result meta-variables (Figure 4). A transformation branch for $r$ from a node $n$, denoted by Branch($r, n$), is the sequence of node identifications in Tree($r$) starting at the root node and finishing at the node $n$. Rules such as unfold, fold and substitution rules may produce several branches. In order to preserve semantics it is needed to explore each branch (Figure 4). Meta-variables consider this fact in the following manner: If unfolding/folding/existential substitution rule has been applied to the node $n_{id}$ then result$_id$ is equal to the disjunction of meta-variables for child nodes of $n_{id}$. If universal substitution has been applied to $n_{id}$ then result$_id$ is equal to the conjunction of meta-variables for child nodes of $n_{id}$. After completion of Tree($r$), a bottom-up evaluation of these meta-variables, propagating substitutions, constructs in result$_1$ the operational specification.

**Definition 21 (Loop-Detection).** In order to prevent the construction of infinite trees, our method restricts the application of substitution rules in the context of a branch. Let Branch($r, n_p$) = $\{n_1, \ldots, n_p\}$ be a branch in Tree($r$) from node $n_p$ and let $\{\sigma_1, \ldots, \sigma_p\}$ be the set of substitutions in such branch of Tree($r$). Then, it is allowed to apply variable substitution $\sigma_p = (v, t)$ in $F_p$ (formula in node $n_p$) iff, after substitution, for each affected atom $a$ defined on recursive symbol $r$ in $F_i$ ($i = 1..p$) there is not any axiom $r(X_i) \Rightarrow R_j(Y_j)$ in $C$ with $P(r(X_i)) > P(\sigma_a \circ \ldots \circ \sigma_p)$. This restriction prevents the unbounded application of substitution rules.

**Example 14.** In Figure 4, it is not allowed to apply universal substitution on $X$ in the node 32112 because, after substitution, the literal nocc($a$, conc($x$, $X$), $z$) in $F_{32112}$ is transformed into nocc($a$, conc($x$, empty), $z$) in $F_{132112}$ and nocc($a$, conc($x$, conc($x$, empty), $z$)) in $F_{4321122}$ respectively and there are some axioms in
Fig. 4. Example of transformation tree for relation \( \text{perm} \)

\[ D_{\text{nocc}} \text{ such as } D_{\text{nocc,2}} \text{ and } D_{\text{nocc,3}} \text{ with} \]
\[ P(\text{nocc}(e, \text{conc}(x, Y), s(z))) >_2 P(\text{nocc}(a, \text{conc}(x, \text{empty}), z)) \]
\[ P(\text{nocc}(e, \text{conc}(x, Y), s(z))) >_2 P(\text{nocc}(a, \text{conc}(x, \text{conc}(u, V)), z)) \]

**Definition 22 (Decomposition Plan).** Let \( r \) be a relation in \( C \) with descriptive specification \( D_r \). A decomposition plan for \( r \), denoted by \( \text{Plan}_{\text{dec}}(C, r) \), establishes which relation symbols are able to be unfolded and substituted in the transformation of \( D_r \). The decomposition plan prioritizes the use of the unfolding rule in situations where it is also possible to apply substitution rules. Our representation of \( \text{Plan}_{\text{dec}}(C, r) \) is by means of directed graphs. A relation symbol \( s \) is included in \( \text{Plan}_{\text{dec}}(C, r) \) iff \( s \) is included in a node of \( D(C, r) \) and there is not any operational relation symbol between \( s \) and \( r \) in \( D(C, r) \) (considering all paths between \( r \) and \( s \)).

**Example 15.**
\[ \text{Plan}_{\text{dec}}(S, \text{eq}) = \{ \text{eq, perm, nocc} \}, \text{Plan}_{\text{dec}}(S, \text{perm}) = \{ \text{perm, nocc} \} \]

**Definition 23 (Composition Plan).** A composition plan for \( r \) with descriptive specification \( D_r \) in \( C \), denoted by \( \text{Plan}_{\text{com}}(C, r) \), represents the sequence of formulas to be used as reference for the similarity-based folding rule during the transformation of \( D_r \). The similarity-based folding rule makes only a search of formulas which are similar to formulas in \( \text{Plan}_{\text{com}}(C, r) \). Formulas in \( \text{Plan}_{\text{com}}(C, r) \) are obtained automatically by successive unfolding of the lhs’ of axioms in \( D_r \) only if such literals are defined on descriptive relation symbols in
Transforming Layered Specifications into Operational Specifications

$\text{Plan}_{\text{dec}}(C, r)$. Only non-trivial formulas (i.e., different to true and false) are considered in $\text{Plan}_{\text{com}}(C, r)$.

**Example 16.**

$$\text{Plan}_{\text{com}}(S, \text{eq}) = \{ \exists Z ((\text{perm}(nL, Z) \land nL = \text{conc}(x, X)))$$

$$\iff (\text{perm}(Z, nS) \land nS = \text{conc}(x, X)),$$

$$\exists Z (\text{nocc}(a, nL, z) \iff \text{nocc}(a, Z, z)),$$

$$\exists Z (\text{nocc}(a, Z, z) \iff \text{nocc}(a, nS, z)) \}$$

$$\text{Plan}_{\text{com}}(S, \text{perm}) = \{ \text{nocc}(a, L, z) \iff \text{nocc}(a, S, z) \} \}$$

After the execution of a transformation plan and due to unfold/fold and loop-detection restrictions, a finite tree is constructed. Leaf nodes in the tree contain either a trivial formula (i.e., true or false) (complete transformation) or a non-trivial formula (incomplete transformation). Each descriptive formula in a leaf node represents the specification of a new relation symbol to be transformed in subsequent transformation steps. Each new symbols (e.g., $r_i$) and their respective specifications (e.g., $D_{r_i}$) extend the initial context in a conservative way.

**Example 17.** The transformation of $D_{\text{perm}}$ produces the following (new) relations:

1. $\text{perm}_1(\text{empty}, \text{empty}) \iff (0 = 0 \iff 0 = 0)$
2. $\text{perm}_2(\text{empty}, \text{empty}) \iff (s(k) = 0 \iff s(k) = 0)$
3. $\text{perm}_3(\text{empty}, \text{conc}(y, Y)) \iff (0 = 0 \iff (\neg y = a \land \text{nocc}(a, Y, 0)))$
4. $\text{perm}_4(\text{empty}, \text{conc}(y, Y)) \iff (s(k) = 0 \iff (y = a \land \text{nocc}(a, Y, k)))$
5. $\text{perm}_5(\text{empty}, \text{conc}(y, Y)) \iff (s(k) = 0 \iff (\neg y = a \land \text{nocc}(a, Y, s(k)))$
6. $\text{perm}_6(\text{conc}(x, X), \text{empty}) \iff (x = a \land \text{nocc}(a, X, 0)) \iff 0 = 0$
7. $\text{perm}_7(\text{conc}(x, X), \text{empty}) \iff (\neg x = a \land \text{nocc}(a, X, k)) \iff 0 = 0$
8. $\text{perm}_8(\text{conc}(x, X), \text{empty}) \iff (\neg x = a \land \text{nocc}(a, X, s(k))) \iff 0 = 0$
9. $\text{perm}_9(\text{conc}(x, X), \text{conc}(y, Y)) \iff (\neg x = a \land \text{nocc}(a, X, 0))$
10. $\text{perm}_{10}(\text{conc}(x, X), \text{conc}(y, Y)) \iff (\text{perm}(X, Y) \land x = a \land y = a) \lor$

$$\neg \text{nocc}(a, X, k) \land x = a \land \neg a = y) \lor$$

$$\neg \text{nocc}(a, Y, k) \land \neg x = a \land y = a) \lor$$

$$\neg x = a \land \neg y = a)$$
11. $\text{perm}_{11}(\text{conc}(x, X), \text{conc}(y, Y)) \iff (x = a \land \text{nocc}(a, X, k)$

$$\iff \neg y = a \land \text{nocc}(a, Y, s(k))$$
12. $\text{perm}_{12}(\text{conc}(x, X), \text{conc}(y, Y)) \iff (\neg x = a \land \text{nocc}(a, X, s(k))$

$$\iff y = a \land \text{nocc}(a, Y, k))$$
13. $\text{perm}_{13}(\text{conc}(x, X), \text{conc}(y, Y)) \iff (\neg x = a \land \text{nocc}(a, X, s(k))$

$$\iff \neg y = a \land \text{nocc}(a, Y, s(k))$$
Eventually, the transformation process of a new relation will yield formulas which are similar to formulas (already) within the context. Therefore, no infinite new relations will be proposed during the process.

**Example 18.** The transformation of \( \text{perm}_9 \) produces a formula similar to the right-hand side of the axiom for \( \text{perm}_3 \), the transformation of \( \text{perm}_6 \) produces a formula similar to the right-hand side of the axiom for \( \text{perm}_6 \), and so on.

Finally, a bottom-up evaluation of meta-variables \( \text{result} \), propagating variable substitutions, completes the structure of the operational specification.

**Example 19.** A bottom-up evaluation for meta-variables in \( \text{Tree}(\text{perm}) \) produces the operational specification \( D^o_{\text{perm}} \):

\[
\begin{align*}
\text{perm}(\text{empty}, \text{empty}) & \Leftrightarrow (\text{perm}_1(\text{empty}, \text{empty}) \land \text{perm}_2(\text{empty}, \text{empty})) \\
\text{perm}(\text{empty}, \text{conc}(y, Y)) & \Leftrightarrow (\text{perm}_3(\text{empty}, \text{conc}(y, Y)) \land \\
& (\text{perm}_4(\text{empty}, \text{conc}(y, Y)) \lor \text{perm}_5(\text{empty}, \text{conc}(y, Y)))) \\
\text{perm}(\text{conc}(x, X), \text{empty}) & \Leftrightarrow (\text{perm}_6(\text{conc}(x, X), \text{empty}) \land \\
& (\text{perm}_7(\text{conc}(x, X), \text{empty}) \lor \text{perm}_8(\text{conc}(x, X), \text{empty}))) \\
\text{perm}(\text{conc}(x, X), \text{conc}(y, Y)) & \Leftrightarrow (\text{perm}_9(\text{conc}(x, X), \text{conc}(y, Y)) \land \\
& (\text{perm}_{10}(\text{conc}(x, X), \text{conc}(y, Y)) \lor \text{perm}_{11}(\text{conc}(x, X), \text{conc}(y, Y)) \lor \\
& \text{perm}_{12}(\text{conc}(x, X), \text{conc}(y, Y)) \lor \text{perm}_{13}(\text{conc}(x, X), \text{conc}(y, Y))))
\end{align*}
\]

5 Conclusions

A transformation method from descriptive specifications has been proposed. The language of the layered contexts constitutes a framework with reasonable expressivity in order to establish automatic transformations. The only manual activity of our method is represented by the totalization of partial relations. Following the classification in [Fle95], our transformation method can be classified as a deductive synthesis method. The constructive definition of our calculus, the automatic generation of transformation plans, and the finite transformations within layered contexts represent key properties in order to consider our method amenable to be compiled and to be executed in practice.

References


Abstract Satisfiability of Linear Temporal Logic

María del Mar Gallardo, Pedro Merino, Ernesto Pimentel

Dpto. de Lenguajes y Ciencias de la Computacion
University of Málaga, 29071 Málaga, Spain
{gallardo,pedro,ernesto}@lcc.uma.es

Abstract. Model Checking has become one of the most powerful methods for automatic verification of software systems. But this technique is only directly applicable to small or medium size systems. For large systems, it suffers from the state explosion problem. One of the most promising ways to solve this problem is the use of Abstract Interpretation to construct simpler models of the system, where the interesting properties can be analyzed. In this paper, we present a theoretical language-independent framework to assist in the definition of correct abstractions for the verification of temporal properties expressed with LTL. The results can be adapted to a number of real modeling languages. In the paper, we present the application of the results to the very popular verification system SPIN, which uses PROMELA as the modeling language. Several completeness and consistency conditions are studied to preserve the (abstract) satisfiability of properties.

1 Introduction

Model Checking [4, 1] represents one of the most useful results of almost twenty years of research in formal methods to increase the quality of software and other related systems. A model checker works with a high level description of a system, a model, and it can automatically inspect the reachable states of the system to check if a given property can be satisfied. Typically, properties are expressed with some variant of temporal logic, Linear time Temporal Logic (LTL) being one of the most employed [15].

Despite the variety of model checkers and formalisms for representing models and properties, existing tools suffer from the state explosion problem when dealing with realistic systems. Among other approaches, the construction of more reduced (abstract) models using the ideas of Abstract Interpretation [2] is becoming a promising way of obtaining information about the satisfaction of formulas without having to inspect the whole state space [6, 14, 7, 8, 14, 3]. In fact, some works have already been oriented to dealing with this problem in several contexts. This is the case of [17], where authors provide a methodology in which abstract interpretation and model checking are combined to carry out flow analysis of programs. A similar objective is considered in [3], where abstract interpretation of temporal calculi and logics is studied. Our approach differs from these other works in several aspects. From a theoretical point of view, we study
the relation between satisfiability of a LTL formula on a model and on its abstraction, giving conditions to ensure the satisfiability preservation. This is made by introducing a notion of non-standard satisfiability, which has not been studied previously. On the other hand, and from a more practical point of view, the way we relate the standard semantics of a model and its abstraction allows us a natural implementation of abstraction by source-to-source transformation, which avoids using different verification tools to analyze a model and its abstraction.

In order to apply abstract interpretation to a concurrent system, we assume it is modeled by a transition system, which is abstracted by obtaining a simpler (and easier to be analyzed) transition system. Of course, to maintain the "relevant" features of the original transition system, the abstracted one has to simulate it in some way, preserving the observable behavior. After the abstraction of a concurrent model, another problem is the relationship between properties in both models (the concrete and the abstract ones). In fact, most of the efforts of this paper are directed to give conditions under which the analysis of properties in a system may be delegated to the analysis of the corresponding "abstracted" properties. An exhaustive study of these conditions is presented, motivating them with several examples, in the context of LTL.

We consider a reference interpretation of the original model, and a non-standard interpretation is obtained from an abstract interpretation. Applying the same technique, we employ two different notions of temporal formula satisfaction: the standard one, determined by the reference interpretation of the propositions, and the non-standard obtained by abstracting the meaning of the logic propositions. Non-standard satisfaction relies on weak Kripke structures, which relax the principle of non-contradiction, thus allowing an (abstract) state to satisfy both a proposition \( p \) and its negation \( \neg p \). These weak Kripke structures move us into the field of paraconsistent logics [16], which are an alternative to classical logic, that allows some contradictions to be true without incurring on the trivialization of classical logic. There are already some works where model checking is applied to these kinds of logics [5].

The problems induced by contradictions are limited by adding consistency and completeness conditions when defining the corresponding abstract notion of satisfiability. In doing so, some practical results about the behaviour of the initial (concrete) systems are obtained. A major contribution of our approach is its ability to be easily instantiated for particular modeling languages. For this purpose, we have to use the labeled transition system semantics of the language, which is usually available in the documentation of model checkers. For example, instances of the correctness conditions and practical results can be found for Promela/Spin [9], and for SDL/MSC [10].

The paper is organized as follows. First, we outline a brief introduction to abstract interpretation. Section 3 is devoted to showing how abstract interpretation can be made on generic transition systems, and Section 4 presents several results concerning the relation between the satisfaction of temporal logic formulas in a model and its abstraction. Section 5 shows an example of the application of the overall theoretical framework to environment Promela/Spin[12].
2 Abstract Interpretation

Abstract interpretation [2] is a static analysis technique that defines a relation between two semantic levels: the concrete and the abstract one. Let $\text{Std}_L$ be a semantics of a programming language $L$ which associates the standard interpretation $\text{Std}_L(P) \in D$ to each program $P$ of $L$, where $D$ denotes the set of concrete denotations. The objective of abstract interpretation is to automatically realize a finite approximation of $\text{Std}_L$, $\text{Std}_L^\alpha : L \to D^\alpha$, such that $\text{Std}_L^\alpha(P)$ is computable and gives us some information about whether some property is fulfilled by the execution of $P$. $\text{Std}_L^\alpha(P)$ is generally obtained by an abstract execution of $P$ that uses data representations (abstractions) instead of the original data. Abstraction always implies some loss of information but instead, applying it, we achieve computability of the analyzed properties.

Abstraction is achieved by means of a pair of functions $\alpha : D \to D^\alpha$ and $\gamma : D^\alpha \to D$ between the sets $(D, \sqsubseteq)$ and $(D^\alpha, \sqsubseteq^\alpha)$ which constitute a Galois connection, that is: $\forall d \in D, d^\alpha \in D^\alpha : \alpha(d) \sqsubseteq^\alpha d^\alpha \Leftrightarrow d \sqsubseteq \gamma(d^\alpha)$.

Functions $\alpha$ and $\gamma$ are called abstraction and concretization functions, respectively. Usually, the values of the abstract domain $D^\alpha$ only capture the information needed for the corresponding analysis. Abstraction function $\alpha$ projects standard data on these values, thus reducing the size of the semantic domain. The partial order $\sqsubseteq^\alpha$ represents the precision degree defined by the abstract semantics: $d^\alpha_1 \sqsubseteq^\alpha d^\alpha_2$ indicates that the abstract property $d^\alpha_1$ is more precise than $d^\alpha_2$, or from the dual point of view, that $d^\alpha_2$ approximates $d^\alpha_1$. Thus, the relation $\alpha(d) \sqsubseteq^\alpha d^\alpha$ means that $d^\alpha$ is an abstraction and an approximation of $d$. With this notation, the objective of abstract interpretation is to find an approximation $\text{Std}_L^\alpha$ of $\text{Std}_L$ such that for each program $P$ of $L$, $\alpha(\text{Std}_L(P)) \sqsubseteq^\alpha \text{Std}_L^\alpha(P)$.

3 Abstracting Concurrent Systems

In this section we apply abstract interpretation to transition systems, in a general way, proving an important result on behavior preservation between a model and its abstract version.

3.1 Modeling Concurrent Systems

Execution of concurrent programs\footnote{We will use the terms programs, systems and models indifferently.} may be defined by means of labeled transition systems such as $M = (A, \Sigma, \rightarrow, s_0)$, where

- $A$ is the set of observable atomic actions.
- $\Sigma$ is the set of states.
- $\rightarrow \subseteq \Sigma \times A \times \Sigma$ is a labeled transition relation. We write $s \rightarrow a s'$ for $(s, a, s') \in \rightarrow$.
- $s_0 \in \Sigma$ is the initial state.
$M$ defines an interpretation of actions over the set of states $\Sigma$: each transition $s \xrightarrow{a} s'$ means that the execution of the atomic action $a$ would transform the system from state $s$ to state $s'$ in a single step. A trace $x$ of $M$ is a sequence $x = t_0 \xrightarrow{a_0} t_1 \xrightarrow{a_1} t_2 \xrightarrow{a_2} \ldots$ of states. Trace $x$ represents a possible computation from the state $t_0$ and $a_0, a_1, \ldots \in A^*$ the sequence of atomic actions executed. Given a trace $x = t_0 \xrightarrow{a_0} t_1 \xrightarrow{a_1} \ldots$ by $x^j$ we will denote the suffix trace $t_j \xrightarrow{a_j} t_{j+1} \xrightarrow{a_{j+1}} \ldots$.

A trace $x = s_0 \xrightarrow{a_0} \ldots$ starting at initial state $s_0$ is called full-trace iff either $x$ is a non terminating trace or it ends at a state from which no transitions evolve.

Operational semantics of programming languages are defined by means of labeled transition systems where the initial state is not fixed. The semantics of a particular program is obtained from an initial state with information about its code. Thus, a transition system with a fixed initial state defines the behaviour of a given program. The set $\mathcal{O}(M) = \{x : x = s_0 \xrightarrow{a_0} \ldots \text{ is a full trace}\}$ defines the trace semantics determined by the transition system $M$.

Let $M_1 = (A, \Sigma_1, \rightarrow_1, s_0^1)$ and $M_2 = (A, \Sigma_2, \rightarrow_2, s_0^2)$ be transition systems giving two different interpretations to the set of actions $A$. Given $\rho \subseteq \Sigma_1 \times \Sigma_2$ a relation from $\Sigma_1$ to $\Sigma_2$, we say that $M_2$ is a $\rho$-simulation of $M_1$ iff (1) $\rho(s_0^1, s_0^2)$ and (2) if $s_1 \xrightarrow{\gamma_1} s_1'$ and $\rho(s_1, s_2)$ then there exists some $s_2' \in \Sigma_2$ such that $s_2 \xrightarrow{\gamma_2} s_2'$ and $\rho(s_1', s_2')$.

### 3.2 Abstract Interpretation of Transition Systems

Let $\text{Std} = (A, \Sigma, \rightarrow, s_0)$ and $\text{Std}^a = (A, \Sigma^a, \rightarrow_a, s_0^a)$ be two labeled transition systems. Let $I_\alpha = (\Sigma, (\Sigma^a, \leq^a), \alpha, \gamma)$ be an abstract interpretation of the set of states $\Sigma$, where $\leq^a$ is a partial order, $\alpha : \Sigma \rightarrow \Sigma^a$ is the abstraction function and $\gamma : \Sigma^a \rightarrow 2^\Sigma$, defined by $\gamma(s^a) = \{s \in \Sigma : \alpha(s) \leq^a s^a\}$, is the concretization function. From now on, we assume that $\text{Std}$ gives the reference meaning, called standard, to the set of actions $A$. In contrast, the interpretation of actions given by $\text{Std}^a$ is intended to represent some abstraction of $\text{Std}$. Thus, we will apply the terms standard/abstract when we refer to the elements of $\text{Std}/\text{Std}^a$.

**Definition 1 (Correctness).** Given $I_\alpha = (\Sigma, (\Sigma^a, \leq^a), \alpha, \gamma)$ an abstract interpretation, we say that $\text{Std}^a$ is correct under the interpretation $I_\alpha$, with respect to $\text{Std}$ iff $\text{Std}^a$ is a $\rho$-simulation of $\text{Std}$, $\rho$ being the binary relation defined as $\rho(s, s^a) \Leftrightarrow \alpha(s) \leq^a s^a$. That is, $\text{Std}^a$ is correct under $I_\alpha$ with respect to $\text{Std}$ iff (1) $\alpha(s_0) \leq^a s_0^a$ and (2) if $s \xrightarrow{a} s'$ and $\alpha(s) \leq^a s^a$ then there exists some $s'^a \in \Sigma^a$ such that $s^a \xrightarrow{a} s'^a$ and $\alpha(s') \leq^a s'^a$.

Given two infinite traces $x = t_0 \xrightarrow{a_0} t_1 \xrightarrow{a_1} \ldots$ and $x^a = t_0^a \xrightarrow{a_0^a} t_1^a \xrightarrow{a_1^a} \ldots$, we write $\alpha(x) \leq^a x^a$ when $\forall i \geq 0. \alpha(t_i) \leq^a t_i^a$.

**Theorem 1.** Given $\text{Std} = (A, \Sigma, \rightarrow, s_0)$, $\text{Std}^a = (A, \Sigma^a, \rightarrow_a, s_0^a)$ and the abstract interpretation $I_\alpha$, if $\text{Std}^a$ is correct under $I_\alpha$ wrt $\text{Std}$ then for every infinite full-trace $x \in \mathcal{O}(\text{Std})$ there exists $x^a \in \mathcal{O}(\text{Std}^a)$ such that $\alpha(x) \leq^a x^a$.

**Proof.** It clearly follows from the definition of correctness.
Note that Theorem 1 only assures the simulation of infinite traces. In order to simulate ending \( a \) (deadlocked/finished) traces, we must impose the following additional condition in the definition of correctness: (3) if \( s \not\in \mathcal{A} \) and \( \alpha(s) \leq_a s^a \) then \( s^a \not\in \mathcal{A} \). This sufficient condition becomes also necessary when \( s \) represents only reachable states in full-traces.

**Example 1.** Let \( \text{Std} = (A, \Sigma, \rightarrow, s_0) \) be a labeled transition system that defines the behavior of a non-deterministic program \( P \), with one variable \( y \) ranging over \( \text{Int} \), the set of the integer numbers, as follows:

- \( A = \{ y = y + 2, \text{skip} \} \).
- \( \Sigma = \text{Int} \times \{ \text{start, end} \} \). A standard state \( s = (s_y, q) \in \Sigma \) defines the current value \( s_y \) stored in \( y \) and the program counter \( q \).
- \( \rightarrow \) is defined as: \((s_y, \text{start}) \xrightarrow{y-y+2} (s_y + 2, \text{start}) \), \((s_y, \text{start}) \xrightarrow{\text{skip}} (s_y, \text{end}) \).
- \( s_0 = (0, \text{start}) \).

Elements in \( \mathcal{O}(\text{Std}) \) are finite traces like \((0, \text{start}) ; \xrightarrow{y-y+2} (2, \text{start}) ; \xrightarrow{y-y+2} \ldots ; \xrightarrow{\text{skip}} (2n, \text{end}) \) and the non-terminating trace \((0, \text{start}) ; \xrightarrow{y-y+2} (2, \text{start}) ; \xrightarrow{y-y+2} \ldots \).

Let \( (\text{Int}_y^\alpha, \leq_y^\alpha) \) be defined by \( \text{Int}_y^\alpha = \{ \text{even, odd} \} \) and the partial ordering \( \leq_y^\alpha \) over this domain being the relation “\( \approx \)”. The classic even-odd abstraction \( \alpha_y : \text{Int} \rightarrow \text{Int}_y^\alpha \), defined as \( \alpha_y(2n) = \text{even} \) and \( \alpha_y(2n + 1) = \text{odd} \) for all \( n \geq 0 \), determines an abstraction \( \alpha : \Sigma \rightarrow (\Sigma^\alpha, \leq^\alpha) \) of standard states over the abstract domain \( \Sigma^\alpha = \text{Int}_y^\alpha \times \{ \text{start, end} \} \) as follows: \( \alpha(s) = (\alpha_y(s_y), q) \). The partial ordering \( \leq^\alpha \) over \( \Sigma^\alpha \) is defined as: \((s_y^\alpha, q^\alpha) \leq^\alpha (s_y^\prime, q^\prime) \) if \( s_y = s_y^\prime \) and \( s_y^\alpha \leq^\alpha s_y^\prime \).

Then, the labeled transition system \( \text{Std}^\alpha = (A, \Sigma^\alpha, \rightarrow^\alpha, s_0^\alpha) \) defines a non-standard interpretation of \( P \) where \( \rightarrow^\alpha \) is defined as: \((s_y^\alpha, \text{start}) \xrightarrow{y-y+2}^\alpha (s_y^\alpha, \text{start}) \) and \((s_y^\alpha, \text{start}) \xrightarrow{\text{skip}}^\alpha (s_y^\alpha, \text{end}) \) and \( s_0^\alpha = (\text{even, start}) \). The set \( \mathcal{O}(\text{Std}^\alpha) \) contains finite traces like \( x = (\text{even, start}) ; \xrightarrow{y-y+2} (\text{even, start}) ; \xrightarrow{y-y+2} \ldots ; \xrightarrow{\text{skip}} (\text{even, end}) \) and the non-terminating trace \( x = (\text{even, start}) ; \xrightarrow{y-y+2} (\text{even, start}) ; \xrightarrow{y-y+2} \ldots \).

Under these definitions, it is clear that \( (\text{Std}^\alpha, \leq^\alpha) \) is correct under the corresponding interpretation \( I^\alpha \) wrt \( \text{Std} \). Therefore, Theorem 1 holds.

In the sequel, we always assume that \( \text{Std}^\alpha \) is correct under the interpretation \( I^\alpha \) wrt \( \text{Std} \).

### 4 Abstracting Temporal Logic

In this section, we define a weak Kripke structure from a labeled transition system and a function to evaluate logic propositions. We will see how to abstract the standard notion of satisfaction of temporal formulas and the conditions under which the model abstraction process preserves the abstract standard satisfaction of temporal formulas. To do this, we will consider Kripke structures where negation is not dealt with as a connective, but as a way of constructing atomic propositions. This is done to get a homogeneous treatment of both standard and abstract satisfaction.
4.1 Our Temporal Logic

Given a set of propositions Prop, we construct the set \( \mathcal{P} = \text{Prop} \cup \neg \text{Prop} \), where \( \neg \text{Prop} = \{ \neg p : p \in \text{Prop} \} \). Let \( \mathcal{F} \) be the set of \( \text{LTL} \) temporal formulas built inductively using the elements of \( \mathcal{P} \), the standard Boolean operators, and the temporal operators: next “\( \circ \)”, always “\( \Box \)”, eventually “\( \Diamond \)” and until “\( \mathcal{U} \)”. The set \( \mathcal{F} \) includes all temporal formulas in negation normal form, that is, negations only appear in atomic propositions, since negated propositions are explicitly included in \( \mathcal{P} \). Let \( \mathcal{F}_\text{S} \subseteq \mathcal{F} \) be the subset of temporal formulas without temporal operators. We will call the elements of \( \mathcal{F}_\text{S} \) state formulas.

A labeled transition system \( M = (A, \Sigma, \rightarrow, s_0) \) may be extended to a \emph{weak Kripke structure} \( \mathcal{K} = \langle M, \tau \rangle \) where \( \tau : \Sigma \rightarrow 2^\mathcal{P} \) is a function that assigns truth values to the propositions of \( \mathcal{P} \) in each state in such a way that the \emph{Principle of Excluded Middle} holds, i. e.

\[
p \in \tau(s) \lor \neg p \in \tau(s), \forall s \in \Sigma, \forall p \in \text{Prop}. \tag{PEM}\]

Besides, \( \mathcal{K} = \langle M, \tau \rangle \) is a \emph{Kripke} structure when the \emph{Principle of Non-Contradiction} holds, i. e.

\[
p \not\in \tau(s) \lor \neg p \not\in \tau(s), \forall s \in \Sigma, \forall p \in \text{Prop}. \tag{PNC}\]

Note that \( \mathcal{K} \) defines an interpretation of both actions and atomic propositions. In the sequel, \( p \in \mathcal{P} \) will denote both non-negated and negated atomic propositions.

The following definition gives the satisfaction of a temporal formula with respect to the given model interpretation \( M \) and the formula interpretation \( \tau \).

\textbf{Definition 2.} Let \( \mathcal{K} = \langle M, \tau \rangle \) be a weak Kripke/Kripke structure. Given a state \( s \in \Sigma \), and a trace \( x = t_0 \xrightarrow{a_0} t_1 \ldots \), we define inductively the relation \( \models_\tau \) as follows:

1. \( s \models_\tau p \iff p \in \mathcal{P} \) and \( p \in \tau(s) \).
2. \( s \models_\tau f \lor g \iff f, g \in \mathcal{F}_\text{S}, s \models_\tau f \lor s \models_\tau g \).
3. \( s \models_\tau f \land g \iff f, g \in \mathcal{F}_\text{S}, s \models_\tau f \land s \models_\tau g \).
4. \( x \models_\tau f \iff f \in \mathcal{F}_\text{S} \) and \( t_0 \models_\tau f \).
5. \( x \models_\tau f \lor g \iff f, g \in \mathcal{F}, x \models_\tau f \lor x \models_\tau g \).
6. \( x \models_\tau f \land g \iff f, g \in \mathcal{F}, x \models_\tau f \land x \models_\tau g \).
7. \( x \models_\tau \bigcirc f \iff f \in \mathcal{F} \) and \( x^1 \models_\tau f \).
8. \( x \models_\tau \Box f \iff f \in \mathcal{F}, t_0 \models_\tau f \) and \( x^1 \models_\tau \Box f \).
9. \( x \models_\tau \bigcirc f \iff f \in \mathcal{F} \) and \( \exists k \geq 0, (x^k \models_\tau f) \).
10. \( x \models_\tau f \text{U} g \iff f, g \in \mathcal{F} \) and \( \exists k \geq 0, (x^k \models_\tau g \land \forall j < k, [x^j \models_\tau f]) \).

We say that \( \mathcal{K} = \langle M, \tau \rangle \) verifies a universal temporal formula \( f \in \mathcal{F} \), and write \( \mathcal{O}(M) \models_\tau f \), iff \( \forall x \in \mathcal{O}(M), (x \models_\tau f) \).
4.2 Abstracting Weak Kripke Structures

Let $\mathcal{K} = \langle \text{Std}, \tau \rangle$ and $\mathcal{K}^\alpha = \langle \text{Std}^\alpha, \tau^\alpha \rangle$ be two weak Kripke structures. We say that $\tau^\alpha$ is (weakly) consistent wrt $\tau$ and $I_\alpha$ when the following condition holds:

$$\alpha(s) \leq^\alpha s^\alpha \Rightarrow \tau(s) \subseteq \tau^\alpha(s^\alpha).$$

(LC)

We denote consistency property with LC because it corresponds to the local consistency condition that usually appears in abstract interpretation. We say that $\mathcal{K}^\alpha$ is correct under $I_\alpha$ wrt $\mathcal{K}$ iff $\tau^\alpha$ is consistent wrt $\tau$ and $I_\alpha$.

Fig. 1. A commutative diagram relating abstraction and satisfiability

Condition LC establishes the commutativity of the diagram depicted in Figure 1. It means that the abstraction $\tau \rightarrow \tau^\alpha$ must be sound wrt $I_\alpha$. Therefore, the set $\tau^\alpha(\alpha(s))$ may contain more propositions than $\tau(s)$. In particular, even if $\mathcal{K}$ is a Kripke structure, $\mathcal{K}^\alpha$ may lose the PNC condition due to the abstraction process as shown in the following example.

Example 2. Consider the program and the abstract interpretation in Example 1. Given $P = \{y \equiv 1, -\neg(y \equiv 1)\}$, the usual interpretation of $y \equiv 1$ and $\neg(y \equiv 1)$ over $\Sigma$ is $y \equiv 1 \in \tau(s) \iff s_y = 1$ and $\neg(y \equiv 1) \in \tau(s) \iff s_y \neq 1$.

Therefore, $\mathcal{K} = \langle \text{Std}, \tau \rangle$ is clearly a Kripke structure. Consider $\mathcal{K}^\alpha = \langle \text{Std}^\alpha, \tau^\alpha \rangle$ such that $\tau^\alpha$ is consistent wrt $\tau$ and $I_\alpha$. Let $s_1 = (1, s_1), s_2 = (3, s_1) \in \Sigma$. Then $\alpha(s_1) = \alpha(s_2) = (\text{odd}, s_1)$. Denote $s^\alpha = (\text{odd}, s_1)$. It is clear that $y \equiv 1 \in \tau(s)$ and $\neg(y \equiv 1) \in \tau(s^\alpha)$ and, by LC, $y \equiv 1 \in \tau^\alpha(s^\alpha)$. Therefore $\mathcal{K}^\alpha$ does not verify PNC, i.e., it is not a Kripke structure.

In the rest of the paper, we assume that $\mathcal{K}^\alpha = \langle \text{Std}^\alpha, \tau^\alpha \rangle$ is correct under $I_\alpha$ wrt $\mathcal{K} = \langle \text{Std}, \tau \rangle$. In addition, according to Definition 2, $\models_\tau$ and $\models_{\tau^\alpha}$ model the satisfaction of a formula under the interpretation given by $\mathcal{K}$ and $\mathcal{K}^\alpha$, respectively. In order to simplify notation, we will write $\models$ and $\models_\alpha$ instead of $\models_\tau$ and $\models_{\tau^\alpha}$, respectively. The next result relates the relationships $\models$ and $\models_\alpha$.

Proposition 1. Let $s \in \Sigma$ be a state and $x = t_0 \overset{a_0}{\rightarrow} t_1 \overset{a_1}{\rightarrow} \ldots$ be a trace and the formulas $f_s \in F_S$ and $f \in F$:

1. $s \models f_s \Rightarrow \alpha(s) \models_\alpha f_s$.
2. Given $x^\alpha = t_0^\alpha \overset{a_0^\alpha}{\rightarrow} t_1^\alpha \overset{a_1^\alpha}{\rightarrow} \ldots$ such that $\alpha(x) \leq^\alpha x^\alpha$ then $x \models f \Rightarrow x^\alpha \models_\alpha f$. 

Abstract Satisfiability of Linear Temporal Logic 169
Proof. Let us consider \( f_s \in \mathcal{P} \). By definition, \( s \models f_s \) means that \( f_s \in \tau(s) \). Then, the consistency condition \( LC \) implies that \( f_s \in \tau^\alpha(\alpha(s)) \), i.e., \( \alpha(s) \models f_s \). Other cases for \( f_s \) and \( f \) are proved by induction on the structure of the formula.

In Definition 3, we say when a standard state satisfies a proposition under a non-standard interpretation of the program.

**Definition 3 (Non-Standard Satisfaction).** We extend \( \models_\alpha \) to standard states as follows:

- A standard state \( s \in \Sigma \) satisfies the state formula \( f_s \in \mathcal{F}_\Sigma \) under a non-standard interpretation \( \mathcal{K}^\alpha \) (\( s \models_\alpha f_s \)) iff \( \alpha(s) \models f_s \).
- A standard trace \( x = t_0 \overset{\alpha}{\rightarrow} t_1 \ldots \) satisfies the formula \( f \in \mathcal{F} \) under a non-standard interpretation \( (x \models_\alpha f) \) iff \( \exists x^\alpha = t_0 \overset{\alpha}{\rightarrow} t_1 \ldots \) such that \( x^\alpha \models_\alpha f \) and \( \alpha(x) \leq^\alpha x^\alpha \).
- \( \mathcal{K}^\alpha \) satisfies the universal property \( f \) under a non-standard interpretation \( \mathcal{K}^\alpha \) \((\mathcal{O}(\text{Std}) \models_\alpha f)\), iff \( \forall x \in \mathcal{O}(\text{Std}) : x \models_\alpha f \).

To illustrate the previous definition, consider the state \( s = (4, s) \) in the context of Example 1. Note that \( s \not\models y = 2 \), but since \( \alpha(s) \models y = 2 \), we have \( s \models_\alpha y = 2 \). Intuitively, we are claiming that although the standard interpretation of the formula is not satisfied by \( s \), its non-standard interpretation, i.e., “\( y \) is even”, is satisfied by \( s \).

This extension of the abstract satisfiability relation will allow us to relate the satisfaction of temporal formulas in the abstract and standard models. Using this notion, we can rewrite Proposition 1 as follows.

**Corollary 1.** Given \( s \in \Sigma \), \( x = t_0 \overset{\alpha}{\rightarrow} \ldots \) a trace, \( f_s \in \mathcal{F}_\Sigma \) and \( f \in \mathcal{F} \), we have:

- \( s \models f_s \Rightarrow s \models_\alpha f_s \).
- If there exists some abstract trace \( x^\alpha \) such that \( \alpha(x) \leq^\alpha x^\alpha \) then \( x \models f \Rightarrow x \models_\alpha f \).

**Proof.** It follows from Proposition 1, using Definition 3.

**Theorem 2.** Let \( f \in \mathcal{F} \) be a universal property. If \( \mathcal{K}^\alpha \) is correct under \( \mathcal{I}_\alpha \) wrt \( \mathcal{K} \), and \( \mathcal{O}(\text{Std})^\alpha \models_\alpha f \), then \( \mathcal{O}(\text{Std}) \models_\alpha f \).

**Proof.** Given \( x = s_0 \overset{\alpha}{\rightarrow} \ldots \in \mathcal{O}(\text{Std}) \), since \( \mathcal{K}^\alpha \) is correct with respect to \( \mathcal{K} \), there exists \( x^\alpha = s_0 \overset{\alpha}{\rightarrow} \ldots \in \mathcal{O}(\text{Std})^\alpha \) such that \( \alpha(x) \leq^\alpha x^\alpha \). As, by hypothesis, \( \mathcal{O}(\text{Std})^\alpha \models_\alpha f \), we have \( x^\alpha \models_\alpha f \), i.e., \( x \models_\alpha f \).

This theorem proves that if an abstract model verifies the formula \( f \), then the model itself verifies \( f \) too under the abstract interpretation given by \( \tau^\alpha \). The example below shows how this result can be applied.

**Example 3.** With the abstraction function \( \alpha \) defined in the Example 1, the abstract satisfaction of the formula \( f = \Box(y = 2) \) could be used to prove that \( y \) remains even during any model execution as follows.
Let $s \in \Sigma$ and $s^\alpha \in \Sigma^\alpha$ be the concrete and abstract states defined by $s = (2, s)$ and $s^\alpha = (even, s)$. To prove that $y \models 2 \in \tau^\alpha(s^\alpha)$, it suffices to note that a) $y \models 2 \in \tau(s)$ b) $\alpha(s) \leq s^\alpha$ and c) $\tau^\alpha$ must verify the property of consistency given by $LC$. In addition, it is clear that if $x^\alpha = s^\alpha \xrightarrow{\beta} s_{j_2} \ldots \in O(Std^\alpha)$, then $\forall j.s^\alpha_j = (even, s_{j_j})$, that is, $y \models 2 \in \tau^\alpha(s^\alpha_j)$. Using Definition 2, this means that $\forall j. s^\alpha_j \models (y \models 2)$, i.e., $x^\alpha \models (y \models 2)$. Since $\tau^\alpha$ is a generic element of $O(Std^\alpha)$, by Definition 2, we have that $O(Std^\alpha) \models (y \models 2)$. Finally, applying Theorem 2, as $K^\alpha$ is correct under $\mathcal{I}_\alpha$ wrt $K^\alpha$, we deduce that $O(Std) \models (y \models 2)$.

\[\square\]

### 4.3 Adding Completeness Condition

It is worth stressing the relevance of Theorem 2. Indeed, if $\alpha$ defines some useful information for the user, this result may be a powerful tool for debugging a model. However, condition $LC$ only forces $\tau^\alpha(s^\alpha)$ to contain all atomic propositions satisfied by the concretizations of $s^\alpha$. Many definitions of $\tau^\alpha$ verify $LC$. In particular, the most conservative one, $\tau^\alpha(s^\alpha) = P$ for all $s^\alpha \in \Sigma^\alpha$, does it. However, this $\tau^\alpha$ is not useful, because it could be used to prove any formula over the (abstract) program. For instance, in the previous example, we could prove that $O(Std) \models (y \models 1)$. The next condition restricts the set $\tau^\alpha(s^\alpha)$.

We say that $\tau^\alpha$ is complete wrt $\tau$ and $\mathcal{I}_\alpha$ iff the following condition holds:

\[\text{If } p \in \tau^\alpha(s^\alpha) \text{ then there exists } s \in \Sigma \text{ such that } \alpha(s) \leq s^\alpha \text{ and } p \in \tau(s). \quad (\text{C})\]

Condition C represents the completeness of the abstraction $\tau \rightarrow \tau^\alpha$ wrt $\mathcal{I}_\alpha$. Every proposition satisfied by $s^\alpha$ must also be satisfied by some concretization of $s^\alpha$. In other words, C states that function $\tau^\alpha$ does not arbitrarily add propositions to $\tau^\alpha(s^\alpha)$. Clearly, the conservative $\tau^\alpha$ defined above is not complete: if $\alpha(s) = (even, s_l)$ then $y \models 1 \not\models \tau(s)$.

In what follows, we always assume that $\tau^\alpha$ is both consistent and complete wrt $\tau$ and $\mathcal{I}_\alpha$. Under these conditions, we obtain the monotonicity of $\tau^\alpha$.

**Proposition 2.** $\tau^\alpha$ is monotonic, that is, if $s_l^\alpha \leq s_2^\alpha$ then $\tau^\alpha(s_l^\alpha) \subseteq \tau^\alpha(s_2^\alpha)$.

**Proof.** Let us assume that $p \in \tau^\alpha(s_l^\alpha)$, then, by condition C, there exists $s \in \Sigma$ such that $\alpha(s) \leq s_l^\alpha$ and $p \in \tau(s)$. As $s_l^\alpha \leq s_2^\alpha$, applying condition LC, we conclude $p \in \tau^\alpha(s_2^\alpha)$.

**Proposition 3.** Let $f_s \in F_S$ and $f \in F$ be a state formula and a temporal formula, respectively.

1. Given $s_l^\alpha, s_2^\alpha \in \Sigma^\alpha$ such that $s_l^\alpha \leq s_2^\alpha$. Then $s_l^\alpha \models f_s$ implies $s_2^\alpha \models f_s$.

2. Given $x_1^\alpha = t_1^0 \rightarrow t_1^1 \rightarrow \ldots$ and $x_2^\alpha = t_2^0 \rightarrow t_2^1 \rightarrow \ldots$ two sequences of states such that $\forall i. t_i^\alpha \leq t_i^\alpha$. Then $x_1^\alpha \models f$ implies $x_2^\alpha \models f$.

**Proof.** Let us assume that $f_s \in P$. By definition, $s_l^\alpha \models f_s$ implies $f_s \in \tau^\alpha(s_l^\alpha)$. By the Proposition 2, since $s_l^\alpha \leq s_2^\alpha$, we also have $f_s \in \tau^\alpha(s_2^\alpha)$ holds, i.e., $s_2^\alpha \models f_s$. The rest of the cases for $f_s$ and $f$ are proved by induction on the structure of the formula.
The next proposition proves that $LC$ and $C$ determine $\tau^a$.

**Proposition 4.** $\tau^a$ verifies conditions $LC$ and $C$ iff $\forall s^a, \tau^a(s^a) = \bigcup_{s \in \gamma(s^a)} \tau(s)$.

**Proof.** Clearly, if $\forall s^a, \tau^a(s^a) = \bigcup_{s \in \gamma(s^a)} \tau(s)$, then $\tau^a$ verifies $LC$ and $C$. Conversely, condition $LC$ implies $\bigcup_{s \in \gamma(s^a)} \tau(s) \subseteq \tau^a(s^a)$. In addition, if $p \in \tau^a(s^a)$, by applying condition $C$, there exists $s \in \gamma(s^a)$, such that $p \in \tau(s)$. Therefore, $p \in \bigcup_{s \in \gamma(s^a)} \tau(s)$.

**Example 4.** Given $\Sigma = Int$ and $\Sigma^a = \{neg, zpos\}$, we define the abstraction function $\alpha: \Sigma \to \Sigma^a$ as $\alpha(neg) = neg$, if $n < 0$ and $\alpha(zpos) = zpos$, if $n \geq 0$. Let $P = \{zero, \neg zero\}$ be the set of atomic propositions. We assume that the order relation $\leq^a$ over $\Sigma^a$ is the $=$ relation. Let $\tau$ be defined as:

$$\tau(n) = \begin{cases} 
\text{\{\neg zero\}} & \text{iff } n < 0 \\
\text{\{zero\}} & \text{iff } n = 0 \\
\text{\{\neg zero\}} & \text{iff } n > 0
\end{cases}$$

Using Proposition 4, we obtain that $\tau$ determines the following $\tau^a$ function:

$$\tau^a(n^a) = \begin{cases} 
\text{\{\neg zero\}} & \text{iff } n^a = neg \\
\text{\{\neg zero, zero\}} & \text{iff } n^a = zpos
\end{cases}$$

Obviously, in this simple case, $\tau^a$ is only useful to check the zero values. □

### 4.4 Adding Strong Consistency Condition

We now present the most relevant result of the paper, that relates the satisfaction of a formula in the abstract model with its satisfaction in the standard one.

In Example 3, we did not prove that during a standard execution the variable $y$ is always equal to the value 2 (this assertion is in fact false). Instead, we proved that $y$ is always equal to the *non-standard interpretation* (given by the function $\alpha$) of the value 2, that is, $y$ is always *even*. In fact, this is the best method for proving property "$y$ remains even during execution" over the standard interpretation of program, which may be interesting in many applications.

However, the user could also be interested in proving that the standard interpretation of the property is also satisfied during the standard interpretation of the program. For instance, we would be interested in proving that $y$ is the value 2 during every standard execution of $P$. Next, we discuss how to relate the standard and non-standard interpretations of temporal formulas.

**Definition 4 (Abstract Implication).** Let $\mathcal{K} = \langle \text{Std}, \tau \rangle$ and $\mathcal{K}^a = \langle \text{Std}^a, \tau^a \rangle$ be two weak Kripke structures such that $\mathcal{K}^a$ is correct under $\mathcal{T}_\alpha$ wrt $\mathcal{K}$.

1. Given two state formulas $f_1, f_2 \in \mathcal{F}$ then $f_1 \Rightarrow_\alpha f_2 \iff \forall s \in \Sigma : s \models_\alpha f_1 \Rightarrow f_2$.
2. Given two temporal formulas $f_1, f_2 \in \mathcal{F}$ then $f_1 \Rightarrow_\alpha f_2 \iff \forall \varphi \in \mathcal{O}($Std$) : \varphi \models_\alpha f_1 \Rightarrow \varphi \models_\alpha f_2$.

In order to be able to make sound backtracking to the standard interpretation when proving temporal formulas we need to impose an additional condition called *strong consistency condition*. 

---

172  M. del Mar Gallardo, P. Merino, E. Pimentel
Definition 5 (Strong Consistency Condition). \( \tau^o \) is strongly consistent wrt \( \tau \) and \( I_\alpha \) when it is weakly consistent and the following condition holds:

\[
\text{If } \tau(s) \subseteq \tau^o(s^o) \text{ then } \alpha(s) \leq^o s^o.
\]

(SC)

Note that LC and SC are reciprocal each other.

Example 5. Consider the sets \( \Sigma \) and \((\Sigma^o, \leq^o)\), and the function \( \alpha : \Sigma \rightarrow \Sigma^o \) defined in Example 4. Function \( \tau^o \) is not strongly consistent wrt \( \tau \) and \( I_\alpha \) because for all \( n < 0, \tau(n) \subseteq \tau^o(zpos) \) but \( \alpha(n) \neq neg \leq^o zpos \). But, if \( P = \{\text{positive, negative, } \neg \text{negative}\} \), \( \tau \) may be redefined as follows:

\[
\tau(n) = \begin{cases} 
\{\text{positive, negative}\} & \text{iff } n < 0 \\
\{\text{negative}\} & \text{iff } n = 0 \\
\{\text{positive, negative}\} & \text{iff } n > 0
\end{cases}
\]

By Proposition 4, \( \tau \) determines \( \tau^o \):

\[
\tau^o(n) = \begin{cases} 
\{\text{negative}\} & \text{iff } n^o = neg \\
\{\text{positive, negative}\} & \text{iff } n^o = zpos
\end{cases}
\]

Now, \( \tau^o \) is strongly consistent wrt \( \tau \) and \( I_\alpha \):

1. if \( n < 0 \) then \( \tau(n) \subseteq \tau^o(n^o) \) implies \( n^o = neg \).
2. \( \tau(0) \subseteq \tau^o(zpos) \) and \( \alpha(0) \leq^o zpos \).
3. if \( n > 0 \) then \( \tau(n) \subseteq \tau^o(n^o) \) implies \( n^o = zpos \). \( \square \)

Proposition 5. Let \( \mathcal{K} = (\text{Std}, \tau) \) and \( \mathcal{K}^o = (\text{Std}^o, \tau^o) \) be two Kripke structures such that \( \mathcal{K}^o \) is correct under \( I_\alpha \) wrt \( \mathcal{K} \), \( \tau^o \) being complete and strongly consistent wrt \( \tau \) and \( I_\alpha \). Then, we have:

(a) Given \( s \in \Sigma \), there is a state formula \( f'_s \in \mathcal{F}_S \) such that \( s \models f'_s \), and for every \( f_s \in \mathcal{F}_S \) such that \( s \models f_s \), \( f'_s \Rightarrow f_s \).

(b) Given \( x = s_0 \equiv \ldots \in \mathcal{O}(\text{Std}) \), there exists a formula \( f \in \mathcal{F} \) such that \( x \models f \) and for all \( f \in \mathcal{F} \) verifying \( x \models f \), \( f \Rightarrow f \).

Proof. a) We have just to take \( f'_s = \land\{q : q \in \tau(s)\} \), which is well defined since the set \( \tau(s) \) is non-empty because \( \mathcal{K} \) is a weak Kripke structure and, therefore, PEM holds. By the construction of \( f'_s \), it is clear that \( s \models f'_s \). Given \( f_s \) such that \( s \models f_s \), let us see that \( f'_s \Rightarrow f_s \). Let \( s' \in \Sigma \) be a state verifying that \( s' \models f'_s \). Using the definition of \( f'_s \), \( s' \models f'_s \) implies \( \tau(s) \subseteq \tau^o(\alpha(s')) \).

Since, by hypothesis, \( \tau^o \) is strongly consistent wrt \( \tau \), this relation implies that \( \alpha(s) \leq^o \alpha(s') \). Finally, by using the monotonicity (Proposition 3), and taking into account that \( s \models f_s \), we infer \( s' \models f_s \).

b) By induction on the structure of the formula:

i. For \( f \in \mathcal{F}_S \) the result is obtained from a).

ii. If \( f = \bigcup f_s \) with \( f_s \in \mathcal{F}_S \). By definition, \( x \models \bigcup f_s \) implies that \( \forall i \geq 0, s_i \models f_s \). Applying a) to each \( s_i \), we obtain a family of propositions \( f'_s \) verifying that \( f'_s \Rightarrow f_s \) and \( s_i \models f'_s \). Let \( f' = \bigcup f'_s \) and \( f'' = \bigcup f''_s \). Then \( f'' \Rightarrow f_s \), that is, \( f'' \Rightarrow f \), what implies that \( x \models f' \).

iii. Proofs for cases \( f = f_s \vee g_s \) and \( f = \bigcap f_s \) are similar to the previous ones.
The previous proposition allows us to define the notion of abstract extension of a formula \( f \). This operation represents the loss of precision about \( f \) due to the abstract interpretation.

**Definition 6 (Abstract extension of a formula).** Given \( f \in \mathcal{F} \) a temporal formula, we define the abstract extension of \( f \), and denote it with \( \overline{f} \), by 
\[
\forall \{ f' : f' \Rightarrow a f \}.
\]

The loss of information above commented may be written as \( f \Rightarrow \overline{f} \), which is easily deduced from the previous definition. In general, the opposite is not true and \( f \) and \( \overline{f} \) do not coincide; this means that the properties represented by \( f \) and the abstract interpretation are orthogonal. For instance, this occurs for the formula \( \Box y = 2 \) in Example 3. A more detailed study will be made below.

**Theorem 3.** Under the same conditions in Proposition 5, if \( \mathcal{O}(\text{Std}^a) \models a f \) then \( \mathcal{O}(\text{Std}) \models \overline{f} \).

**Proof.** It follows directly from Proposition 5 b).

This theorem gives us the relationship between the formulas proved under the abstract interpretation and the formulas which hold under the standard interpretation. This result is particularly useful when \( f = \overline{f} \), i.e., when the abstraction process preserves all the information represented by \( f \). Under this hypothesis, whenever we have \( \mathcal{O}(\text{Std}^a) \models a f \), Theorem 3 guarantees \( \mathcal{O}(\text{Std}) \models f \), as desired by the user.

**Corollary 2.** Given a temporal formula \( f \in \mathcal{F} \), if \( \mathcal{O}(\text{Std}) \models a f \) and \( \overline{f} = f \), then \( \mathcal{O}(\text{Std}) \models f \).

Thus, we need a procedure to compute \( \overline{f} \) in order to check whether \( \overline{f} = f \). Fortunately, \( \mathcal{O}(\text{Std}) \) preserves the temporal operators. That is, \( \overline{f \lor g} = f \lor \overline{g} \), \( \overline{f \land g} = f \land \overline{g} \), \( \overline{f \Rightarrow g} = f \Rightarrow \overline{g} \), \( \overline{f \land g} = f \land \overline{g} \), \( \overline{f \lor g} = f \lor \overline{g} \), and \( \overline{f U g} = f \lor \overline{g} \).

Corollary 2 gives the analyzer the possibility of directly using information about a given concrete property obtained from an abstract model (easier to be verified than the concrete one). Thus, it is important to provide conditions (automatically decidable) which ensure the applicability of the corollary. The next definition presents these conditions, as Proposition 6 will prove.

**Definition 7.** We say that \( p \in \mathcal{P} \) is not abstracted by \( I_a \), if \( \forall s^a : p \in \tau^a(s^a) \Rightarrow \neg p \not\in \tau^a(s^a) \).

**Proposition 6.** If \( p \in \mathcal{P} \) is not abstracted by \( I_a \) then \( \overline{p} \Rightarrow p \).

**Proof.** Let us see that \( \overline{p} \Rightarrow p \). By definition, \( s \models \overline{p} \) means that \( p' \in \mathcal{P} \) exists such that \( s \models p' \) and \( p' \Rightarrow a p \). Using LC, we have that \( \alpha(s) \models a p' \), that is, \( \alpha(s) \models p^a \), or equivalently, \( p \in \tau^a(\alpha(s)) \). If \( \neg p \in \tau(s) \) then, by LC, \( \neg p \in \tau^a(\alpha(s)) \). But this is not possible, since \( p \) is not abstracted by \( I_a \) and we know that \( p \in \tau^a(\alpha(s)) \). Therefore, it stands that \( p \in \tau(s) \), i.e., \( s \models p \).
The next corollary presents two practical results of interest when we want to prove some very frequent temporal formulas by using abstract interpretation.

**Corollary 3.** Let us assume that \( p, q \in \mathcal{P} \) are not abstracted by \( I_a \). Then if \( f \) is any of the temporal formulas \( \square p, \Diamond q, \Box p, \) or \( p \lor q \), and \( \mathcal{O}(\text{Std}^a) \models_a f \), then \( \mathcal{O}(\text{Std}) \models f \).

## 5 An Example in PROMELA

In section we illustrate the usefulness of our approach by applying it to SPIN [12, 13], a widely used model checking tool that uses PROMELA as the modeling language. A PROMELA model \( F = \text{Proc}_1 \ldots \text{Proc}_n \) consists of a finite set of concurrent processes, global and local channels, and global and local variables. Processes communicate via message passing through channels. Communication may be asynchronous using channels as bounded buffers, and synchronous using channels with size zero. Global channels and variables determine the environment in which processes run, while local channels and variables establish the internal local states of processes.

The operational semantics of a PROMELA model \( \mathcal{O}(\text{Std}) \) is defined in [11] by means of a transition relation. In that work, a generalized semantics is proposed by considering a transition relation defined on a set of environments \( \text{Env} \), and parameterized by two mappings, \( \tau : \text{Env} \rightarrow 2^{\text{BExp}} \) and \( \varphi : \text{Basic} \times \text{Env} \rightarrow \text{Env} \). The first one evaluates Boolean expressions \( (\text{BExp}) \) in a given environment, whereas the second mapping gives the effect of executing a basic action \( \text{(Basic)} \) on an environment. As a PROMELA model depends on these two parameters, we will denote \( \text{Std} \) as \( M(\tau, \varphi) \). Note that different definitions of \( \tau \) and \( \varphi \) yield different interpretations of the model. In particular, to give an abstraction of \( \text{Std} \) it is enough to give abstractions for \( \tau \) and \( \varphi \).

We present an example illustrating the applicability of the previous results to this popular modeling language.

**Example 6.** Consider the model Collatz in Figure 2, where the process \( p2 \) uses the values generated by the Collatz function to send messages to process \( p1 \) using a shared channel. Let \( \text{Env} = \{v, n, ch\} \rightarrow \{\alpha, b, c, \ldots, \text{error}\} \cup \text{Int} \) be the set of states. Each element \( e \in \text{Env} \) associates the variables \( v, n \) and the channel \( ch \) with its actual value. Assume that we want to prove the temporal formula \( F = \Diamond (v \equiv \text{error}) \). To do it, we consider the set of abstract states \( \text{Env}^a = \{v, n, ch\} \rightarrow \{0, 1\} \cup \text{Int} \). Let us take \( (\text{Env}^a, \leq^a) \), where \( \leq^a \) is the relation “\( = \)”. Let \( \alpha_{eh} : \text{Int} \rightarrow \{0, 1\} \) be defined by \( \alpha_{eh}(\text{error}) = 0 \) and \( \alpha_{eh}(m) = 1, \forall m \neq \text{error} \). Finally, let \( \alpha : \text{Env} \rightarrow \text{Env}^a \) be the abstraction function given by \( \alpha(e)(n) = e(n), \alpha(e)(v) = \alpha_{eh}(e(v)) \) and \( \alpha(e)(ch) = \alpha_{eh}(e(ch)) \). Denote \( I_a = (\text{Env}, (\text{Env}^a, \leq^a), \alpha, \gamma) \). Finally, let \( \text{Std}^a = M(\tau^a, \varphi^a) \) the abstract interpretation of the program given by the functions \( \tau^a \) and \( \varphi^a \) defined by

\[- v \equiv \text{error} \in \tau^a(e^a) \text{ iff } e^a(v) = 0.\]
\[ v \equiv m \in \tau^a(e^a) \text{ iff } (e^a(v) = 1), \forall m \neq \text{error}. \]
\[ \varphi^a(\text{out } m, e^a) = e^a[1/\text{out}], \forall m \neq \text{error}. \]
\[ \varphi^a(\text{out } \text{error}, e^a) = e^a[0/\text{out}]. \]

It can be proved that $\text{Std}^a$ is correct under $I_a \text{ wrt Std}$. In addition, $\tau^a$ is strongly consistent and complete under $I_a \text{ wrt } \tau$.

Furthermore, if $v \equiv \text{error} \in \tau^a(e^a)$ then, by definition, $e^a(v) = 0$. Therefore, $\neg(v \equiv \text{error}) \notin \tau^a(e^a)$; and, by Definition 7, $v \equiv \text{error}$ is not abstracted by $I_a$. Using Proposition 6, this means that $v \equiv \text{error}^a = v \equiv \text{error}$, that is, \( (\text{Corollary 3}) \Downarrow v \equiv \text{error}^a \equiv (\Downarrow v \equiv \text{error}) \).

A simple analysis of the abstract behaviour of the program shows that $\mathcal{O}(\text{Std}^a) \models_\alpha \Downarrow v \equiv \text{error}$. Since $\tau^a$ is strongly consistent wrt $\tau$ we conclude that $\mathcal{O}(\text{Std}) \models_\alpha \Downarrow v \equiv \text{error}$, that is, $\mathcal{O}(\text{Std}) \models \Downarrow v \equiv \text{error}$. \(\square\)

\begin{verbatim}
ntype = {a,b,c,...,error}
nstype y; int x = ...;
class ch = [1] of (ntype);
active proctype p0(){
    active proctype p1(){
        do
            ...
            do
                ...
        od
    od
}
}

Fig. 2. Model Collatz

6 Conclusions

We have introduced a framework to apply abstract interpretation to LTL, in the context of model checking. In particular, we have defined a set of (weak) consistency conditions that guarantee that the abstract interpretation of a model preserves the abstract satisfiability relation. However, these conditions are satisfied by trivial abstractions, which clearly are not useful in practice. In order to solve this problem, some completeness conditions are introduced. Finally, strong consistency provides a way to guarantee that the abstract satisfaction of a formula implies its standard satisfaction. These results have been illustrated by means of an example.

A relevant contribution of this paper is that all results are independent of the actual modeling language. It can be adapted to a wide range of practical modeling languages with available model checking tools. We are currently implementing an extension of SPIN that supports abstraction. Current and future versions of this tool can be found at http://www.lcc.uma.es/~gisum/fmse/tools.
Acknowledgements
We would like to thank David de Frutos for his comments to improve this paper. This research is partially supported by the CICYT projects TIC98-0445-C03-03 and TIC99-1083-C02-01.

References
Declarative Reflection and its Application as a Pattern Language

Juan José Moreno-Navarro and Ángel Herranz-Nieva

LSIS - Facultad de Informática, Universidad Politécnica de Madrid
{jjmoreno,aherranz}@ifi.upm.es

Abstract. The paper presents the reflection facilities of the specification language Slam-sl. Slam-sl is an object oriented specification language where class methods are specified by pre and postconditions. The reflection capabilities permit managing these pre and postconditions in specifications. What means that semantic reflection is possible. The range of interesting applications is very wide: formal specification of interfaces and abstract classes, specification of component based software, formalization of design patterns, using Slam-sl as a pattern language, etc. The paper discusses the last two advantages in some detail.

1 Motivation

We have recently presented the SLAM system [11, 16], which includes an object oriented specification language, Slam-sl, that is supported by a development environment that, among other features, is able to generate readable code in a high level object oriented language. The Slam-sl language is a formal specification language that integrates algebraic specifications (like those proposed by the OBJ language family [9, 4] or Larch-LSL [10]), and model-based specifications (as Z [17], VDM [12], or Larch interfaces languages [10]). Specification of class methods uses two predicates (pre/post-conditions) that describe the relationship between the input and the output by means of logic formulas.

In this paper we present the reflective features of the language which can be used for some interesting applications, like the formalization of design patterns in terms of class operators. An object-oriented reflective system is one that is itself built out of programmable, first-class objects. In Slam-sl it is possible to dynamically inspect and manipulate classes, objects, methods, and other language elements.

Reflection capabilities can be used in many useful applications: component based software, compiler construction, debugging, module operations, etc. The reflective features of a language can be categorized as linguistic, structural, and behavioral. Linguistic and structural reflection is basically syntactical and is present in a number of programming languages either in the object-oriented imperative community (Smalltalk [7], Java, C# [3]) and in the declarative programming community (Lisp, Prolog, Maude [4]). However, behavioral reflection allows managing the semantics of the elements of the language (Maude).
We call *declarative reflection* those behavioral reflection characteristics that rely on declarative models (logics, functions, etc.) Declarative reflection not only allows manipulating the semantics of classes and objects, but also to formally reason about them. In the previous list, only the specification language Maude, based on rewriting logic what in turns is a reflective logic, can be considered to have declarative reflective features. However, they are focused mainly on module operations.

Slam-sl is equipped with behavioral/declarative reflection:

- It is possible to inspect and manipulate object oriented elements: classes, methods, etc.
- It is possible to manage the declarative behaviour of such elements, as class invariants, or methods pre and postconditions.
- Declarative properties of those elements are included and formal reasoning is enforced. For instance, inheritance is only permitted under certain conditions and these conditions are preserved even in dynamically generated classes.
- It is possible to declaratively model some object oriented features, as interfaces or abstract classes.

We are going to present in detail a pair of interesting application: the formalization of design patterns as class operators and, consequently, the use of a formal method as a pattern language.

The remainder of the paper is organized as follows: an introduction to Slam-sl in section 2, reflective properties of slam-sl in section 3, and composite, state and builder design pattern specification in section 4. We present some conclusions in section 5.

2 Object oriented specifications. Slam-sl

This section presents the main constructions of the language focused on reflective features that will used in the following sections.

Slam-sl is part of the SLAM project, a software construction development environment that is able to synthesize (reasonable) efficient and readable code in different high level object oriented target languages like C++ or Java. Among other features, the user can write specifications in a friendly way, track her hand-coded optimizations, or check in debug mode those optimizations through automatically synthesized assertions.

In order to facilitate the understanding of Slam-sl we will show its elements with a concrete syntax that does not necessarily correspond neither with an internal representation nor the environment presentation\(^1\), so the reader should not pay attention to the concrete syntax but to the abstract one.

A Slam-sl program is a collection of specifications that defines classes and class properties. The specification of method behaviour is given through preconditions and postconditions but with a functional flavour as we will see.

\(^1\) In fact, Slam-sl programs are stored in XML format and its presentation in the environment can be customized.
2.1 Slam-sl toolkit

As many others specification languages Slam-sl has a powerful toolkit with pre-defined types representing booleans, numbers, characters and strings, records and tuples, collections (sequences, sets, etc.), dictionaries (maps, relations, etc.). Slam-sl type syntax reflects value syntax, for instance, the type 'sequence of integers' is written as `[Integer]` and its values are written as `[1,2]`, a tuple type can be written as `(Char, Integer)` and its values as `(‘a’,32)`, etc.

**Records and tuples** Slam-sl hardly distinguish tuples from records. The type `(x : float, y : float)` defines the Cartesian product `float x float` but, in order to avoid 'boilerplate' in specifications, introduces field selectors. Let `p` be `(x -> 0.1, y -> 0.0)`, `p` represents the tuple `(0.1, 0)` and `p.x` is the projection of the first component of the tuple. Some syntactic sugar for modifying a record have been added to Slam-sl through the `except` operator (\`). For instance, the formula `q = p \ y -> 1.0` states that `q` represents `(x -> 0.1, y -> 1.0).

**Sets and sequences** The type expression `{String}` groups together values like `"Hello","world"` or {}. Usual mathematical operations over sets are pre-defined (union +, intersection *, etc.).

Sequences like `"Hello","world"` or `{}` belongs to the type `[String]`. Sequences are indexed data collections. If `s` is a sequence, then `s(i)` with is the i-th element, 1 is the first index, of the sequence (if exists), and `t = s \ 2 -> "universe"` establishes that `t` represents `"Hello","universe"`. Usual operations over sequences are pre-defined (append +, insert, dom, rng, in, etc).

Both sets and sequences inherit the properties of Collection. Collection is a class over which quantifiers are allowed.

**Collections and ‘quantifiers’** Set and list comprehension, restricted quantifiers, and iteration follow a common abstract scheme of ‘traversing’ collections and Slam-sl introduces the following expressive syntax:

```
Q x in d [where F(x)] with E(x)
```

The above Slam-sl expression is a quantified\(^2\) expression. `Q` is the quantifier symbol that indicates the meaning of the quantification by a binary operation and a starting value. `d` is an object of the class `Collection`. `x` is the variable the quantifier ranges over. `F` is an optional boolean expression that filters elements in the collection. And `E` represents the function previously applied to elements in the collection.

Some predefined quantifiers appears in the following table with an informal description:

\(^2\) We have maintained the term ‘quantifier’ because it is a generalization of quantification in logic
Symbol Generalizes
\begin{align*}
\text{exists} & \quad \forall \text{ with true} \\
\text{exists1} & \quad \text{as } \text{exists} \text{ but limiting the count to 1} \\
\text{forall} & \quad \forall \text{ with false.} \\
\text{sum} & \quad + \text{ with 0} \\
\text{prod} & \quad \times \text{ with 1} \\
\text{count} & \quad \text{inc with 0 (counting)} \\
\text{select} & \quad \text{searching} \\
\text{max} & \quad \text{max} \\
\text{maxim} & \quad \text{maximizers} \\
\text{filter} & \quad \text{filters} \\
\text{map} & \quad \text{apply a function to every element in a collection}
\end{align*}

Some examples and its intended meaning will be useful in the remainder of the paper:

\begin{verbatim}
Example = Meaning
forall x in \{1,2,4,7,8\} with x < 11 = true
forall x in \{1,2,4,7,8\} with x \&\& prime = 3
sum x in [1..10] where x < 5 with x.pow(2) = 30
map x in [1..10] with x / 2 = [0,1,2,3,4,5]
map x in [1..10] with x / 2 = [0.1,1.2,2.3,3.4,4.5]
\end{verbatim}

2.2 Classes and class properties

In Slam-sl, a class is defined by specifying its properties: name, relationships with other classes, and method specifications. We will specify the class Stack for representing stacks of objects:

class Stack
Stack inherits Collection

Stack state Empty
Stack state NonEmpty (top_ : Object, rest_ : Stack)

Line 1 declares a new class called Stack, line 3 establishes that class Stack inherits properties from Collection, and lines 5 and 6 define attributes that are the internal representation of the class instances. Slam-sl permits defining algebraic types to indicate that a syntactical construction represents class instances, in our example, the values Empty and NonEmpty (5, Empty) represent the state of an empty stack and the state of a stack with a unique object (the constant 5) respectively.

Class relationships Slam-sl can be considered as a programming language. As in any other (OO) programming languages, we cannot distinguish every kind
of relationships between classes as UML allows. For instance, *aggregation* cannot be distinguished from *composition*, and some *associations* are implicit through the semantics of methods. Anyway, we can list the following relationships that can be caught statically:

**Aggregation:** the state specification of a class defines an aggregation or composition between class instances and instances of other classes.

**Inheritance:** class properties can be defined from scratch or by inheriting them from already defined classes. Overriding of such properties are constrained in Slam-sl, not only the signatures but also the meaning (see subsection 2.2). **Polymorphism:** generic polymorphism is introduced by permitting introducing arguments in types. Slam-sl allows deferring classes in the style of Eiffel but adding some features from theories (in OBJ terminology [9]) as well as type classes (à la Haskell [13]) playing a more powerful role than C++ templates.

**Method specifications** The standard methods in stack objects permit creating an empty stack, decide if a stack is empty, read the top of the stack, and push and pop elements. Slam-sl helps the user to classify different kinds of methods: *constructors*, *modifiers* and *observers*. Let us complete the stack class specification with the definition of its methods:

```plaintext
constructor empty : Stack
pre := true
empty
post := result = Empty

modifier Stack.push(Object)
pre := true
push(x)
post := result = NonEmpty (rest_ -> self, top_ -> x)

observer Stack.isEmpty : Bool
pre := true
isEmpty
post := result = (self = Empty)

observer Stack.top : Object
pre := not self.isEmpty
top
post := result = self.top_
```

In Slam-sl an operation is specified by a set of *rules*, every rule involves a *guard or precondition* that indicates if the rule can be triggered, an *operation call scheme*, and a *postcondition* that relates input state and output state.

The general form of a rule is the following:

```plaintext
pre := P(x, self)
op (x)
post := Q(x, self, result)
```

3 'constructor' methods are not instance members but class members
where $P(x, \text{self})$ is a Slam-sl formula involving variables in the argument $(x)$ and the recipient of the message $(\text{self})$ in case of the operation to be either an observer or a modifier. $Q(x, \text{self}, \text{result})$ is another formula involving variables in the argument, the reserved symbol $\text{result}$ that represents the computed value of the function and $\text{self}$ that represents the state of the receipt of the message before the method invocation.

Some 'shorthands' help the user to write formulas concisely and readable: $\text{self}$ can be omitted for record fields, as in VDM, explicit function definitions are allowed and unconditionally true preconditions can be skipped.

Let us explain in detail how Slam-sl handles method overriding. Suppose you have a class $C$ with a method $m$ with precondition $P$ and postcondition $Q$. Now, a subclass $C'$ of $C$ is declared supplying a new specification for $m$: precondition $P'$ and postcondition $Q'$. As Slam-sl is a formal specification language, it is forces that the following statement holds:

\[
\text{Inheritance Property : } (P \rightarrow P') \land (P \land Q' \rightarrow Q)
\]

**Encapsulation** Encapsulation is an important distinctive in programming languages which permits the user to control coupling and maximize cohesion. Nevertheless, encapsulation is not encourage in formal methods. In Slam-sl, as in other object oriented programming languages, the user can indicate the visibility scope of each method: public, protected or private. If an attribute is indicated as public the user get for free an observer, for instance, in the stack example the definition of the observer top could have been avoided in this way:

\[
\text{Stack state Empty} \\
\text{Stack state NotEmpty (public top : Object, rest_ : Stack)}
\]

The language introduce a broad notion of inheritance via aggregation. Let us see an example, the following Slam-sl spec defines a read only wrapper for stacks:

\[
\text{class ROSTack} \\
\text{ROStack state (target : Stack accept top, isEmpty)} \\
\text{constructor wrap (Stack) : ROSTack wrap (s) = (target --> s)}
\]

Now, the user can return a wrapper instead of the stack if she does not want stack instances to be modifier by clients. This is a pretty unexplored feature.

Applying the ‘shorthands’ introduced through the section, we could redefine the stack example in a more concise way:
class Stack(El:Class)

Stack inherits Collection

Stack state

Empty

NonEmpty (public top : El:Class,
rest_ : Stack)

constructor empty : Stack
empty = Empty

observer Stack.isEmpty : Bool
isEmpty = (self = Empty)

modifier Stack.push(El:Class)
push(x) = NonEmpty (rest_ --\rightarrow self,
top --\rightarrow x)

modifier Stack.pop
pre := not self.isEmpty
pop = self.rest_

Abstract classes In Slam-sl it is quite easy to declare interfaces, i.e. classes with no state and methods that must be redefined in the subclasses. The way to declare such methods is to indicate that the precondition is false. This means that this method is not applicable in any case. Notice that it is still possible to supply an adequate postcondition. This postcondition must be preserved in all derived classes. Those methods that have no definition are implicitly considered to have precondition false and postcondition true.

3 Reflective features

In this section we will present some Slam-sl reflective constructions that we will use in the next section.

Informally, a reflective language is a language in which interesting aspects of its model can be represented and manipulated in the language itself. Reflection makes possible advanced meta-programming applications, like refilction of Slam-sl or other languages, and development of interpreters and component based systems.

3.1 Classes and class relationships

Like many others object oriented languages, Slam-sl classes are represented as instances of a (meta)class called Class. The declaration of a class introduces an immutable instance of the class Class. Let us start with the definition of Class:

class Class

Class state (name : String, st : State, inh : {Class},
inv : Formula, meths : {Method})

Class invariant :-

forall m1, m2 in meths with m1.differ (m2)
We have made a natural reading of 'what a class is': a name, aggregation relationships, inheritance relationships, and methods. The class name is a string, aggregation is represented by an instance of State plus an instance of Formula representing the invariant, inheritance is a set of instances of Class, and, finally, we have added a set of instances of Method.

The invariant in Class establishes that two methods of the class must differ in the signature\(^4\). In other words, method overloading is allowed, but there must be an argument of different type. Notice that thanks to this declarative specification Slam-sl is able to identify those properties that a class must fulfill what is much more powerful than the reflective features of Java or C\# that are merely syntactic.

Let us see the definition of State plus auxiliar classes:

```java
class Declaration

    Declaration state (name : String, type : Class)

    Declaration invariant :- name. isidentifier

public constructor makeDecl (String, Class) : Declaration

observer Declaration. isSubclassOf (Declaration) : Bool
isSubclassOf (d) = self. type = d.type or
    self. type in d.type.inh

---

class DecCollection

DecCollection state [Declaration]

public constructor makeEmptyDecl : DecCollection

public modifier DecCollection. add (Declaration)

---

class State

State inheritance DecCollection

Among the interesting operations of classes, let us show a couple of them. First of all we will introduce a constructor to create a class. Slam-sl automatically generates the field observers\(^5\). When a class is just an interface is detected by checking if the state is empty and if all the precondition of the methods are false.

```java
public constructor makeClass
    (String, State, {Class}, Formula, {Methods}) : Class
```

\(^4\) In fact, the invariant should include some other needed properties related to the inheritance rules stated in section 2.2 set but we omit them for shake of simplicity.

\(^5\) only if the user states attributes are 'public', but in the paper we have assumed it.
makeClass (name, st, inh, inv, meths) = (name, st, inh, inv, meths)

```
public observer Class. isInterface : Bool
  isInterface = st.isEmpty andforall m in meths with m.doNothing
```

### 3.2 Methods

The class modelling methods could be specify in the following way:

```
class Method
Method state (kind : MethodKind, visibility : Visibility,
  name : String, sig : ArgSig, return : Class,
  prec : Formula, postc : Formula)
```

```
public constructor makeMethod
  (MethodKind, Visibility, String, ArgSig, Class,
  Formula, Formula) : Method
makeMethod (k, v, n, sg, rt, pr, ps) = (k, v, n, sg, rt, pr, ps)
```

```
public observer Method.typeSig : [Class]
typeSig = map d in sig with sig.type
```

```
public observer Method.call : [String]
call = map d in sig with sig.name
```

with the following previous definitions:

```
class ArgSig
ArgSig inherits DecCollection
```

We have introduced a couple of useful operations: constructing a method, abstracting the type signature just using the argument types (the names are almost irrelevant except for the pre and postconditions), and composing a method call with the argument names.

On top of them, we can describe a number of interesting operations on methods. The first one (isCompatible) indicates when two method are equivalent (same name, types and equivalent pre and postconditions). The second one (canInherit) specifies when a method can override another definition. They must have a coherent definition (same name and arguments/return type) and the inheritance property must hold.

```
public observer Method.isCompatible (Method) : Bool
isCompatible (m) =
  kind = m.kind and name = m.name and typesig = m.typesig and
  return = m.return and
  (prec implies m.prec[m.call / call ] and postc [implies m.postc[m.call / call ]]
```

```
public observer Method.canInherit (Method) : Bool
canInherit (m) =
```
kind = m.kind and name = m.name and sig_length = m.sig_length and 
(forall i in sig.dom with sig(i).isSubclassOf(m.sig(i))) and 
return = m.return and 
(m.prec implies prec (call /m.call)) and (postc implies m.postc[m.call /call])

Finally, we specify operations to decide when two methods are really different (up to argument names) and when a method implements an interface method (i.e. precondition false):

public observer Method.differ (Method) : Bool
differ (m) =
    name /= m.name or 
    (name = m.name and 
      (sig_length /= m.sig_length or 
       (exists i in sig.dom with sig(i).type /= m.sig(i).type)))

public observer Method.doNothing : Bool
doNothing = (prec = false and postc = true)

For sake of simplicity, we assume that all record components of classes Method and Class are public.

3.3 Formulas

Possibly, the most interesting Slam-sl reflective properties are those related to formula management. Slam-sl runtime environment can manage formulas in the same way the compiler does, this means formulas can be created and compiled at runtime so the user can specify programs that manage classes and class behaviors. The following specification of formulas reflects its abstract syntax in Slam-sl:

class Formula

    Formula state Constant (Bool)
    Formula state Variable (String)
    Formula state And (Formula, Formula)
    Formula state Or (Formula, Formula)
    Formula state Implies (Formula, Formula)
    Formula state Equiv (Formula, Formula)
    Formula state Expression (Expression)

public constructor makeTrue : Formula
    makeTrue = Constant (true)

public constructor makeFalse : Formula
    makeFalse = Constant (false)

public constructor makeVariable (String) : Formula
    makeVariable (s) = Variable (s)
public constructor makeVariable (String): Formula
makeVariable (s) = Variable (s)

public constructor makeAnd (Formula, Formula): Formula
makeAnd (f1, f2) = $f1$ and $f2$

... public modifier substitute (String, Expression)
substitute (var, expr) =
  result =
    case self
    | Constant (c) -> self
    | Variable (v) -> if v = var then Expression (expr) else self
    | And (f1, f2) -> And (f1.substitute (var, expr), f2.substitute (var, expr))
    | Or (f1, f2) -> Or (f1.substitute (var, expr), f2.substitute (var, expr))
    | Implies (f1, f2) -> Implies (f1.substitute (var, expr), f2.substitute (var, expr))
    | Equiv (f1, f2) -> Equiv (f1.substitute (var, expr), f2.substitute (var, expr))
    | Expression (e) -> e[x/expr]

Writing formulas with the above interface would produce unreadable specifications so we write instances of Formula using the Slam-al own notation between $\neq$ symbols and permitting the compiler to parse the sentence and generate the expression. See the definition of the constructor makeAnd.

Sintactic sugar for the substitution operation have been introduced: $f[x/e]$ is the formula $f$ replacing all the references to the variables $x$ by the expression $e$. See that we have used that syntax for expressions in the last line of the definition of substitution operation.

4 Application. Design Patterns as Class Operators

As an application of the reflective features of Slam-al let us show how design patterns [8] can be formalized as class operators. A given (preliminary) design is the input of a design pattern. This design is modeled as a collection of classes. The result of the operation is another design obtained by modifying the old classes and/or creating new ones, taking into account the description of the design pattern.

For instance, consider you have a collection of classes leafs (e.g. Line, Circle, Rectangle, ...) that share some operations (e.g. draw, rotate, resize, ...) and you want to compose all of them in a wider object that either has all of them as particular cases and also can collect some of them inside (e.g. a Figure). The Composite pattern considered as an operator accepts classes (leafs) as input and returns two new classes Component (merely an interface) and Composite
(for the collection of components) with the common operations as methods, and modifying classes in leaves to inherit from Component.

More specifically, design patterns are modeled as a class with a single function apply that is a class operator. This precondition for this function collects the logical conditions required to use the pattern with success. Basically, this means that the pattern precondition establishes the applicability of the pattern, talking in terms of the sections in the pattern description. For instance, in the Composite pattern we mentioned above, the precondition needs to ensure that all the classes in leaves define the common methods with the same signature.

On the other hand the postcondition encompasses most of the elements of the intent and consequences sections of the pattern description. In the Composite pattern, the postcondition establishes that input classes leaves now inherit from Component and classes Composite and Component are introduced, the first one inheriting from the second one. The Composite state is a collection of Components and its methods are described by iterative calls to the corresponding leaves methods.

In order to describe all this elements, the reflective features play a significant role because they allow inspecting argument classes and to describe new classes as result. Design patterns can be described by a (polymorphic) class DPattern. The method apply describes the full behaviour of the pattern by accepting a collection of classes as arguments (the previous design) and returning a new collection of classes. The class argument (coming from the polymorphic definition) is occasionally needed to instruct the pattern about the selection of classes, methods, etc. that take part in the pattern. This argument is stored in the pattern by a dedicated constructor.

```java
class DPattern (T)

DPattern state (arg: T)
```

```java
public constructor instantiate (T) : DPattern

instantiate (x)

post :~ result.arg = x
```

```java
public function apply ([Class ]): [ Class ]
```

Inheritance is used to derive concrete design patterns. It is also needed to instantiate the type argument and supplying a value for the state. Notice that design pattern variants is easily supported in our model.

Let us describe the method by some examples taken from [8]. We have chosen one pattern for each component of the classification: creational, structural, and behavioral patterns. A graphical description complements the formal definition using an OMT-based notation taken again from [8]. More examples can be found in [15].

4.1 Composite pattern

The Composite pattern is part of the object structural patterns. It is used to compose objects intro tree structures to represent part-whole hierarchies. Us-
ing the pattern the clients treat individual objects and compositions of object uniformly.

When we treat it as a class operator, we have the collection of basic objects as argument (called the leaves). The result "invents" two new classes Component and Composite. Component is just an interface for all the common methods in all the leaf classes plus some methods to add, remove and consult internal objects. Composite inherits from Component and stores collection of components. The result also collects all the classes in leaves that are modified by inheriting from Component. The methods in Composite can be grouped in two parts. On one hand, we have methods to add and remove a component, and also to consult the ith element in the component collection (getChild). On the other hand, we have all the common methods of the leaves that have a very simple specification by iterative calling the same operation in all the components. See figure 1 for the complete Slam-sl specification.

4.2 State pattern

The State pattern belongs to the object behavioral classification. It can be used to allow an object to alter its behavior when its internal state changes. The object will appear to change its class. When studied as a class operator, it takes a collection of concrete state classes as argument. All these classes are present in the result, except that they inherit from the State class described below. The result adds two classes: one to abstract the behavior of all the concrete states, called State, that represents an interface containing all the common methods in all the concrete states. The second one is Context that is designed for calling state operations. It contains a State as attribute and all the common methods, described as merely calls to the corresponding operation of the attribute. This class can be refined by inheritance to introduce more functionality. The complete specification can be found in figure 2.

4.3 Builder pattern

The Builder pattern (belonging to the object creational patterns) is designed to separate the construction of a complex object from its representation, so that the same construction process can create different representations.

As a class operator, the Builder patterns takes a collection of concrete builders as an argument. Another class director is part of the arguments and it is assumed that it contains the algorithm to construct objects. It is also assumed that all the concrete builders share some operations that are used to build objects. Those methods are called builders and need to be defined in all the concrete builders. The argument class is a boolean function isBuilder that is applied to methods in the concrete builders, detecting if they are builders or not. Methods classified as builders are abstracted into the Builder class. The concrete builders appear in the result but they are forced to inherit from Builder. The director class is modified in the following way: once an attribute belongs to one of the
concrete classes it is abstracted to the Builder class. See figure 3 for the detailed description.

4.4 Slam-sl as a pattern language

The formalization of design patterns in terms on class operators and using the (declarative) reflective features of an specification language has a number of advantages:

- Coherent specifications of patterns are essential to improve their comprehension and to reason about their properties.
- It is possible to develop tools for supporting design patterns. In fact we are interested in introducing them in existing development environments (as Visual Studio, Visual Age, etc.). The tool can allow applying a design pattern to the project you are working on. The project should be modified according to our description adopting the design pattern. In this way, we can apply design patterns to already existing code and with "every day" existing CASE environments.
- Patterns can be combined by simply applying function composition.
- The functional semantics of Slam-sl can be modified to support functional-logic semantics. Functional-logic languages amalgamate the main features of functional languages and Prolog-like languages, in such a way that inverse functions can be computed. This means that, in principle, we can identify a concrete design patterns into an existing design/specification.

In the literature we can find some other formalizations of design patterns. The work in [1] is focused on the formalization of architectural design patterns based on an object oriented model integrated with a process oriented method to describe the patterns. [14] presents a way to formalize temporal behaviors of design patterns, so communication between objects is the main goal of the specification that uses primitives of a process algebra. Although both use an specification language for the formalization they do not propose any supporting tool and reflection is not used. The project proposed in [5,6] are more focused on providing tools that interact with existing code. They use a metaprogramming language based on a (limited form of) verbal specification. A tool can read it and produce what they call a trick, basically an algorithm to manipulate programs. They have also designed a visual language for specifying patterns (LePus). They share some of our goals and even more, but we claim that we can get a similar power with a simpler approach.

In fact, thanks to its declarative reflection features, Slam-sl can be considered as a pattern language. Once you can model a patterns as a class operator, Slam-sl can be used to specify it and this specification can be used to instruct the associated tool to apply the pattern to existing designs and programs.

5 Conclusion

The precise definition of software design patterns is a prerequisite for allowing tool support in their implementation. Thus comprehensive specifications
of patterns are essential not only to improve their understanding and property reasoning, but also for supporting an automatization of their use.

We have proposed a formal specification of design patterns as class operators. Of course, our approach is not necessarily “better” than others. In fact, different formalizations focused on a particular aspect yields to different tools. We only encourage the fact that it is very simple and easy to automatize in existing development tools. On the contrary, it is not already clear that all the design patterns can be modeled as class operators (for instance the Factory Method that can only be seen as a class operator in a very tricky way).

We have also presented the declarative reflection characteristics of SLAM. The main advantage with respect to other reflective languages is that the semantics of a class method can be inspected (by consulting pre and postconditions) what is very useful for a number of applications like component based systems and the definition of grey box frameworks [2]. Our conclusion is that declarative reflection is a key feature for:

- A simple formalization of design patterns in terms of class operators.
- Supporting concrete tools that permit applying design patterns to existing code.

References

8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns - Elements of Reusable Object Oriented Software. Addison-Wesley, 1995.
class Composite
    extends DParen
    implements DRunnable

    public function apply ( [Class] : [Class] )
    pre := ( not leaf.isEmpty ) and ( not commonMethods.isEmpty )
    apply ( leaves )
    post := result = [component, composite] +
        map c in leaves with c \ inh . insert ( component )
where
    m in commonMethods equiv ( for all cl in leaves with m in cl.methods )
    component = makeClass ( "Component", emptyDec, [], $true$,
        map m in ( builderMethods + [ create, add, remove, getChild ] )
            with m \ prec = $false$ and postc = $true$ )
    composite = makeClass ( "Composite", [ makeDec ( children, [ component ] ) ], [], $true$,
        [ create, add, remove, getChild ] +
        map m in builderMethods gen ( m )
        create = makeMethod ( $constructor$, $public$, "create", emptyDec, $true$, $result = []$ )
        add = makeMethod ( $modifier$, $public$, "add", [ makeDec ( "c", component ) ], $true$,
            $result = children . insert ( c )$ )
        remove = makeMethod ( $modifier$, $public$, "remove", [ makeDec ( "c", component ) ], $true$,
            $result = children . remove ( c )$ )
        getChild = makeMethod ( $observer$, $public$, "getChild", [ makeDec ( "i", Nat ) ], $true$,
            $result = children [ i ]$ )
        gen ( m ) = m \ prec = $forall c in children with m . prec ( this / c )$ and
            postc = $result = map c in children with makeCall ( m . name, [ c ] + m . call )$

Fig. 1. Composite pattern specification.
class State
State inherits DPattern()

public function apply ([Class]): [Class]
pre :- (not concreteState.isEmpty) and (not commonMethods.isEmpty)
apply (concreteStates)
post :- result = [context, abs.state] +
     map c in concreteStates with c \ inh. insert (state)

where
m in commonMethods equiv (forall cl in leafs with m in cl.meths)
abs.state = makeClass("State", emptyDec, {}, true,
     map m in commonMethods with m \ prec = $false$ and
     postc = $true$)
context = makeClass("Context", [makeDec (stt.abs.state)], {}, $true$,
     map m in commonMethods transfer (m)
transfer (m) = m \ postc = $(result = stt.makeCall (m.name, m.call))$

Fig. 2. State pattern specification.
class Builder
Builder inherits DPattern (Methods:Bool)

public function apply ([Class]; [Class]
pre := (classes.length > 2) and (not builderMethods.isEmpty)
apply (classes)
post := result = [builder] +
[director \ st = map d in director.st with abstractToBuilder (d)] +
map c in concreteBuilders with c \ inh. insert (builder)

where
director = classesprefix (1)
concreteBuilders = classessuffix (1)
isBuilder = arg
m in commonMethods equiv (forall cl in concreteBuilders with m in cl.meths)
builderMethods = filter m in commonMethods with isBuilder (m)
builder = makeClass ("Builder", emptyDec, {}, $true$, $true$, $true$, $true$)
abstractToBuilder (d) =
if d.type in concreteBuilders
then d \ type = builder
else d

Fig. 3. Builder pattern specification.
Un lenguaje modelo para la definición y análisis de refactorizaciones

Yania Crespo, Valentín Cardenoso, and José Manuel Marqués

Departamento de Informática, Universidad de Valladolid, España.
{yania, valen, jmanc}@infor.uva.es

Resumen: En este trabajo se propone el lenguaje modelo MOON con suficiente poder expresivo para representar las construcciones abstractas necesarias en la definición y análisis de refactorizaciones. Gracias a este modelo, el esfuerzo de refactorización para un lenguaje particular se reparte entre la tarea de representar en MOON el código y la de refactorización propiamente dicha en el lenguaje modelo. Este último esfuerzo, no obstante, es claramente reutilizable, lo que hace que en conjunto el empleo de nuestro lenguaje modelo reduzca sustancialmente el esfuerzo de refactorización. También se ha definido una arquitectura y un proceso para integrar el modelo en herramientas de refactorización. Como caso concreto se detalla la reducción a MOON de un lenguaje de programación bien conocido y se ilustra la viabilidad de nuestra propuesta describiendo un algoritmo de refactorización concreto.

Palabras clave: transformación de programas, refactorización, abstracción de lenguajes, modelo de familia de lenguajes.

1 Introducción

El propósito fundamental de esta comunicación es presentar brevemente un lenguaje modelo definido con un objetivo muy concreto, facilitar la labor de la definición y análisis de refactorizaciones y avanzar hacia la posibilidad de establecer una cierta independencia del lenguaje en este campo. Fundamentalmente presentaremos el análisis razonado de las decisiones del diseño del lenguaje modelo y de la forma en que este apoya las tareas de refactorización. Hemos empleado el modelo en una aplicación concreta que se presenta como caso de estudio de la factibilidad de proseguir por este camino.

En lo que sigue, la presentación se organiza de la siguiente forma. La Sección 2 introduce el problema de la refactorización y la dependencia del lenguaje de los elementos objetos de la transformación. En la Sección 3 se exponen otros trabajos que se relacionan con la aproximación que hemos tomado. Posteriormente en la Sección 4 se define y se analiza el modelo propuesto y en la Sección 5 se explica el papel del modelo en la definición y análisis de refactorizaciones. La Sección 6 es la presentación de un caso de estudio, mientras que en la Sección 7 se concluye.
2 Refactorización de software

El término refactorización fue introducido por primera vez por Opdyke en [1]. Las refactorizaciones son transformaciones de elementos de software Orientado a Objetos (OO) que, realizando reestructuración y reorganización, preservan el comportamiento. Reestructurar es transformar la estructura interna de un elemento de software. Una reestructuración que transforma la manera en que se relacionan diferentes elementos se dice una reorganización.

En el área de la refactorización de software nos encontramos con una intensa actividad actualmente. En este campo se trabaja para abordar diferentes aspectos tales como la propia definición de operaciones de refactorización [1], la construcción de herramientas que apoyen la ejecución de refactorizaciones [2], la elaboración de catálogos de “patrones” de refactorización que ayuden a los desarrolladores a decidir cuándo refactorizar y qué operaciones aplicar [3], la incorporación de técnicas para poder llegar a inferir automáticamente cuándo y qué refactorizar [4]. También se encuentran trabajos centrados en el análisis y demostración de preservación del comportamiento al aplicar una refactorización [5] y en la introducción en el ciclo de vida de métodos de desarrollo [6].

Definir una refactorización pasa por definir las operaciones a realizar, mostrar que las reglas de validez se conservan y proponer cuándo ejecutar la refactorización. Finalmente, para que una refactorización definida pueda ser aplicada eficazmente, es deseable contar con una herramienta que la realice automáticamente o al menos asista en su ejecución.

La herramienta debe garantizar que a partir de elementos correctos, una vez aplicada la transformación, se obtengan elementos correctos. Para esto es necesario conocer de manera precisa la estructura de los elementos a transformar y sus reglas de validez. Esto ha condicionado que los trabajos realizados hasta el momento en refactorización sean dependientes del lenguaje que define a los elementos objeto de las transformaciones, haciendo que se multipliquen los esfuerzos en la definición de las refactorizaciones por cada lenguaje en particular y por cada proceso de desarrollo de las herramientas que las ejecutan.

Sin embargo, la experiencia en el área nos lleva a las siguientes ideas. En primer lugar, las refactorizaciones no modifican el comportamiento sino la arquitectura que se ha diseñado para la solución, de modo que no todas las construcciones de los lenguajes son importantes a la hora de definir e implementar refactorizaciones. Por otra parte, de la experimentación con diferentes lenguajes podemos detectar que aunque dichos lenguajes tengan características diferentes, si enfocamos aquellas que han sido fundamentales para definir e implementar una refactorización, encontramos muchos conceptos comunes. La perspectiva esbozada por las ideas anteriores se resume en que podemos avanzar hacia establecer, al menos, una cierta independencia del lenguaje.

Nuestro enfoque para abordar esta nueva perspectiva consiste en concentrarnos en los aspectos comunes y en las construcciones que interesan desde el punto de vista de la refactorización, definiendo un lenguaje modelo que responda a estas características. De esta forma, contando con dicho modelo, como primer paso, podemos definir las refactorizaciones para las construcciones del modelo y
reducir los esfuerzos, de particularizar para un lenguaje, a mirar cada lenguaje por el prisma del modelo.

Otros aspecto muy importante, y muchas veces pasado por alto, en el ámbito de las refactorizaciones es el análisis de las consecuencias que tiene su aplicación para los elementos que dependen de los transformados y para los objetos persistentes que fueron creados previos a la modificación. La definición de las refactorizaciones sobre el modelo también ayuda a abordar esta tarea. En [7] hemos definido un lenguaje modelo para el análisis y definición de refactorizaciones, que posteriormente hemos aplicado en casos de estudio.

3 Trabajos relacionados

En [8] se presenta el modelo FAMIX como un meta-modelo para almacenar información en un repositorio que permita la integración de diferentes entornos de desarrollo de software con soporte para la refactorización y por otro lado se presenta un estudio de factibilidad, a partir del análisis de dos lenguajes, JAVA y SMALLTALK, para validar su propuesta de meta-modelo que abstrae las características necesarias para realizar refactorizaciones. JAVA y SMALLTALK son lenguajes con diferencias muy marcadas como la presencia de meta-clases en SMALLTALK y de interfaces en JAVA, y de tipo estático en JAVA y dinámico en SMALLTALK. De esta forma obtienen un modelo que considera la abstracción de estas características.

En cambio en nuestro modelo nos hemos concentrado en abstraer características de lenguajes basados en clases, estática y fuertemente tipados y con genericidad, haciendo especial énfasis en la consideración de características avanzadas en cuanto a herencia y genericidad. Hemos comprobado en la implementación para JAVA de la refactorización que definimos en [9], que la adaptación para considerar interfaces a lo JAVA no es un problema para nuestro modelo, ni la ausencia de tipos. Sin embargo creemos que la consideración de meta-clases es un problema mayor para nuestro modelo, al igual que la inclusión en FAMIX de genericidad al mismo nivel que en el nuestro, es también un problema mayor. Una línea interesante sería intentar fusionar ambos modelos.

Hasta donde conocemos, éste es el único trabajo que se ha publicado con un propósito igual al nuestro: ir hacia un motor de refactorizaciones independiente del lenguaje.

La idea, en el sentido de brindar soporte para múltiples lenguajes en una herramienta de refactorización, fue esbozada en [10], aunque también puede inspirarse en las direcciones de investigación que propone [11] en lo que denomina “tecnologías para lenguajes genéricos”.

A continuación presentamos el enfoque de nuestra propuesta mediante la descripción, primeramente, del lenguaje modelo definido y su posterior aplicación a la definición, el análisis y la implementación de refactorizaciones, mostrando finalmente un caso de estudio.
4 El lenguaje MOON

La esencia del lenguaje modelo que hemos propuesto radica en intentar ser minimal, en el sentido de enfocar solamente en las construcciones que aportan información fundamental a la hora de definir y analizar refactorizaciones, y a la vez intentar que sea lo más general posible para dar cabida a una amplia familia de lenguajes. Con este objetivo se han estudiado un conjunto de lenguajes para abstractar sus características. El resultado de dicho estudio se puede consultar en [7].

Con este objetivo se han definido variantes en el modelo. La variación se centra en las reglas de tipos. Tener bien definidas las reglas de las construcciones del lenguaje y del sistema de tipos permiten analizar si las refactorizaciones, una vez aplicadas, dan lugar a elementos correctos en cuanto a estructura y tipos.

El lenguaje modelo minimal-general que se define se denomina MOON$^1$. MOON no intenta ser un lenguaje para programar sino para abstractar cómo se representan estructuralmente los principales conceptos y construcciones a tener en cuenta para definir y analizar refactorizaciones. En este lenguaje las clases son implementaciones implícitas de tipos y se considera la presencia de herencia múltiple. La estructura de los tipos que dichas clases implementan, estará dada por un modelo de especificación de tipos basado en estructura y signatura.

MOON se basa en ISTBOPL, lenguaje definido en [12]. El lenguaje ISTBOPL es una extensión que se hace a partir de otro lenguaje nombrado BOPL$^2$, añadiendo a este último: declaración de tipos, herencia y genericidad. BOPL fue definido como un lenguaje básico a partir del cual se van añadiendo propiedades, hasta llegar a ISTBOPL. BOPL es un lenguaje basado en clases mientras que ISTBOPL es un lenguaje OO, según la clasificación definida en [13]$^3$. Estos lenguajes fueron diseñados para ilustrar construcciones básicas, son lenguajes básicos sin adornos. En la Figura 1 se muestra la relación de las características de MOON respecto a sus antecesores.

Se ha decidido basar MOON en ISTBOPL porque este último constituye un tronco común a la familia de lenguajes estudiados y porque ya asume simplificaciones que nos ayudan en el intento de buscar minimalidad. La presentación de MOON la haremos en dos partes, la primera refleja la búsqueda de aún más minimalidad a partir de ISTBOPL y la segunda la obtención de más generalidad a partir de la consideración de más complejidad en la herencia, la genericidad y el sistema de tipos.

---

$^1$ MOON es un acrónimo de Minimal Object-Oriented Notation.
$^2$ BOPL e ISTBOPL son acrónimos de Basic Object Programming Language y de Inheritance, Substitution and Types for BOPL, respectivamente.
$^3$ Breve resumen de la clasificación de Wegner: 1- lenguaje basado en objetos, ofrece construcciones para definir objetos, 2- lenguaje basado en clases, ofrece construcciones para definir clases y objetos, 3- lenguaje orientado a objetos, ofrece construcciones para definir clases, objetos e incorpora el mecanismo de herencia.
Un lenguaje modelo para la definición y análisis de refactorizaciones

Figura 1. Precedentes de MOON.

4.1 Simplificaciones a partir de ISTBOPL

En el modelo MOON se proporciona una gramática de la sintaxis concreta para la descripción de las clases, definiciones de la sintaxis abstracta y reglas formales del sistema de tipos (comunes y variantes). A partir de la gramática de la descripción de una clase se proporciona la definición de la estructura de los objetos y de los objetos persistentes.

La gramática que se presenta en el Apéndice A define la sintaxis concreta de MOON y por tanto determina la estructura de las clases que se pueden definir. Un módulo en MOON se corresponde con la definición de una clase (regla 1). Cada clase tiene atributos y métodos que pueden ser rutinas o funciones. Una rutina ejecuta un conjunto de instrucciones y una función computa un resultado. Los atributos y los métodos definen las propiedades de las clases. Las signaturas de las propiedades intrínsecas de las clases se declaran en el grupo de signaturas (reglas 5 y 13). La regla 23 define las anotaciones de tipo que se pueden hacer en MOON, a la que conducen las declaraciones de entidades (reglas 19 y 22).

Sintácticamente un envío de mensaje a una entidad se denota, como es habitual, con la notación de punto, es decir: entidad.mensaje. En MOON se define sin pérdida de generalidad, que no se permiten envíos de mensajes (llamadas) en cascada. Todo envío de mensajes en cascada se puede transformar en una sucesión de envíos de mensajes. Las llamadas en cascada, $e_1, e_2, \ldots e_n$ ($n > 2$), que aparecen en los elementos de un lenguaje en particular, que se mira desde el prisma de MOON, al margen de los argumentos que pueda tener cada mensaje involucrado en la cascada, se pueden reducir considerando la siguiente secuencia de instrucciones MOON:

\[
t_1 := e_1, e_2; \\
t_2 := t_1, e_2; \ldots
\]
\[ t_{i-1} := t_{i-2} e_i \quad (2 < i < n) \]

si \( e_1, e_2, \ldots, e_n \) es una expresión, \( t_{n-1} := t_{n-2} e_n \)

si \( e_1, e_2, \ldots, e_n \) es una instrucción, \( t_{n-2} e_n \) 

(1)

Esta reducción del envío de mensajes en cascada no va encaminada a dar una alternativa al tratamiento dinámico ni a la generación de código, sino a hacer un tratamiento estático simplificado de la cascada. De esta forma las dependencias implícitas de tipos que ocasiona una cascada se eliminan, quedando todas las dependencias expresadas de forma clara y explícita como propone la ley de Deme ter [14]. La eliminación de las llamadas en cascada está recogida en las reglas 36 y 41.

Por otra parte, los argumentos reales en los envíos de mensajes solamente podrán venir dados por expresiones atómicas (una entidad o una constante manifesta). Este caso tampoco implica una pérdida de generalidad puesto que si se quiere analizar la presencia de una expresión no atómica \( \exp \triangleq e_1, \ldots, e_n \) (\( n > 1 \)) como argumento real en un envío de mensaje tal que \( a(f(e_1, \ldots, e_n)) \), el análisis se puede hacer sobre la aplicación de (1) a \( e_1, \ldots, e_n \) y la sustitución \( a(f(t_{n-1})) \). Esta consideración se refleja en la regla 44.

En el lenguaje \textsc{istBOPL}, que se tomó como punto de partida para definir MOON, las formas de expresiones admitidas son: expresiones atómicas, expresiones de envío de mensajes, expresiones binarias y expresiones unarias. De las formas anteriores de expresiones, en MOON no se incluyen las dos últimas (regla 39). Las expresiones binarias o unarias se pueden considerar como una expresión de envío de mensaje (e.g. \texttt{a+b} como \texttt{a\texttt{+}(b)}).

También en \textsc{istBOPL} el conjunto de instrucciones está formado por: instrucción compuesta, instrucción de asignación, instrucción de creación de objetos, instrucción de envío de mensajes, instrucción condicional (\texttt{if \texttt{EXPR} then \texttt{INSTR} else \texttt{INSTR}) e instrucción de repetición (\texttt{while \texttt{EXPR} do \texttt{INSTR}}). De este conjunto de instrucciones, en MOON se eliminan las dos últimas, condicional y de repetición que son las instrucciones de control del flujo de ejecución definidas en \textsc{istBOPL} (regla 32). Esto se debe a que desde el punto de vista de los análisis que interesan a la hora de refactorizar, la presencia en un lenguaje particular de instrucciones como estas se miran desde el prisma de MOON como:

\[
c_1 \colon \text{Boolean}; \quad c_2 \colon \text{Boolean};
\]

\[
c_1 := \text{EXPR}; \quad c_2 := c_1 \texttt{eq} \texttt{(} \text{BOOLEA\_CONST} \text{)};
\]

\text{INSTR}

De esta forma se pueden analizar claramente las relaciones de dependencias entre clases, las asociaciones de entidades, la corrección en cuanto a tipos, etc., sin prestar atención a más detalles.

### 4.2 Reglas y variantes del sistema de tipos de MOON

A partir de \textsc{istBOPL}, para definir MOON en la búsqueda de reflejar la riqueza de conceptos de una mayor familia de lenguajes, se añade la capacidad de declarar herencia múltiple de clases y una amplia gama de modificadores de herencia,
se añade también la capacidad de acotar los parámetros formales de una clase
genérica y se enriquece el sistema de tipos considerando algunas variantes.

En el sistema de tipos de MOON:

- se considera la existencia un tipo universal \( U \) que es implementado por una
clase predefinida de nombre Object. Toda clase hereda de Object de forma
implícita, directa o indirectamente.
- se considera la existencia de clases predefinidas que son implementación de
tipos básicos de objetos.
- no se definen relaciones entre los tipos básicos.
- se define un núcleo de reglas comunes y variantes.

Las reglas comunes se definen de forma similar al sistema de tipos de Fun
descrito en [15], teniendo en cuenta las observaciones aportadas en [16], y la
correspondencia de renombramiento que se utiliza en la definición de subtipo
de [17]. En la variante 1) las reglas de la herencia y la asignación polimórfica
se basan en las reglas de subtipo (contravarianza) y los tipos paramétricos
se acotan mediante subtipo, un ejemplo de lenguaje particular que se adapta
da esta variante de MOON es TRELIS/Owl. En la variante 2) las reglas de
la herencia se basan en las reglas de subtipo (contravarianza) y los tipos pa-
ramétricos se acotan mediante cláusulas \( tal \ que \)\(^4\); un ejemplo de lenguaje que se
adapta a esta variante de MOON es Theta. En la variante 3) las reglas de la
herencia se basan en reglas de conformidad (covarianza) y los tipos paramétricos
se acotan mediante conformidad, un ejemplo de lenguaje que se adapta a esta
variante de MOON es Eiffel.

Un lenguaje particular que no incluya genericidad se puede mirar fácilmente
desde el prisma de MOON, igualmente lenguajes como C++ que incluyen una
forma muy rudimentaria de genericidad en la que no hay acotación. En estos
casos lo único que sucede es que se simplifica aún más el modelo. Lo mismo
podemos decir de lenguajes sin anotaciones de tipo.

La definición de MOON tiene como limitación que no permite fácilmente el
tránsito a poder modelar lenguajes con nivel meta avanzado, como Smalltalk.
Pero este es un aspecto que podremos aclarar en un futuro tomando como base
lo que se ha desarrollado en FAMIX para ese caso.

4.3 Estructura de los objetos

La definición de la estructura de los objetos depende en última instancia de la
estructura de las clases y de la definición de los tipos completamente instancia-
dos que dichas clases implementan. Conociendo la estructura de los objetos de
un tipo dado, se podrá analizar las consecuencias de la aplicación de las refac-
torizaciones para evaluar si son respetuosas con los objetos, lo que conduciría a
saber si es necesario hacer, lo que en bases de datos se denomina, migración de
poblaciones.

\(^4\) cláusulas \( tal \ que \), del término en inglés \( where \) \( clauses \).
Los objetos forman parte de la extensión de algún tipo no paramétrico o de algún tipo paramétrico completamente instanciado. El modelo de tipos asumido define que la estructura de un tipo está dada por un record cuyos campos se corresponden con las signaturas de las propiedades del tipo, atributos y métodos. Por eso es inmediato pensar en la estructura de un objeto como el valor de un record. El valor de un record contiene los valores de cada campo definido en el record.

Digamos entonces que en el record que representa el objeto tenemos los valores de los campos que se corresponden con atributos, el valor correspondiente al identificador del tipo del objeto, un campo cuyo valor será el identificador único del objeto al que se le suele denominar oid y un campo que indica dónde encontrar las implementaciones de los métodos. Este campo se conoce como referencia a la tabla de métodos.

Los valores de los atributos de un objeto en MOON pueden ser: una referencia a otro objeto o constantes manifiestas (regla 40, Apéndice A). Como referencia a otro objeto bastaría tener el valor de su oid. Esto hace homogéneo el valor de los atributos en la estructura de los objetos persistentes y de los objetos en ejecución. Por claridad asumimos que los valores son homogéneos, es decir, las referencias a objetos son oids.

La estructura de los objetos MOON, en presencia de herencia de sus clases, es una estructura basada en concatenación [18]. Esta es la forma más frecuente, según hemos detectado, en las implementaciones de los lenguajes revisados.

Se define que los objetos persistentes tienen la misma estructura que los objetos en ejecución excepto la referencia a la tabla de métodos pues esta se actualiza cuando el objeto se carga desde el soporte externo durante una ejecución, utilizando el valor del identificador del tipo del objeto.

El análisis de las consecuencias para los objetos de la aplicación de las refactorizaciones, se hace a partir de la estructura de los objetos persistentes. Puede parecer que para este análisis también haría falta definir la estructura de la tabla de métodos pues, evidentemente, cuando se aplica una refactorización, esta puede cambiar. Una vez que se realice la refactorización, hay que volver a “compilar” (o realizar una acción con un efecto similar) las clases transformadas. De esta forma la tabla de métodos se regenera. Esto no se refleja en la estructura del objeto pues solamente tiene una referencia a la tabla. Para lo que sí tiene consecuencias es para los mensajes que el objeto es capaz de responder. Pero determinar esto se corresponde con el análisis de las consecuencias de la refactorización para los clientes. A partir de la estructura del objeto lo que se analiza es si los objetos persistentes, que están dados fundamentalmente por los valores de sus atributos, siguen siendo válidos en una sesión posterior a la refactorización.

5 MOON y refactorización

Se ha definido una arquitectura y un proceso para integrar el modelo en herramientas de refactorización. En este apartado presentamos ambos aspectos.
5.1 Arquitectura

En el marco de MOON definimos una arquitectura de refactorizadores en la que entran clases a un repositorio pasando por un analizador que extrae los elementos relevantes (definidos por MOON) y almacena el resultado en forma de un objeto persistente cuya estructura responde a un modelo de clases que describe las partes más relevantes del árbol sintáctico que se genera del análisis de una clase MOON. A este formato le llamamos formato de clases preanalizadas. La entrada de una clase al repositorio implica la actualización de las relaciones entre las clases preanalizadas. Las relaciones entre las clases preanalizadas se han definido formalmente en el modelo de MOON como un Grafo de Dependencias que tiene en cuenta las relaciones de herencia y cliente entre las clases, así como las relaciones de herencia y cliente con sustitución (para los parámetros formales) en presencia de clases genéricas. El modelo formal está definido mediante un grafo dirigido etiquetado.

Se establecen reglas de formación del Grafo de Dependencias a partir de la estructura de las clases. Las reglas de formación consideran la presencia de relaciones con sustitución para expandir las dependencias. También se definen consultas que permiten extraer información del repositorio a partir de la estructura de las clases preanalizadas y del Grafo de Dependencias (e.g. clases de las que depende una clase, clases que dependen de una clase, origen de una propiedad, etc.).

A partir de estas definiciones, en la arquitectura propuesta, la estructura de la nueva clase que llega al repositorio determina la forma en que se actualiza el Grafo de Dependencias.

Las herramientas de refactorización actúan sobre las clases preanalizadas que se encuentran en el repositorio. De los objetos persistentes que representan las clases preanalizadas del repositorio se debe poder extraer el texto equivalente en el lenguaje de partida. De esta forma se obtienen los resultados de la refactorización. La Figura 2 muestra un esquema de dicha arquitectura.
Con este enfoque se disminuye la complejidad de la definición de una herramienta de refactoring pues ésta se centra en elementos que responden a una estructura que contiene sólo lo que tiene que considerar la herramienta para refactorizar, apartándole de liñar con los detalles propios del análisis de un lenguaje particular.

Como ya hemos dicho, es muy importante analizar si la refactorización obtiene elementos correctos, cómo se afectan las clases que dependen de las clases modificadas y cómo se afectan los objetos persistentes que hayan sido creados previamente a la refactorización. De la misma forma que la definición, el análisis se ve simplificado por el modelo. El Grafo de Dependencias nos permite analizar las clases que deben ser transformadas y la forma en que se afectan las clases que dependen de las transformadas. La correción de los nuevos elementos y las consecuencias para los objetivos se analiza desde la estructura y las reglas definidas en MOON.

5.2 Proceso

El proceso que guía la inserción de MOON en el ámbito de las refactorizaciones tiene dos partes fundamentales: definición y análisis, e implementación, que se resumen a continuación.

1. Definición y Análisis
   (a) Poner en correspondencia el lenguaje objetivo con MOON
   (b) Definir en MOON la refactorización
   (c) Analizar sus consecuencias

2. Implementación
   (a) Construir un analizador del lenguaje objetivo que obtiene como resultado para cada clase un objeto cuya estructura viene dada por el modelo de clases que se deriva del árbol sintáctico si se analizan textos MOON. Estos objetos están enlazados por un grafo que representa el Grafo de Dependencias del Repositorio.
   (b) Construir un recuperador de textos en el lenguaje objetivo a partir de dicho Grafo. Este paso y el anterior se condicionan.
   (c) Implementar la refactorización definida, actuando sobre el Grafo de Dependencias del Repositorio.

En este escenario, la intención de agregar una nueva refactorización para el mismo lenguage objetivo implica trabajar en 1(b), 1(c) y 2(c), mientras que aplicar la misma refactorización para elementos de un nuevo lenguaje implica trabajar en 1(a), 2(a) y 2(b).

6 Un caso de estudio

En esta sección se presenta un caso de estudio del modelo en dos partes. En primer lugar un caso de estudio del modelo como abstracción de lenguajes, y
en segundo lugar un caso de estudio del modelo como base para la definición y análisis de refeactorizaciones. Para esto se han elegido el lenguaje Eiffel y una refeactorización definida por nosotros y presentada en [19] y [20].

La selección de Eiffel viene dada por sus recursos en cuanto a generacidad (incluyendo acotaciones) y de la herencia que permiten explotar bien el modelo. Además, la claridad de la sintaxis, el tratamiento uniforme de las entidades, la ausencia de punteros y el manejo automático de la memoria, facilitan la definición e implementación de los analizadores necesarios.

6.1 Eiffel visto desde MOON

El lenguaje seleccionado (Eiffel) se corresponde con la variante 3) de MOON, teniendo en cuenta además algunos aspectos particulares como los tipos ancla y expandidos, su conjunto de instrucciones y sus formas de expresiones.

Todas las construcciones se pueden reducir, a la hora de realizar un análisis estático de las clases Eiffel, de la misma forma que se presentaron las simplificaciones de MOON en la Sección 4. Las instrucciones de creación, asignación, y envío de mensajes y las formas de expresiones de envío de mensajes y constantes manifestadas son la base de las demás.

Evidentemente, hay que tener en cuenta, tal y como se analizó, en la presentación de MOON (al no incluir las instrucciones if y while) que ciertas instrucciones exigen un tipo determinado de expresiones, e.g. expresiones de tipo Boolean, como en la instrucción if, o de tipo Integer, como en la parte variant de una instrucción loop de Eiffel. La solución que se presentó en las simplificaciones de MOON: asumir que existe una asociación con una constante manifiesta del tipo requerido, es válida para este caso.

Los aspectos relativos al conjunto de instrucciones de Eiffel se vuelven a reducir a la instrucción de creación, la instrucción de asignación (el intento de asignación ?: en Eiffel se analiza desde MOON como una asignación), y la instrucción de envío de mensajes. Los envíos de mensajes en cascada se simplifican. En cuanto a las formas de expresiones que se corresponden con expresiones con operadores binarios y unarios en Eiffel, se tratarán como expresiones de envío de mensajes, tal y como se expuso en Sección 4. Esto se puede hacer directamente en Eiffel, sin tener que cambiar ninguna consideración, porque todos los tipos básicos se corresponden con una clase de la biblioteca Kernel.

6.2 Desarrollo de una refeactorización: parameterize

En este apartado se presenta una refeactorización que hemos definido y denominado parametrización. El objetivo de esta refeactorización es obtener clases genéricas a partir de clases que no lo son y transformar el software basado en estas clases para utilizar las nuevas clases genéricas y sus instancias.

Dada una orden de parametrización

```plaintext
C.parameterize(e as T)
```
donde $C$ es la clase objetivo, la clase que se quiere hacer genérica; $e$ es la enti-
dad guiad, la entidad cuyo tipo específico va a pasar a estar dado por el nuevo
parámetro formal $T$; la ejecución de la refactorización pasa por:

1. Determinar el grafo que forman las clases que participarán en la obtención
de las entidades genérico-dependientes a partir de la entidad guiad y las rela-
ciones entre dichas clases. A este grafo se le denomina universo de trabajo y
se denota $G^I$.

Las entidades genérico-dependientes son aquellas que deben modificarse su ti-
po a genérico producto del cambio del tipo de $e$. El grafo del universo de
trabajo se especifica muy fácilmente según el modelo del Grafo de Depen-
dencias establecido a partir de MOON. Este grafo estará determinado por
los condicionantes de $C$ (ancestros y proveedores), y por sus descendientes.

2. Determinar las entidades genérico-dependientes directas e indirectas.

Las entidades genérico-dependientes pueden ser directas o indirectas, en alu-
sión a la forma en que cambia su tipo. El tipo de la entidad $e$ cambia para
ser el nuevo parámetro genérico formal, algunas entidades (las directas) cam-
biarán de la misma forma, mientras que otras (las indirectas) cambiarán para
que su tipo pase a ser un tipo genérico instanciado con el nuevo parámetro
(si su tipo era A ahora sería A[T].

La clara especificación de las reglas del sistema de tipos de MOON y la
abstracción en sus construcciones simplifican en gran medida la definición
de cómo obtener estas entidades y el análisis de la corrección de su selección.

Hay que garantizar corrección de tipos una vez ejecutada la transformación.

Las reglas fundamentales a tener en cuenta son las que guían la corrección
de las asociaciones entre entidades y expresiones. Las reglas 34, 35, 42 y 44
son las que determinan, y la ausencia de envíos de mensajes en cascada, así
como la restricción sobre las expresiones que pueden ser argumentos reales
en un envío de mensaje, clarifican la definición formal y el análisis de la
formación de los conjuntos de entidades genérico-dependientes.

3. Determinar si es posible proseguir con la parametrización y con qué clases.

Con esto se obtiene un subgrafo del grafo $G^I$. Este subgrafo pasa a constituir
el grafo de las clases candidatas a participar en la parametrización y se denota
$G^C$.

Nuevamente las reglas de tipos y las asociaciones determinan. Se hace nece-
sario un análisis exhaustivo de los cambios que pueden ocurrir y de las reglas
que se deben cumplir para determinar qué circunstancias conducirían a ob-
tener elementos incorrectos, y prohibirlas. Por ejemplo, no se puede obtener
una entidad cuyo tipo es un parámetro formal asociada con una constante
manifesta.

4. Si del paso anterior se determina que no se puede proseguir o sí no se desea:
Terminar

5. Eliminar de $G^C$ las clases a las que no es necesario propagar la operación
(las clases que no contienen ninguna de las entidades genérico-dependientes
determinadas).
El grafo resultante se denomina grafo de clases finales y se denota $G_F$. Consultar si se desea progresar y con qué clases, rearranando $G_F$ cada vez que se elimine una clase interactivamente.

Las entidades que tienen que cambiar su tipo determinan las clases que tienen que ser modificadas y las relaciones de dependencia de éstas determinan si la eliminación de una clase conduce a la eliminación de otras.

6. Analizar la generosidad acotada.

Se analiza la necesidad de acotar los nuevos parámetros formales. Se determina cuáles deben ser las restricciones. La forma de expresarlas varía de acuerdo a la variante de acotación modelada en MOON. La que mejores resultados da es la variante 2) (acotación mediante cláusulas tal que).

7. Comenzar la parametrización a partir de las clases ancestros en el grafo de herencia que subyace en $G_F$ hasta las descendientes (considerando el análisis de generosidad acotada).

Las estructuras que pueden ser afectadas son las dadas por las siguientes reglas:

regla 3, porque de no haber sido genérica la clase, habría que modificar la construcción que da paso a los parámetros formales.

regla 7, porque habría que añadir el nuevo parámetro formal, y en el caso de la variante 2), en esta regla se da paso a la construcción para acotar el nuevo parámetro formal si es necesario (regla 49W).

regla 8, porque en las variantes 1) y 3) es aquí donde hay que dar paso a la construcción para acotar el nuevo parámetro formal si es necesario (regla 48S).

reglas 49W y 50W, en el caso de la variante 2).

regla 48S, en las variantes 1) y 3).

regla 10, porque hay que verificar si el padre, tiene como clase determinante una clase que ha sido modificada por la parametrización. En ese caso, el tipo del padre debe modificarse añadiéndole el parámetro real correspondiente a la sustitución.

reglas 19 y 22, porque las entidades genérico-dependientes declaradas a partir de estas reglas deben cambiar a variable su tipo.

Los cambios en las reglas 48S, 10, 19 y 22 condicionan que hay que reformar las dependencias expresadas en el Grafo de Dependencias del Repositorio. Las dependencias cambian en las siguientes formas: una relación puede pasar a ser de sustitución, puede cambiar la sustitución de una relación, puede aparecer una nueva relación producto de la inclusión de acotación para los nuevos parámetros formales en el caso de las variantes 1) y 3)

8. Reajustar las clases que no están en $G_F$ y que dependen directamente (hijos o clientes directos) de las clases parametrizadas para que pasen a depender de las nuevas clases genéricas instanciadas.

Este último indica que la refactorización parameterize no respeta a los clientes (directos) de las clases transformadas. El otro aspecto significativo a analizar sobre las consecuencias de la refactorización es su impacto en los objetos persistentes. El modelo de objetos basado en concatenación, la estructura de los
objetos persistentes con el identificador de su tipo (ausencia de referencia a los métodos), y la gestión del repositorio sobre la identificación de los tipos hace que la transformación no tenga consecuencias para los objetos persistentes.

Esta refactorización se implementó en el caso de estudio de Eiffel, siguiendo la arquitectura diseñada (Figura 2). Los únicos aspectos que hubo que tener en cuenta por encima del modelo de clases preanalizadas según MOON fueron: no transformar el tipo de una entidad declarada como tipo ancla pues la declaración como ancla es más significativa que tener las dos entidades declaradas del mismo tipo; y tener en cuenta la presencia de tipos expandidos en el análisis de generalidad acotada, pues si el resultado de la acotación es un tipo expandido, debido a las reglas de conformidad de tipos, se limitan las posteriores instanciaciones.

Se construyeron las siguientes herramientas:

- un preanalizador de clases Eiffel: genera objetos persistentes representando las clases preanalizadas según el modelo de MOON
- un gestor de repositorio que mantiene el Grafo de Dependencias y permite extraer información de los objetos que representan las clases preanalizadas.
- un refactorizador que implementa la operación *parameterize*

Con este caso de estudio hemos comprobado la viabilidad del modelo y de la arquitectura diseñada para la definición y análisis de refactorizaciones. Estamos trabajando actualmente en una versión de Java con generalidad para migrar automáticamente bibliotecas de clases no genéricas. El modelo y arquitectura MOON está mostrándose muy efectivo en la reducción del esfuerzo de implementar la refactorización para otro lenguaje, lo que era uno de los objetivos fundamentales de la propuesta.

7 Conclusiones

Hemos presentado un lenguaje modelo, MOON, para la definición y análisis de refactorizaciones, así como una arquitectura para su integración en herramientas de refactorización. El propósito fundamental es disminuir el esfuerzo de definición y desarrollo de herramientas para refactorizar elementos de diferentes lenguajes. Se ha realizado un caso de estudio de la viabilidad de este propósito dando resultados muy positivos.

Nuestro trabajo parece estar en la misma línea que el iniciado por Tichelaar et al. [8] (Sección 3). A diferencia de este último, que define un modelo de intercambio de información mediante un esquema entidad-relación y su especificación se basa en CDIF, nosotros definimos un lenguaje modelo dado por una gramática de atributos y las reglas de tipos del lenguaje, entre otros aspectos, que conduce a una representación de las clases analizadas según MOON como objetos persistentes cuya estructura viene dada por el modelo de clases que se deriva del árbol sintáctico. Derivar de esta estructura un modelo de intercambio resulta inmediato. Por otra parte, creemos que esta forma de abordar el problema nos establece un punto de partida inmejorable para abordar la refactorización independiente del lenguaje. En este sentido, nuestras actividades apuntan ya a la generación
automática de herramientas de refactoring partiendo de las gramáticas y del sistema de tipos de MOON y del lenguaje de programación objetivo.

Referencias


A Sintaxis concreta de MOON

1. MODULE
2. CLASS_DEF
3. CLASS_NAME
4. HEADER
5. SIGNATURES
6. CLASS_BODY
7. FORMAL_PARAMETERS
8. INHERITANCE_LIST
9. INHERITANCE_CLAUSE
10. CLASS_TYPE
11. MODIFIER
12. STATIC
13. ATTRIBUTES
14. METHODS
15. METHODS
16. METHODS
17. METHODS
18. METHODS
19. METHODS
20. METHODS
21. METHODS
22. METHODS
23. METHODS
24. METHODS
25. METHODS
26. METHODS
27. METHODS
28. METHODS
29. METHODS
30. METHODS
31. METHODS
32. METHODS
33. METHODS

continúa en la página siguiente...
... continuación de la página anterior

34 CREATION INSTR \[create VARJD\]
35 ASSIGNMENT INSTR \[VARJD := 'EXPR\]
36 CALL INSTR \[CALL INSTR LONG1 | CALL INSTR LONG2 no hay llamadas en cascada\]
37 CALL INSTR LONG1 \[METHOD JD | REAL ARGUMENTS\]
38 CALL INSTR LONG2 \[CALL EXPR LONG1, CALL INSTR LONG1\]
39 EXPR \[MANIFEST CONSTANT | CALL EXPR se reducen las formas de expresiones\]
40 MANIFEST CONSTANT \[null | REAL CONSTANT | INTEGER CONSTANT | BOOLEAN CONSTANT | CHAR CONSTANT | STRING CONSTANT\]
41 CALL EXPR \[CALL EXPR LONG1 | CALL EXPR LONG2 no hay llamadas en cascada\]
42 CALL EXPR LONG1 \[ENTITY | REAL ARGUMENTS\]
43 CALL EXPR LONG2 \[ENTITY, CALL EXPR LONG1\]
44 REAL ARGUMENTS \[\(\ldots\)\] \[\text{EXPR ATOM} \[\ldots\]+ \]
45 EXPR ATOM \[MANIFEST CONSTANT | CALL EXPR LONG1\]
46 ENTITY \[VARJD | result | self\]
47 PROP JD \[VARJD | METHOD JD\]

Los símbolos \textit{BOUND\textsubscript{S}} y \textit{BOUND\textsubscript{W}} que aparecen en las reglas anteriores, dependen de que variante de acotación de parámetros genéricos formales esté en consideración. Las reglas correspondientes a estos símbolos se definen a continuación de acuerdo a la variante de lenguaje a la que dará lugar.

48 \textit{BOUND\textsubscript{S}} \[\rightarrow\textit{CLASS TYPE} es cliente\]
49 \textit{WHERE\textsubscript{S}} \[e\]
50 \textit{BOUND\textsubscript{W}} \[e\]
51 \textit{WHERE\textsubscript{W}} \[WHERE\textsubscript{CLAUSE} \[\ldots\]+\]
52 \textit{WHERE\textsubscript{CLAUSE}} \[\textit{FORMAL\textsubscript{GEN JD}} has SIG\textsubscript{LIST}\]

Cuando la forma de acotación de los parámetros genéricos formales que se defina para el lenguaje esté dada por acotación mediante subtiempo o conformidad, las reglas que se utilizan son las marcadas con S (reglas 48S y 49S).

Cuando la forma de acotación de los parámetros genéricos formales que se defina para el lenguaje esté dada por acotación mediante cláusulas tal que, las reglas que se utilizan son las marcadas con W (reglas 48W, 49W y 50W).
El Visitante Genérico Extensible*

Francisco Moya, José M. Moya y Juan Carlos López

Universidad de Castilla-La Mancha
Escuela Superior de Informática
E-13071 Ciudad Real, España
{fmoya, jmmoya, jclopez}@inf-cr.uclm.es

Resumen En el presente artículo describimos la implementación de una versión del patrón *Visitante* [2]. Esta implementación fue diseñada en el contexto del desarrollo del núcleo de simulación *Kepler*, con el fin de añadir soporte de almacenamiento persistente de la mayoría de las estructuras de datos empleadas. Frente a un visitante tradicional, el *Visitante Genérico Extensible* permite extender una jerarquía cualquiera de elementos concretos visitables sin tener que modificar todos los visitantes, incluso en tiempo de ejecución.

1 Introducción

El patrón *Visitante* [2] es usado con mucha frecuencia en todas las labores que requieren recorrer estructuras de datos realizando operaciones específicas en cada elemento de la estructura (e.g. almacenamiento persistente). Las operaciones de recorrido quedan delegadas en la propia estructura de datos mientras que las operaciones concretas que hay que realizar se agrupan en una clase visitante.

El patrón visitante consta de dos jerarquías de clases paralelas: las clases concretas visitables y las clases visitantes, que realizan operaciones específicas con objetos de cada clase concreta. En la descripción de Gamma et al., los primeros se caracterizan por tener un método *accept* que admite como argumento un objeto visitante e invoca un método de ese objeto visitante. Los objetos visitantes tienen una serie de métodos específicos *visitXXX* para cada clase concreta. Esta técnica suele también llamarse *despachado dinámico doble*.

El patrón clásico resulta muy conveniente cuando las estructuras de datos están prefijadas en la propia aplicación, pero hoy en día es cada vez más frecuente que las aplicaciones puedan extenderse mediante bibliotecas de carga dinámica (*plugins*) y se tiende a utilizar formatos de almacenamiento igualmente extensibles, como XML. Estas características impiden el uso del patrón visitante tradicional en C++, puesto que no es posible extender las tablas de despachado dinámico de la clase visitante en tiempo de ejecución.

Nosotros proponemos una implementación del visitante que abstrae la tabla de funciones virtuales en forma de un objeto función, que denominamos *specializer*. Así mismo se propone una implementación genérica del patrón en forma

* Este trabajo ha sido financiado por el Programa Nacional de I+D+I (CICYT) a través del proyecto CORE, TIC2000-0583-C02-01
2 Implementación del visitante

Nuestra implementación comparte con el Extrínseco Visitor [3] y con el Indirect Visitor [5,6] las motivaciones de reducción de acoplamiento entre los objetos participantes, pero es considerablemente más simple su implementación y su uso. De forma análoga a los ya citados, la capacidad de poder ampliar un visitante en tiempo de ejecución obligará a transgredir un principio básico del patrón visitante [7]:

The Visitor pattern is the classic technique for recovering lost type information without resorting to dynamic casts.

El Visitante Genérico Extensible necesita recurrir al operador de conversión dinámica de tipos de C++ dynamic_cast para permitir que elementos visitables concretos puedan ser incorporados en tiempo de ejecución mediante la carga de bibliotecas dinámicas. Por contra, un visitante tradicional debe conocer todos los elementos visitables concretos presentes en el sistema con antelación. Con el esquema propuesto en este artículo los visitantes preexistentes funcionarán sin problemas si se extiende el número de elementos concretos, pero sólo los nuevos visitantes podrán aprovechar la existencia de nuevos elementos concretos.


Figura 1. Diagrama de clases UML para el Visitante Genérico Extensible

En la figura 1 se muestra un diagrama de clases UML de nuestra implementación. El usuario debe proporcionar el interfaz común para todas las clases visitables relacionadas (llamado abstract en el diagrama). Con este interfaz se
puede instanciar la plantilla visitable_base para constituir la raíz de la jerarquía de clases visitables. Esta plantilla simplemente añade el método virtual accept y servirá como clase base para todos los visitantes de la jerarquía:

```cpp
template <class T>
class visitable_base: public T {
    typedef visitable_base<T> _Self;
public:
    visitable_base() {}
    virtual ~visitable_base() {} // 
public:
    struct visitor_base {
        typedef _Self base_type;
        virtual void visit(_Self& vb) {} // 
    };
    virtual void accept(visitor_base& v) = 0;
};
```

Como puede apreciarse, el método visit tiene una implementación vacía en lugar de ser un método abstracto, como en el caso del visitante tradicional. La intención es similar a la del Default Visitor descrito en [3]. Este método se invocará en caso de no disponer de un método visit específico para una determinada clase visitable concreta.

Para cada clase visitable concreta definimos una clase visitante diferente, que simplemente añade un método sobrecargado visit especializado. Este método es equivalente a los métodos visitXXX en el patrón visitante tradicional:

```cpp
template <class T>
struct visitor: public virtual T::visitor_base {
    typedef T base_type;
    virtual void visit(T& cc) const = 0;
};
```

El parámetro de plantilla T se asume que deriva de una instancia de la clase visitable_base definida anteriormente (raíz de la jerarquía de visitables). El caso normal, en el que se requiere que un mismo visitante pueda visitar múltiples tipos concretos, se puede implementar bien utilizando herencia múltiple, o bien utilizando un adaptador [2] que realice una búsqueda más sofisticada del visitante concreto. Ambos casos serán analizados a continuación.

El caso más simple, en el que se utiliza herencia múltiple para crear una clase visitante, justifica el uso de herencia virtual al derivar de visitable_base.

En cualquiera de las dos posibilidades mencionadas se necesita un mecanismo para obtener un visitante especializado a partir de la referencia al visitante base que se pasa al método accept. Esta traducción se lleva a cabo mediante un specializer. En el caso más simple, visitantes compuestos mediante herencia múltiple, la especialización corresponde a un simple dynamic_cast:

```cpp
template <class T>
class trivial_specializer {
    typedef visitor<T> visitor_type;
};
```
public:
    operator visitor_type*() { return _q; }
    template <class Q> trivial_specializer(Q & t)
    : _q(dynamic_cast<visitor_type*>(&t)){} 
private:
    visitor_type* _q;
};

En este caso se trata de un simple envolvente de dynamic_cast. En otros casos el objeto función encargado de la especialización puede permitir la carga dinámica de módulos y consultas a una base de datos.

El caso más simple está prácticamente completo. Tan solo falta definir un método accept adecuado para cada una de las clases visitables. La implementación es idéntica en todos los casos, se trata de una invocación al método visit especializado. Por tanto es un candidato ideal para ser implementado como una plantilla de C++:

```
template <class T, class specializer = trivial_specializer<T>>
class visitable: public T {
    typedef visitable<T> _Self;
public:
    typedef typename T::visitor_base visitor_base;
    typedef visitor<T> visitor_type;
public:
    void accept(visitor_base & v) {
        visitor_type* cv = specializer(v);
        if (cv) cv->visit(*this);
        else v.visit(*this);
    }
};
```

Como en el caso del Default Visitor propuesto en [3], si el visitante no puede ser especializado para una clase concreta invoca el método visit no especializado, definido en la base de la jerarquía de visitantes. Este método no especializado también puede redefinirse si se desea (la plantilla visitable_base lo define con una implementación nula).

3 Ejemplo de uso

Un pequeño ejemplo de uso puede servir para clarificar nuestra implementación:

```
#include "visitor.hh"
#include <iostream>

class Root {}

typedef visitable_base<Root> VisitableRoot;

struct A_base: public VisitableRoot {
    void sayHello() { cout << "Hello_from_A" << endl; }
```
struct B_base: public VisitableRoot {
void sayGoodbye() { cout "Goodbye_from_B" endl; }
};
typedef visitable<A_base> A;
typedef visitable<B_base> B;

struct Visitor: public A::visitor_type, public B::visitor_type {
void visit(A_base& a) const { a.sayHello(); }
void visit(B_base& b) const { b.sayGoodbye(); }
}

int main()
{
A a; B b; Visitor v;
a.accept(v); b.accept(v);
return 0;
}

El código de este ejemplo no es significativamente más complejo que la implementación de un visitante tradicional, pero tampoco explota las capacidades de extensión que ofrece el Visitante Genérico Extensible. La siguiente sección describe cómo extender un visitante de este tipo.

3.1 Visitantes extensibles

Para jerarquías de objetos basadas en prototipos resulta muy conveniente disponer de un visitante que se pueda extender durante la ejecución de la aplicación. En ese caso el visitante contendrá un conjunto de visitantes específicos, que podrían haberse añadido en tiempo de compilación o de ejecución:

class visitor_dynamic: public visitor_base {
    typedef const type_info key_type;
    typedef map<key_type*, visitor_base*> container_type;
    public:
        void append_visitor(const key_type& k, visitor_base& v) {
            _known[k]= &v;
        }
        void remove_visitor(const key_type& k, visitor_base& v) {
            _known.erase(&k);
        }
        visitor_base* lookup(const key_type& k) {
            return _known[k];
        }
    private:
        container_type _known;
    };
Todas las clases visitantes específicas deben derivarse de visitor_base y el visitante que las contiene debe ser una instancia de visitor_dynamic.

En este caso el objeto función encargado de especializar al visitante deberá consultar la tabla de clases concretas visitables conocidas (_known) utilizando el soporte de información de tipos en tiempo de ejecución de C++. Esto obliga a una invocación adicional de dynamic_cast con respecto al caso del trivial_specializer.

```cpp
template <class T>
class dynamic_specializer {
    typedef visitor<T> visitor_type;
    typedef typename T::visitor_dynamic visitor_dynamic;

    public:
    operator visitor_type*() { return _q; }
    template <class Q>
    dynamic_specializer(Q& t) : _q(dynamic_cast<visitor_type*>(&t))
    {
        if (!_q) {
            visitor_dynamic* v = dynamic_cast<visitor_dynamic*>(&t);
            if (v)
                _q = dynamic_cast<visitor_type*>(v->lookup(
                    typeid(visitor_type)->lookup()));
        }
    }

    private:
    visitor_type* _q;
};
```

Nótese que dynamic_specializer también funciona con visitantes normales (no dinámicos) con una pequeña penalización en tiempo de ejecución.

### 3.2 Cuellos de botella

La generalidad y flexibilidad de nuestra implementación de visitantes implica una serie de ineficiencias respecto al visitante tradicional:

1. La necesidad de usar dynamic_cast para no requerir un conocimiento previo de toda la jerarquía de clases concretas visitables. Esto es causa de una penalización relativamente importante en muchos compiladores. En la figura 2 se muestra una gráfica de la sobrecarga introducida por el uso de dynamic_cast en lugar de una simple llamada a un método virtual utilizando diversos compiladores. En la figura se comparan dos versiones recientes del compilador de GNU. Como puede apreciarse, el impacto puede ser muy diferentes dependiendo del compilador utilizado.

2. La necesidad de usar herencia virtual desde la clase raíz de la jerarquía de visitantes, para evitar ambigüedades al definir un visitante para múltiples clases concretas. Esto añade un nivel de indirección adicional, aunque el impacto global puede ser despreciado en la mayoría de las situaciones (ver fig. 3).
Figura 2. Representación gráfica de la sobrecarga debida al uso de dynamic_cast.

Figura 3. Representación gráfica de la sobrecarga debida a la herencia virtual.
3. La necesidad de tres niveles de herencia. Esto no es causa de degradación de prestaciones, pero un número excesivo de niveles de herencia es un síntoma de un diseño pobre. En nuestro caso los dos niveles de herencia adicionales con respecto al visitante tradicional son debidos a extensiones comunes aplicadas mediante plantillas C++ a clases provistas por el usuario. Realmente estas extensiones no introducen acoplamiento innecesario, puesto que las plantillas C++ de nuestro patrón visitante se suponen estables.

Además de las ineficiencias mencionadas es frecuente encontrar rechazo a la herencia múltiple. En ese caso el usuario puede simplemente proporcionar un objeto función de especialización, diferente del trivial_specializer, que retorne un nuevo visitante especializado en lugar de convertir explícitamente el tipo del visitante utilizado en la llamada a accept.

4 Conclusiones

Nuestra implementación ofrece un compromiso entre generalidad y prestaciones. Mantiene la integridad de tipos característica de C++ y permite detectar la mayoría de los errores en tiempo de compilación. Por otro lado logra un alto grado de desacoplamiento entre los integrantes de la jerarquía de clases concretas visitables y clases concretas visitantes. Frente a otras versiones del patrón visitante, el Visitante Genérico Extensible añade:

- Desacoplamiento de visitantes y visitables hasta un nivel similar al Extrinsic Visitor [3].
- Implementación genérica utilizando plantillas de C++ para evitar la replicación de código.
- Capacidad de extensión en tiempo de ejecución o de compilación.

A pesar de todo, para algunas aplicaciones es muy importante evitar la pérdida de rendimiento debida al uso de dynamic_cast. En esos casos C++ no puede proporcionar un método para automatizar la declaración y manejo del patrón visitante, pero podría implementarse utilizando herramientas de preprocesado como OpenC++ [1] que añaden propiedades reflexivas a C++.

Referencias

2. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
Generic Description, Behavior and Animation of Visual Modeling Languages

Hartmut Ehrig, R. Bardohl, and C. Ermel


Visual modeling techniques including UML as well as graph and net based techniques are of growing interest for software system specification and development. The GENGED approach developed at the Technical University of Berlin allows already the generic description of visual modeling languages based on formal graph transformation and graphical constraint solving techniques and tools.

In this paper, the GENGED approach is reviewed and extended in order to allow the description of dynamic behavior and animation of systems. The basic idea is to define visual behavior and animation rules on top of the rules defining the corresponding visual modeling language and to allow a domain specific layout for the animation view of the system. A simple version of a traffic light system is used as running example, where the system view is given by a Petri net. The animation view shows directly the dynamic changes of the colors of traffic lights at a street crossing.
Modelling Implementations as Institution Morphisms*

C. Domínguez, L. Lambán, M. V. Pascual and J. Rubio

Departamento de Matemáticas y Computación, Universidad de La Rioja. Edificio Vives, Luis de Ulloa s/n, E-26004 Logroño (La Rioja, Spain). {cedomin,lalamban,mvico,jurubio}@dmc.unirioja.es

Abstract. In this paper, the role of coalgebras for specifying a concrete software system is explained. We show how, from this particular goal, a generic construction of institutions for coalgebras (on certain constant functors) has been obtained. This also reveals a new application field in which the two concepts algebra/coalgebra (and the initial/final matters) are fruitfully confronted.

Keywords. Institution, algebraic specification, coalgebra, symbolic computation

1 Introduction and Motivation

When an analyst undertakes the task of specifying a symbolic computation system in Algebraic Topology he finds a challenging context. This was the case when we started the study of the data structures in the EAT (Effective Algebraic Topology) system [17, 16]. In such a system two layers of data structures exist. In the first layer, one finds the usual data structures as (finite) lists or trees of symbols or integer numbers (for representing linear combinations, polynomials and so on). In the second layer, one must deal with algebraic structures as (graded) groups or rings in which the elements are data belonging to the first layer. Besides, two additional features of these layers enrich even more the problem. On the one hand, structures of the two layers have to be created and handled at runtime. This should be compared with other mathematical packages (in the field of Commutative Algebra or in general systems such as Mathematica or Maple) in which the two above mentioned layers obviously exist, but in which only one (or very few) algebraic structure is created in each session (in a pre-processing time, one can say) and then a massive work on usual (first layer) data is carried out inside this structure. In contrast, in a typical calculation with EAT, several hundred algebraic structures must be created and handled through several hours of CPU time (see examples in [17, 16]).

On the other hand, the interesting algorithms in Algebraic Topology use, in an essential way, infinite structures as intermediaries for computing the sought

* Partially supported by DGES, project PB98-1621-C02-01 and by Universidad de La Rioja, project API-00/B28
(finite) results. (The loop space construction is the more important of this kind [16].) Let us stress that these infinite data structures are not the usual sequential or hierarchical structures, as infinite lists or streams or infinite trees, in which canonical ways of traversing the structure are known. In EAT, one must implement infinite groups, rings and so on, in which the aspects of behavior and observation are more important than the storing and traversing aspects.

These particular characteristics imply that, even if for the first-layer data structures the usual algebraic specification techniques (and the initial algebra construction, in particular) are enough for the modelling task, for the second-layer data structures, a research effort is required, since the direct application of known techniques is not possible.

Our first remark was that in a system such as EAT we are not only implementing an Abstract Data Type, or ADT (as a group, for instance), but also dealing, at runtime, with implementations (in the example, several hundred implementations of the ADT group would populate the program memory). Working with the classical notion of ADT implementation due to Hoare [8] and dealing with the semantic subtleties of the programming language in which EAT was developed (Common Lisp), we were able in [11] of proving that the EAT (second-layer) data structures were as general as possible, in the sense that they are ingredients of final objects in certain categories of ADT implementations. Later on, directed by this finality property found, we reinterpreted in [4, 12] our results related to hidden and coalgebraic technology in a purely (co)algebraic setting (without considering implementation or programming language issues).

In this paper, we continue this line and we introduce some constructions in the institutional framework with the aim of illustrating that our previous results give some generic insights on the field of (co)algebraic specifications.

After this motivating introduction, the paper is organized as follows. Section 2 is devoted to collect some well known definitions and facts on (co)algebraic specification. In Section 3, a general procedure to obtain an institution for coalgebras from another institution is introduced. Then, Section 4 particularizes this general construction to the case in which the source institution is one of the well-known algebraic institutions with certain constraints on the carrier sets. In this same section, the interpretation of this institution as for hidden specifications and implementation matters is briefly considered too. The paper ends with a section of conclusions and future work.

2 Preliminaries

In order to ease the reading of the paper, we start recalling some well-known definitions presented as in [13].

**Definition 1 (Signature and Algebra).** A signature $\Sigma$ is a pair $(S, \Omega)$ of sets, whose elements are called sorts and operations respectively. Each operation consist of a $(k + 2)$-tuple, $\omega : s_1 \cdots s_k \rightarrow s$ with $s_1, \ldots, s_k, s \in S$ and $k \geq 0$. In the case $k = 0$, the operation is called a constant of sort $s$. 
Let $\Sigma = (S, \Omega)$ and $\Sigma' = (S', \Omega')$ be two signatures. A signature morphism $\mu : \Sigma \to \Sigma'$ from $\Sigma$ to $\Sigma'$ is a pair $\mu = (\mu_S : S \to S', \mu_\Omega : \Omega \to \Omega')$ of functions such that for each operation $n : s_1 \ldots s_k \to s \in \Omega, k \geq 0$, there exists an operation $m$, with $\mu_\Omega(n : s_1 \ldots s_k \to s) = (m : \mu_S(s_1) \ldots \mu_S(s_k) \to \mu_S(s))$.

Let $\Sigma = (S, \Omega)$ be a signature. A total algebra for $\Sigma$ (or $\Sigma$-algebra) assigns a set $A(s)$ to each sort $s \in S$, called the carrier set of the sort $s$, and a total function $A(\omega : s_1 \ldots s_k \to s) : A(s_1) \times \ldots \times A(s_k) \to A(s)$ to each operation $\omega \in \Omega$.

The $\Sigma$-algebras can be organized as a category $\mathbf{Alg}(\Sigma)$ using the following natural notion of morphism. Let $A, B$ be two $\Sigma$-algebras, $\Sigma = (S, \Omega)$. A $\Sigma$-homomorphism $h : A \to B$ from $A$ to $B$ is a family $\{h_s : A(s) \to B(s)\}_{s \in S}$ of functions such that

$$h_s(A(\omega)(a_1, \ldots, a_k)) = B(\omega)(h_{s_1}(a_1), \ldots, h_{s_k}(a_k))$$

for $\omega : s_1 \ldots s_k \to s \in \Omega$ and for all $a_i \in A(s_i), i = 1, \ldots, k$.

Next, we introduce the notion of coalgebra (which first appeared in the field of algebraic specification in [14] and which has been developed in [18, 9, 10], among other papers).

**Definition 2 (Coalgebra).** Let $F : \text{Set} \to \text{Set}$ be an endofunctor on $\text{Set}$, the category of sets. Then a $F$-coalgebra is a couple $(A, c_A)$, where $A$ is a set and $c_A : A \to F(A)$ is a map. A morphism between $F$-coalgebras $(A, c_A), (B, c_B)$ is a map $f : A \to B$ such that $F(f)c_A = c_Bf$.

The $F$-coalgebras, together with the morphisms between them, define a category which is denoted by $\text{CoAlg}(F)$.

The following definitions of institution and institution morphism have been extracted from [6].

**Definition 3 (Institution).** An institution $\mathcal{I}$ consists of

1. a category $\mathbf{Sign}$, whose objects are called signatures,
2. a functor $\mathbf{Sign} \to \text{Set}$, giving for each signature a set whose elements are called sentences over that signature,
3. a functor $\mathbf{Mod} : \mathbf{Sign} \to \mathbf{Cat}$ giving for each signature $\Sigma$ a category whose objects are called $\Sigma$-models, and whose arrows are called $\Sigma$-(model)morphisms, and
4. a relation $\models \subseteq \text{Obj}(\mathbf{Mod}(\Sigma)) \times \mathbf{Sen}(\Sigma)$ for each $\Sigma \in \text{Obj}(\mathbf{Sign})$, called $\Sigma$-satisfaction,

such that for each morphism $\phi : \Sigma \to \Sigma'$ in $\mathbf{Sign}$, the Satisfaction Condition

$$m' \models_{\Sigma'} \mathbf{Sen}(\phi)(e) \iff \mathbf{Mod}(\phi)(m') \models_{\Sigma} e$$

holds for each $m' \in \text{Obj}(\mathbf{Mod}(\Sigma'))$ and each $e \in \mathbf{Sen}(\Sigma)$.
Definition 4 (Institution morphism). Let \( I \) and \( I' \) be institutions. Then, an institution morphism \( \Phi : I \rightarrow I' \) consists of

1. a functor \( \Phi : \text{Sign} \rightarrow \text{Sign}' \),
2. a natural transformation \( \alpha : \Phi; \text{Sen}' \Rightarrow \text{Sen} \), and
3. a natural transformation \( \beta : \text{Mod} \Rightarrow \Phi; \text{Mod}' \)

such that the following Satisfaction Condition holds

\[
m \models_\Sigma \alpha_S(e') \iff \beta_S(m) \models_{\Phi(\Sigma)} e'
\]

for any \( \Sigma \)-model \( m \) from \( I \) and any \( \Phi(\Sigma) \)-sentence \( e' \) from \( I' \).

As usual, the first example of institution is the one obtained from the equational algebraic specification [13]:

Example 1 (Equational algebraic institution). We can organize the equational algebraic specifications as an institution considering

- the category \( \text{Sign} \) as the category with objects the signatures \( \Sigma \) and morphisms the signature morphisms,
- the functor \( \text{Sen} : \text{Sign} \rightarrow \text{Set} \), giving for each signature \( \Sigma \) the set of sentences \( EL(\Sigma) \), that is to say, pairs of \( \Sigma \)-terms,
- the functor \( \text{Mod} : \text{Sign} \rightarrow \text{Cat}^{\text{op}} \), giving for each signature \( \Sigma \) the category \( \text{Alg}(\Sigma) \),

and considering as \( \Sigma \)-satisfaction the usual equational satisfaction definition obtained from the interpretation of a \( \Sigma \)-term in a \( \Sigma \)-algebra.

To finish this preliminary section, we define the notion of hidden algebra which will be used in Section 4.

Definition 5 (Hidden algebra). Let \( V \Sigma = (V_S, V_\Omega) \) be a signature. Let us fix a \( V \Sigma \)-algebra \( D \) and let us include in \( V_\Omega \), as constants, the elements of the carrier sets of \( D \) which do not correspond to constants previously in \( V_\Omega \).

The elements of \( V_S \) are called visible sorts and those of \( V_\Omega \) are called visible operations. The \( V \Sigma \)-algebra \( D \) is called data domain. Then a hidden signature, on \( V \Sigma \) and \( D \), is a signature \( H \Sigma = (S, \Omega) \) such that:

- \( S = HS \cup V_S \); the elements of \( HS \) are called hidden sorts of \( H \Sigma \).
- \( \Omega = H_\Omega \cup V_\Omega \) and for each operation \( \omega : s_1 \ldots s_n \rightarrow s \) in \( H_\Omega \) the following property hold: in \( s_1, \ldots, s_n \) there is one and only one hidden sort and it is assumed that this hidden sort appears in the first position (that is, it is \( s_1 \)).

(This definition only covers a particular case of the notion introduced in [7], but it is enough for our purposes in this paper.)

A hidden algebra \( A \) for a hidden signature \( H \Sigma \), on \( V \Sigma \) and \( D \), is a \( H \Sigma \)-algebra such that \( A_{V \Sigma} = D \) (in other words, the restriction of \( A \) to the visible part is equal to the data domain \( D \)). A hidden morphism between two hidden algebras is a \( H \Sigma \)-homomorphism \( f \) such that \( f_D \) is the identity on \( D \).

The hidden algebras for \( H \Sigma \) on \( V \Sigma \) and \( D \), together with the hidden morphisms, define a category \( \text{HAlg}^D(H \Sigma) \).

Hidden specifications can be organized as an institution (see, for instance, [2]) but this fact will not be used in this paper.
3 Coalgebric Institution Associated to an Institution

3.1 Definition of Institutions for Coalgebras

The starting point is an institution $I$, with the property that for each $\Sigma \in \text{Obj}(\text{Sign}_I)$, $\text{Mod}_I(\Sigma)$ is a small category, that is to say, the class $\text{Obj}(\text{Mod}_I(\Sigma))$ is a set. This condition could seem very restrictive, but in fact, it is not. For instance, if we are specifying a software system by means of an algebraic institution, it is quite natural to impose that any $\Sigma$-algebra has carrier sets which are subsets of a fixed data universe, namely the set of all objects definable in a particular programming language. Then the corresponding institution has the aforementioned property.

From such an institution $I$, a coalgebraic institution $\text{CoAlg}(I)$ is introduced. Let us stress that these institutions do not cover general coalgebras; only coalgebras on (particular) constant functors are considered, but this poor class of coalgebras is enough for our modelling purposes. The detailed construction is given in the following definition.

Definition 6 (Coalgebraic institution associated to an institution). Let $I = (\text{Sign}_I, \text{Sen}_I, \text{Mod}_I, \models^I)$ be an institution so that for each $\Sigma \in \text{Obj}(\text{Sign}_I)$, $\text{Mod}_I(\Sigma)$ is a small category. We define the coalgebraic institution associated to $I$, denoted by $\text{CoAlg}(I)$ as follows:

1. The category $\text{Sign}_{\text{CoAlg}(I)}$ is the category with
   - objects: for each $\Sigma \in \text{Obj}(\text{Sign}_I)$ we include as object of $\text{Sign}_{\text{CoAlg}(I)}$ the endofunctor $F^\Sigma : \text{Set} \to \text{Set}$ which is constant on the set $\text{Obj}(\text{Mod}_I(\Sigma))$,
   - morphisms: for each $\mu : \Sigma \to \Sigma' \in \text{Morph}(\text{Sign}_I)$, we include as morphism between $F^\Sigma$ and $F^{\Sigma'}$ in $\text{Sign}_{\text{CoAlg}(I)}$ the natural transformation $F^{\Sigma'} \Rightarrow \mu F^\Sigma$, between the endofunctors $F^\Sigma$ and $F^{\Sigma'}$, defined in the natural way through the signature morphism $\mu$.

2. The functor $\text{Sen}_{\text{CoAlg}(I)} : \text{Sign}_{\text{CoAlg}(I)} \to \text{Set}$ is defined by $\text{Sen}_{\text{CoAlg}(I)}(F^\Sigma) = \text{Sen}_I(\Sigma)$ for each endofunctor $F^\Sigma$ with $\Sigma \in \text{Obj}(\text{Sign}_I)$. For a morphism $F^{\Sigma'} \Rightarrow \mu F^\Sigma$, between the endofunctors $F^\Sigma$ and $F^{\Sigma'}$, the functor is defined in the natural way through the signature morphism $\mu$.

3. The functor $\text{Mod}_{\text{CoAlg}(I)} : \text{Sign}_{\text{CoAlg}(I)} \to \text{Cat}^{\text{op}}$ is defined by $\text{Mod}_{\text{CoAlg}(I)}(F^\Sigma) = \text{CoAlg}(F^\Sigma)$ for each endofunctor $F^\Sigma$ with $\Sigma \in \text{Obj}(\text{Sign}_I)$. For a morphism $F^{\Sigma'} \Rightarrow \mu F^\Sigma$, between the endofunctors $F^\Sigma$ and $F^{\Sigma'}$, the functor is defined in the natural way through the natural transformation.

4. The satisfaction condition $\models^I_{F^\Sigma} : \text{Mod}_{\text{CoAlg}(I)}(F^\Sigma) \times \text{Obj}(\text{Sign}_{\text{CoAlg}(I)})$ is defined as

$1$ The contravariant flavour of the definition is introduced in order to maintain the variancy which is usual when dealing with institutions (that is to say, the target of $\text{Sen}$ is $\text{Set}$ and that of $\text{Mod}$ is $\text{Cat}^{\text{op}}$).
3.2 Institution Morphism between an Institution $I$ and its Coalgebraic Institution $CoAlg(I)$

It is intuitively clear from the construction above that the institution $I$ can be mapped in $CoAlg(I)$. Nevertheless, to achieve this mapping it is necessary to impose a new condition on $I$: each morphism in $Mod_I(\Sigma)$ must be an endomorphism. We will see in Section 4 that this condition is naturally held in our context (and, in fact, in the context of hidden specifications, too).

Thus, let $I = (Sign_I, Sen_I, Mod_I, |-^\Sigma)$ be an institution such that for each $\Sigma \in Obj(Sign_I)$, $Mod_I(\Sigma)$ is a small category in which a morphism is always an endomorphism. We define the following mappings.

1. A functor $\Phi_I : Sign_I \to Sign_{\text{CoAlg}(I)}$ is defined as:
   - for each $\Sigma \in Obj(Sign_I)$, $\Phi_I(\Sigma) = F_{\Sigma}$, the constant endofunctor on $Obj(Mod_I(\Sigma))$.
   - for each $\mu : \Sigma \to \Sigma'$ in $\text{Morph}(Sign_I)$, $\Phi_I(\mu) = F_{\Sigma'} \Rightarrow \mu F_{\Sigma}$ is the natural transformation between the corresponding endofunctors $F_{\Sigma'}$ and $F_{\Sigma}$ (recall that this is a $Sign_{\text{CoAlg}(I)}$-morphism from $F_{\Sigma}$ to $F_{\Sigma'}$).

2. A natural transformation $\alpha_I : \Phi_I; Sen_{\text{CoAlg}(I)} \Rightarrow Sen_I$, is defined as a family of applications $\alpha_I : Sen_{\text{CoAlg}(I)}(\Phi_I(\Sigma)) \to Sen_I(\Sigma)$, where $\alpha_I$ is the identity for each $\Sigma \in Obj(Sign_I)$.

3. A natural transformation $\beta_I : Mod_I \Rightarrow \Phi_I; Mod_{\text{CoAlg}(I)}$ is defined by a family of functors $\beta_I : Mod_I(\Sigma) \Rightarrow Mod_{\text{CoAlg}(I)}(\Phi_I(\Sigma))$, one for each $\Sigma \in Obj(Sign_I)$ such that $\beta_I(A) = \{\ast\}, \alpha_A : \{\ast\} \to \Phi_I(\Sigma)(\{\ast\})$ where $\alpha_A$ is defined by $\alpha_A(\ast) = A$, i.e. the coalgebra defined by a singleton as set and the map applying the unique element to $A$, for each $A \in Mod_I(\Sigma)$.

It is straightforward to check that these mappings define an institution morphism, and then we obtain the following result.

**Theorem 1.** Let $I$ be an institution so that for each $\Sigma \in Obj(Sign_I)$, $Mod_I(\Sigma)$ is a small category and each morphism in $Mod_I(\Sigma)$ is an endomorphism. Then, there exists a canonical institution morphism between $I$ and $CoAlg(I)$.
We have preferred to explain our ideas in the more precise and comfortable setting of equational algebraic specifications.

First of all, a set $U$ (the universe of sorts) is fixed. Then, for each $s \in U$ a (non-empty) set $D_s$ is also fixed. The family $D = \{D_s\}_{s \in U}$ is called data universe and, if no confusion can arise, we will not explicitly refer to the universe of sorts $U$.

**Definition 7 (Algebraic institution defined over a data universe).**

The equational algebraic institution defined over a data universe $D$, $\mathcal{I}^D = (\text{Sign}_{\mathcal{I}^D}, \text{Sen}_{\mathcal{I}^D}, \text{Mod}_{\mathcal{I}^D}, \models_{\mathcal{I}^D})$ is defined as the equational algebraic institution evoked in the Example 1, except for the following modifications. The objects in $\text{Sign}_{\mathcal{I}^D}$ are signatures $\Sigma = (S, \Omega \cup C)$, where $S \subseteq U$, $\Omega$ is a set of operations without constants and $C = \{d : s \mid \text{for each } d \in D_s, \text{ for each } s \in S\}$ is a set of constants. The morphisms in $\text{Sign}_{\mathcal{I}^D}$ are morphisms between signatures such that the constants in $C$ are invariant. For each $\Sigma = (S, \Omega \cup C) \in \text{Obj}(\text{Sign}_{\mathcal{I}^D})$, $\text{Mod}_{\mathcal{I}^D}(\Sigma)$ is the category whose objects are $\Sigma$-algebras $A$ such that $A(s) = D_s$ for each $s \in S$ and $A(d) = d$ for each $d \in C$, and the only morphisms on $\text{Mod}_{\mathcal{I}^D}(\Sigma)$ are the identities.

Thus, in the institutions $\mathcal{I}^D$, the two conditions on the size and the morphisms in the categories of models are satisfied and we conclude the existence of a canonical institution morphism between $\mathcal{I}^D$ and the corresponding coalgebraic institution $\text{CoAlg}(\mathcal{I}^D)$. Anyone who knows something about hidden specifications will observe a parallelism between our constructions and terminology and the hidden techniques. But, before making even more explicit this relationship, we want to give an interpretation with respect to our initial modelling problem.

**4.2 Coalgebraic Institutions and ADT Implementations**

Let us come back to the ideas presented in the introduction. It is quite obvious that the data universe $D$ is related with the first-layer of data structures in EAT. In fact, even if in the previous subsection the data universe was fixed once and as a whole, the practical way of working is defining each $D_s$ as the initial model for a previous signature (usually based on the built-in Common Lisp operations for managing list, arrays and so on). In this context, it is not necessary to introduce in each signature the complete set of constants $C$, condition which is quite unrealistic in most cases (see [5] and [15] for details on this constructing process).

It is clear that the second-layer data structures in EAT should be related to the models of the institution $\mathcal{I}^D$. Nevertheless, it is also clear that each model represents one algebraic structure (one group, for example), but in EAT we must deal not only with one group but with families of groups (which will be created at runtime) and, besides, we must model the implementations (that is to say, the computer-memory counter-part of $\Sigma$-algebras) of such models. Then, the conclusion is that second layer EAT data structures are implementations of the models in $\mathcal{I}^D$ and therefore that algebraic institutions such as $\mathcal{I}^D$ are too poor to formally specify the features of the EAT system.
In [11], we rely on Hoare’s notion of ADT implementation [8] in order to model these data structures and then we are obliged to deal with invariant, equality, partiality and Common Lisp semantics matters. If all these technicalities are skipped and we place ourselves in the simpler setting of pure algebraic specifications (even if doing so, we move away from our main goal: modelling the actual features of the EAT system), we realize that our second layer data structures are implementing nothing but families of $\Sigma$-algebras. That is to say, we move from a category of models in $\mathcal{I}^D$ to an indexed (or fibered) category on it (see [20]). Or, from another point of view, we can identify each second layer EAT data structure with a coalgebra recovering the corresponding $\Sigma$-algebras. This is exactly what was formally expressed in the previous section in the transfer from $\mathcal{I}^D$ to $\text{CoAlg}(\mathcal{I}^D)$. Let us note that the idea of seeing a coalgebra as an implementation is not really fulfilled by the institution morphism. This is true from the syntactic point of view (that is to say, the functors in the image of the morphism can be accurately interpreted as the signatures for the implementations of $\Sigma$-algebras) but from the semantic point of view, the models in the image of the morphism represent almost trivial implementations (concretely, they are implementations of a unique datum). However, we will see in Subsection 4.4 how these images can be gathered (as coproducts) to produce final objects which directly correspond to the data structures in EAT. But before stating this result, the relationship with hidden specifications is presented.

4.3 The Hidden Specification Point of View

Let $\Sigma = (S, \Omega)$ be a signature in $\mathcal{I}^D$, and let us suppose that $\Omega = (\omega_1, \ldots, \omega_m)$ is an enumeration of the operation symbols (here we are not considering the constants $C$, for the reasons explained in the previous subsection). A new signature $\Sigma_{\text{Imp}} = (S_{\text{Imp}}, \Omega_{\text{Imp}})$ is defined as follows:

- $S_{\text{Imp}} = \{\text{imp}_\Sigma\} \cup S$ where $\text{imp}_\Sigma$ is a fresh symbol (that is to say, a symbol which does not belong to the universe of sorts $U$ of $\mathcal{I}^D$),
- $\Omega_{\text{Imp}} = (\text{imp}_\omega_1, \ldots, \text{imp}_\omega_m)$ in which for each operation $\omega : s_1 \ldots s_n \rightarrow s \in R$, an operation $\text{imp}_\omega : \text{imp}_\Sigma s_1 \ldots s_n \rightarrow s \in \Omega_{\text{Imp}}$ is included.

This signature can be considered a hidden signature [7] by declaring $\text{imp}_\Sigma$ as a unique hidden sort. Then, the $\Sigma_{\text{Imp}}$-hidden algebras can be interpreted as families of $\Sigma$-algebras and the relation with EAT data structures becomes clear.

The category of hidden algebras $\text{HAlg}^D(\Sigma_{\text{Imp}})$ is canonically equivalent to the category of coalgebras $\text{Mod}_{\text{CoAlg}(\mathcal{I}^D)}(\Phi_{\Sigma}(\Sigma))$ (as it can be deduced, for instance, from [3]), showing that, as it is widely considered, the two formalism are roughly equivalent in expressiveness. In addition, it is not difficult to check that not only signatures and models, but also sentences and satisficability, are well-transferred through hidden logics. In other words, the mappings of Subsection 3.2 factorize through a hidden institution [2], completing the picture of our approach².

² But, please note that we are not claiming that our institution morphism can be expressed as the composition of two institution morphisms.
4.4 Final Objects

The existence of a final object in the category $\text{Mod}_{\text{CoAlg}(I^D)}(F\Sigma)$ can be reached from a good number of sources: from the indexed or fibered category theory [20], from general results on coalgebras [1], from the coalgebraic specification area [14], from the hidden approach [7], or in our very particular case, simply by an elementary category theory argument, because that category is nothing but a category of sets over a fixed set (slice category).

This final object can be described as the identity map $\mathbb{1}^D : \text{Obj}(\text{Mod}_{I^D}(\Sigma)) \to \text{Obj}(\text{Mod}_{I^D}(\Sigma))$. Interestingly enough, to take advantage of such an inoffensive theoretical object it was necessary to use all the power of Common Lisp as a functional programming language [11]. This simple object can be also understood as the cause of the “universal” and very general scope of EAT and, in particular, of its capability for dealing with infinite data structures (this is not surprising because it is well-known that coalgebras have always been proposed for the specification of infinite data structures; see [18] for instance).

In EAT [16], the final object is encoded by means of a record of Common Lisp functions (lexical closures [19]) with a field for each operation in $\Sigma$. Note that the elements of each algebraic structure (which is represented by an instance of the record) are chosen among the data $D$ which is fixed in advance, or constructed from initial semantics, and so they do not require an explicit storing. If we consider that each particular implementation (i.e., each particular instance of the record) is “summed” in the record structure, we obtain a practical interpretation of the following result, which shows that the models in the image of the institution morphism enable the re-construction of the final objects (as in the case of the existence of final objects, the proof of this theorem can be interpreted as a particular case of more general results on coalgebras [18]).

**Theorem 2.** The final object $\mathbb{1}^D$ is the coproduct of the image on models of the canonical institution morphism from $I^D$ to $\text{CoAlg}(I^D)$.

**Proof.** For each $A \in \text{Obj}(\text{Mod}_{I^D}(\Sigma))$, we have the $F\Sigma$-coalgebra $\beta_{I^D}(A) = (\{\ast\}, \alpha_A : \{\ast\} \to \Phi_{I^D}(\{\ast\})\{\ast\})$, where $\alpha_A$ is defined by $\alpha_A(\ast) = A$. We can consider a morphism $i_A$ between this coalgebra and $\mathbb{1}^D$, concretely $i_A : \{\ast\} \to \text{Obj}(\text{Mod}_{I^D}(\Sigma))$, defined also by $i_A(\ast) = A$. Let $(X, \alpha : X \to F\Sigma(X))$ be another $F\Sigma$-coalgebra so that a morphism $h_A : \{\ast\} \to X$ between the coalgebras $\beta_{I^D}(A)$ and $(X, \alpha)$ exists for each $A \in \text{Obj}(\text{Mod}_{I^D}(\Sigma))$. We can then define $f : \text{Obj}(\text{Mod}_{I^D}(\Sigma)) \to X$ as $f(A) = h_A(\ast)$, $\forall A \in \text{Obj}(\text{Mod}_{I^D}(\Sigma))$, and it is clear that this is the (unique) morphism so that $f(i_A(\ast)) = h_A(\ast)$ holds, $\forall A \in \text{Obj}(\text{Mod}_{I^D}(\Sigma))$.

5 Conclusions and Future Work

In this paper we have showed the interest of the coalgebraic methods for modelling features of actual software systems such as EAT, a symbolic computation
system for Algebraic Topology. In addition, we have also illustrated how this kind of applied research can have theoretical relevance, in our case, casting light on a new algebra-coalgebra relationship and introducing a family of coalgebraic institutions with an intuitive and practical interpretation.

Without leaving the field of coalgebraic methods, further work will be necessary to know if our coalgebraic institutions can be generalized in order to cover coalgebras associated to non-constant functors without losing our intuitive and practical interpretation. This will be the first step to establish an institutional comparison between the hidden and the coalgebraic approaches, comparison which, up to the authors’ knowledge, is still missing in the literature (this missing bridge has prevented us from presenting our institution morphism as the composition of two institution morphisms through a hidden institution).

References


Tight and Loose Semantics for Transformation Systems

Fernando Orejas\textsuperscript{2}, Hartmut Ehrig\textsuperscript{1}, and Elvira Pino\textsuperscript{2}

\textsuperscript{1} FB 13 Informatik, Tech. Univ. Berlin, Franklinstrasse 28/29, 10587 Berlin, Germany.

ehrig@cs.tu-berlin.de

\textsuperscript{2} Dpto de L.S.I., Universidad Politécnica de Catalunya, Campus Nord, Módul C6, Jordi Girona 1-3, 08034 Barcelona, Spain. \{pino,orejas\}@lsi.upc.es

Abstract. When defining the requirements of a system, specification units typically are partial or incomplete descriptions of a system component. In this context, providing a complete description of a component means integrating all the existing partial views for that component. However, in many cases defining the semantics of this integration operation is not an easy task. In particular, this is the case when the framework used at the specification level is, in some sense, an “operational” one (e.g. a Petri net or a statechart). Moreover, this problem may also apply to the definition of compositional semantics for modular constructs for this kind of frameworks.

In this paper, we study this problem, at a general level. First, we define a general notion of framework whose semantics is defined in terms of transformations over states represented as algebras, and characterize axiomatically the standard tight semantics. Then, inspired in the double-pullback approach defined for graph transformation, we axiomatically present a loose semantics for this class of transformation systems, exploring their compositional properties. In addition, we see how this approach may be applied to a number of formalisms.

1 Introduction

When defining the requirements of a system, specification units typically are partial or incomplete descriptions of a system component. For example, a viewpoint [8] describes a component of a system from a certain perspective (e.g., a external user’s point of view), or a use case [18] describes (perhaps partially) one possible behaviour of a component or system. In this context, providing a complete description of a component means integrating all the existing partial views for that component. However, in many cases defining the semantics of this integration operation is not an easy task. In particular, this is the case when the framework used at the specification level is, in some sense, an “operational” one (e.g. a Petri net or a statechart). Moreover, this problem may also apply to the definition of compositional semantics for modular constructs for this kind of frameworks.

In this paper, we study this problem at a general level. First, we define a general notion of framework whose semantics is defined in terms of transformations over states represented as algebras, and characterize axiomatically the
standard **tight semantics**. The approach is inspired in the *double-pullback approach* [12, 13] defined for graph transformation and, especially in [7]. In that paper, the double pullback approach is, in a way, extended in a conceptual way to deal with other frameworks. In this paper, the approach is different. In particular, we approach the problem in an axiomatic way characterizing tight and loose semantics for a general class of transformation systems, studying some compositionality properties for the loose case. The basic idea in the case of the loose semantics consist in considering that the transformations defined by a given specification should be interpreted as the minimal changes that should be performed over a given state. In contrast to the case of the tight semantics where the transformations define exactly the changes that should be performed over a given state. This is equivalent to eliminating the so-called frame assumption.

We believe that our approach may also be applicable for the definition of compositional semantics of modular constructs for certain classes of programming or specification languages. Actually, another source of inspiration was the definition of compositional semantics for logic programs with negation [15]. This was considered a difficult task due to the non-monotonic nature of negation as failure. In that paper, we avoided that problem by defining a loose semantics for this class of logic programs and, then, by defining adequate algebraic constructions to define a least model semantics.

In order to show the application of our approach, we use as running examples the cases of graph transformation using the double pushout approach, the single pushout approach and the double pullback approach, and the case of place/transition nets. Moreover, at the end of the paper we present the essential ideas underlying a loose semantics for logic programs with negation.

The paper is organized as follows. In section 2, we present a general formulation of tight transformation frameworks, showing how graph transformation by double and single pushouts and place/transition nets fit in our approach. In section three, we modify the previous definitions to characterize loose transformation frameworks, showing that graph transformation by double pullbacks is a loose framework, and we show how these ideas can be used to provide a loose semantics to tight transformation frameworks. Moreover, we define an operation for combining transformation rules, that may be considered complementary, and we provide an informal example of how our approach can be used to give semantics to viewpoints specifications. Finally, in section four, we show how one can restrict the loose interpretation of transformation rules by means of constraints and we apply these ideas to the definition of semantics of logic programs.

In this paper, we assume that the reader has a certain knowledge of the most standard algebraic concepts, together with the most usual notation (for
details, for example, one may consult [1]). We also assume that the reader has some very basic knowledge of algebraic graph transformation (see e.g. [17]), place-transition nets [16] and logic programming [2].

2 Transformation systems

In this section we present the basic ideas underlying our approach. First, we define a general notion of transformation framework and then we axiomatize the standard tight semantics that these formalisms usually have. As running examples to discuss our definitions we study graph transformation (double and single pushout approaches) and place-transition nets.

2.1 Basic notions

We consider that a transformation system is some kind of specification that defines a relation (a transformation relation) over a well-defined class of states. In this sense, a transformation framework is a formalism for specifying transformation systems.

Some examples of transformation formalisms that we will consider along this paper are graph transformation systems (using the double pushout approach and the single pushout approach) and place-transition nets. Other obvious candidates are all kinds of approaches based on the transformation of algebras, such as Abstract State Machines (formerly called evolving algebras) [4, 11], D-oids [3], algebraic approaches to object specification [6] or Algebra Transformation Systems, [10]. Moreover, in the last section we will deal with the special case of logic programs. In the case of graph transformation systems, states are graphs and transformation systems are sets of graph productions. In the case of the double pushout approach, productions are pairs of injective graph morphisms \( L \leftarrow H \rightarrow R \), while in the case of the single pushout approach, productions are injective partial graph morphisms \( L \rightarrow R \). In the case, of place-transition nets, states are net markings and the transformation systems are the nets themselves. The transformations denoted by a net are the possible firings.

In our case, to achieve a reasonable degree of generality, we assume that states are partial \( \Sigma \)-algebras. This covers most reasonable cases. For instance, in the case of graph transformation, graphs can be seen as total algebras over a signature including two sorts (one for vertices and one for edges) and two operations providing, respectively, the source and target of each edge. Similarly, in the case of place-transition nets, markings can be seen as algebras over a signature including the signature of the natural numbers and, in addition, having as many constants of sort natural as places in the net. However, we may consider
that not all the algebras over this signature are valid states. In particular, we may consider that the only valid states are the algebras whose carrier associated to the sort natural corresponds to the set of natural numbers and where the interpretation of the natural number operators coincide with the corresponding natural numbers operations. On the other hand, we consider that a state transformation consists of two $\Sigma$-algebras (the source and target states) together with a tracking map, which is an injective partial map relating the elements of the source and target states that remain unchanged by the transformation. However, again, we may consider that not all the possible tracking maps are valid for a given formalism. For instance, in the case of place/transition nets only the identity should be a valid tracking map. The reason is that this is the only reasonable relation between the elements (the natural numbers) of the source and target states.

Following these intuitions, we may define a transformation framework $T$ as a 5-tuple consisting of: a class of signatures $\text{Sig}_T$; a mapping, $\text{States} : \text{Sig}_T \rightarrow \mathcal{P}(\text{PAlg})$, that associates to each signature $\Sigma$ the class of partial $\Sigma$-algebras that can be considered allowable states in the given framework; another mapping, $\text{Tmaps} : \text{Sig}_T \rightarrow \text{PlnMaps}$, that associates to each signature $\Sigma$ the class of partial injective $\text{Sorts}(\Sigma)$-maps that can be used as tracking maps in transformations, a set of the transformation rules, $\text{Rules}$, allowed in the given framework; and a mapping, $\text{Trafos} : \text{Rules} \rightarrow \text{StateTrafos}$, that associates to each transformation rule the set of transformations defined by that rule. In particular, $\text{StateTrafos}$ is the set of all possible state transformations, i.e. the set of all 4-tuples consisting of a signature $\Sigma$, two $\Sigma$-states which are the source and target of the transformation, and a tracking map $t$. We do not assume that the set of rules of a given transformation framework has any specific form.

**Definition 1.** A transformation framework $T$ is a 5-tuple $(\text{Sig}_T, \text{States} : \text{Sig}_T \rightarrow \mathcal{P}(\text{PAlg}), \text{Tmaps} : \text{Sig}_T \rightarrow \text{PlnMaps}, \text{Rules}, \text{Trafos} : \text{Rules} \rightarrow \text{StateTrafos})$, where $\text{Sig}_T$ is included in $\text{Sig}$, the class of all signatures, and $\text{StateTrafos}$ is the set of all possible state transformations in $T$. In particular, a state transformation in $\text{StateTrafos}$ is a 4-tuple $(\Sigma, A_1, A_2, t)$, where $A_1$ and $A_2 \in \text{States}(\Sigma)$ are called the source and target states, respectively, and $t = \{t_1 : A_1 \rightarrow A_2^1\}_{\text{Sorts}(\Sigma)}$ is the tracking map of the transformation, which is a family of partial injective functions, written $t : A_1 \rightarrow A_2$. In addition, we assume that, for every rule $r$, $\text{Trafos}(r)$ is closed up to isomorphism and tracking maps, i.e. if $(\Sigma, A_1, A_2, t)$ is in $\text{Trafos}(r)$, $h_1 : A_1 \rightarrow A_1'$ and $h_2 : A_2 \rightarrow A_2'$ are $\Sigma$-isomorphisms and $(t' : A_1' \rightarrow A_2') \in \text{Tmaps}(\Sigma)$, such that $\text{dom}(t') = h_1(\text{dom}(t))$ and $t' \circ h_1 = h_2 \circ t$, then $(\Sigma, A_1', A_2', t')$ is also in $\text{Trafos}(r)$.

**Example 2.** The double pushout approach to graph transformation can be defined formally as follows. $\text{Sig}_T$, contains only the graphs signature:
\[\text{GRAPHS} = \text{sorts } \text{vertex, edge}\]

\[\text{opns } \text{source, target : edge } \rightarrow \text{ vertex}\]

States are total algebras over the above signature. Rules are spans of injective homomorphisms, \(L \leftarrow K \rightarrow R\). Tracking maps are pairs of arbitrary injective partial maps, \(t_{\text{vertex}} : G_{\text{vertex}} \rightarrow H_{\text{vertex}}\) and \(t_{\text{edge}} : G_{\text{edge}} \rightarrow H_{\text{edge}}\), defined as follows:

- For every \(n\) in \(D_{\text{vertex}}\), \(t_{\text{vertex}}(l^* (n)) = r^* (n)\)
- For every \(n\) in \(G_{\text{vertex}} \setminus l^* (D)\), \(t_{\text{vertex}}(n)\) is undefined
- For every \(e\) in \(D_{\text{edge}}\), \(t_{\text{edge}}(l^* (e)) = r^* (e)\)
- For every \(e\) in \(G_{\text{edge}} \setminus l^* (D)\), \(t_{\text{edge}}(e)\) is undefined

Finally, \((\Sigma, G, H, t)_r\), is a transformation associated to the previous rule if there is a graph \(D\) and morphisms \(m, d, m^*, l^*\) and \(r^*\) such that diagrams (1) and (2) in figure 1 are pushouts. Then, for every rule \(r\), \(\text{Trafos}(r)\) is closed under isomorphism and tracking maps by the definition of the associated tracking maps and because pushouts are closed under isomorphism.

\[\begin{array}{c}
L \\
m \\
G
\end{array}
\begin{array}{c}
K \\
d \\
D
\end{array}
\begin{array}{c}
R \\
m^* \\
H
\end{array}
\]

\(m\) \quad (1) \quad d \quad (2) \quad m^*

**Fig. 1. Double pushout**

**Example 3.** The single pushout approach to graph transformation can be defined formally as follows. \(\text{Sig}_T\), \(\text{States}\) and \(\text{Tmaps}\) are defined as for the double pushout approach. \(\text{Rules}\) are partial injective homomorphisms, \(p : L \rightarrow R\). Finally, \((\Sigma, G, D, t)_r\), is a transformation associated to the previous rule if there are morphisms \(m, m^*\) and \(p^*\) such that the diagram in figure 2 is a pushout in the category of graphs and partial graph morphisms, and where \(t\) is the restriction of \(p^*\) to the category of sets and partial mappings.

Again, \(\text{Trafos}(r)\) is closed under isomorphism and tracking maps because pushouts are closed under isomorphisms.
Example 4. Place/transition nets can be defined formally as follows. As said above, \( \text{Sig}_T \) consists of signatures extending the signature of natural numbers with a number of constants of sort \( \text{nat} \), a constant representing each place in the net. \( \text{States} \) are total algebras extending the natural numbers. The only tracking maps considered are the identity functions. As said above, the reason is that, in this case, the states only contain value sorts (the natural numbers) and they should remain unchanged under transformation. \( \text{Rules} \) are the sets of transitions in a net. Finally, \( (\Sigma, A_1, A_2, T) \) is a transformation associated to a net \( N \) if we can pass from the marking \( A_1 \) to the marking \( A_2 \) by the firing of some transition in \( N \).

### 2.2 Tight transformations frameworks

Two basic ideas underlie standard (tight) transformation frameworks. The first one is what we may call the locality assumption. This principle says that, if a given rule defines a transformation on a state \( A \), producing as result a state \( B \), then this rule may also be applied to a larger state \( A' \), including \( A \) in some well-defined sense, obtaining a state \( B' \), including \( B \). The second basic idea is the so-called frame assumption. This assumption implies that, when applying a local transformation on part of a state \( A \), the rest of the state should remain unchanged in the result.

In order to formalize these two assumptions, we must, first, define what we mean by saying that a state \( A' \) includes a state \( A \). On the one hand, we may consider that if \( A \) is a subalgebra of \( A' \) then \( A' \) includes \( A \). For instance, dealing with graphs as algebras, the subgraph relation would correspond to the subalgebra relation. On the other hand, we may also consider that if a signature \( \Sigma \) is included in \( \Sigma' \), and the states \( A \) and \( A' \) are, respectively a \( \Sigma \)-algebra and a \( \Sigma' \)-algebra, such that \( A'|_E = A \), then \( A' \) also includes \( A \). For instance, if \( A \) denotes a marking of a certain net, then an extension of \( A \) with respect to a larger signature would correspond to a marking of a net including more places. Mixing these
two conditions, we may say that a $Σ$-state $A$ is included in a $Σ'$-state $A'$ if $A$ is a subalgebra of $A'|_Σ$.

According to these ideas, we may say that a transformation framework is a tight transformation framework if the locality and the frame assumptions hold. This may be formalized as follows:

**Definition 5.** $T = \langle \text{Sig}_τ, \text{States}, \text{Tmaps}, \text{Rules}, \text{Trafos} \rangle$ is a tight transformation framework if the following two properties hold:

1. If $τ = (Σ, A, B, t) ∈ \text{Trafos}(r)$, then for every $Σ' ∈ \text{Sig}_τ, A', B' ∈ \text{States}(Σ')$, such that $Σ ⊆ Σ', A'|_Σ = A$ and $B'|_Σ = B$, and every tracking map $t : A' → B'$ such that:
   - For every $s$ in $\text{Sorts}(Σ), t'_s = t_s$
   - For every $s$ in $\text{Sorts}(Σ') \setminus \text{Sorts}(Σ), t'_s = \text{id}$
   - For every $σ$ in $Σ_{s_1...s_n, s}, σ_H = σ_H$
   - For every $σ$ in $Σ'_{s_1...s_n, s} \setminus Σ$ and for all $(a_1', ..., a_n')$ in $B'_s \times ... \times B'_s$

   $$σ_H(a_1', ..., a_n') = t'_s(σ_H(t^{-1}_{s_1}(a_1'), ..., t^{-1}_{s_n}(a_n')))$$

   where $=$ should be interpreted as strong equality (i.e. $e = e'$ if both expressions are defined and equal or both are undefined), and where if one subexpression in undefined (e.g., $t^{-1}_{s_1}(a_1')$) then the result of the whole expression is considered undefined.

   we have that the transformation $(Σ', A', B', t') ∈ \text{Trafos}(r)$.

2. If $τ = (Σ, A, B, t) ∈ \text{Trafos}(r)$, then for every $A', B' ∈ \text{States}(Σ)$, such that $A ⊆ A'$, and $B \subseteq B'$, and every tracking map $t : A' → B'$ such that:
   - For every $s$ in $\text{Sorts}(Σ)$,
     - $B'_s = (A'_s \setminus A_s) + B_s$, where, as usual, $+$ denotes disjoint union.
     - For every $a ∈ A_s, t'_s(a) = t_s(a)$.
     - For every $a ∈ A'_s \setminus A_s, t'_s(a) = a$.
   - For every $σ$ in $Σ_{s_1...s_n, s}$ and for all $(a_1', ..., a_n')$ in $B'_s \times ... \times B'_s$

   $$σ_H(a_1', ..., a_n') = t'_s(σ_H(t^{-1}_{s_1}(a_1'), ..., t^{-1}_{s_n}(a_n')))$$

   we have that the transformation $(Σ', A', B', t') ∈ \text{Trafos}(r)$.

Properties (1) and (2) take care of the locality and frame assumptions for both kinds of state inclusion described above, and the combination of both properties takes care of the general notion of state inclusion. We could have provided a single property combining (1) and (2) in an obvious way. However we think that the splitting into two properties is simpler to deal with.
Proposition 6. 1. The double pushout approach for graph transformation is a
tight transformation framework.
2. The single pushout approach for graph transformation is a tight transfor-
mation framework.
3. Place/transition nets are a tight transformation framework.

Proof. 1. Property (1) holds trivially for graph transformation since there is
just one signature in $\text{Sig}_T$. With respect to property (2), let us suppose that
$(\Sigma, G, H, t) \in \text{Trafos}(r)$ by means of the double pushout diagram in figure
1, and suppose that $G', H'$ and $t'$ are as in property (2) above. Then, we can
build the double pushout diagram in figure 3, where all the vertical arrows
are inclusions, and where the pushout complement $D'$ is $D' = D + (G' \setminus G)$.

$$
\begin{array}{c}
G & \xleftarrow{l^*} & D & \xrightarrow{r^*} & H \\
\downarrow & & \downarrow & & \downarrow \\
G' & \xleftarrow{l'} & D' & \xrightarrow{r'} & H'
\end{array}
$$

Fig. 3.

To show that $D'$ is indeed a pushout complement we must show that the top-
most left corner satisfies the so-called gluing conditions. In particular, we
must show that the “dangling condition” holds (the so-called identification
condition holds trivially in our case), i.e. that there cannot be an edge $e$ in
$G' \setminus G$ which is incident to a node in $G \setminus D$. Suppose that $n = \text{source}_{G'}(e)$
(or, similarly, $n = \text{target}_{G'}(e)$) is in $G \setminus D$ and $e$ is in $G' \setminus G$. Then we would
have that $t_{\text{node}}(n)$ is undefined, by the definition of the tracking map asso-
ciated to the first double pushout, and, as a consequence, so is that $t'_{\text{node}}(n)$.
However, according to the conditions stated in property (2), we would have
that $t'_{\text{edge}}(e) = e$ and $\text{source}_{H'}(e) = t'_{\text{edge}}(\text{source}_{G'}(e))$, which means that $H'$
would be ill-formed.

Now, the gluing of the two double pushout diagrams provides us with the
desired transformation and the corresponding tracking map coincides with
the definition in property (2).

2. Again, it is enough to prove that property (2) holds. The proof is similar.
Suppose that $(\Sigma, G, H, t) \in \text{Trafos}(r)$ by means of the pushout diagram in
figure 2, and suppose that $G$, $H$ and $t'$ are as in property (2) above. Then, the diagram in figure 4, where the vertical arrows are inclusions, is a pushout.

![Diagram](image)

Fig. 4.

Finally, the gluing of the two pushout diagrams provides us with the desired transformation and the corresponding tracking map coincides with the definition in property (2).

3. In this case it is property (2) that holds trivially, since the only states allowed differ only on the number and the value of the constants representing the places of the nets. This means that there do not exist two states $A$ and $B$ such that $A$ is a strict subalgebra of $B$. With respect to property (1), it should be clear that, independently of the places in a net the effect of a firing only modifies the values of the constants associated to the positions involved in the transition.

\[ \text{Fig. 4.} \]

3 \textbf{Loose transformation frameworks}

As discussed in the introduction, requirements specifications are typically incomplete specifications, i.e. they usually describe some partial aspects of the behaviour of a system. In the case of specifications using some kind of transformation framework, this means that the transformation rules associated to a given event only describe, partially, the state transformations that should happen when this event occurs. According to this intuition, we may consider that, in this context, an event may cause some additional transformations not specified by the given transformation system. This is equivalent to eliminating the frame assumption when defining the semantics of a transformation system. This can be obtained by relaxing properties (1) and (2) of tight transformation systems, just asking for the satisfaction of the locality assumption. In particular, the below properties, (3) and (4), state that if a certain transformation can be
applied to a state \( A \), yielding a state \( B \), then this transformation should be also applicable to any larger state \( A' \). In this case, any state \( B' \) including \( B \), in some well-defined sense, could be a possible result of the transformation. The idea is that the concrete state \( B' \) obtained depends on the additional changes caused by the “environment”.

In our opinion, this kind of loose transformation frameworks are adequate for certain forms of requirements specifications. This is the case, for instance, of the so-called viewpoints approach. The idea is that, when specifying a given system, one describes its functionality by means of several complementary points of view (e.g. the external user view, the system manager view, etc.) Then, the complete specification of the system is given by the integration or combination of all the views. In what follows, we informally describe an example of the specification of a system using this approach, to motivate the rest of the section.

**Example 7.** Let us consider as a simple example the specification of a logistics information system for a certain company. We assume that this company has a certain number of stores, where the items produced by the company are kept in stock, and certain number of shops that sell these items to the end customer. Moreover the company has also a number of trucks, whose location is not fixed a priori, for delivering the items. From time to time, a certain shop may ask the system for a supply of \( N \) units of a certain item. Then the system must decide from which store are these items supplied and register the delivery on the company accounts. In what follows, we briefly describe this supply operation from different viewpoints.

- **A shop manager viewpoint:** From the point of view of the manager of a shop, the effect of asking for the supply of \( N \) units of a certain item could be that the stock of one store (maybe more than one as possible choices) should be decreased by \( N \) units and the stock in the shop of that item should be increased by \( N \) units.

- **The logistics manager viewpoint:** From the point of view of the person in charge of the logistics of the company, the effect of asking for the supply of \( N \) units of a certain item to a given shop could be that the stock of that item on a certain store should be decreased by \( N \) units and the stock in the shop of that item should be increased by \( N \) units. In addition, that store is selected among the stores having enough stock for that delivery, on the basis of the transportation cost, considering the distance of the store to the shop and to the closer free truck. Moreover, the selected truck should be marked as busy.

- **The accounting manager viewpoint:** From the point of view of the person in charge of the accounting of the company, the effect of asking for the supply of \( N \) units of a certain item to a given shop could be that the global value
of the stocked items should be decreased by the value of the items delivered and the income of the company should be increased by that value.

Now we could formalize these viewpoints by means of some kind of algebra transformation approach. Each viewpoint would be described by a transformation rule that describes only part of the transformations associated to the supply operation. It may be noted that, in this example, the second and the third (or the first and the third) viewpoints are, in a way, orthogonal. Their associated transformations would deal with different parts of the “global” state of the system. However, in the case of the shop manager viewpoint and the logistics manager viewpoint, their associated transformations partially overlap. In particular, with respect to that overlapping the logistics manager viewpoint can be seen as a refinement of the shop manager viewpoint, since in the latter case the choice of the store is nondeterministic.

Then, the complete specification of the supply operation would be the combination, using the operation described at the end of the section, of the transformation rules associated to these three viewpoints.

**Definition 8.** \( \mathcal{T} = (\Sigma_T, \text{States}, \text{Rules}, \text{Trafos}) \) is a loose transformation framework if the following two properties hold:

1. If \( \tau = (\Sigma, A, B, t) \in \text{Trafos}(r) \), then for every \( \Sigma' \in \Sigma_T \) and every \( \Sigma', A', B' \in \text{States}(\Sigma') \), such that \( \Sigma \subseteq \Sigma', A'|_{\Sigma} = A \) and \( B'|_{\Sigma} = B \), and for every \( t' \), such that \( t'|_{\text{Sorts}(\Sigma')} = t \), we have that the transformation \( (\Sigma', A', B', t') \in \text{Trafos}(r) \).
2. If \( \tau = (\Sigma, A, B, t) \in \text{Trafos}(r) \), then for every \( A', B' \in \text{States}(\Sigma) \), such that \( A \subseteq A' \) and \( B \subseteq B' \), and for every \( t' \), such that \( t'|_{A} = t \), we have that the transformation \( (\Sigma', A', B', t') \in \text{Trafos}(r) \).

where \( t'|_{\text{Sorts}(\Sigma)} \) and \( t'|_{A} \) denote, respectively, the restriction of \( t' \) to the sorts of the signature \( \Sigma \) and the restriction of \( t' \) to the elements of the subalgebra \( A \).

**Definition and Proposition 9.** The double pullback approach for graph transformation is a loose transformation framework

**Proof.** The double pullback approach to graph transformation can be defined formally as follows. \( \Sigma_T, \text{States}, \text{Rules} \) and tracking maps are as for the double pushout approach. Then, \( (\Sigma, G, D, t) \), is a transformation associated to the rule \( L \leftarrow K \rightarrow R \), if there is a graph \( D \) and morphisms \( m, d, m^*, l^* \) and \( r^* \) such that \( l^* \) and \( r^* \) are injective homomorphisms and diagrams (1) and (2) in figure 5 are pullbacks, and where \( t \) is defined as follows:

- For every \( n \) in \( D_{\text{vertex}} \), \( t_{\text{vertex}}(l^*(n)) = r^*(n) \)
- For every \( n \) in \( G_{\text{vertex}} \setminus l^* (D) \), \( t_{\text{vertex}} (n) \) is undefined

- For every \( e \) in \( D_{\text{edge}} \), \( t_{\text{edge}} [l^* (e)] = r^* (e) \)

- For every \( e \) in \( G_{\text{edge}} \setminus l^* (D) \), \( t_{\text{edge}} (e) \) is undefined

\[ \begin{array}{c}
L \quad \quad m \quad (1) \quad \quad K \\
G \quad \quad j^* \quad \quad D \\
\end{array} \quad \quad \begin{array}{c}
D \quad \quad d \quad (2) \quad \quad R \\
H \quad \quad r^* \quad \quad m^* \end{array} \]

Fig. 5. Double pullback

Moreover, for every rule \( r \), \( Trafos (r) \) is closed under isomorphism and tracking maps by the definition of the associated tracking maps and because pullbacks are closed under isomorphism.

In this case, we just have to prove property (4). Suppose that \( (\Sigma, G, H, t) \in Trafos (r) \) by means of the double pullback diagram in figure 5, and suppose that \( G \subseteq G' \), \( H \subseteq H' \) and \( t' \mid _{G} = t \). Then, we can build the double pullback diagram in figure 6, where \( D' = D + (\text{dom}(t') \setminus \text{dom}(t)) \) and where all the vertical arrows are inclusions. The gluing of the two double pullback diagrams provides us with the desired transformation.

\[ \begin{array}{c}
G \quad \quad j^* \quad \quad D \\
G' \quad \quad D' \end{array} \quad \quad \begin{array}{c}
D \quad \quad r^* \quad \quad H \\
H' \end{array} \]

Fig. 6.

Additionally, the tracking map associated to this double pullback coincides, by construction, with \( t' \).

Obviously, every loose transformation framework is also a tight transformation framework, but the converse is not true in general.
**Fact 10.** If $\mathcal{T}$ is a loose transformation framework then $\mathcal{T}$ is a tight transformation framework.

This fact may seem counterintuitive, since, together with the previous proposition, it states that the double pullback approach is a tight transformation framework. An alternative definition of tight and loose transformation frameworks could consist in, first, considering that, in a transformation framework, every rule $r$ defines a set of minimal transformations, $\text{MinTrafo}(r)$. Then, we would say that a transformation framework is tight (resp. loose) if, for every rule $r$, $\text{Trafo}(r)$ is the closure of $\text{MinTrafo}(r)$ with respect to properties (1) and (2) (resp. with respect to properties (3) and (4)). According to that alternative definition, in general, loose transformation frameworks would not be tight. Instead, we could consider tight transformations as special cases of loose transformations, as double pushout transformations can be seen as special cases of double pullback transformations, as we will see below. This would probably fit better our intuition.

On the other hand, the previous fact gives us a way of defining a loose transformation semantics for tight transformation frameworks. In particular, it is enough to make a closure, with respect to the properties (3) and (4) above, of the transformations associated to the given rules:

**Definition 11.** Let $\mathcal{T} = (\text{Sig}_T, \text{States}, \text{Tmaps}, \text{Rules}, \text{Trafos})$ be a transformation framework and $r$ be a rule in $\text{Rules}$, we define the loose extension of $\text{Trafos}(r)$, denoted $\text{Loose}(\text{Trafos}(r))$ as the least set of transformations, satisfying properties (3) and (4) that includes $\text{Trafos}(r)$. We also define the loose transformation framework associated with $\mathcal{T}$, $\text{Loose}(\mathcal{T})$ as:

$$\text{Loose}(\mathcal{T}) = (\text{Sig}_T, \text{States}, \text{Tmaps}, \text{Rules}, \text{LTrafos})$$

where, for every $r$, $\text{LTrafos}(r) = \text{Loose}(\text{Trafos}(r))$

One may wonder whether the double pullback approach is the loose version of the double pushout approach. The answer is negative. There are some double pullback transformations which can not be obtained by the loose closure of the double pushout transformations. The reason is that, given a production $L \leftarrow K \rightarrow R$ and a morphism from $L$ to a graph $G$ we can not always build a double pushout to define a transformation. It is required that certain conditions (the so-called gluing conditions) hold. However, for building a double pullback the required conditions are looser. In this context, a double pullback transformation, where the gluing conditions are not satisfied, would not correspond to any loose extension of a double pushout transformation.
Definition and Proposition 12. The faithful double pullback approach for graph transformation is the loose transformation framework associated with the double pushout approach.

Proof. The double pullback transformation in figure 7 is faithful if $m$ satisfies the identification condition. This means that if $x_1$ and $x_2$ are elements (vertices or edges) of $L$ such that $m(x_1) = m(x_2)$ then, either $x_1 = x_2$, or there exist $y_1$ and $y_2$ in $K$ such that $l(y_1) = x_1$ and $l(y_2) = x_2$. Then the faithful double pullback approach is defined as the double pullback approach, but where only faithful transformations are allowed. The proof that this framework is a loose transformation framework is like the one for the general double pullback approach. The proof that every faithful double pullback transformation is an extension, with respect to property (4), of a double pushout transformation can be found in [13].

The loose versions of the single pushout approach and of place/transition nets are quite obvious: we can loosely transform graph $G$ into graph $H$ by means of the graph production $L \rightarrow R$, if there are graphs $G'$ and $H'$ such that $G' \subseteq G$, $H' \subseteq H$ and the diagram in figure 8 is a pushout.

By analogy with the case of double pushouts/pullbacks, one may wonder whether this can be characterized by means of a simple pullback, but the answer is no. The problem is that a pushout diagram in the category of graphs and partial morphisms, such as the one in figure 8 may be not a pullback in that category as the following counter-example shows. If $L$ is a graph with just two nodes, say $n_1$ and $n_2$, $R$ and $G'$ are graphs with just one node, say $n_1$, $p$ is a partial morphism binding $n_1$ in $L$ to $n_1$ in $R$ and $m$ is a morphism binding $n_1$ and $n_2$ in $L$ to $n_1$ in $G'$. Then the result of the pushout, $H'$ would be the empty graph. However, the pullback of $p^*$ and $m^*$ would be the empty graph and not $L$. 
With respect to the case of place/transition nets, the result of a loose transformation associated to a given transition for a certain marking $M$ would be any other marking such that the value of the positions associated to the transition are the ones specified by the transition, but the values of the rest of the positions may have changed in any arbitrary way.

As said above, the intuition behind loose transformation frameworks is that rules specify incompletely a certain class of transformations. In this context, we define an operation for combining the effects of two (or more rules). The idea is that, using this operation, one can complete or refine an incomplete transformation specification. The definition of this operation is quite simple: one can transform a state $A$ into a state $B$ if a “parallel” application of the combined rules provide this effect:

**Definition 13.** Let $\mathcal{T} = (\text{Sig}, \text{States}, \text{Tmaps}, \text{Rules}, \text{Trafos})$ be a loose transformation framework, the combination of two rules $r$ and $r'$, $r \parallel r'$ is a rule that defines the following set of transformations:

$\text{Trafos}(r \parallel r')$ if and only if there exist $\Sigma, A, B, C$, $t : B \rightarrow C$ and $t' : B' \rightarrow C'$, such that:

- $\Sigma, A, B, C, t, t' \in \text{Trafos}(r \parallel r')$
- $\Sigma, B, C, t \in \text{Trafos}(r)$
- $\Sigma, B', C', t' \in \text{Trafos}(r')$
- $B \subseteq A_1 \parallel A_2$ and $B' \subseteq A_1 \parallel A_2$
- $C \subseteq A_2 \parallel A_2$ and $C' \subseteq A_2 \parallel A_2$
- $t = t|_{B_0}$ and $t' = t'|_{B_0}$ where $t|_{B_0}$ and $t'|_{A}$ denotes, respectively, the restriction of $r'$ to the sorts of the signature $\Sigma_0$ and to the elements of the subalgebra $B_0$.

As one can expect, the combination of two rules has a very simple compositional semantics:

**Proposition 14.** $\text{Trafos}(r \parallel r') = \text{Trafos}(r) \cap \text{Trafos}(r')$
Proof. If \((\Sigma, A_1, A_2, t) \in \text{Trafos}(r|r')\) then, according to the previous definition, there exist \(\Sigma_0 \in \text{Sig}_T, B_0 \in \text{States}(\Sigma_0), B_0' \in \text{States}(\Sigma_0')\), and a tracking map \(t_0 : B_0 \rightarrow C_0\), such \((\Sigma_0, B_0, C_0, t_0) \in \text{Trafos}(r), B_0 \subseteq A_1|_{\Sigma_0}\) and \(t_0 = t|_{B_0}\). This means that \((\Sigma, A_1|_{\Sigma_0}, A_2|_{\Sigma_0}, t|_{\Sigma_0}) \in \text{Trafos}(r)\), since the framework is closed under (4). But this implies that \((\Sigma, A_1, A_2, t) \in \text{Trafos}(r)\) since the framework is closed under (3). The proof for \(r'\) is similar. Conversely, if \((\Sigma, A_1, A_2, t) \in \text{Trafos}(r)\) and \((\Sigma, A_1, A_2, t') \in \text{Trafos}(r')\) then \((\Sigma, A_1, A_2, t) \in \text{Trafos}(r|r')\): it is enough to take \(\Sigma_0 = \Sigma = \Sigma_0', B_0 = A_1 = B_0', C_0 = A_2 = C_0', \) and \(t_0 = t = t_0'\).

4 Loose transformation frameworks with constraints

One may consider that loose transformation frameworks are too loose for many practical applications. In particular, we may know that state transformations defined by a given rule should not produce any effect on a certain part of the state, i.e., we may want to have some restricted form of the frame assumption. This is handled in the double pullback approach by defining open transformations over typed graphs [14, 12], allowing us to express that part of the given graph should not change after a transformation. We could have defined something similar by assuming that, given a certain signature \(\Sigma_0\), when applying a transformation over a \(\Sigma\)-state \(A\), with \(\Sigma\) including \(\Sigma_0\), this transformation should not produce any change on the \(\Sigma_0\) part of \(A\). This means that, if \(B\) is the result of the transformation, then \(B|_{\Sigma_0} = A|_{\Sigma_0}\). However, this may be not enough. We may also want that the resulting state satisfies certain conditions with respect to the source state. For instance, if the given signature \(\Sigma\) includes a constant \(c\) of sort integer, we may want to restrict the transformations over \(\Sigma\)-states so that the value of \(c\) in the target state is always greater than the value of \(c\) in the source state. A simple way of doing this is to assume that transformation frameworks may be equipped with a set of constraints that may be imposed on the rules. We may assume that these constraints are some kind of logic formula. However, in order to be general enough, we will not assume that constraints have any specific form. Instead, we will define constraints in a kind of institution-independent way [9], similarly to the related notion of logic of constraints in [5].

**Definition 15.** Let \(\mathcal{T} = (\text{Sig}_T, \text{States}, \text{Tmaps}, \text{Rules}, \text{Trafos})\) be a transformation framework, a logic of constraints for \(\mathcal{T}\) is a pair \((\text{Constraints}, \models)\), where \(\text{Constraints} : \text{Sig}_T \rightarrow \text{Set}\) is a functor associating to every signature \(\Sigma\) in \(\text{Sig}_T\) a set of constraints over state transformations on that signature, and \(\models \in \{\models \subseteq \text{StateTrafos}(\Sigma) \times \text{Constraints}(\Sigma)\}_{\Sigma \in \text{Sig}_T}\) is a \(\Sigma\)-indexed family of relations between \(\text{Constraints}(\Sigma)\) and \(\Sigma\)-transformations, where \(\text{StateTrafos}(\Sigma)\) is the set
of all possible $\Sigma$-transformations $(\Sigma, A, B, t)$. In addition we require the following satisfaction condition:

For every $\Sigma_1, \Sigma_2 \in \Sig_T$, every signature morphism $h : \Sigma_1 \to \Sigma_2$, every $\Sigma_1$-constraint $c_1$, every $\Sigma_2$-states $A_2, B_2$ and every tracking map $t : A_2 \to B_2$, we have that:

$$\Sigma_2(A_2, B_2, t) \models_{\Sigma_2} c_2 \iff \Sigma_1(A_2|_h, B_2|_h, t|_h) \models_{\Sigma_1} c_1$$

where $c_2 = \text{Constraints}(h)(c_1)$, and where $t|_h$ denotes the reduction of $t$ along $h$, i.e. for every $s \in \Sigma_1$, $(t|_h)_s = t|_{h(s)}$.

As usual, the satisfaction condition ensures that constraints over a given signature are uniformly translated over signature morphisms. This implies, in particular, that if a constraint is satisfied by a transformation over a given state, then this constraint (or, rather, its translation) would also be satisfied by any extension of this transformation over a state having a larger signature.

Now, we can extend the previous definition of loose transformation frameworks to include constraints. The idea, on the one hand, is to consider that each rule considered is equipped with a set of constraints. To this aim, we will first define the class of all constraints associated to a given logic and, then, extend the constraints satisfaction relation to sets of constraints over different signatures.

**Definition 16.** Let $L = (\text{Constraints}, \models)$ be a logic of constraints over a transformation framework $T$, we define $\text{Constr}(L)$, the class of all constraints associated to $L$, as:

$$\text{Constr}(L) = \bigcup_{\Sigma \in \Sig_T} \{ (\Sigma, c) / c \in \text{Constraints}(\Sigma) \}$$

If $C$ is a set of constraints $C \subseteq \text{Constr}(L)$, we say that a transformation $(\Sigma, A, B, t)$ satisfies $C$, denoted $(\Sigma, A, B, t) \models C$ if and only if $\forall (\Sigma, c) \in C, (\Sigma, A, B, t) \models_{\Sigma} c$.

On the other hand, it is assumed that properties 3) and 4) should apply only to transformations satisfying the given constraint:

**Definition 17.** Let $T = (\Sig_T, \text{States}, \text{Tmaps}, \text{Rules}, \text{Trafos})$ be a transformation framework and $L = (\text{Constraints}, \models)$ be a logic of constraints over $T$, then $T$ is a loose transformation framework with respect to the set of constrained rules $\text{CRules} \subseteq \text{Rules} \times \mathcal{P}(\text{Constr}(L))$ if and only if for every $(r, C) \in \text{CRules}$ the following two properties hold:

5) If $\tau = (\Sigma, A, B, t) \in \text{Trafos}(r)$ and $(\Sigma, A, B, t) \models C$, then for every $\Sigma' \in \Sig_T$ and every $A', B' \in \text{States}(\Sigma')$, such that $\Sigma \subseteq \Sigma'$, $A'|_{E} = A$ and $B'|_{E} = B$, and for every $t'$, such that $t'|_{\text{Sorts}(\Sigma')} = t$, if $(\Sigma', A', B', t') \models C$ then $(\Sigma', A', B', t') \in \text{Trafos}(r)$.
(6) If \( \tau = (\Sigma,A,B,t) \in \text{Trafos}(r) \) and \( (\Sigma,A,B,t) \models C \), then for every \( A',B' \in \text{States}(\Sigma) \), such that \( A \subseteq A' \) and \( B \subseteq B' \), and for every \( t' \), such that \( t'|_A = t \), if \( (\Sigma',A',B',t') \models C \) then \( (\Sigma',A',B',t') \in \text{Trafos}(r) \).

Now, we can extend the composition operation defined in the previous section to the composition of constrained rules:

**Definition 18.** Let \( \mathcal{T} = (\mathcal{Sig},\mathcal{States},\mathcal{Tmaps},\mathcal{Rules},\text{Trafos}) \) be a loose transformation framework, the combination of two constrained rules \((r,C)\) and \((r',C')\), \( (r,C)\mid (r',C') \) is the constrained rule \( (r|r',C \cup C') \).

**Proposition 19.** If \( \mathcal{T} \) is a loose transformation with respect to the set of constrained rules \( \mathcal{CRules} \), then for all rules \((r,C)\) and \((r',C')\) in \( \mathcal{CRules} \):
\[
\text{Trafos}((r,C)\mid (r',C')) = \text{Trafos}((r,C)) \cap \text{Trafos}((r',C'))
\]

**Proof.** If \((\Sigma,A1,A2,t) \in \text{Trafos}(r|r')\) and \((\Sigma,A1,A2,t) \models C \cup C'\) then we have that \((\Sigma,A1,A2,t) \in \text{Trafos}(r)\) and, obviously, \((\Sigma,A1,A2,t) \models C\). The converse is trivial. \(\blacksquare\)

Now, using the approach of loose transformation frameworks with constraints, we will see how we can define a compositional semantics for logic programs with negation. For simplicity, we will just consider the propositional case. The approach presented in the example is a (very) simplified reformulation of part of the work presented in [15], where the general first-order case is considered.

**Example 20.** A propositional logic program with negation is a set of ground clauses having the form:
\[
a : \neg \ell_1, \ldots, \neg \ell_n
\]
where \(a\) is an atom and each \(\ell_i\) is a positive or a negative literal, i.e. \(a_i\) or \(-a_i\), where \(a_i\) is an atom. There are several ways to define the semantics of a logic program \(P\). Here we will just consider the semantics in terms of a (continuous) immediate consequence operator \(T_P\). This semantics is used to define a least model semantics by means of the least fixpoint of \(T_P\). This operator is defined as a mapping transforming logical structures into logical structures, where a logical structure, in this context, is a 3-valued Herbrand structure that is, a pair \(\mathcal{A} = (A^+,A^-)\) of sets of atoms (representing the positive and the negative information in the structure) satisfying \(A^+_i \cap A^-_i = \) (consistency condition). The idea of the immediate consequence operator is that \(T_P(\mathcal{A})\) should contain all the atoms that are one-step consequences of the program \(P\) and the information in \(\mathcal{A}\). This means that \(T_P(\mathcal{A}) = (B^+,B^-)\), where:
\[ B^+ = A^+ \cup \{ a \exists a : -\ell_1, \ldots, \ell_n \in P \text{ such that } \ell_1, \ldots, \ell_n \in A \} \] (a)

\[ B^- = A^- \cup \{ a \forall a : -\ell_1, \ldots, \ell_n \in P, \exists i \text{ such that } -\ell_i \in A \} \] (b)

where \( \ell \in A \) means that \( \ell \in A^+ \), in case \( \ell \) is positive, or \( \ell \in A^- \), in case \( \ell \) is negative. Definition (a) is the standard one when dealing with positive programs (without negation). On the other hand, Definition (b) is a very weak version of negation-as-failure.

Now, we can reformulate the above definitions as a tight transformation framework. States are sets of atoms (which can easily be formulated in several ways as algebras over an appropriate signature \( \Sigma \)), rules are programs, states are 3-valued Herbrand structures, and the transformation defined by a program \( P \) on a state \( A \) is \( T_P(A) \). Finally, since the transformations associated to a program always add atoms to the given structure, we may consider that the tracking maps are always the identity.

In order to define a compositional semantics, we will now present a loose semantics for this class of programs. However, this definition is not a direct application of definition 11. The key idea is to consider that, if we think that a program \( P \) is incomplete, then the transformation associated to \( P \), when applied to a state \( A = (A^+, A^-) \), should at least add to \( A^+ \) all the immediate positive consequences (i.e. \( \{ a \exists a : -\ell_1, \ldots, \ell_n \in P \text{ such that } \ell_1, \ldots, \ell_n \in A \} \)) and should at most add to \( A^- \) all the immediate negative consequences (i.e. \( \{ a \forall a : -\ell_1, \ldots, \ell_n \in P, \exists i \text{ such that } -\ell_i \in A \} \)). The reason is that adding more clauses to \( P \) would provide additional positive consequences but fewer negative consequences.

Now, we may obtain the desired semantics as follows. First, we reformulate the tight semantics considering programs as if they were positive. This means that \( (\Sigma, A, B, t) \) is a transformation associated to \( P \) if \( B^+ = A^+ \cup \{ a \exists a : -\ell_1, \ldots, \ell_n \in P \text{ such that } \ell_1, \ldots, \ell_n \in A \}, B^- = A^- \) and \( t \) is the identity.

Then, we consider the loose extension of this framework. In this case, we can see that this loose extension \( LTrafo(P) \) would include all transformations \( (\Sigma, A, B, t) \) such that there are structures \( A' \subseteq A \) and \( B' \subseteq B \), such that \( B^+ = A^+ \cup \{ a \exists a : -\ell_1, \ldots, \ell_n \in P \text{ such that } \ell_1, \ldots, \ell_n \in A \} \) and \( B^- = A^- \). However, this loose extension is not yet what we want. We have to make sure that the negative atoms that can be added in \( B \) with respect to \( B' \) are consistent with the negation as failure rule. This can be done by imposing a constraint on rules. In particular, this constraint \( C \) would be:

\[ \forall a \in B' \setminus A^- \forall a : -\ell_1, \ldots, \ell_n \in P, \exists i, \text{ such that } -\ell_i \in A \]

It is not difficult to prove that the following compositionality property holds:

\[ LTrafos((P_1 \cup P_2, C)) = LTrafos((P_1, C)) \cap LTrafos((P_1, C)) \]
5 Conclusions

The work presented in this paper has been motivated by the need of providing a loose semantics to transformation systems, in order to deal with requirements specifications and to provide compositional semantics to modular units. In this sense, we have introduced a general axiomatic approach to deal with transformation systems, showing how this approach can be applied to a number of cases. In particular, the double and single pushout approaches and the double-pullback approach to graph transformation, place/transition nets and logic programs with negation have been considered.

First, we have defined a notion of tight transformation framework, characterizing the locality and frame assumptions that underlie most operational approaches. Then, we have seen how, by eliminating the frame assumption, one can define loose transformation frameworks that can be used to give loose semantics to tight transformation approaches. Finally, we have seen how one can impose some constraints to restrict, as needed, the semantics of loose frameworks.

Acknowledgements This work has been partially supported by the CICYT project HEMOSS (ref. TIC98-0949-C02-01) and CIRIT GRC 1999SGR-150.

References

A Comparison between three Non-determinism Analyses in a Parallel-Functional Language

Ricardo Peña and Clara Segura
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
e-mail: {ricardo,segura}@sip.ucm.es

Abstract. The paper compares three analyses to determine when an Eden expression is sure to be deterministic, and when it may be non-deterministic. This work extends previous works where the three analyses were presented and compared with respect to expressiveness and efficiency. The first analysis has linear cost, the second one exponential cost, and the third one polynomial (cubic) cost. In this paper: (1) the second and third analyses are completed with polymorphism, (2) we prove that the domains in the second and third analyses form a category in which the morphisms are embedding-closure pairs of functions; and (3) we formally relate the analyses and prove that the first analysis is a safe approximation to the third one (in a previous paper we already proved that the third one is a safe approximation to the second one). So the three analyses become totally ordered by increasing cost and precision.

1 Introduction

The parallel-functional language Eden [2] extends the lazy functional language Haskell by constructs to explicitly define and communicate processes. It is implemented by modifying the Glasgow Haskell Compiler (GHC) [13]. The three main new concepts are process abstractions, process instantiations and the non-deterministic process abstraction merge. Process abstractions of type Process a b can be compared to functions of type a -> b, and process instantiations can be compared to function applications. An instantiation is achieved by using the predefined infix operator (#) :: Process a b -> a -> b. Each time an expression e1 # e2 is evaluated, a new parallel process is created to evaluate (e1 e2). Non-determinism is introduced in Eden by means of a predefined process abstraction merge :: Process [[a]] [a] which fairly interleaves a set of input lists, to produce a single non-deterministic list. The presence of non-determinism creates some problems [11] in Eden: It affects the referential transparency of programs [8, 16] and invalidates some optimisations done in the GHC [15].

The paper compares three analyses to determine when an Eden expression is sure to be deterministic, and when it may be non-deterministic. This work is a continuation of [11] and [10]. In [11], the first and second analyses were presented and compared with respect to expressiveness and efficiency. In that paper some open questions were left which are extensively treated here: (1) to complete the second analysis with polymorphism; (2) to prove that the domains in the second analysis form a category in which the morphisms are embedding-closure pairs of functions, respectively called abstraction and concretisation functions; and (3) to
prove that the first analysis is a safe approximation to the second one, i.e. whenever it characterizes an expression as being deterministic, so will do the second one. In the second paper [10], a third, intermediate analysis was developed as a widening [4, 5, 7, 14] of the second one in order to have a less costly representation of functions and to speed up the fixpoint computation. There, it was proved that this new analysis is a safe approximation to the second one. It also described an algorithm to annotate the program expressions with non-determinism information, so that it can be used to avoid the harmful transformations.

The plan of the paper is as follows: In Section 2 the language and the three analyses are briefly summarized. All of them are based on abstract interpretation. In Section 3 the properties of the abstraction-concretisation function pairs are proved. In Section 4 polymorphism in the second and third analysis is treated in detail. Section 5 shows that the first one is a safe approximation to the third one. Finally, some conclusions and open questions are drawn. All the proofs can be found in [10], available at authors’ web page.

2 Three Analyses for Non-determinism in Eden

2.1 The Language

The language being analysed is an extension of Core-Haskell [13], i.e. a simple functional language with second-order polymorphism, so it includes type abstraction and type application. A program is a list of possibly recursive bindings from variables to expressions. Such expressions include variables \( v \), lambda abstractions, applications of a functional expression to an atom \( x \) (a variable or a literal \( k \)), constructor applications, primitive operators applications, and case and let expressions. Constructor and primitive operators applications are saturated. The variables contain type information, so we will not write it explicitly in the expressions. When necessary we will write \( e : : t \) to make it explicit. A type may be a basic type \( K \), a tuple type \( (t_1, \ldots, t_m) \), an algebraic type \( T_t_1 \ldots, t_m \), a functional type \( t_1 \rightarrow t_2 \) or a polymorphic type \( \forall \beta.t \).

The new Eden expressions are a process abstraction \( \text{process } v \rightarrow e \), and a process instantiation \( v \neq x \). There is also a new type \( \text{Process } t_1, t_2 \) representing the type of a process abstraction \( \text{process } v \rightarrow e \) where \( v \) has type \( t_1 \) and \( e \) has type \( t_2 \). Frequently \( t_1 \) and \( t_2 \) are tuple types and each tuple element represents an input/output channel of the process.

2.2 The First Analysis

Figure 2 shows the abstract domains for the first analysis. There is a domain \( \text{Basic} \) with two values: \( d \) represents \textit{determinism} and \( n \) possible \textit{non-determinism}, with the ordering \( d \sqsubseteq n \). This is the abstract domain corresponding to basic types and algebraic types (except tuples). Tuples are specially treated as tuples of basic abstract values\(^1\) (see [11] for justification). The ordering between basic values is naturally extended to tuples. Several least upper bound (lub) operators as well

\(^1\) We use \( b \) to denote a basic abstract value and \( a \) to denote either a basic value or a tuple of basic values.
A Comparison between three Non-determinism Analyses in ...

\begin{tabular}{ll}
Basic = \{d, n\} where d \subseteq n & \\
D_{i,K} = D_i & \\
D_{i,(t_1, \ldots, t_m)} = D_i \cup \{b_i \in Basic\} & \\
D_{i,1 \rightarrow t_2} = D_i \cup \{b_{i} \mid t_1 \rightarrow t_2\} \quad t_2 = D_{i,t_2} & \\
D_{\text{Var}, t} = D_{i,t} & \\
\end{tabular}

Fig. 1. Abstract domains for the first analysis

\begin{tabular}{ll}
\neg_i : Basic \rightarrow D_{i,t} & \\
b_{i_1, \ldots, t_m} = (b_1, \ldots, b_m) & n \sqcup b = n \\
b_{t_1, t_2} = (b_{t_1, t_2}) & d \sqcup b = b \\
b_{\text{Var}, t} = b_t & \sqcup \{b_1, \ldots, b_m\} = \bigcup_i b_i \\
b_t = b \text{ if } t = K, \alpha, T, t_1, \ldots, t_m & \\
\end{tabular}

Fig. 2. Adaptation function definition, lub and flattening operators

as an operator to flatten the internal tuples can be defined, shown in Figure 2. The domains corresponding to functions and processes are identified with their range domains (this is the most important difference with the second analysis). The abstract domain of a polymorphic type is that of its smallest instance \cite{1}, i.e. that one in which \(K\) is substituted for the type variable. So the domain corresponding to a type variable is Basic.

In Figure 3 the abstract interpretation is shown. The interpretation of a tuple is a tuple of basic abstract values, so the abstract value of each component is previously flattened (by using \(\sqcup\)). In the interpretation of a constructed value, we take a step further: Once the components are flattened, the lub is applied so that a final basic value is obtained. This means that the information about the components is lost. As merge process is the only source of non-determinism, we will say that an expression may be non-deterministic when it 'contains' any instantiation of this process. So we will consider that a function/process is deterministic if it does not generate non-deterministic results from deterministic arguments. Then, the interpretation of a function/process is the interpretation of its body when the argument is given a deterministic abstract value.

Such value is in fact an adaptation of the basic abstract value \(d\) to the type of the argument, see Figure 2. Given a type \(t\), \(\neg_t\) takes a basic abstract value \(b\) and produces an abstract value in \(D_{i,t}\). Its behaviour is the opposite to the flattening operator: If \(t\) is a \(m\)-tuple, it replicates \(b\) to obtain the \(m\)-tuple \((b_1, \ldots, b_m)\). Note that \(\sqcup b = b\) and \((\sqcup a)_t \subseteq a\).

The result of an application is non-deterministic either when the function is non-deterministic or when the argument is non-deterministic.

In the recursive let expression, the fixpoint can be obtained by using the Kleene's ascending chain. The number of iterations is linear with the number of tuple components in the bindings.

We have three different kinds of case expressions (for tuple, algebraic and primitive types). The more complex one is the algebraic case. It is non-deterministic if either the discriminant expression or any of the expressions in the alternatives is non-deterministic. Note that the abstract value of the discriminant \(e\),
\[ [\epsilon], \rho_0 = \rho_0, v \\
[k], \rho_0 = d \\
[(x_1, \ldots, x_m)], \rho_0 = (\bigcup_i \{[x_i], \rho_0\}), \ldots, \bigcup \{[x_m], \rho_0\}) \\
[C, x_1, \ldots, x_m], \rho_i = \bigcup_i \bigcup([x_i], \rho_0) \\
[e, x], \rho_i = (\bigcup_i \{[x], \rho_0\}) \cup \{[e], \rho_0\} \\
\text{where } op \colon t_1 \rightarrow (t_2 \rightarrow \ldots (t_m \rightarrow t)) \\
[p\#], \rho_1 = \bigcup \{[p], \rho_0\} \cup \{[\lambda v.e], \rho_0 = [e], \rho_0 [v \mapsto d_i] \text{ where } v \mapsto t \\
\text{process } v \rightarrow e, \rho_1 = [e], \rho_0 [v \mapsto d_i] \text{ where } v \mapsto t \\
\text{merge}_1, \rho_1 = n \\
\text{let } v = e \text{ in } e', \rho_1 = [e'], \rho_0 [v \mapsto [e]], \rho_0 \\
\text{let rec } \{v_i = e_i\} \text{ in } e', \rho_1 = [e'], \rho_0 [v \mapsto \pi_i([e]), \rho_0] \\
\text{case } e \text{ of } \{v_i = e_i\} \rightarrow e', \rho_1 = [e'], \rho_0 [v_i \mapsto \pi_i([e]), \rho_0] \text{ where } v_i : t_i \\
\text{case } e \text{ of } k \rightarrow e_i, \rho_1 = [e], \rho_0 [v_i \mapsto b_{ij}], v_{ij} : t_{ij} \\
\text{where } b = [e]; \rho_1 \text{ and } \rho_0 [v_i \mapsto b_{ij}], v_{ij} : t_{ij} \\
\text{[\lambda e], \rho_1 = [e], \rho_0; \rho_0] \text{ where } (c t) : t \text{ in } st \\
\]

Fig. 3. Abstract interpretation of the first analysis

let us call it \( b \) belongs to \textit{Basic}. That is, when it was interpreted the information about the components was lost. We want now to interpret each alternative’s right hand side in an extended environment with abstract values for the variables \( v_{ij} : t_{ij} \) in the left hand side of the alternative. We do not have such information, but we can safely approximate it by using the adaptation function, see Figure 2.

When a type application \( e t \) is interpreted, it is necessary to adapt the abstract value of the \textit{type} abstraction to the instantiated \textit{type}, as new structure may arise from the instantiation. If no new structure arises (i.e. the domain corresponding to the smallest instance is already a cartesian product), then we do not need to make any adaptation. So, by overloading the adaptation function notation, we could write \((b_1, \ldots, b_m | t_1, \ldots, t_m) = (b_1, \ldots, b_m)\).

2.3 The Second Analysis

To overcome the limitations of the first analysis (see [11] for a thorough discussion and an example) we define a new analysis where interpreting functions and processes as abstract functions allows us to express several levels of determinism and non-determinism, for instance \( \lambda z.d \subseteq \lambda z.n \subseteq \lambda z.n \). It is also an abstract interpretation based analysis, in the style of Burn, Hankin and Abramsky [3].

In figures 4 and 5 the abstract domains and the abstract interpretation for the second analysis are shown. The interpretation of a constructor belongs to \textit{Basic} but, as each component \( x_i : t_i \) has an abstract value belonging to \( D_{at_i} \), we cannot directly apply the hub. First the information of each component must
be flattened. The function responsible for this is called the abstraction function \( \alpha_t : D_{2t} \to \text{Basic} \), defined in the following section.

Application and process instantiation are interpreted as functions applications. In a recursive let expression the fixpoint can be calculated using again Kleene’s ascending chain. As there are functional domains, the number of iterations is, in the worst case, exponential in the number of bindings.

In the algebraic case expression we again find a safe representative of the discriminant’s abstract value in the domain of the variables \( v_{ij} \), now \( D_{2t_{ij}} \). This is obtained with a function \( \gamma : \text{Basic} \to D_{2t} \), called the concretisation function. In particular, \( n \) is mapped to the top of the domain \( D_{2t} \), and \( d \) to a value in \( D_{2t} \) that reflects our original idea of determinism: We will see below that \( \gamma(d) \) is the biggest value in \( D_{2t} \) that has the property of preserving the determinism. Polymorphism will be treated in Section 4.4

### 2.4 The Third Analysis

The high cost of \( \llbracket \| \) is due to the fixpoint calculation. At each iteration, a comparison between abstract values is done. Such comparison is exponential in case
functional domains are involved. So, a good way of speeding up the calculation of the fixpoint is finding a quickly comparable representation of functions. Some different techniques have been developed in this direction, such as frontiers algorithms [12] and widening/narrowing operators [4, 7, 14]. In this analysis, functions are represented by signatures in a way similar to [14] and the fixpoint calculation is speeded up by using a widening operator, see Figure 6.

Three (not shown) useful functions, nArg{}, rType and aTypes can be defined. Given a type t, the first one returns the number of arguments of t; the second one returns the (non-functional) type of its result (it is the identity in the rest of cases); and the third one returns the list (of length nArg(t)) of types of the arguments.

A signature for a function is obtained by probing such function with some explicitly chosen combinations of arguments. We probe a function of m arguments with m + 1 combinations of arguments: $\gamma_1(n), \gamma_2(d), \ldots, \gamma_m(d)$; $\gamma_1(d), \gamma_2(n), \ldots, \gamma_m(d)$; $\gamma_1(d), \gamma_2(d), \ldots, \gamma_m(n)$ and $\gamma_1(d), \gamma_2(d), \ldots, \gamma_m(d)$. The last combination is very important for us, as we want the analysis to be more powerful than \cite{r}, where the functions were probed with only this combination.

As we probe only with some arguments, different functions may have the same signature and consequently some information is lost, see \cite{r} for details. When we want to recover the original value, we can only return an upper approximation to it. This leads us to the definition of the operator $W_t : D_{2^k} \rightarrow D_{2^k}$, in Figure 6. This definition is obtained by applying first the probing process to obtain a signature and then trying to recover the original value. In \cite{r} we prove that $W_t \cong id_{D_{2^k}}$ for each type $t$.

The abstract domains for this third analysis are the same as the domains for the second analysis, see Figure 4. The representation of functions by signatures is only done in the fixpoint calculation and the details can be found in the implementation algorithm shown in \cite{r}. We will use a 3 underscript to identify it. The only expression where there are differences with respect to the second analysis is the recursive let expression:

$$\begin{align*}
\text{let rec } & \{ q_i = e_i \} \text{ in } \{ e' \} ; \\
& \text{let } \rho_3 = \{ e' \} ; \\
& \text{fix } \{ \lambda \rho_1, \rho_3 \{ q_i \mapsto W_t (\{ e_i \} ; \rho_i) \} \}
\end{align*}$$

where $e_i :: t_i$. In \cite{r} it is shown that the cost of fixpointing a recursive function definition $f = \lambda \eta_1 \ldots \lambda \eta_m. e$ with this analysis, including the complete annotation of the body $e$, is in $O(m^2 \text{ size}(e))$, being $m$ the number of arguments and size(e) (roughly speaking) the size of the definition body.

Notice that, by modifying the widening operator, we can have several different variants of the analysis. We can express them parameterised by the (collection of) widening operator $wop_{\alpha}$, $\{ \}^{wop}_{\alpha}$.

3 Abstraction and Concretisation Functions

The contributions of this paper begin in this section\footnote{All the results obtained in this section remain true after including polymorphism.}. In the second and third analyses, and for each type $t$, we need two functions called the abstraction function, $\alpha_t$, and the concretisation function, $\gamma_t$, both defined in Figure 7. They are
\[ \begin{align*}
 W_k & : D_{\beta t} \to D_{\beta t} \\
 W_K & = W_{\beta t} \cdot \cdots \cdot \gamma_{\alpha t} = \tilde{id}_{\beta t} \\
 W_{\beta t} & = (W_{\beta t}(e_1), \ldots, W_{\beta t}(e_n)) \\
 W_{\beta t}(f) & = \lambda z_i \in D_{\gamma_{\beta t}} \ldots \lambda z_m \in D_{\gamma_{\beta t}} . \\
 & \begin{cases} \\
 W_{\beta t}(f \gamma_1(d) \cdots \gamma_m(d)) & \text{if } \bigwedge_{i=1}^{m} z_i \in \gamma_i(d) \\
 W_{\beta t}(f \gamma_1(d) \cdots \gamma_i(n) \cdots \gamma_m(d)) & \text{if } \bigwedge_{j=1, j \neq i}^{m} z_j \in \gamma_j(d) \land z_i \notin \gamma_i(d) \quad i \in \{1, m\} \\
 \gamma_{\beta t}(n) & \text{otherwise } (m > 1) 
 \end{cases} \\
 \end{align*} \]

where \( t = t'_1 \to t'_2 \) or \( \text{Process } t' t'_1 \), \( m = n \text{Args}(t) \), \( t_r = r \text{Type}(t) \), \( t_1, \ldots, t_m = r \text{Type}(t) \)

**Fig. 6.** The widening operator

<table>
<thead>
<tr>
<th>( \alpha_t : D_{\beta t} \to \text{Basic} )</th>
<th>( \gamma_t : \text{Basic} \to D_{\beta t} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha_K = \alpha_{\beta t} \cdot \cdots \cdot \alpha_{\beta t} = id_{\text{Basic}} )</td>
<td>( \gamma_K = \gamma_{\beta t} \cdot \cdots \cdot \gamma_{\beta t} = id_{\text{Basic}} )</td>
</tr>
<tr>
<td>( \alpha_{\beta t} = \alpha_{(t_1, \ldots, t_m)}(e_1, \ldots, e_m) = \prod_{t \in {1, \ldots, m}} \alpha_t(e_i) )</td>
<td>( \gamma_{\beta t} = \gamma_{(t_1, \ldots, t_m)}(b) = (\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) )</td>
</tr>
<tr>
<td>( \alpha_{\text{Process } t_1} = \alpha_{t_2}(f) )</td>
<td>( \gamma_{\text{Process } t_1} = \gamma_{t_2}(b) )</td>
</tr>
</tbody>
</table>
| \( \alpha_{t_1} \to \alpha_{t_2} = \alpha_t(f(\gamma_{t_1}(d))) \) | \( \gamma_{t_1} \to \gamma_{t_2} = \{
\begin{cases}
\lambda z_i \in D_{\gamma_{t_1}(n)} & \text{if } b = n \\
\lambda z_i \in D_{\gamma_{t_1}(n)} & \text{if } b = d \\
\gamma_{t_1}(z) & \text{otherwise } (m > 1) 
\end{cases}
\) |
| \( \alpha_{t_1} \cdot \alpha_{t_2} = \alpha_t \) | \( \gamma_{t_1} \cdot \gamma_{t_2} = \gamma_t \) |

**Fig. 7.** Abstraction and concretisation function definitions

mutually recursive and are respectively used in constructor applications and in case expressions. In [11] they were explained in detail and an example was given to illustrate their definitions. Here we only recall the main ideas. Given a type \( t \), the abstraction function takes an abstract value in \( D_{\beta t} \) and flattens it to a value in Basic. The idea is to flatten the tuples (by applying the Hub operator) and apply the functions to deterministic arguments.

Given a type \( t \), the concretisation function \( \gamma_t \) unflattens a basic abstract value and produces an abstract value in \( D_{\beta t} \). The idea of this function is to obtain the best safe approximation both to \( d \) and \( n \) in a given domain. The function type needs explanation, the rest of them being immediate. We have said that a function is deterministic if it produces deterministic results from deterministic arguments; but if the argument is non-deterministic, the safest we can produce is a non-deterministic result. So, the unflattening of \( d \) for a function type is a function that takes an argument, flattens it to see whether it is deterministic or not and again applies the concretisation function corresponding to the type of the result. The unflattening of \( n \) for a function type is the function that returns a non-deterministic result independently of the argument.

The first contributions of this paper are to prove that \( \alpha_t \) and \( \gamma_t \) are a Galois insertion [9], or equivalently a Galois surjection [6], or an embedding-closure pair [1] (see Proposition 1), and also that \( \gamma_{t_1} \to \gamma_{t_2}(d) \) faithfully reflects the notion of deterministic function (see Proposition 3).

In the following proposition we prove that \( \alpha_t \) and \( \gamma_t \) are monotone and continuous, and that they are a Galois insertion. This fact guarantees that in the

---

3 Where \( \gamma_t \) is the embedding and \( \alpha_t \) is the closure.
interpretation of case expressions, the information recovered from the discriminant is not only safe but also the best we can obtain considering how its abstraction was done (giving a non-deterministic abstract value to each variable in the alternative is also safe, but less accurate). Item 1(d) says that the unflattening of \( n \) is the top of the corresponding domain.

**Proposition 1** For each type \( t \):

(a) The functions \( \alpha_t \) and \( \gamma_t \) are monotone and continuous.
(b) \( \alpha_t \cdot \gamma_t = id_{Basic} \)
(c) \( \gamma_t \cdot \alpha_t \sqsubseteq id_{D_2} \)
(d) \( \forall e \in D_2, e \sqsubseteq \gamma_t(n) \)

As a direct consequence of this proposition, we have the following (quite standard) one, asserting that all the values below \( \gamma_t(d) \) are flattened to \( d \). That is, \( \gamma_t(d) \) represents \( d \). But it also says that any other value, bigger that \( \gamma_t(d) \) or incomparable with it, is flattened to \( n \). This confirms that \( \gamma_t(d) \) is the biggest value representing \( d \).

**Proposition 2** For each type \( t \): \( \forall e \in D_2, e \sqsubseteq \gamma_t(d) \Leftrightarrow \alpha_t(e) = d \)

A function/process is considered as deterministic if when applied to a deterministic argument its result is also deterministic. This means that, when a function (resp. process) of type \( t_1 \rightarrow t_2 \) (resp. Process \( t_1 t_2 \)) is applied to a value less than or equal to \( \gamma_{t_1}(d) \), the result is less than or equal to \( \gamma_{t_2}(d) \). The following proposition says that a function of type \( t_1 \rightarrow t_2 \) is deterministic if and only if it is less than or equal to \( \gamma_{t_1 \rightarrow t_2}(d) \), i.e. \( \gamma_{t_1 \rightarrow t_2}(d) \) is the biggest representative of the deterministic functions of that type. The same happens for processes.

**Proposition 3** Given a function \( f \in D_2 \), where \( t = t_1 \rightarrow t_2 \) or \( t \) = Process \( t_1 t_2 \):
\( f \in \gamma_t(d) \Leftrightarrow \forall e \in \gamma_{t_1}(d) \cdot f(e) \in \gamma_{t_2}(d) \)

4 Polymorphism

Polymorphism is now incorporated to both second and third analyses, see Figure 8. The abstract interpretation of a polymorphic expression is the abstract interpretation of its ’smallest instance’, i.e. that instance where \( K \) (the basic type) is substituted for the type variables. This is the reason why the abstract domain corresponding to a type variable \( \beta \) is \( Basic \), and the abstract domain corresponding to a polymorphic type is that of the type without qualifier. When an application to a type \( t \) is done, the abstract value of the appropriate instance
Polymorphic abstraction and concretisation functions definitions

\[
\begin{array}{|c|c|c|c|}
\hline
p \in F_p & \alpha_{\text{inst}t_1}(p) & f(\alpha_{\text{inst}t_1}(p)) & \gamma_{t_2 \text{ inst}}(f(\alpha_{\text{inst}t_1}(p))) \\
\hline
(\lambda z \in \text{Basic}, \lambda z \in \text{Basic}d) & (d, d) & d & \lambda z \in \text{Basic}u \\
(\lambda z \in \text{Basic}, \lambda z \in \text{Basic}d) & (d, n) & d & \lambda z \in \text{Basic}u \\
(\lambda z \in \text{Basic}, \lambda z \in \text{Basic}c) & (d, n) & d & \lambda z \in \text{Basic}u \\
(\lambda z \in \text{Basic}, \lambda z \in \text{Basic}c) & (d, n) & n & \lambda z \in \text{Basic}n \\
(\lambda z \in \text{Basic}, \lambda z \in \text{Basic}c) & (d, n) & n & \lambda z \in \text{Basic}u \\
\hline
\end{array}
\]

Fig. 9. Polymorphic abstraction and concretisation functions definitions

An example of polymorphism

must be obtained. Such abstract value is in fact obtained as an approximation constructed from the abstract value of the smallest instance. From now on, the instantiated type \( t[\beta := t] \) will be denoted as \( \text{tinst} \). The approximation to the instance abstract value is obtained by using a polymorphic concretisation function \( \gamma_{t \text{ inst}} : D_{2t} \rightarrow D_{2\text{tinst}} \), which is defined and studied in detail below. Another function, \( \alpha_{t \text{ inst}} : D_{2\text{tinst}} \rightarrow D_{2t} \), which we will call the polymorphic abstraction function, will also be defined. They are mutually recursive and will be shown to be a Galois insertion. We study also some other properties.

Given two types \( t, t' \) and a type variable \( \beta \) functions \( \gamma_{t \text{ inst}} \) and \( \alpha_{t \text{ inst}t'} \) are formally defined in Figure 9 (where \( \text{tinst}_t \) denotes \( t[\beta := t] \)). These functions are a generalisation of \( \alpha_t \) and \( \gamma_t \). In case \( t' = \beta \) they coincide with \( \alpha_t \) and \( \gamma_t \).

In [1] the category of domains and embedding-closure pairs is presented. Two functions, \( \times \) (and \( \times^m \)) and \( \rightarrow \), can be defined in this category (see Figure 9 bottom). They build a new embedding-closure pair from two (or more in the case of the cartesian product) embedding-closure pairs. We define \( \gamma_{t \text{ inst}} \) and \( \alpha_{t \text{ inst}t'} \) by means of these functions.

As an example, let the polymorphic type \( \forall \beta . t' \), where \( t' = (\beta, \beta) \rightarrow \beta \), be applied to the type \( t = \text{Int} \rightarrow \text{Int} \). Then \( \text{tinst} = t'[t/\beta] = (\text{Int} \rightarrow \text{Int}, \text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \). To abbreviate, we will call \( E_p \) to \( \text{Basic} \times \text{Basic}d \) and \( F_p \) to \( \text{Basic} \rightarrow \text{Basic} \times [\text{Basic} \rightarrow \text{Basic}] \). Let \( f \in D_{2t} \) be \( f = \lambda p \in E_p.\pi_1(p) \). By definition we have that \( \gamma_{t \text{ inst}}(f) = \lambda p \in F_p.\gamma_{t \text{ inst}t'_1}(f(\alpha_{t \text{ inst}t_1}(p))) \). Figure 10 shows a table for \( \gamma_{t \text{ inst}}(f) \) and the intermediate steps that have been followed to compute it. The following proposition is trivially true by definition.
Proposition 4 Given two types \( t, t' \) and a type variable \( \beta \):

(a) \( \gamma_{t' \text{inst}} \) and \( \alpha_{t' \text{inst}} \) are monotone and continuous.
(b) \( \alpha_{t' \text{inst}} \cdot \gamma_{t' \text{inst}} = \text{id}_{D_{t'}} \)
(c) \( \gamma_{t' \text{inst}} \cdot \alpha_{t' \text{inst}} \supseteq \text{id}_{D_{t'}} \)

In the following proposition we study the relationship between the abstraction/concretisation functions and its polymorphic counterparts. They have some commutativity properties, shown in Figure 11.

Proposition 5 Given two types \( t, t' \) and a type variable \( \beta \):

(a) \( \gamma_{t' \text{inst}} \cdot \gamma_{t} = \gamma_{t \text{inst}} \)  
(c) \( \alpha_{t' \text{inst}} \cdot \gamma_{t' \text{inst}} = \alpha_{t} \)  
(e) \( \gamma_{t' \text{inst}} \cdot \alpha_{t' \text{inst}} \supseteq \gamma_{t' \text{inst}} \)
(b) \( \alpha_{t'} \cdot \alpha_{t' \text{inst}} = \alpha_{t' \text{inst}} \)  
(d) \( \alpha_{t' \text{inst}} \cdot \gamma_{t' \text{inst}} = \gamma_{t'} \)  
(f) \( \gamma_{t'} \cdot \alpha_{t' \text{inst}} \supseteq \alpha_{t' \text{inst}} \)

A generalization of the pair \( (\gamma_{t' \text{inst}}, \alpha_{t' \text{inst}}) \) can be defined, where several type variables are instantiated in sequence. This corresponds to a type \( \forall \beta_{1}, \ldots, \forall \beta_{m}. t' \) and an instantiation \( \text{tinst} = t'[\beta_{1} := t_{1}, \ldots, \beta_{m} := t_{m}] \). The analysis domains together with these pairs as morphisms form a category (by propositions 5(a) and 5(b)).

5 Relation between the Analyses

In [10] the relation between the second and the third analyses was studied. There, it was shown that all variants of the third analysis produces less accurate results than the second one:

Proposition 6 Let \( \mathcal{W}_{t} : D_{t \ast} \to D_{\ast t} \) be a widening operator for each type \( t \). Given \( \rho_{2} \) and \( \rho_{3} \) such that for each variable \( v : t \varepsilon ; \rho_{2} (v) \subseteq \rho_{3} (v) \), then for each expression \( e : t \varepsilon ; [e]_{\rho_{2}} \rho_{2} \subseteq [e]_{\rho_{3}} \).

In this section the relation between the first analysis and the other two analyses is studied. Grosso modo we will see that the first analysis is worse than some variants of the third analysis, but in a safe way, that is, it is a safe approximation to them. In particular we will prove that the first analysis is an
\[
e = \text{let rec}
\]
\[
f = \lambda p. \lambda x. \text{case } p \text{ of}
\]
\[
(p_1, p_2) \to \text{case } p_2 \text{ of}
\]
\[
0 \to (p_1, x)
\]
\[
z \to (f (p_1 \ast p_2, p_2 - 1)) (x \ast p_1)
\]
\[
\text{in let}
\]
\[
f_1 = (f [q, 3])
\]
\[
f_2 = (f (1, 2))
\]
\[
x_1 = \text{case } f_1 \text{ of } (f_{11}, f_{12}) \to f_{11}
\]
\[
x_2 = \text{case } f_2 \text{ of } (f_{21}, f_{22}) \to f_{21}
\]
\[
\text{in } (x_1, x_2)
\]
\[
\mathbb{F}_1 [q \mapsto n] = (n, n) \sqsubseteq \mathbb{F}_3^{\mathbb{W}} [q \mapsto n] = (n, d) \sqsubseteq \mathbb{F}_4, [q \mapsto n] = (d, d)
\]

**Fig. 13.** Differences in power between the analyses.

An upper approximation to those variants of the third analysis that fulfill a property (see Theorem 14). We will prove that the particular widening operator \(\mathbb{W}_i\) defined in Figure 6 fulfills such property. In [10] we mentioned other possible widening operators that could be defined (\(\mathbb{W}_b\), \(\mathbb{W}_c\) and \(\mathbb{W}_d\)) and their relation with \(\mathbb{W}\). The main difference between them lies in their treatment of tuples in the arguments and/or in the result of functions, either as indivisible entities or as componentwise entities. As all the variants of the third analysis are upper approximations to the second one, it will also happen that the first analysis is an upper approximation to the second analysis. In Figure 12 we illustrate the relation between the first analysis, the second analysis and some variants of the third analysis. In [10] an example was given to show the differences in precision between \(\mathbb{F}_1\), \(\mathbb{F}_2\) and \(\mathbb{F}_3^{\mathbb{W}}\), also shown in Figure 13.

In Figure 14 a scheme of the functions defined in this section and their definition domains are shown. Many propositions in this section can be easily visualized using this figure.

In the first analysis there are neither nested tuples nor abstract functions. However, the abstract value \(a\) of a function in such analysis represents an abstract function. It is a function that takes an abstract value as argument; if that value is non-deterministic the result is also non-deterministic, and if it is deterministic it gives as result the abstract value \(a\). This leads us to define a function \(\eta_\delta\), see Figure 15 (left), called the expansion function that expands the abstract values of the first analysis domains into abstract values belonging to the second and third analyses abstract domains. Recall also that in the first analysis there are no nested tuples, so it will be necessary to adapt each of the basic values to the types of the components.

We can also look at the second and third analyses from the point of view of the first one. We just have to flatten the tuples up to the first level and apply the functions to \(\gamma(d)\), to obtain the behaviour of the function when it is applied to a deterministic argument. This is represented by the compression function \(\delta_\eta\) defined in Figure 15 (right). We prove now that \(\delta_\eta\) and \(\eta_\delta\) are a Galois insertion.
**Proposition 7** For each type \( t \):

(a) The functions \( \delta_t \) and \( \eta_t \) are monotone and continuous.
(b) \( \delta_t \cdot \eta_t \equiv \text{id}_{D_t} \)
(c) \( \eta_t \cdot \delta_t \equiv \text{id}_{D_t} \)

The following proposition asserts that given an abstract value in \( D_{2t} \), if we apply \( \delta_t \) to it to obtain a value in \( D_{1t} \) and then we apply \( \cup \) to the result [it could be a tuple of basic abstract values] we obtain the same basic value as applying \( \alpha_t \) directly to it. This shows a similarity between the abstraction function \( \alpha_t \) used in constructor applications and the compression function \( \delta_t \). In the end the ideas are the same.

**Proposition 8** For each type \( t \): \( \cup \cdot \delta_t = \alpha_t \)

Proposition 7 tells us that \( \eta_t \cdot \delta_t \) is a widening operator in \( D_{2t} \). This composition will be widely used in the following, so we define \( \nabla_t = \eta_t \cdot \delta_t \). We will call it the rising function (just to differentiate it from \( \mathcal{V}_t \)).

It also tells us that for each type \( t \), the range of \( \nabla_t \) is isomorphic to \( D_{1t} \):

\( \nabla_t(D_{2t}) \cong D_{1t} \).

This means that the range of \( \nabla_t \) is a subdomain of \( D_{2t} \) where we have lost the additional information provided by the second and third analyses.

For example, we still have nested tuples, but they are maintained in a fictitious way, that is, they have been flattened up to the first level and then unflattened again. So, all the internal tuples will be formed by only \( n \)'s or only \( d \)'s. For example both \( ((n,d),d,(d,d)) \) and \( (d,n),(d,d)) \) are transformed by \( \nabla_t \) into \( (n,d),d,(d,d)) \). We also have abstract functions, but only some of them: Those that can be represented with an abstract value in the first analysis domains. These are the functions such that for all values below \( \gamma_t(d) \), give the same result as the one obtained for \( \gamma_t(d) \), and for the rest of the values give as result the top of the corresponding domain, \( \gamma_t(n) \).

Theorem 14 below asserts the correctness of the first analysis with respect to the third one, but first we will see some more properties that are needed to prove it. The following proposition relates the adaptation function of the first analysis and the concretisation function \( \gamma_t \). The are made equal through the application of \( \delta_t \).
A Comparison between three Non-determinism Analyses in ...

\[ \eta : D_{2t} \rightarrow D_{2t} \]
\[ \eta' = \eta \cdot \eta_t \] with the abstraction and concretisation functions \( \alpha_t \) and \( \gamma_t \). The first two items tell us that once we have gone up in the domain with \( \nabla_t \), the functions \( \alpha_t \) and \( \gamma_t \) behave in the same way in the subdomain of \( D_{2t} \) that is the range of \( \nabla_t \), than in the whole domain. This means that in fact \( \alpha_t \) and \( \gamma_t \) are moving inside this subdomain. In Figure 14 this fact is represented by two \( D_{2t} \) domains encircled by a dashed line, where \( \alpha_t \) and \( \gamma_t \) appear between the range of \( \nabla_t \) and \( \text{Basic} \).

**Proposition 9** For each type \( t \), \( \forall h \in \text{Basic} \). \( b = (\delta_t \cdot \gamma)(b) \)

The following proposition relates the rising function \( \nabla_t \) with the abstraction and concretisation functions \( \alpha_t \) and \( \gamma_t \). The first two items tell us that once we have gone up in the domain with \( \nabla_t \), the functions \( \alpha_t \) and \( \gamma_t \) behave in the same way in the subdomain of \( D_{2t} \) that is the range of \( \nabla_t \), than in the whole domain. This means that in fact \( \alpha_t \) and \( \gamma_t \) are moving inside this subdomain. In Figure 14 this fact is represented by two \( D_{2t} \) domains encircled by a dashed line, where \( \alpha_t \) and \( \gamma_t \) appear between the range of \( \nabla_t \) and \( \text{Basic} \).

**Proposition 10** For each type \( t \): \( (a) \alpha_t \cdot \nabla_t = \alpha_t \) (b) \( \nabla_t \gamma_t = \gamma_t \) (c) \( \nabla_t \subseteq \gamma_t \alpha_t \)

The following proposition adds some results about the morphic polynomials. The first one tells us that we obtain the same result if we adapt an abstract value belonging to \( D_{2t} \) to the type \( \text{tinst} \) and then rise the result than if we first rise it and then adapt the result. The second one tells us that the adaptation of an abstract value in \( D_{2t} \) to obtain an approximation to the abstract value of an instance \( \text{tinst} \) is basically equal to the adaptation made with \( \gamma_t \cdot \text{tinst} \) in the domains of the second and third analyses.

**Proposition 11** Given two types \( t, t' \) and a type variable \( \beta \):

(a) \( \nabla_t \cdot \gamma_t \cdot \text{tinst} = \gamma_{t \cdot \text{tinst}} \cdot \nabla_{t'} \) (b) \( \forall a \in D_{1t} \cdot \alpha_t \cdot \text{tinst} = \delta_t \cdot \text{tinst} \cdot \eta_t(a) \)

A very important and useful property to prove the correctness is the semi-homomorphic property of \( \delta_t \) with respect to the application of a function. But as there are no functional domains in the first analysis, the property holds with respect to the pseudocapplication we use in such domains, that is, the way in which application of a function is interpreted: \( f(x) = \{ [x] \} \cup f \) (see Figure 3).

**Proposition 12** Let \( f \in [D_{2t} \rightarrow D_{2t}], e \in D_{2t}. \) Then: \( \delta_t(f(e)) \subseteq (\hat{\delta}_t(e)) \cup \delta_t(e) \)

To prove the corollary of Theorem 14 we will need the following lemma.

**Lemma 13** For each type \( t \): \( \eta_t(d_t) = \gamma_t(d_t) \)

The following theorem establishes that the first analysis is a safe (upper) approximation to the transformation by \( \delta_t \) of those variants of the third analysis \( \mathbb{I}_a \) such that \( \delta_t \cdot \mathbb{I}_a = \delta_t \).
Theorem 14 Let $\mathcal{W}_t'$ be a widening operator for each type $t$, such that $\delta_t \cdot \mathcal{W}_t' = \delta_t$. If for each variable $v : t_v$, $\rho_1(v) \sqsupseteq \delta_t, (\mathcal{W}_t') \quad \text{then:} \quad \forall e :: t_e. \quad [\{e\}, \rho_1 \sqsubseteq \delta_t, (\mathcal{W}_t')]$. Or equivalently, $\forall e :: t_e. \quad \gamma_t, (\mathcal{W}_t') \sqsupseteq \gamma_t, (\mathcal{W}_t')$

The theorem can be proved by structural induction on $e$ and by propositions 1(a), 1(c), 4(a), 7(a), 8, 9, 11 and 12.

As a corollary, and by Lemma 13 and Proposition 7(c), we obtain the correctness of the first analysis with respect to these variants: If the first analysis tells us that an expression is deterministic then the corresponding variant of the third analysis also tells us that it is deterministic, probably with some additional detail as the independence of the output with respect to the input in a function/process.

Corollary 15 Let $\mathcal{W}_t'$ be a widening operator for each type $t$, such that $\delta_t \cdot \mathcal{W}_t' = \delta_t$. If for each variable $v : t_v$, $\rho_1(v) \sqsubseteq \delta_t, (\mathcal{W}_t')$ then: $\forall e :: t_e. \quad [\{e\}, \rho_1 \sqsubseteq \delta_t, (\mathcal{W}_t')]$

Proposition 16 asserts that the particular widening operator defined in Figure 6 has this property (the widening operator $\mathcal{W}_t$ mentioned in [10] also fulfills it)

Proposition 16 For each type $t$, $\delta_t \cdot \mathcal{W}_t = \delta_t$.

The correctness of the first analysis with respect to the second one is trivially obtained from Theorem 14, as the second analysis is in fact a variant of the third analysis where the widening operator is the identity function.

6 Conclusions

We have presented and formally compared three non-determinism analyses for a functional language with second-order polymorphism and non-deterministic expressions. All of them are based on abstract interpretation. We have proved interesting properties of some flattening and unflattening functions which appear when polymorphism is formally treated, and also have shown that the first analysis is a safe approximation to the second and the third ones.

Although the main motivation for this work has been the correct compilation of our language Eden, everything presented here can be applied to any other non-deterministic polymorphic functional language. Related work concerning other abstract interpretation and type-based analyses was already reported in previous papers [11, 10]. We have not found any previous analyses for this problem in the literature.

The first paper on this subject [11] presented our preliminary ideas in the form of two not completely satisfactory analyses. The first one was efficient but not powerful enough, while the second one was powerful but not practical in terms of efficiency. The second paper [10] developed a third analysis that was both powerful enough and practical in implementation terms. This paper has presented the theoretical results supporting the correctness of that implementation. For our purposes, this closes the initial problem. A proof of correctness
of the second analysis with respect to the standard semantics of the language remains to be done. Unfortunately there is still no such semantics available for Eden. Nevertheless, a simplified form of semantics could be used to prove part of the analysis correctness.

References

Sincronización Multiproceso en Programas Concurrentes.
Selección Completamente $k$–justa de Interacciones *

David Ruiz, Rafael Corchuelo, José A. Pérez y Miguel Toro

Universidad de Sevilla
Departamento de Lenguajes y Sistemas Informáticos
Avenida de la Reina Mercedes s/n, Sevilla, 41.012
e-mail: druiz@isi.us.es

Resumen. La selección completamente justa surge en el contexto de los programas no deterministas y sirve para garantizar que todos los elementos que se habitan infinitamente a menudo se seleccionan infinitamente a menudo. Esta noción de selección presenta dos anomalías: la finitud justa y las conspiraciones. Este artículo se centra en la selección justa de interacciones y presenta una nueva noción llamada selección completamente $k$–justa cuya principal ventaja sobre otras propuestas es que da solución a las dos anomalías de forma simultánea y que el valor de $k$ se puede establecer a priori para caracterizar su bondad. Para ello, hemos descrito un modelo abstracto de interacción que hace independiente el criterio de selección del lenguaje de programación y que se puede acomodar a gran variedad de modelos de interacción. También presentamos un algoritmo general para implementar la selección completamente $k$–justa de interacciones que no requiere acceder al estado local de los procesos del sistema.

Palabras clave: programación concurrente y distribuida, modelo abstracto de interacción, acciones conjuntas, eventos compartidos, exclusión mutua, selección justa, conspiraciones.

1 Introducción

La selección justa surge en el contexto de los programas cuya ejecución no es determinista para poder garantizar propiedades de integridad y de viveza [7] en sus ejecuciones. Este factor no determinista puede ser introducido por (i) lenguajes de programación no deterministas, por (ii) el entrelazado de código en los programas concurrentes o por (iii) la configuración de la red en el caso de programas distribuidos.

En la bibliografía no hemos encontrado ninguna definición de justicia que prevalezca sobre las otras, pero en el contexto de los lenguajes de programación

* Este trabajo está subvencionado por la Comisión Interministerial de Ciencia y Tecnología: proyecto GEOZOCO (TIC2000-1106-C02-01)
parece que la selección completamente justa es la más adecuada [7] ya que permite garantizar propiedades fundamentales de viveza como son la terminación o la respuesta a eventos. Por definición, la ejecución de un programa es completamente justa cuando los elementos\(^1\) que se habilitan infinitamente a menudo resultan seleccionados infinitamente a menudo.

La figura 1 muestra una solución al problema de los filósofos comensales en IP\(^*\), en la que, además de garantizar la exclusión mutua de los tenedores compartidos, también podemos garantizar (asumiendo selección completamente justa de interacciones) que todos los filósofos cogen sus dos tenedores para comer infinitamente a menudo.

$$S ::= \exists_{i=1}^{N} F_i \mid T_i$$
$$F_i ::= (Coger_i,) \rightarrow (Soltar_i,)$$
$$T_i ::= (\begin{array}{c} Coger_i, \rightarrow Soltar_i, \end{array} (\begin{array}{c} Coger_{i+1 \text{ mod } N}, \rightarrow (Soltar_{i+1 \text{ mod } N}) \end{array}$$

**Figura 1.** Problema de los filósofos comensales en IP*.

El problema que presenta este criterio de selección es que por definición se deduce:

**Finitud justa:** Que toda ejecución finita de un programa es justa por definición. La figura 2a muestra una posible ejecución del programa de la figura 1 para \(N = 5\). La notación \(p.\chi\) indica que el proceso \(p\) ha ofrecido el conjunto de interacciones \(\chi, x\) indica que la interacción \(x\) se ha ejecutado. El hecho de que para cualquier valor de \(n\) finito la ejecución es completamente justa implica que interacciones como \(Coger_2\) estén en \(n\) ocasiones habilitadas y nunca resulten seleccionadas.

**Conspiraciones:** Aunque una interacción sea ofrecida por todos sus participantes infinitamente a menudo un entrelazado desafortunado puede provocar que nunca llegue a habilitarse. Esta situación se conoce con el nombre de conspiración y la definición de selección completa no la tiene en cuenta. La figura 2b muestra un ejemplo de ejecución completamente justa en la que la interacción \(Coger_2\) es ofrecida por todos sus participantes infinitamente a menudo pero nunca resulta habilitada.

\(^1\) En nuestro caso los elementos objeto de selección justa son las interacciones entre procesos.
F₁,{Coper₁}, F₂,{Coper₂},
(T₅,{Coper₅}, Coper₁), T₂,{Coper₂, Coper₃}, T₁,{Coper₁, Coper₂}, Coper₁];
F₁,{Sollar₁}, T₁,{Sollar₁, Sollar₂}, T₆,{Sollar₃, Sollar₁}, Sollar₃; F₁, {Coper₁})

F₁,{Coper₁}, F₂, {Coper₂} F₃, {Coper₃},
(T₅,{Coper₅, Coper₁}, T₃,{Coper₃, Coper₄}, T₁,{Coper₁, Coper₂}, Coper₃],
T₂,{Coper₂, Coper₃}, Coper₃], F₁,{Sollar₁}, T₁,{Sollar₁, Sollar₂},
T₅,{Sollar₅, Sollar₁}, Sollar₁]; F₈,{Sollar₃}, T₃,{Sollar₅, Sollar₂}, Sollar₃,
T₃,{Sollar₃, Sollar₄}, Sollar₃], F₁,{Coper₁}, F₃, {Coper₃})

Figura 2. Anomalías de la selección completamente justa de interacciones.

Estas anomalías han sido estudiadas por varios autores dando lugar a nuevos criterios de selección más restrictivos. Entre ellos destacaremos: la selección justa finita [1] y la selección hiperjusta [2].

La selección justa finita intenta paliar el primer problema. Para ello lo que hace es sustituir el término "infinitamente a menudo" de la definición por "al menos una vez cada k veces" siendo k un natural desconocido a priori. Las principales desventajas de esta propuesta es que (i) el valor de k se conoce a posteriori, (ii) que no resuelve el problema de la finitud y las conspiraciones, (iii) no hemos encontrado ningún algoritmo para implementarla.

La selección hiperjusta surge como respuesta al problema de las conspiraciones. Se dice que una ejecución hiperjusta cuando es finita y toda interacción que puede habilitarse infinitamente a menudo lo hace infinitamente a menudo. Con esta noción se pretende garantizar la habilitación de las interacciones y no la selección de las mismas, por lo que se hace necesario combinarla con otra noción de selección de interacciones. Sus principales desventajas son que (i) no resuelve el problema de la finitud y (ii) no se conoce ningún algoritmo general para implementarla.

Nosotros proponemos una nueva noción para resolver simultáneamente las dos anomalías presentadas: la selección completamente k-justa. Una ejecución será completamente k-justa cuando ninguna interacción se ejecuta más de k veces sin conocer el estado definitivo de las interacciones con las que tiene que obtener la exclusión mutua y además cuando se selecciona la interacción más antigua del grupo (este concepto lo definiremos más adelante).

Nuestra noción está definida sobre un modelo de interacción abstracto basado en el concepto de acción conjunta que abarca a modelos de interacción de alto nivel (como el de múltiples participantes) hasta modelos de interacción más

2 Los autores proponen un esquema de transformación que puede ser aplicado a programas expresados con autómatas de Büchi [10, 5].
3 Los autores proponen un esquema de transformación de programas IP que implica modificación del código de los procesos y la existencia de gestores a medida para cada programa.
primitivos como el paso de mensajes o el rendezvous [8]. Con este modelo de interacción conseguimos hacer independiente el lenguaje de programación del criterio de selección que se adopte. Por último, presentamos un algoritmo general para implementar nuestra noción que no necesita acceder al estado de los procesos.

2 Modelo abstracto de interacción

Entendemos que nuestros programas concurrentes y/o distribuidos estarán formados por un conjunto fijo no vacío de procesos y un conjunto fijo no vacío de acciones conjuntas (en adelante interacciones) entre dichos procesos. Todo proceso puede participar durante su ejecución en un conjunto fijo no vacío de interacciones.

Los procesos se ejecutarán de forma concurrente (real o simulada) y en cada instante de tiempo sólo podrán ejecutar una interacción, es decir, suponemos que los procesos sólo tienen un único hilo de ejecución. El único mecanismo que se proporciona para sincronizar procesos distintos son las interacciones, es decir, los procesos no comparten variables ni se envían mensajes entre ellos.

Los procesos se pueden encontrar en tres estados distintos:

Realizando cálculos locales: Cuando un proceso se encuentra en dicho estado sólo podrá ejecutar acciones sobre su estado local, es decir, acciones que no requieren interactuar con otros procesos para llevar a cabo ninguna tarea.

Esperando interactuar con otros procesos: Cuando un proceso se encuentra en este estado diremos que está ofreciendo un conjunto de interacciones y que se encuentra bloqueado a la espera de que alguna ellas se habilite y resulte seleccionada para ser ejecutada.

Finalizado: Un proceso se encuentra en este estado cuando termina su ejecución. En realidad lo que a nosotros nos interesa es que un proceso que se encuentra en este estado no puede ofrecer (ni ejecutar) ninguna interacción en el futuro ni tampoco efectuar cálculos locales.

La vida de un proceso puede describirse con la siguiente expresión regular:

\[ p :: (p,t[p,\chi,x])^*, p \emptyset \]  

(1)

Donde \( p,t \) representa la ejecución de cálculos locales, \( p,\chi \) el ofrecimiento de un conjunto \( \chi \neq \emptyset \) de interacciones, \( x \) la sincronización a través de una interacción \( x \in \chi \) y \( p,\emptyset \) el evento de terminación de \( p \).

Las interacciones (acciones conjuntas sincronas entre un número arbitrario y fijo de procesos) sólo se ejecutan cuando todos los procesos que pueden ofrecerla lo han hecho. Cuando dos o más interacciones pueden ser ofrecidas por los mismos procesos, es decir, tienen algún participante en común diremos que son potencialmente conflictivas. Cuando dos o más interacciones que son potencialmente conflictivas se encuentran habilitadas el mismo tiempo diremos que se ha producido un conflicto y se tendrá que decidir qué interacción ejecutar y
rechazar el resto, ya que como hemos dicho los procesos sólo tienen un único hilo de ejecución.

De esta forma, entenderemos que la ejecución de un programa es la secuencia de ofrecimiento de interacción / ejecución de interacción que provoca el cambio de estado en el mismo. Es decir, asumimos que los cálculos locales que realizan los procesos no son visibles durante la ejecución.

2.1 Descripción detallada

A continuación haremos algunas definiciones para introducir todos estos conceptos de forma precisa. Tomaremos como ejemplo para poder ilustrar estas definiciones el ejemplo de los filósofos comensales visto en la figura 1.

Modelo de los sistemas

**Sistema:** Un sistema $\Sigma$ se define como la la tupla $(P_\Sigma, I_\Sigma)$. $P_\Sigma \neq \emptyset$ es un conjunto finito de procesos (con un único hilo de ejecución y un estado local independiente) e $I_\Sigma \neq \emptyset$ es un conjunto finito de interacciones que permiten sincronizar y comunicar a un número arbitrario y fijo de procesos de $P_\Sigma$.

**Participantes de una interacción:** Dada interacción $x \in I_\Sigma$ se define un conjunto estático de procesos $P(x) \subseteq P_\Sigma$ que estará formado por todos los procesos a los que $x$ puede sincronizar.

**Interacciones de un proceso:** Todo proceso $p \in P_\Sigma$ define un conjunto también estático de interacciones $P(p) \subseteq I_\Sigma$ de forma que sólo se podrá sincronizar y comunicar con otros procesos a través de estas interacciones.

**Configuración:** Una configuración es una estructura matemática que denota el estado en el que se encuentra la ejecución un programa en un instante determinado. Generalmente las configuraciones incluyen el estado de los procesos y alguna información de control adicional. Las denotaremos con $C, C', C_1, C_2, \text{etc.}$

**Eventos:** Las transiciones entre configuraciones de un sistema vienen dadas por eventos. Nuestro modelo define tres eventos: $p.\ell$ cuando el proceso $p$ ejecuta una instrucción local, $p.x$ cuando el proceso $p$ ofrece interacción en alguna interacción $x \in \chi$ y $x$ cuando el sistema ejecuta la interacción $x$.

**Ejecución:** Se define una ejecución $\lambda$ de un sistema $\Sigma$ como la tupla $(C_0, \alpha, \beta)$, donde $C_0$ es la configuración inicial, $\alpha = [C_1, C_2, \ldots]$ es una secuencia maximal (finita o infinita) de configuraciones y $\beta = [e_1, e_2, \ldots]$ también es una secuencia maximal (finita o infinita) de eventos, donde $e_i$ caracteriza la transición entre las configuraciones $C_{i-1}$ y $C_i$ ($i \geq 1$).

**Trazas:** Sea $\lambda = (C_0, \alpha, \beta)$ una ejecución de un programa $\Sigma$. Denotaremos $\alpha$ como $\lambda_0$ (traza de configuraciones) y $\beta$ como $\lambda_3$ (traza de eventos).

**Semántica:** Asumiremos que nuestros programas han sido escritos en un lenguaje $L$ cuya semántica operativa está definida por la regla de transición $\xrightarrow{e}_L$, donde $e$ es un evento del sistema. Dada una ejecución $\lambda = (C_0, \alpha, \beta)$ también la escribiremos de la siguiente forma:

$$C_0 \xrightarrow{e_1}_L C_1 \xrightarrow{e_2}_L C_2 \xrightarrow{e_3}_L \cdots$$ (2)
Modelo de los procesos

**Ofrecimiento de interacción:** Un proceso $p$ estará ofreciendo la interacción $x$ en la configuración $i$-ésima de su ejecución si o sólo si llega a un punto en el que la ejecución de la interacción $x$ se une posible continuación.

$$\text{Readies}(\lambda, p, x, i) \equiv \exists l \leq k \leq i \cdot (\lambda(k) = p \cdot x \in \chi) \wedge \exists j < i \cdot \lambda_\beta(j) = z \wedge z \in \chi)$$  \hspace{1cm} (3)

**Esperando interactuar:** Un proceso $p$ estará esterando interactuar en la configuración $i$-ésima de una ejecución $\lambda$ si y sólo si está ofreciendo alguna interacción.

$$\text{Waiting}(\lambda, p, i) \equiv \exists x \in \mathcal{I}_\Sigma \cdot \text{Readies}(\lambda, p, x, i)$$  \hspace{1cm} (4)

**Proceso finalizado:** Un proceso $p$ habrá finalizado su ejecución en la configuración $i$-ésima de su ejecución $\lambda$ si y sólo si ya no puede volver a ejecutar ni cálculos locales ni interacción.

$$\text{Finished}(\lambda, p, i) \equiv \exists l \leq k \leq i \cdot \lambda_\beta(k) = p \cdot \emptyset$$  \hspace{1cm} (5)

Modelo de las interacciones

**Interacciones enlazadas:** El conjunto de interacción enlazadas a la interacción $x$ en la configuración $i$-ésima de la ejecución $\lambda$ estará formado por aquellas interacciones que han sido ofrecidas por procesos comunes.

$$\text{Linked}(\lambda, x, i) = \{ y \in \mathcal{I}_\Sigma \mid y \neq x \wedge \exists p \in \mathcal{P}_\Sigma \cdot (\text{Readies}(\lambda, p, x, i) \wedge \text{Readies}(\lambda, p, y, i)) \}$$  \hspace{1cm} (6)

**Interacciones habilitadas:** Una interacción $x$ está habilitada en la configuración $i$-ésima de la ejecución $\lambda$ si y sólo si todos los procesos de $\mathcal{P}(x)$ la están ofreciendo.

$$\text{Enabled}(\lambda, x, i) \equiv \forall p \in \mathcal{P}(x) \cdot \text{Readies}(\lambda, p, x, i)$$  \hspace{1cm} (7)

**Interacciones deshabilitadas:** Una interacción $x$ estará deshabilitada en la configuración $i$-ésima de la ejecución $\lambda$ si y sólo si no está habilitada y todos los procesos de $\mathcal{P}(x)$ están esperando interactuar o finalizados.

$$\text{Disabled}(\lambda, x, i) \equiv \neg \text{Enabled}(\lambda, x, i) \wedge \forall p \in \mathcal{P}(x) \cdot (\text{Waiting}(\lambda, p, i) \lor \text{Finished}(\lambda, p, i))$$  \hspace{1cm} (8)

**Interacciones semihabilitadas:** Una interacción $x$ estará semihabilitada en la configuración $i$-ésima de la ejecución $\lambda$ si y sólo si no está habilitada y ha sido ofrecida por al menos un proceso.

$$\text{SemiEnabled}(\lambda, x, i) \equiv \neg \text{Enabled}(\lambda, x, i) \wedge \exists p \in \mathcal{P}(x) \cdot \text{Readies}(\lambda, x, i)$$  \hspace{1cm} (9)

**Interacciones consolidadas:** Una interacción $x$ estará consolidada en la configuración $i$-ésima de la ejecución $\lambda$ si y sólo si está habilitada o deshabilitada.

$$\text{Stable}(\lambda, x, i) \equiv \text{Enabled}(\lambda, x, i) \lor \text{Disabled}(\lambda, x, i)$$  \hspace{1cm} (10)
Ejecuciones de interacción: Se define el conjunto de ejecuciones de una interacción $x$ en la configuración $i$-ésima de la ejecución $\lambda$ como las posiciones de $\beta$ en las que la interacción $x$ ha sido ejecutada.

$$\text{ExeSet}(\lambda, x, i) = \{1 \leq k \leq i \mid \lambda_\beta(k) = x\}$$  \hspace{1cm} (11)

Edad de una interacción: Se define la edad de una interacción $x$ en la configuración $i$-ésima de su ejecución $\lambda$ como el número de configuraciones que hay desde la última ejecución de $x$ o $\infty$ si la interacción nunca se ha ejecutado.

$$\text{Age}(\lambda, x, i) = \begin{cases} i - \max \text{ExeSet}(\lambda, x, i) & \text{si ExeSet}(\lambda, x, i) \neq \emptyset \\ \infty & \text{en otro caso} \end{cases}$$  \hspace{1cm} (12)

Nuestro modelo abstracto está compuesto por dos módulos cuya relación se muestra en la figura 3. El programa informa de los ofrecimientos al módulo de selección de interacciones, que se encargará de detectar cuándo una interacción se habilita y debe ser ejecutada por el programa.

![Diagrama de la relación entre el programa y el algoritmo de selección de interacciones](image)

**Figura 3.** Modelo abstracto: (a) Relación entre el programa y el algoritmo de selección de interacciones. (b) Esquema abstracto de programa. (c) Esquema abstracto de algoritmo de selección.

Esta relación puede modelarse haciendo uso de reglas de inferencia. La figura 3.b muestra el aspecto que tendrían las reglas que describen el comportamiento de los programas ($\rightarrow_S$) y la figura 3.c el algoritmo de selección de interacciones ($\rightarrow_L$). Como se observa, $\rightarrow_S$ aparece en el antecedente de la regla de inferencia del programa. Esto implica que la transición de ejecución de interacción viene dada por el algoritmo de selección de interacciones al programa. Asimismo, $\rightarrow_L$ aparece en el antecedente de la regla de inferencia del algoritmo de selección por lo que la transición de ofrecimiento de interacción viene dada por el programa.
Hay que destacar que este modelo abstracto de interacción sólo muestra el aspecto que deben tener las reglas de inferencia que lo describen, y que para valores concretos de \(\rightarrow_L\) y \(\rightarrow_S\) tendremos distintos casos de modelos de interacción.

La figura 4 muestra dos reglas de inferencia que caracterizan (desde un punto de vista abstracto) a nuestros programas. La regla 13 describe la semántica de un ofrecimiento por parte de un proceso \(p\). El consecuente sólo exige que la transición venga dada por un evento de ofrecimiento. Por otro lado, el antecedente obliga a que para que un proceso \(p\) pueda ofrecer interacción primero llegue a un punto de su ejecución en la que ya no puede ejecutar cálculos locales, y además, para poder volver a ofrecer interacción o ejecutar cálculos locales (\(\omega \in \{p\eta, p\chi\}\)) es necesario que el programa ejecute una interacción en la que \(p\) sea participante. La regla 14 captura la semántica de la terminación de un proceso \(p\). La transición \(\rightarrow_1\) del consecuente esta caracterizada por el evento de terminación. Con el antecedente expresamos que cuando un proceso \(p\) termina su ejecución ya no volverá a ejecutar ni cálculos locales ni a ofrecer interacciones (\(\omega \in \{p\eta, p\chi\}\)).

\[
\begin{align*}
\chi \subseteq 3(p) & \land \chi \neq \emptyset \land (3C' \cdot C_1 \xrightarrow{p\xi} \downarrow C') \land \\
\exists C'', C''', C_2 \cdot (C_2 \xrightarrow{L} C'' \xrightarrow{L} C_3 = C_2 \xrightarrow{L} C''') \quad \text{(13)}
\end{align*}
\]

\[
\begin{align*}
3C', C'' \cdot C_2 & \xrightarrow{\ast} \downarrow C' \xrightarrow{\downarrow C''} \xrightarrow{L} C_3 \\
C_1 & \xrightarrow{L} C_2 \xrightarrow{L} C_3 \quad \text{(14)}
\end{align*}
\]

**Figura 4.** Modelo abstracto de interacción entre procesos de un programa

En el caso del criterio de selección podemos encontrar en la bibliografía varios propuestos de selección de interacciones. De forma abstracta, todos ellos se comportan según el modelo de la figura 3, lo único que cambia entre ellos es la condición de selección que cada uno adopta. Por ejemplo, en [3] se reciben los ofrecimientos y tan pronto como una interacción habilitada consigue la exclusión mutua se selecciona (no tiene en cuenta la selección justa). En [4] se trata el problema de la selección justa de alternativas en instrucciones de selección múltiple no deterministas seleccionando aquella alternativa que hace más tiempo que no se ejecuta haciendo uso de contadores de selección. En [9] se reciben ofrecimientos aleatorios y se selecciona aquella interacción que se habilita primero (se apoya en la Teoría de los Grandes Números para garantizar la selección justa de interacciones). En [6] se espera a recibir todos los ofrecimientos y selecciona aquella interacción que puede ser rechazada en menor número de ocasiones haciendo uso de contadores de rechazo.
3 Selección completamente \( k \)-justa de interacciones

De forma intuitiva entenderemos por ejecución completamente \( k \)-justa aquella que garantiza que ninguna interacción se ejecuta más de \( k \) veces sin que las interacciones con las que está enlazada en ese momento estén consolidadas, además, en caso de conflicto debe seleccionarse la interacción de mayor edad. Formalmente la definimos de la siguiente forma:

**Definición 1 (Selección Completamente \( k \)-Justa)** Sea una ejecución \( \lambda = (C_0, \alpha, \beta) \) y \( k \) un natural no nulo. Diremos que \( \lambda \) es una ejecución completamente \( k \)-justa si el predicado \( \mathcal{SKF}(\lambda, k) \) se satisface.

\[
\mathcal{SKF}(\lambda, k) \equiv \forall x \in I_x, i \in \text{ExeSet}(\lambda, x, \infty) \cdot \text{Enabled}(\lambda, x, i) \land \\
(\text{LStable}(\lambda, x, i) \land \text{LOldest}(\lambda, x, i) \land \Delta(\lambda, x, i) \leq k)
\]

Esta definición hace uso de varias funciones y predicados auxiliares que detallaremos a continuación. \( \text{LStable}(\lambda, x, i) \) es un predicado que utilizaremos para saber si las interacciones enlazadas con \( x \) en la configuración \( i \)-ésima están en un estado consolidado.

\[
\text{LStable}(\lambda, x, i) \equiv \forall y \in \text{Linked}(\lambda, x, i) \cdot \text{Stable}(\lambda, y, i)
\]

\( \text{LOldest}(\lambda, x, i) \) es un predicado que se satisface cuando la interacción \( x \) es la de mayor edad en la configuración \( i \)-ésima de la ejecución \( \lambda \) de entre todas las interacciones que están enlazadas con ella.

\[
\text{LOldest}(\lambda, x, i) \equiv \forall y \in \text{Linked}(\lambda, x, i) \cdot \text{Age}(\lambda, x, i) \geq \text{Age}(\lambda, y, i)
\]

\( \Delta(\lambda, x, i) \) es una función que devuelve el número de veces que la interacción \( x \) se ha ejecutado en presencia de interacciones enlazadas no consolidadas desde la última configuración que se ejecutó hasta la configuración \( i \).

\[
\Delta(\lambda, x, i) = \sum_{0 \leq k \leq i} |\{ k \in \text{ExeSet}(\lambda, x, i) \mid \text{LStable}(\lambda, x, k) \land \text{Linked}(\lambda, x, k) \neq \emptyset \land \neg \text{LStable}(\lambda, x, k) \}| \tag{18}
\]

donde \( \sum_{a \in A} P(a) \triangleq |\{ a \in A \mid P(a) \}| \), y \( \phi \) se define de la siguiente forma (observe que el máximo de un conjunto vacío es \( \bot \)):

\[
\phi \triangleq \begin{cases} j \text{ si } j = \max \{k \in \text{ExeSet}(\lambda, x, i) \mid \text{LStable}(\lambda, x, k) \} \land j \neq \bot \\
1 \text{ en otro caso} \end{cases} \tag{19}
\]

De la definición se deduce que toda ejecución que contenga menos de \( k \) ejecuciones de interacción es completamente \( k \)-justa (ya que ninguna interacción se ha podido ejecutar más de \( k \) veces) y que además \( k \) debe ser conocido a priori\(^4\) y debe verificar que:

\[
k \geq \max \{|1(p) \cap 1(q)| \mid p, q \in P_x \land p \neq q\} \tag{20}
\]

\(^4\) Observe que \( k \) caracteriza a la noción de selección completamente \( k \)-justa, es decir, es un dato de partida. En la noción de selección justa finita el valor de \( k \) se calcula a posteriori sobre la propia ejecución.
4 Algoritmo de selección de interacciones completamente $k$-justo

A continuación vamos a describir en detalle un algoritmo que a partir de cualquier programa cuyo modelo abstracto de interacción sea el descrito en la sección 2 obtenga ejecuciones que sean completamente $k$-justas para un valor de $k$ dado.

Primero haremos una descripción informal a grandes rasgos del algoritmo. Después haremos una descripción más detallada haciendo uso de reglas de inferencia para describir el cambio de estado (configuración) que provocan los distintos eventos.
4.1 Descripción informal

La idea del algoritmo consiste en ordenar todas las interacciones en una cola ($\tau$) de forma que las interacciones que están al final son las que se han ejecutado más recientemente (seleccionar la primera interacción habilitada de una cola ordenada garantiza que nuestras ejecuciones sean completamente justas [6]). Además a cada interacción $x$ se le asocia un contador ($\delta(x)$) que sirve para llevar la cuenta del número de veces que otras interacciones enlazadas con $x$ se han ejecutado cuando el grupo no estaba consolidado.

Para conocer el estado de los procesos y las interacciones utilizaremos un mapa de ofrecimientos ($\varphi$), de forma que cada interacción tiene asociado en tiempo de ejecución cuáles son los procesos que la han ofrecido. Para ello cada vez que se produce una transición de ofrecimiento/interacción se actualiza dicho mapa de la forma adecuada.

Así, nuestro algoritmo siempre seleccionará para su ejecución aquella interacción habilitada que se encuentre primero en la cola y que comparta procesos con un conjunto de interacciones consolidado o que el valor de $\delta$ de ninguna de ellas haya superado el valor de $k$.

4.2 Descripción detallada

Vamos a describir la semántica operativa de nuestro algoritmo de selección completamente $k$-justo haciendo uso de la regla de transición $\rightarrow^*_{SK\Gamma}$ sobre configuraciones que denotaremos como $D, D', D_1, \ldots$. Dichas configuraciones estarán compuestas por la configuración $C$ del programa más las estructuras de datos necesarias para llevar a cabo la selección.

A continuación definiremos las estructuras de datos necesarias así como las funciones que las actualizan.

Estructuras de datos

Mapa de ofrecimientos: Definimos el mapa $\varphi$ de forma que a cada interacción $x \in I_X$ se le asocia un conjunto de procesos $\varphi(x)$,

$$ \varphi \in \{ f : I_X \rightarrow 2^{\mathcal{P}_X} \land \text{dom} \ f = I_X \} \quad (21) $$

Este mapa sirve para disponer en tiempo de ejecución de cuáles son los procesos que han ofrecido cada interacción. De esta forma, podremos saber cuándo una interacción se habilita o cuándo comparte procesos con otras interacciones.

Procesos que finalizados: Se define un conjunto de procesos $\theta \subseteq \mathcal{P}_X$ que contendrá a todos los procesos que han finalizado su ejecución. En este conjunto mantendremos todos los procesos que han terminado su ejecución de forma que se podrá detectar cuándo un proceso ha terminado su ejecución o cuando una interacción no volverá jamás a habilitarse.
Mapa de semiabilitaciones: Definimos un mapa $\delta$ de forma que a cada interacción se le asocia un número natural.

$$\delta \in \{ f : I_\Sigma \to \mathbb{N} \land \text{dom } f = I_\Sigma \}$$  \hspace{1cm} (22)

En este mapa mantendremos información acerca del número de veces que una interacción ha sido rechazada en presencia de otras interacciones con las que compartía algún proceso. De esta forma podremos saber cuándo dentro de un grupo de interacciones que comparten procesos alguna interacción puede estar siendo marginada.

Cola de interacciones: Se define la secuencia de interacciones $\tau$ de forma que las interacciones que se encuentran en las primeras posiciones son aquellas que hacen más tiempo que no se ejecutan.

$$\tau = [x_1, \ldots, x_n] \land n = |I_\Sigma| \land \text{img } \tau = I_\Sigma$$  \hspace{1cm} (23)

Configuración extendida: La configuración extendida del algoritmo en un instante $i$ viene dada por la tupla $(\mathcal{C}_i, \tau_i, \varphi_i, \delta_i, \vartheta_i)$, donde $\mathcal{C}_i$ es la configuración del programa en el instante $i$, $\tau_i$ es la cola de interacciones, $\varphi_i$ es el mapa de ofrecimientos, $\delta_i$ es el mapa de semiabilitaciones y $\vartheta_i$ es el conjunto de procesos finalizados.

La configuración inicial del algoritmo viene determinada por la configuración inicial del programa, cualquier cola de interacciones (no importa el orden), un mapa de ofrecimientos $\varphi_0$ de forma que $\forall x \in \text{dom } I_\Sigma \cdot \varphi_0(x) = \emptyset$, un mapa de semiabilitaciones $\delta_0$ de forma que $\forall x \in I_\Sigma \cdot \delta_0(x) = 0$ y una secuencia de procesos finalizados $\vartheta_0 = \emptyset$.

Funciones de consulta y actualización de las estructuras de datos

Actualización de $\varphi$ y $\vartheta$: Cuando un proceso $p$ decide interactuar a través de conjunto de interacciones $\chi \subseteq I_\Sigma$ utilizaremos la función AddOffer($\varphi, p, \chi$) para crear un nuevo mapa de ofrecimientos actualizado de la siguiente forma:

$$\text{AddOffer}(\varphi, p, \chi) = \{ x \mapsto \varphi(x) \cdot x \in \text{dom } \varphi \land x \notin \chi \} \cup \{ x \mapsto \varphi(x) \cup \{ p \} \cdot x \in \text{dom } \varphi \land x \in \chi \}$$  \hspace{1cm} (24)

Cuando una interacción $x$ se selecciona para ser ejecutada será la función RemoveOffer($\varphi, x$) la que se encargue de crear un nuevo mapa de ofrecimientos de la siguiente manera:

$$\text{RemoveOffer}(\varphi, x) = \{ x \mapsto \varphi(x) \setminus P(x) \cdot x \in \text{dom } \varphi \}$$  \hspace{1cm} (25)

Cuando un proceso $p$ finaliza su ejecución $\gamma$ ofrece el conjunto de interacciones $\chi$ vacío ($p, \emptyset$) la función AddFinished($\vartheta, p$) crea un nuevo conjunto de procesos finalizados en el que incluye a $p$.

$$\text{AddFinished}(\vartheta, p) = \begin{cases} \vartheta \cup \{ p \} & \text{si } \chi \neq \emptyset \\ \vartheta & \text{si } \chi = \emptyset \end{cases}$$  \hspace{1cm} (26)
Actualización de $\tau$: Cuando una interacción $x$ resulta seleccionada para ser ejecutada usaremos la función Order para retrasar la posición de $x$ en $\tau$.

$$\text{Order}(\tau, \delta) = \tau' \text{ si } \text{dom } \tau = \text{dom } \tau' \land \text{ran } \tau = \text{ran } \tau' \land \forall x_1, x_2 \in \text{ran } \tau \cdot \tau'^{-1}(x_1) \leq \tau^{-1}(x_2) \Rightarrow \delta(x_1) \geq \delta(x_2) \quad (27)$$

Interacciones habilitadas disyuntas: Dada una cola de interacciones $\tau$ y un mapa de ofrecimientos $\varphi$ se define el conjunto de todas las interacciones habilitadas disyuntas $\text{Ready}(\tau, \varphi)$ de la siguiente forma:

$$\text{Ready}(\tau, \varphi) = \{ x \in \text{dom } \varphi \cdot \text{P}(x) = \varphi(x) \land \# y \in S \cdot \text{P}(y) = \varphi(y) \land \tau^{-1}(y) < \tau^{-1}(x) \} \quad (28)$$

Siendo $S = \{ z \in \text{dom } \varphi \cdot \varphi(z) \cap \varphi(x) \neq \emptyset \}$.

Interacciones consolidadas: Dado un conjunto de interacciones $\mathcal{T}$, un mapa de ofrecimientos $\varphi$ y un conjunto de procesos finalizados $\theta$ definimos el predicado $\text{Consolidated}(\mathcal{T}, \varphi, \theta)$ que se verifica cuando todos los procesos de todas las interacciones de $\mathcal{T}$ han ofrecido interacción$^5$.

$$\text{Consolidated}(\mathcal{T}, \varphi, \theta) \text{ si } \forall x \in \mathcal{T} \cdot \forall p \in \text{P}(x) \cdot p \in \text{ran } \varphi \lor p \in \theta \quad (29)$$

Actualización de $\delta$: Cuando una interacción $x$ es seleccionada para su ejecución utilizaremos la función $\text{Update}(\varphi, \delta, x)$ para crear un nuevo mapa de semihabilitaciones de la siguiente forma:

$$\text{Update}(\varphi, \delta, x) = \delta \ominus \{ x \rightarrow 0 \} \ominus \{ y \rightarrow \delta(y)+1 \cdot y \in S \setminus \{ x \} \land \text{Offered}(S, \varphi, \theta) \} \quad (30)$$

Definiéndose $S$ igual que en la función $\text{Ready}(\tau, \varphi)$.

Hemos descrito el algoritmo de selección completamente $k$-justo de interacción haciendo uso de las dos reglas de inferencia que se muestran en la figura 5.

En la regla 31 describimos cómo cambian las estructuras de datos cuando se produce una transición de ofrecimiento, es decir, cada vez que un proceso $p$ ofrece el conjunto de interacciones $\gamma$ nuestro algoritmo añade dicho ofrecimiento en el mapa de ofrecimientos. En el caso de que un proceso $p$ finalice lo que hacemos es añadir dicho proceso al conjunto de procesos finalizados.

En la regla 32 se describe qué interacción debe seleccionar para que la ejecución que se genere sea completamente $k$-justa (obsérvese que $C_1 \rightarrow C_2$ aparece en el consecuente). El antecedente de esta regla asume que $x$ es la primera interacción habilitada de la cola $\tau$, de forma que el criterio de selección se satisface si se da alguna de las siguientes condiciones:

1. Si $x$ no es conflictiva con ninguna interacción, es decir, $S = \{ x \}$.
2. Las interacciones con las que $x$ está enlazada están consolidadas.
3. El contador de semihabilitaciones asociado de todas las interacciones enlazadas con $x$ es menor que $k$.

$^5$ Fíjese en que consolidación no implica habilitación pero en cambio habilitación sí implica consolidación.
\[
\begin{align*}
C & \xrightarrow{\text{LK}_L} C' \wedge \\
\varphi' &= \text{AddOffer}(\varphi, p, \chi) \wedge \vartheta' &= \text{AddFinished}(\vartheta, p) \\
(C, \tau, \varphi, \delta, \vartheta) & \xrightarrow{\text{SK}_F} (C', \tau', \varphi', \delta', \vartheta') \quad (31)
\end{align*}
\]

\[
\begin{align*}
x & \in \text{Read}(\tau, \varphi) \wedge \mathcal{S} = \{z \in \text{dom} \varphi : (\varphi(z) \cap \varphi(x) \neq \emptyset)\} \\
\tau' &= \text{Order}(\tau, \delta') \wedge \varphi' = \text{RemoveOffer}(\varphi, x) \wedge \delta' = \text{Update}(\varphi, \delta, x) \wedge \\
(S = \{x\} \cup \text{Offered}(S, \varphi, \vartheta)) \wedge (S \neq \{x\} \wedge \max_{y \in \mathcal{S}(x)} \delta(y) < k) \\
(C, \tau, \varphi, \delta, \vartheta) & \xrightarrow{\text{SK}_F} (C', \tau', \varphi', \delta', \vartheta) \wedge C \xrightarrow{\text{LK}_L} C' \quad (32)
\end{align*}
\]

Figura 5. Algoritmo de selección $SK_F$.

5 Conclusiones

En este artículo hemos presentamos un nuevo criterio de selección de interacciones para poder garantizar propiedades fundamentales en programas cuyo comportamiento es no determinista que con otros criterios resultan imposibles. Este criterio de selección de interacciones resuelve las anomalias de la selección completamente justas mostradas en la figura 2.a y 2.b y sus principales ventajas son:

- No asume que cualquier ejecución finita sea justa por definición, lo que nos permite llegar a ejecuciones como las mostradas en 2.a.
- El valor de $k$ se ajusta empíricamente a priori para cada programa. Dicho valor caracteriza las ejecuciones que cumplen el criterio de selección.
- Garantiza la ejecución de todas las interacciones que pueden habilitarse en un tiempo finito y acotado superiormente (el valor de dicha cota varía en función de $k$).
- Resuelve el problema de las conspiraciones con una bondad que aumenta conforme disminuye el valor de $k$.
- Proporcionamos un algoritmo para implementarlo que no requiere acceder al estado local de los procesos.

Una de las aportaciones más importantes es el concepto de umbral de semihabilitaciones $k$, que utilizamos para detectar las posibles situaciones de conspiración y resolverlas a tiempo. Este umbral caracteriza la ejecución de nuestros programas de forma que nos sirve para regular la velocidad a la que se ejecutarán las interacciones que comparten procesos en tiempo de ejecución. Si $k$ es mínimo entonces las interacciones enlazadas se ejecutarán a la velocidad del participante más lento, ya que en este caso ninguna interacción se selecciona hasta conocer el estado final de todas las interacciones con las que comparte renunciamientos. En este contexto nuestro algoritmo es similar a [4] resolviendo el problema de las conspiraciones a costa de ralentizar los procesos que componen un programa concurrente. Si $k$ es máximo (tiende a infinito) entonces las interacciones enlazadas se ejecutan a la velocidad del participante más rápido, ya que en este caso
toda interacción se selecciona tan pronto como obtiene la exclusión mutua con las que es conflictiva. En este contexto nuestro algoritmo se parece a [3] y [11], en los que no se tiene en cuenta el problema de la selección justa de interacciones pero se consigue ejecutar un gran número de interacciones por segundo.

Referencias

Compatibility of Linda Interfaces

Antonio Brogi\(^1\), Ernesto Pimentel\(^2\), and Ana M. Roldán\(^3\)

\(^1\) Dipartimento di Informatica, Università di Pisa, Italy
\(^2\) Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain
\(^3\) Dpto. de Ingeniería Electrónica, Sistemas Informáticos y Automática, Universidad de Huelva, Spain

Abstract. Linda is a coordination language, originally presented as a set of inter-agent communication primitives which can virtually be added to any programming language.

In this paper, we analyse the use of Linda to specify the interactive behaviour of software components. We first introduce a process algebra for Linda and a we define a notion of process compatibility that ensures the safe composition of components. In particular, we prove that compatibility implies successful computation.

We also argue that Linda features some advantages with respect to similar proposals in the context of dynamic compatibility checking. In this perspective, we propose an alternative definition of compatibility that takes into account the state of a global store, which gives some relevant information about the current execution of the system.

The results presented in this paper correspond to the preliminary steps towards a depth exploration of different formalisms to model interaction in open distributed systems, always oriented to their practical application by means of automatic checking tools.

Keywords: Coordination languages, components, compatibility, interaction, process algebras.

1 Introduction

Component-Based Software Engineering (CBSE) is an emergent discipline growing up in the field of Software Engineering. Although it appeared recently, there is a lot of people working on this field. The reason is the necessity for developing open systems and plug-and-play reusable applications which lead to the concept of “commercial off-the-self” (COTS) component. Historically, the most important previous platforms were CORBA and DCE which were developed by OSF (Open Software Foundation) and OMG (Object Management Group). The following step was formed by several platforms, like COM/DCOM, CCM, EJB, and recently .NET, developed by Microsoft.
It is worth observing, however, that traditional IDLs were defined for describing the services that objects offer, rather than the services they need from other objects or the relative order in which they expect their methods will be called. It is therefore necessary to add protocol information to object interfaces.

The objective of this work is to explore the usability of the coordination language Linda as a language to express protocol information of components. This has been already dealt with in previous works, from different perspectives. Thus, in the field of Software Architecture, several attempts have been made to express, at certain level of detail, the behavior of components involved in a software system architecture. In fact, some formal approaches (e.g. CSP, π-calculus, etc.) have been used to describe protocols governing the interaction of a component with its environment [1, 9]. Other similar works have been developed in the context of component-based software engineering, where also process algebras have been applied [4, 10] to give a more practical view, where both feasibility of automatic tools and the application to real component platforms are the main motivations.

In this work, we explore a different approach based on the coordination model Linda [11], trying to exploit the specific features which characterize it. Linda can be presented as a set of interaction primitives which allow agents to read, delete and add tuples in a tuple space, which is shared by all interacting agents. This way, we define a compatibility relation among agents which takes into account the situation of the store, so making the compatibility checking easier during the system’s execution, and ensuring the safe composition of components. We are interested in using Linda as a specification language for describing component protocols, that is, we do not worry about component computation, but about its interaction behavior. Thus, a component would be defined by two orthogonal descriptions, its execution code (providing the computation), and its Linda-based specification (giving an abstraction of the interaction behavior). In order to have a process calculus semantically equivalent to Linda, we have considered an algebraic view of its communication primitives [8].

The rest of the paper is organized as follows. Section 2 presents a Linda-based process calculus, and we illustrate by means of a simple example how it can be used to specify component protocols. Next section is devoted to introduce the notion of safe composition of components, giving several conditions to ensure it. In this section, we define a notion of compatibility, which is firstly presented in an intuitive way, and then we give an alternative definition which is store sensitive. We also prove
the equivalence between both compatibility presentations. Finally, we give some concluding remarks.

2 Specifying component protocols in Linda

2.1 A Linda calculus

Linda [11] was the first coordination language [12], originally presented as a set of inter-agent communication primitives which can virtually be added to any programming language. Linda's communication primitives allow processes to add, delete and test for the presence/absence of tuples in a shared tuple space. The tuple space is a multiset of data (tuples), shared by concurrently running processes. Delete and test operations are blocking and follow an associative naming scheme that operates like select in relational databases.

In this paper, following [8], we shall consider a process algebra \( L \) containing the communication primitives of Linda. These primitives permit to add a tuple \((\text{out})\), to remove a tuple \((\text{in})\), and to test the presence/absence of a tuple \((\text{rd}, \text{urd})\) in the shared dataspace. The language \( L \) includes also the standard prefix, choice and parallel composition operators in the style of CCS [13].

The syntax of \( L \) is formally defined as follows:

\[
P ::= 0 \mid A.P \mid P + P \mid P \parallel P \mid \text{recX}.P
\]

\[
A ::= \text{rd}(t) \mid \text{urd}(t) \mid \text{in}(t) \mid \text{out}(t)
\]

where 0 denotes the empty process and \( t \) denotes a tuple.

Following [8], the operational semantics of \( L \) can be modeled by a labelled transition system defined by the rules of Table 1. Notice that the configurations of the transition system extend the syntax of processes by allowing parallel composition of tuples. Formally, the transition system of Table 1 refers to the extended language \( L' \) defined as:

\[
P' ::= P \mid P' || (t)
\]

Rule (1) states that the output operation consists of an internal move which creates the tuple \( (t) \). Rule (2) shows that a tuple \( (t) \) is ready to offer itself to the environment by performing an action labelled \( \text{F} \). Rules (3), (4) and (5) describe the behaviour of the prefixes \( \text{in}(t), \text{rd}(t) \) and \( \text{urd}(t) \) whose labels are \( t, \bar{t} \) and \( -t \), respectively. Rule (6) is the standard rule for choice composition. Rule (7) is the standard rule for the synchronization between the complementary actions \( t \) and \( \text{F} \): It models the
effective execution of an \textit{in}(t) operation. Rule (8) defines the synchronization between two processes performing a transition labelled $\tau$ and $\bar{\tau}$, respectively. Notice that the process performing $\bar{\tau}$ is left unchanged, since the read operation $\text{rd}(t)$ does not modify the dataspace. The usual rule (9) for the parallel operator can be applied only to labels different from $\neg t$. Indeed a process $P$ can execute a $\text{nr}(t)$ action in parallel with $Q$ only if $Q$ is not able to offer the tuple $\langle t \rangle$, as stated by rule (10). Notice that, following [8], there are no rules for recursion since its semantics is defined by structural axiom $\text{recX}.P \equiv P[\text{recX}.P/X]$ which applies an unfolding step to a recursively defined process.

The rules of Table 1 are used to define the set of derivations for a Linda system. Following [8], both reductions labelled $\tau$ and reductions labelled $\neg t$ are considered. Formally, this corresponds to introducing the following derivation relation:

$$P \xrightarrow{*} P' \text{ iff } (P \xrightarrow{\tau} P' \text{ or } P \xrightarrow{\neg t} P').$$

Notice that the above operational characterization of $\mathcal{L}$ employs the so-called \textit{ordered} semantics of the output operation [7]. Namely, when a sequence of outputs is executed, the tuples are rendered in the same order as they are emitted. It is also worth noting that also the store can be seen as a process which is the parallel composition of a number of tuples.

Let us finally introduce another derivation relation that will be used as a shorthand in the rest of the paper:

$$P \xrightarrow{\alpha} P' \text{ iff } (P \xrightarrow{*} \alpha \rightarrow P').$$

where $\alpha \in \{t, \neg t\}$.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
(1) $\text{out}(t).P \xrightarrow{\tau} \langle t \rangle \parallel P$ & (6) $P \xrightarrow{\tau} P'$ \\
(2) $\langle t \rangle \xrightarrow{0}$ & (7) $P \parallel Q \xrightarrow{\tau} Q'$ \\
(3) $\text{in}(t).P \xrightarrow{0}$ & (8) $P \xrightarrow{0} P' Q \xrightarrow{\tau} Q'$ \\
(4) $\text{rd}(t).P \xrightarrow{\neg t}$ & (9) $P \xrightarrow{\neg t} P' \parallel Q$ \\
(5) $\text{nr}(t).P \xrightarrow{\neg t}$ & (10) $P \parallel Q \xrightarrow{\neg t} P' \parallel Q$ \\
\hline
\end{tabular}
\caption{Transition system for $\mathcal{L}$.}
\end{table}
2.2 Component protocols in Linda

We now describe how the Linda language can be effectively used to specify the interactive behaviour of components. To exemplify the appropriateness of Linda for specifying component protocols, we will illustrate its application to the standard client/server interaction model.

The typical basic behaviour of a server can be described by the following protocol:

\[
\text{SERVER} = \text{in}(c, \text{tos}, \text{qry}). \text{out}(c, \text{ans}). \text{SERVER}
\]

The server repeatedly exhibits the same interactive behaviour: It first inputs a request and then outputs the answer it computed for the received request. The input operation has three parameters which denote the name (c) of the client who produced the request, the type of service required (some constant tos), and the actual request (qry). The server then returns its answer to the query by placing a tuple of the form \(<c, ans>\) in the shared dataspace.

The typical basic behaviour of a client is instead described by the following protocol:

\[
\text{CLIENT} = \text{out} (#me, \text{tos}, \text{qry}). \text{in} (#me, \text{ans}). \text{CLIENT}
\]

where \(#me\) is the identifier of the client process [11].

Notice that, in the above specification, the client request does not refer to the name of a specific server. Most importantly, a client does not need to be aware of which servers are currently available. Notably, the above specification allows several clients and servers to be dynamically and transparently plugged in an open system.

The above specification describes the basic behaviour of clients and servers. A more refined specification may include for instance the brokerage of the servers currently available for a given type of service. Indeed, the server protocol may be rewritten so that the first operation a server performs is to inform the system that she is a server featuring a certain type of service. This can be done by outputing a tuple that associates the process identifier with a certain type of service, as specified in the following protocol:

\[
\text{SERVER} = \text{out}(\text{tos}, #me). \text{CYCLE}
\]

\[
\text{CYCLE} = \text{in}(c, #me, \text{qry}). \text{out}(#me, c, \text{ans}). \text{CYCLE}
\]

where \(\text{CYCLE}\) is a process name. Notice also that the \(\text{SERVER}\) protocol employs the tuple format \(<\text{sender}, \text{receiver}, \text{message}>\) for the message exchanged between clients and servers on the shared dataspace.
Server brokerage can be then easily included in the client protocol as follows:

\[
\text{CLIENT} = (\text{rd}(\text{tos}, \text{srv}).\text{out}(\#\text{me}, \text{srv}, \text{qry}).\text{in}(\text{srv}, \#\text{me}, \text{ans}).\text{CLIENT}) \\
+ (\text{nr}(\text{tos}, \text{srv}).\text{EXCEPTION})
\]

Namely the client determines the name of a server offering the desired type of service by means of the \text{rd}(\text{tos}, \text{srv}) operation. If there is no server available for such type of service (\text{nr}(\text{tos}, \text{srv})) then the client will have to handle the unexpected situation by means of some process \text{EXCEPTION}.

3 Correct composition of components

3.1 Components compatibility

We now introduce a notion of compatibility among processes in order to determine whether two processes — specified as two Linda agents — conform one another.

Let us first define the notion of successful computation which, intuitively speaking, denotes the absence of deadlock in all possible alternative executions of a process.

**Definition 1 (Successful computation).** A process \( P \) is a possible failure if there exists an agent \( P' \) such that \( P \xrightarrow{\ast} P' \xleftarrow{\ast} \) and \( P' \) is not structurally equivalent to a store (i.e., a parallel composition of tuples). On the contrary, a process \( P \) is successful if \( P \) is not a possible failure.

Before defining the notion of compatibility among processes, we introduce the notion of synchronizable processes. This notion is needed for technical reason, and its need will be better clarified later.

**Definition 2 (Synchronizable processes).** A process \( P \) provides an input \( a \) for an agent \( Q \) if there exist two processes \( P' \) and \( Q' \), such that \( P \xrightarrow{a} P' \) and \( Q \xrightarrow{a} Q' \), where \( a \in \{a, \bar{a}\} \). Two processes \( P \) and \( Q \) are synchronizable if \( P \) provides an input for \( Q \) or \( Q \) provides an input for \( P \).

**Definition 3 (Compatible processes).** A process \( P \) is semi-compatible with a process \( Q \), written \( P \preceq Q \) (and the relation \( \preceq \) is called a semi-compatibility), iff:

1. If \( P \) is not successful then \( P \) and \( Q \) are synchronizable
2. If $P$ only can proceed by $\not \rightarrow t$ transition then $Q \not \rightarrow^{*} T$  
3. If $P \not \rightarrow^{*} P'$ then $P' C Q$ 
4. If $P \not \rightarrow^{*} P'$ and $Q \not \rightarrow^{*} Q'$ then $P' C Q'$ 
5. If $P \not \rightarrow^{*} P'$ and $Q \not \rightarrow^{*} Q'$ then $P' C Q'$ 
6. If $P \not \rightarrow^{*} P'$ and $Q \not \rightarrow^{*} Q'$ then $P' C Q'$ 

A relation $C$ is a compatibility if both $C$ and $C^{-1}$ are semi-compatibilities. We say that two processes $P$ and $Q$ are compatible, and we denote it by $P \circ Q$, if there exists a compatibility relation $C$, such that $P C Q$.

Intuitively speaking, two Linda processes are compatible if for each possible action offered by one of them there is a corresponding answer from the other one, and vice-versa. Notice that condition (i) has a technical justification as it avoids considering two unrelated processes (viz., two processes that do not share any action) compatible.

If we consider the notion of bisimilarity defined in other process calculi [14], we can observe that compatibility provides a different way of comparing processes. In fact, whereas two bisimilar processes present the "same" behavior, two compatible processes describe two "complementary" behaviors.

When processes are defined with a finite number of states (even if they present an infinite behavior), it is worth observing that it is possible to implement a tool capable of automatically checking the compatibility of two processes. Obviously, depending on the structural complexity of the processes, the cost of checking might be very high. In any case, even when infinite behavior is dealt with, the usefulness of a tool is clear. Thus, a negative answer showing the non-compatibility of two components could prevent from wrong compositions. Obviously, the compatibility of two generic processes is not always decidable. However, from a practical point of view, it is usual to complement the description of a component with state finite processes. We can, for instance, verify that the processes:

$\text{CLIENT} = \text{out}(\#me, \text{tos}, \text{qry}) . \text{in}(\#me, \text{ans}) . \text{CLIENT}$

and:

$\text{SERVER} = \text{in}(c, \text{tos}, \text{qry}) . \text{out}(c, \text{ans}) . \text{SERVER}$

are compatible. Since $\text{CLIENT} \not \rightarrow^{*} \text{CLIENT}'$ and $\text{SERVER} \not \rightarrow^{*} \text{SERVER}'$, we check the compatibility of the two new processes $\text{CLIENT}'=\text{in}(\#me, \text{ans}) . \text{CLIENT}$ and $\text{SERVER}'=\text{out}(c, \text{ans}) . \text{SERVER}$.

Now, $\text{CLIENT}' \not \rightarrow^{*} \text{CLIENT}$ and $\text{SERVER}' \not \rightarrow^{*} \text{SERVER}$. Then, both pro-
cesses belong to a compatibility relation, and then, we can conclude that CLIENT and SERVER are compatible.

On the other hand, we can observe that an "eager" server (that may terminate if there are no pending queries) such as:

\[ \text{SERVER2} = \text{in(c,tos,qry).out(c,ans).SERVER2 + nrd(c,tos,qry).0} \]

is not compatible with CLIENT.

We now prove that the compatibility of two processes implies the success of their parallel composition.

**Theorem 4.** If \( P \circ Q \) then \( P \parallel Q \) is successful.

**Proof.** Suppose that \( P \circ Q \) but \( P \parallel Q \) is not successful. Then, there exists \( F \) (a possible failure) such that \( P \parallel Q \xrightarrow{\tau} F, F \) is not a set of tuples and \( F \xrightarrow{\tau} \). We will prove that it is not possible by induction on the number \( n \) of \( \tau \)-transitions leading to \( F \).

1. **Base Case.** Suppose \( n = 0 \). Then \( P \parallel Q = F \xrightarrow{\tau} \), therefore \( P \xrightarrow{\tau} \) and \( Q \xrightarrow{\tau} \). If \( P \) or \( Q \) is a stuck process, then, by the first condition of compatibility, we infer that \( P \) and \( Q \) are synchronizable (it means that \( a \) exists such that \( P \xrightarrow{a} P' \) and \( Q \xrightarrow{a} Q' \), where \( a \in \{a, b\} \)). And then, we have \( P \parallel Q \xrightarrow{\tau} P' \parallel Q' \xrightarrow{\tau} \), which is a contradiction. Another possibility is that \( P \xrightarrow{\tau} \) and \( Q \xrightarrow{\tau} \) for some action \( t \) (or vice versa); but this is contradictory with the second condition of compatibility.

2. **Inductive hypothesis.** \( \forall P', Q'. P' \circ Q', \text{if } P' \parallel Q' \xrightarrow{\tau} F \text{ with } k < n \), then either \( F \xrightarrow{\tau} \) or \( F \) is structurally equivalent to a store (i.e. a parallel composition of tuples).

3. **General Case.** Suppose that \( (P \parallel Q) \xrightarrow{\tau} (P' \parallel Q') \xrightarrow{\tau}^{n-1} F \). Then the initial transition is due to one of the following situations:
   (a) \( P \xrightarrow{\tau} P' \). Then, since \( Q = Q' \) we have that \( P' \circ Q' \)
   (b) \( P \xrightarrow{\tau} P' \) and \( Q \xrightarrow{\tau} Q' \). Then, we have that \( P' \circ Q' \)
   (c) \( P \xrightarrow{\tau} P' \) and \( Q \xrightarrow{\tau} \). Then, we have \( P' \circ Q' \)
   (d) \( P \xrightarrow{\tau} P' \) and \( Q \xrightarrow{\tau} Q' \). Then, we have \( P' \circ Q' \)
   (e) Or the symmetrical situations for \( Q \).

Because of \( P \circ Q \) in every situation, we can apply the inductive hypothesis and deduce that either \( F \xrightarrow{\tau} \) or \( F \) is a parallel composition of tuples, again obtaining a contradiction.

\[ \square \]

For instance, the previously described processes CLIENT and SERVER are compatible, and Theorem 4 ensures that their parallel composition is a success.
3.2 Store sensitive compatibility

In Linda inter-process communication occurs only via a shared store (or dataspaces) which is a (multi)set of tuples inserted, extracted or deleted by the concurrent processes.

In order to have an explicit treatment of the store, we now define a compatibility relation that takes into account the situation of the store. As we will see, we can obtain a similar result concerning successful computation in the presence of compatibility. An advantage of having an explicit reference to the store is the possibility of establishing dynamic compatibility checking. Indeed, a Linda-based computation is characterized by the store’s evolution, so that the set of tuples included into the store governs each computation step. This way, the aim of the following definition is to enable run-time, store-sensitive compatibility checking.

**Definition 5 (Compatible processes with respect to a store).** A process $P$ is semi-compatible with a process $Q$ w.r.t. a store $\text{Store}$, written $P \succeq_{\text{Store}} Q$, iff:

1. If $P$ is not successful then a partition of $\text{Store}$, $\text{Store}_P || \text{Store}_Q$, exists such that $P || \text{Store}_P$ and $Q || \text{Store}_Q$ are synchronizable.
2. If $P \xrightarrow{t} P'$ then $P' \succeq_{\text{Store}} Q$.
3. If $P \xrightarrow{t} P'$ and $\text{Store} \xrightarrow{i} \text{Store}'$ then $P' \succeq_{\text{Store}} Q$.
4. If $P \xrightarrow{t} P'$ and $Q \xrightarrow{i} Q'$ then $P' \succeq_{\text{Store}} Q$.
5. If $P \xrightarrow{t} P'$ and $\text{Store} \xrightarrow{i} \text{Store}'$ then $P' \succeq_{\text{Store}} Q'$.
6. If $P \xrightarrow{t} P'$, $Q \xrightarrow{i} Q'$, and $\text{Store} \xrightarrow{i} \text{Store}'$ then $P' \succeq_{\text{Store}} Q$.

A relation $\succeq_{\text{Store}}$ is a compatibility w.r.t. the store $\text{Store}$ if both $\succeq_{\text{Store}}$ and $\succeq_{\text{Store}}^{-1}$ are semi-compatibilities w.r.t. the same $\text{Store}$. We say that two processes $P$ and $Q$ are compatible w.r.t. $\text{Store}$, and we denote it by $P \circ_{\text{Store}} Q$, if there exists a compatibility relation $\succeq_{\text{Store}}$, such that $PC_{\text{Store}}Q$.

As in Definition 3, condition $(i)$ is introduced for technical reasons to avoid two unrelated processes to be considered compatible. In this case, since the compatibility relation is relative to a certain store, we allow to have unrelated (without sharing complementary actions) compatible processes, whenever they might be related through the store. That is, when two processes do not present any common (complementary) behavior, but one of them, after consuming a tuple from the store, synchronizes
with the other process, then they may present a compatible behavior. In another case, these processes should not be considered compatible.

Notice that two processes which are not compatible in the sense of Definition 5 can be compatible with respect to a convenient store. For example, if we consider the process \texttt{CLIENT} \texttt{2} \texttt{in} \langle \#me, srv, ans \rangle. \texttt{CLIENT}, it is not compatible with the process \texttt{SERVER}, but they are compatible w.r.t. a store containing the tuple \langle c, tos, qry \rangle. Here, we can see how the new introduced notion makes more flexible the compatibility between two processes. In fact, the compatibility of a client and a server could depend on the actions already made by a third component already created (the one which has put the convenient tuple into the store.)

The result of Theorem 4 can be extended to \texttt{Store}, obtaining the following theorem.

**Theorem 6.** If \( P \circ \texttt{Store} \ Q \) then \( P || Q || \texttt{Store} \) is successful.

**Proof.** Analogous to the proof of Theorem 4. \qed

Theorem 6 ensures the success of the computation of a pair of processes in presence of a suitable store. In practice, Theorem 6 can be used:

- both for checking the compatibility of a component and of a running system w.r.t. the current store (characterizing the current state of the execution),
- and for conditioning the acceptance of a given component into an open running system so as to wait for a suitable state of the store in order to ensure the success of the overall system.

It is worth observing that the two relations \( \circ \) and \( \circ \texttt{Store} \) are closely related. Although we have not fully proved the relationship between both notions, we conjecture the following result: “if \( P \circ \texttt{Store} \ Q \) then \( (P || \texttt{Store}_P) \circ (Q || \texttt{Store}_Q) \) for some partition \( \texttt{Store} = \texttt{Store}_P || \texttt{Store}_Q \)”. This is rather natural, because the notion of compatibility w.r.t. a store is defined in terms of the complementary behavior of a process with respect to another one (as it is made in the notion of compatibility), and with respect to the store, which is dealt with as one more process (a parallel composition of tuples). Informally, \( \circ \) can be seen as a different presentation of \( \circ \texttt{Store} \). What we mean with this is that the compatibility with respect to a store could be defined in terms of the compatibility relation, where the store can be seen as one more process; a parallel composition of tuples. The advantage of using the presentation given by \( \circ \texttt{Store} \) is its usefulness from the automatic checking perspective. Although the new
compatibility relation is relevant per se (because the store plays an important role in the interaction of components, and it is explicitly considered), a more interesting point is the possibility of building an automatic checking tool capable of determining which is the store (if any) that makes two given processes compatible.

4 Concluding remarks

Linda is a coordination language where inter-process communication can only occur through a set of tuples, and the main novelty of our proposal consists of defining a compatibility relation taking into account the situation of the store. The advantage of this is the possibility of establishing dynamic compatibility checking. That is, when a component has to be incorporated into an already executing system (seen as another component), the compatibility has to be analyzed dynamically, and the "static" specification is not enough because it presents the behavior of a component from its instantiation. Indeed, the advantage of using a Linda-based formalism is that a Linda computation is characterized by the store's evolution, in such a way that the set of tuples included in the store governs each computation step. This is not made in other proposal, where other formalisms, like CSP or \( \pi \)-calculus, are used. We believe that this Linda's feature can be potentially be used to establish the compatibility of executing components, by using the store to have information about the current state of the component.

Indeed, some of the issues covered in this paper have also been dealt with in other proposals. In the context of software architecture Allen and Garlan [1] use the process algebra CSP to describe synchronization of components and connectors, while having some limitations concerning the dynamic change of configurations. Another proposal improving the expressiveness of interaction descriptions by using \( \pi \)-calculus was presented by Canal [9]. Some of the ideas proposed in [9] have already been applied to CORBA in [10]. In this case, dynamic interaction among components (dynamic change of topology) can be better expressed than in CSP. Other works, like [4], propose the use of (a subset of) \( \pi \)-calculus to describe interaction patterns for components so as to reduce the cost of verifying correctness properties in dynamic, open systems. Our proposal somehow combines these two last lines by defining a notion of process compatibility in the style of [9,10], while focusing on the automatic, run-time checking of properties in dynamic, open systems in the style of [4].
Our future work will be devoted to define an inheritance relation over processes in order to promote the reusability and substitutability of interaction descriptions, and to study how this affects compatibility and successful computations. We are also planning to develop an automatic tool (by applying model checking techniques) to check compatibility in order to explore the practical application of our proposal and to analyze and experiment the cost of checking properties in practical real-word cases.

New generation component-based platforms (e.g., .NET) will allow protocol information to be directly included in the metalanguage description (e.g., in XML) of a component. In this perspective, our future work will be devoted to develop a methodology for coding protocol information as metalanguage descriptions and for checking composition properties by analyzing their metalanguage descriptions.

References

A Pattern-based Language to Coordinate HPF Tasks*

Manuel Díaz, Bartolomé Rubio, Enrique Soler, and José M. Troya

Dpto. Lenguajes y Ciencias de la Computación. Málaga University.
29071 Málaga, SPAIN
{mdr, tolo, esc, troya}@lcc.uma.es

Abstract. This paper describes how to use a pattern-based approach to integrate task and data parallelism. Coordination patterns are used to express task parallelism among a collection of data parallel HPF tasks. Patterns specify the interaction among domains involved in the application along with the processor and data layouts. The use of domains, i.e., regions together with some interaction information, improves pattern reusability. Data distribution belonging to the different HPF tasks is known at the coordination level. This is the key for both computational code reutilization and an efficient implementation of the communication among tasks. Besides that, our system implementation requires no change to the runtime system support of the HPF compiler used. In addition, a set of different implementation templates are provided in order to ease the programmer task. The suitability, expressiveness and efficiency of the language are shown by means of some examples.

Keywords: Coordination patterns, Task and data parallelism integration, Pattern and computational code reusability.

1 Introduction

There has been a tendency in the parallel programming paradigm to ignore high level issues, particularly in the area of programming model and language design. This may be reasonable since performance is the main purpose of parallel programming. Currently, an important effort is being carried out in order to apply structured programming principles to parallel programming. This is justified by the experience, as real parallel programs rarely consist of random collections of processes interacting in an unpredictable way, but these interactions are usually well structured and fit a set of patterns. In this sense, in [15] a methodology for structured development of parallel software is proposed. This methodology is based on the establishment of a fixed set of patterns or constructors of parallelism, which are the only way to express the parallel structure of the program.

In the same way, the coordination paradigm [6] provides parallel computing with a high level way of facing the development of parallel software. It is based on

* This work was supported by the Spanish project CICYT TIC-99-0754-C03-03
the separation of coordination and computational aspects. Coordination models and languages offer a good solution to the problem of managing the interactions among concurrent processes. The purpose of a coordination model and the associated language is to provide a mean of integrating a number of possibly heterogeneous components in such a way that the collective set forms a single application that can execute on and take advantage of parallel and distributed systems.

A research area that can take advantage of both structured parallel programming and coordination models and languages is the integration of task and data parallelism. This area tries to overcome the constrain that arises when using a pure data parallel language such as High Performance Fortran (HPF) [12]. It is widely accepted that many important parallel applications cannot be efficiently implemented following a pure data parallel paradigm. For these applications, rather than having a single data parallel program, it is more appropriate to subdivide the whole computation into several data parallel pieces, where these run concurrently and co-operate, thus exploiting task parallelism. Although several approaches have been proposed [10][11][19], integrating the two forms of parallelism cleanly and within a coherent programming model is difficult [4]. The use of coordination languages and structured parallel programming is proving to be a good alternative, providing high level mechanisms and supporting different forms of task parallelism structures in a clear and elegant way [3][7][17].

In this paper we describe DIP (Domain Interaction Patterns), a new approach to integrate task and data parallelism using patterns. DIP is a high level coordination language to express task parallelism among a collection of data parallel HPF tasks, which interact according to static and predictable patterns. Patterns specify the interaction among domains involved in the application, the relationship between domains and tasks, and the mapping of processors and data distribution. In order to ease the programmer task, a set of different implementation templates are provided. This way, the programmer uses a higher level of abstraction to manage communication and synchronization aspects, and some other low level details can also be avoided in the computational part.

On the one hand, the use of domains, which are regions together with some interaction information, makes the language suitable for the solution of numerical problems, especially those with an irregular surface that can be decomposed into regular, block structured domains. Moreover, other kinds of problems that take advantage of integrating task and data parallelism and have a communication pattern based on (sub)array interchange may also be defined and solved in an easy and clear way. The use of domains also avoids that some computational aspects involved in the application, such as data types, have to appear at the coordination level, as it occurs in other approaches [7][17]. This improves pattern reusability.

On the other hand, the knowledge at the coordination level of data distribution belonging to the different HPF tasks is the key for both computational code reutilization and an efficient implementation of the communication and
synchronization among them. In DIP, the inter-task communication schedule is established at compilation time.

DIP implementation is based on BCL [8][9], a Border-based Coordination Language focused on the solution of numerical problems. BCL provides a simple parallelism model where coordination and computational aspects are clearly separated. The former are established using the coordination language and the latter are coded using HPF (together with only a few extensions related to coordination). This way, there is a coordinator process that is in charge of both creating the different HPF tasks and establishing the communication and synchronization scheme among them. In the coordination part, processor and data layouts are also specified. BCL is implemented on top of the MPI communication layer, but no change to the underlying HPF compiler has been needed.

The rest of the paper is structured as follows. Section 1.1 discusses related work. DIP is described in section 2. Section 3 discusses the implementation issues and preliminary results and, finally, in section 4, some conclusions are sketched.

1.1 Related work

In recent years, several proposals have addressed the integration of task and data parallelism. We shall state a few of them and discuss the relative contributions of our approach.

In HPF/MPI [10], the message-passing library MPI has been added to HPF. In an HPF/MPI program, each task constitutes an independent HPF program in which one logical thread of control operates on arrays distributed across a statically defined set of processors. At the same time, each task is also one logical process in an MPI computation. In our opinion, the adoption of a message-passing paradigm to directly express task parallelism is too low-level. Moreover, in our approach, the inter-task communication schedule is established at compilation time from the information provided at the coordination level related to the inter-domain connections and data distribution. In this case, expressiveness and good performance are our relative contributions.

The SkIE [3] environment is also focused on heterogeneous and multidisciplinary applications and uses another coordination language (SkIECL) but, in this case, a pattern-based approach is adopted. The basic idea of this integrated heterogeneous environment is to allow the rapid prototyping and development of complex applications on several platforms. This work evolves from P3L [2], a structured parallel programming language that embeds a set of parallel constructs expressing basic parallel paradigms into C. In SkIE, different sequential languages for computational tasks have been considered [e.g., C, C++, F77, F90, Java, ...]. The user can also encapsulate parallel code using MPI and specialized libraries.

Another work that evolves from P3L is taskHPF [7]. It is a high level coordination language to define the interaction patterns among HPF tasks in a declarative way. Considered applications are structured as ensembles of independent data parallel HPF modules, which interact according to static and predictable
patterns. taskHPF is possibly the closest proposal to DIP. However, the differences are substantial: a) we work with domains, without considering data types at the coordination level, improving pattern reusability; b) our pattern provides information about the future data distribution together with the processor layout, which allows scheduling the inter-task communication pattern at compilation time. On the other hand, DIP provides a multi-block pattern that makes the language suitable for the solution of domain decomposition-based problems and multi-block codes. Both proposals provide implementation templates. However, DIP improves computational code utilization as the data distribution is specified in the pattern instead of the task.

The implementation of taskHPF is based on COLT$_{HPF}$ [14], a runtime support specifically designed for the coordination of concurrent and communicating HPF tasks. It is implemented on top of MPI (there is a new version using PVM) and requires small changes to the runtime support of the HPF compiler used.

2 The DIP coordination language

DIP is a high level coordination language which allows the definition of a network of cooperating HPF tasks. Tasks interact according to static and predictable patterns and can be composed using predefined structures, called patterns. We have established two patterns in DIP. The multi-block pattern is focused on the solution of multi-block and domain decomposition-based problems, which conform an important kind of problems in the high performance computing area. The other pattern provided by DIP is the pipeline pattern, which pipelines sequences of tasks in a primitive way.

DIP is based on the use of domains. HPF tasks receive the domains they need and use them to establish the necessary variables for computation. Local computations are achieved by means of HPF sentences while the communication and synchronization among tasks are carried out through some predefined DIP primitives. The programmer can also use a higher level of abstraction to manage communication and synchronization aspects, by means of implementation templates.

2.1 The MULTIBLOCK pattern

Domain decomposition methods are successfully being used for the solution of linear and non-linear algebraic equations that arise upon the discretization of partial differential equations (PDEs) [18]. Programming such applications is a difficult task because we have to take into account many different aspects, such as: the physics of each domain; the different numerical methods applied; the conditions imposed at the borders; the equations used to solve them; overlapping or non-overlapping techniques; the problem geometry, which may be complex and irregular and, finally, the possible integration of task and data parallelism.

In order to express this kind of problems in an easy, elegant and declarative way, the MULTIBLOCK pattern has been defined. The following code shows the general scheme of this pattern:
MULTIBLOCK pattern_name domain definitions
  task1(domain1:(data distribution)) processor layout
  ....
  taskm(domainm:(data distribution)) processor layout
WITH BORDERS
  border definitions
END

A domain definition is achieved by means of an assignment of Cartesian points, i.e. the region of the domain is established. For example, in the following code:

MULTIBLOCK example  u/1,1,Nxu,Nyu/, v/1,1,Nvx,Nvy/
solve(u:(BLOCK,BLOCK)) ON PROCES(4,4)
solve(v:(*,BLOCK)) ON PROCES(2)
WITH BORDERS
  u(Nxu,Ny1,Nxu,Ny2) <- v(2,1,2,Nvy)
  v(1,1,1,Nvy) <- u(Nxu-1,Ny1,Nxu-1,Ny2)
END

the expression u/1,1,Nxu,Nyu/ assigns to the two-dimensional domain u the region of the plane that extends from the point (1,1) to the point (Nxu,Nyu). In general, a region will consist of 2x numbers, where x is the problem dimensionality (1 ≤ x ≤ 4).

Different borders can be defined among the specified domains. In the previous example, the expression u(Nxu,Ny1,Nxu,Ny2) <- v(2,1,2,Nvy) indicates that the zone of u delimited by points (Nxu,Ny1) and (Nxu,Ny2) will be updated by the values belonging to the zone of v delimited by points (2,1) and (2,Nvy) (see Figure 1). A border definition can be optionally labeled with a number that indicates the connection type in order to communicate different borders in different phases of the algorithm.

In the task call specification, the name of the domain to be solved by the task and the data distribution are specified (optionally, an arbitrary number of additional arguments needed by the task can also be passed). The processor

Fig. 1. Communication between two HPF tasks
layout where the task is going to be executed is also indicated. Each task is solved by a disjoint set of processors. In the example above, the first task is executed on a $4 \times 4$ mesh of processors while the second one, on an array of 2 processors. The distribution types correspond to those of HPF. In Figure 1, dotted lines represent data distribution, and it shows a (BLOCK,BLOCK) distribution for $u$ and a (*,BLOCK) for $v$. This declaration indicates the future distribution of the data associated to the specified domain (see section 2.3). A task knows the distribution of its domain and the distribution of every domain with a border in common with its domain by means of the information declared in the pattern. So, the part of the border that needs to be sent to which processor of other task can be deduced. This is achieved at compilation time.

2.2 The PIPE pattern

The PIPE pattern pipelines sequences of HPF tasks. The structure of a general $n$-stage pipeline corresponding to the PIPE pattern is shown in the following code:

```
PIPE _pattern_name_ domain definitions
  stage1
  .......
  stageN
END
```

Each stage in the pipeline consumes and produces a data stream, except the first and the last stages that only produces and consumes, respectively. The data stream consists of a number of elements. This number, i.e. the stream length, may or may not be statically known. Since our approach is based on the use of domains, the actual data type of the input/output channels connecting each pair of interacting tasks, i.e. the element type, is not specified. This improves the reusability of the coordination part of the application.

A stage of the pipeline can be one of the following:

- A task call, which has a similar form to the task call specification in the MULTIBLOCK pattern.
- A pipeline call, i.e. the name of a nested PIPE pattern together with the domains it needs.
- A REPLICATE directive that is used to replicate a non-scalable stage, which can be a task call or a pipeline call. This improves the pipeline throughput as different data sets can be computed in parallel on different sets of processors.

We use the following example to explain the different characteristics of our PIPE pattern:

```
PIPE _nested_pipeline_ f/1,1,N,N/
  task2(f:(*,BLOCK)) ON PROC (4)
  task3(f:(BLOCK,*)) ON PROC (4)
```
Fig. 2. Structure of the nested pipeline

In the example above, task5 is replicated in such a way that two instances of this task are executed on two processors each. This way, task4 and task6 must carry out some special work due to the presence of the replicate structure. Thus, task4 must dispatch the different elements of its output stream to the two instances of task5, and task6 has to collect the elements received from both instances. Different dispatch methods can be carried out, e.g. round-robin, on demand, etc.

Note that PIPE patterns allow us a high level description of an application structure. A programmer can completely change this structure by only modifying this concise, simple and high level description, so that several alternative implementations can be tested and evaluated. Unlike in other proposals [7], the
absence of input/output directives improves the reutilization of simple, nested and replicated stages.

2.3 Computational tasks

A task achieves local computations by means of HPF sentences while the communication and synchronization among tasks are carried out through some incorporated primitives. A new type and a new attribute have also been included. The following code shows the general scheme of a task:

```fortran
subroutine task_name(list_of_domains)
  domain declarations
  grid declarations
  grid distribution
  grid initialization
  body code
end subroutine
```

By means of the type \texttt{DOMAINxD} \((1 \leq x \leq 4)\), a task declares variables for the received domains. For example, in the following task:

```fortran
subroutine solve(d)
  DOMAIN2D d
  double precision,GRID(d)::g,g_old
  !hpfs distribute(BLOCK,BLOCK)::g,g_old
  call initGrid(g)
  do i=1,niters
    g_old = g
    UPDATE_DATA(g)
    call computeLocal(g,g_old)
    error = computeNorm(g,g_old)
    REDUCE(error,maxim)
    Print *, Max norm: , error
  enddo
end subroutine
```

the expression \texttt{DOMAIN2D d} declares the two-dimensional domain variable \texttt{d}. We use the attribute \texttt{GRID(d)} to declare array variables associated to the domain \texttt{d}. In the previous example \texttt{double precision,GRID(d)::g,g_old} declares two-dimensional arrays of real numbers, which are dynamically created with the size of the domain region.

The actual data distribution is achieved by means of the corresponding HPF directive. The task body of the example consists of a loop where besides local computations, communication and synchronization aspects are carried out by means of two primitives: \texttt{UPDATE_DATA(g)} and \texttt{REDUCE}. The data belonging to one task that are needed by another (as defined in the corresponding coordination pattern) are interchanged by means of the instruction \texttt{UPDATE_DATA(g)}.
where \( g \) is a variable with \texttt{GRID} attribute. This instruction actually calls sequentially \texttt{PUT\_DATA}(g) and \texttt{GET\_DATA}(g), two other primitives provided by the model in order to send and receive data, respectively. The use of these two instructions separately can take advantage of both the implementation of complex communication patterns and the overlapping of communication and computation, which is the key for the efficiency of some applications. \texttt{PUT\_DATA}(g), \texttt{GET\_DATA}(g) and \texttt{UPDATE\_DATA}(g) instructions may optionally have a second argument, an integer number that represents the kind of border that is desired to be "sent" and/or "received" (see the \texttt{MULTIBLOCK} pattern).

When the \texttt{MULTIBLOCK} pattern is used, it may be necessary to establish certain communication among tasks in order to determine whether the convergence criteria of a method have been reached. The instruction \texttt{REDUCE(vbles, ProcName)} is in charge of this aspect. \texttt{vbles} is a scalar variable of any type and \texttt{ProcName} is a subroutine name. This instruction produces a reduction of the scalar value used as first argument by means of the subroutine \texttt{ProcName}.

**Implementation templates** Besides this direct way of codifying the computational tasks, we provide a set of different implementation templates in order to ease the programmer task. A template is a code skeleton of an HPF task cooperating with other tasks according to a fixed interaction pattern. In order to obtain the actual implementation of the tasks, the templates must be instantiated by including programmer-provided code. Templates are parameterized by means of arguments to receive this code. In addition to Fortran predefined types, we have established two new argument types: \texttt{HPF\_DECL} and \texttt{HPF\_CODE} for variable declarations and HPF code, respectively. The system compiler is in charge of automatically carrying out this instantiation. The template argument types aid the compiler to detect possible errors in the programmer specification.

The next two sections describe the different templates we have initially established. A programmer can add new templates appropriate to the kinds of applications he/she is dealing with.

**Multiblock templates**

We have defined two implementation templates in order to generate HPF tasks for the \texttt{MULTIBLOCK} pattern: the \texttt{Elliptic} and the \texttt{Parabolic} templates. Both fit the problem of solving PDEs by means of finite difference methods using domain-decomposition, which has been used in order to evaluate our approach. Obviously, these templates are not the only ones that can be defined for this kind of problems and more complex templates can be established.

Figure 3 shows the DIF specification of the \texttt{MULTIBLOCK} pattern together with a task "called" from it that solves a parabolic equation. The programmer will fill the different necessary code sections of the task using HPF code. The figure also depicts the instantiation of the corresponding template. The template we show is organized as two nested loops. The outermost is used to evolve in the time variable, the innermost iterates until the convergence conditions among domains are achieved. Inside this convergence loop, the borders among tasks are updated by means of the \texttt{UPDATE\_DATA} instruction. The instantiation is carried out by the
DIP compiler. The compiler adds some necessary information to the programmer variable declarations and initializations before instantiating the template. From a domain definition, its distribution specified in the MULTIBLOCK pattern and the variables declared in the task specification, the compiler generates: a) The domain dimensionality; b) The dimensionality of the grid associated to the domain; c) The distribution of the grid; d) The dynamic allocation of the grid.

The rest of the programmer code sections are directly used for the template instantiation.

Note that if the same task is called from the MULTIBLOCK pattern but using different data distributions, the task specification will be the same and the compiler will generate different instances from the template. This way, the computational code reusability is improved. On the other hand, a template may be
independent of the problem dimensionality (1D, 2D, etc.). The templates for the
PIPE pattern take advantage of both characteristics as well.

The Elliptic template we have defined only requires a loop since, in this
case, no time variable is taken into account. In this way, the programmer does
not have to provide the termination, preconverge and post converge code
sections. The instantiation process is similar to that shown in Figure 3.

Pipeline templates

The PIPE pattern establishes a chain of data-flow stages, which consume and
produce an input and output data stream, respectively. Thus, the implementa-
tion template of a generic stage has to be organized as a loop that receives
elements from the input stream, executes some code and sends the resulting data
to the output stream.

We have considered two different cases to deal with the stream length. The
first one assumes that all the stages know the stream length by means of the num-
ber of iterations established by the programmer. In the second one, the stages
do not know it, so that a distributed termination protocol has been established
associating a termination mark with each element of the stream.

Figure 4 shows the implementation template where the first approach has
been considered. Note that we do not need special templates for the first and
last stages of the pipeline, since the compiler will deduce how many \texttt{PUT\_DATA}
and \texttt{GET\_DATA} instructions have to be introduced in the instantiation process
from the information described in the pattern and task specification.

When the stream length is not known by the stages, the task specification
does not need the \texttt{number\_of\_iterations} section. The loop is repeated until the
\texttt{end\_of\_stream} mark is received. However, with this approach, we need a special
template for the first stage of the pipeline, which is in charge of generating the
mark. A logical expression must be provided in the task specification in order
to control the loop of this first stage.

Finally, when the \texttt{REPLICATE} directive is used, the round-robin policy has
been considered. Thus, when the number of iterations is provided, the compiler
automatically decides how many iterations have to be done by each instance of
the replicated stage. On the other hand, when the stream length is unknown, the

![Fig. 4. The Stage template. The stream length is known](image)
compiler introduces special code for the stages before (emitter) and after (collector) the replicated stage. Thus, when the emitter sends the \texttt{end\_of\_stream} mark at the end of its execution, it must actually send as many marks as instances in the replicated stage. In the same way, the collector must receive the \texttt{end\_of\_stream} marks from all the instances before sending its own mark.

### 3 Implementation issues and results

In order to evaluate the performance of DIP, a prototype has been developed on a cluster of 4 DEC AlphaServer 4100 nodes interconnected by means of Memory Channel. Each node has 4 processors Alpha 22164 (300 MHz) sharing a 256 MB RAM memory. The operating system is Digital Unix V4.0D (Rev. 878).

The implementation is based on source-to-source transformations together with the necessary libraries and has been realized on top of the MPI communication layer and the public domain HPF compilation system ADAPTOR [5]. The DIP compiler translates DIP code into BCL code. As mentioned in section 1, BCL is a previous work and it has inspired different aspects of DIP.

If it is required, the DIP compiler will use the implementation templates to generate the BCL code. The BCL compiler translates the code into an SPMD program that takes advantage of the \texttt{task\_region} HPF 2.0 facility, so that the worker processes can be executed on different processor subsets. Communication among worker processes are achieved by means of calls to the BCL library (BCLIB), which is implemented on top of MPI. The resulting HPF program is compiled by the ADAPTOR compiler.

Several examples have been used to test the prototype and the obtained preliminary results have successfully proved the efficiency of the proposal. Table 1 shows the results obtained for a system of two non-linear reaction-diffusion equations. A detailed explanation of this problem and the employed numerical method can be found in [16]. The equations are solved for an irregular domain decomposed in three regular subdomains. Different grid sizes have been considered (for example, in the first row, the grid sizes of each subdomain are 64 $\times$ 64, 32 $\times$ 32 and 64 $\times$ 64, respectively). For both HPF and DIP implementations, 5, 9 and 16 processors have been considered. In the case of HPF, all the processors execute each domain. In the case of DIP, when 5 processors are used, 2 of them execute the first subdomain, 1 the second and 2 the last one; for 9 processors, the mapping is 4/1/4 and for 16 processors it is 7/2/7. Note that when 5 processors are used, the one executing the second subdomain is idle most of the time since its total number of grid points is 1/4 of the other two domains. However, DIP offers a better performance than HPF except when the problem size becomes larger. In the rest of tested cases, DIP is better than HPF.

Table 2 shows the execution time per input array for HPF and DIP implementations of the 2-D FFT application. Given an N $\times$ N array of complex values, a 2-D FFT entails performing N independent 1-D FFTs on the columns of the input array, followed by N independent 1-D FFTs on its rows. In order to in-
Table 1. Computational time (in hours) for the non-linear reaction-diffusion equations

<table>
<thead>
<tr>
<th>Grid Sizes</th>
<th>Sequential</th>
<th>HPF vs. DIP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>5 Processors</td>
</tr>
<tr>
<td>64/32/64</td>
<td>0.21</td>
<td>0.28/0.16</td>
</tr>
<tr>
<td>128/64/128</td>
<td>2.07</td>
<td>1.34/1.05</td>
</tr>
<tr>
<td>256/128/256</td>
<td>21.12</td>
<td>11.14/11.88</td>
</tr>
</tbody>
</table>

Table 2. Computational time (in milliseconds) and HPF/DIP ratio for the 2-D FFT problem

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Sequential</th>
<th>HPF vs. DIP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4 Processors</td>
</tr>
<tr>
<td>32 x 32</td>
<td>1.505</td>
<td>0.947/0.505</td>
</tr>
<tr>
<td>64 x 64</td>
<td>5.165</td>
<td>2.189/1.005</td>
</tr>
<tr>
<td>128 x 128</td>
<td>20.536</td>
<td>7.738/7.010</td>
</tr>
</tbody>
</table>

crease the solution performance and scalability, a pipeline solution scheme is preferred as proved in [10] and [7].

Results are given for different problem sizes. Again, the performance of DIP is generally better. However, HPF performance is near DIP as the problem size becomes larger and the number of processors decreases, as it also happens in other approaches [10]. In this situation HPF performance is quite good and so, the integration of task parallelism does not contribute so much.

The third example has been taken from the NAS Parallel Benchmark [13]. It is the Fourier Transform (FT) to solve a 3-D diffusion equation. As in the previous example, a pipeline scheme is preferred [1] in order to improve the solution performance by means of the overlapping of communication and computation. In addition, our solution takes advantage of another level of parallelism as the time step iterations are independent since the result of one iteration is not used for the next one. We achieve this by means of the REPlicate directive. Tables 3 and 4 show the results considering different number of instances in the replicated stage (R) and data sizes. Since each stage is executed on at least one processor, the version with R=4 can not be executed on 4 processors as there are two stages per replica. We can observe in table 3 that for small data sizes, DIP is better than HPF. As in the previous examples, the better results are generally for the greater number of processors. On the other hand, as the number of instances grows, the performance improves except in the case where the replication avoids the integration of task and data parallelism (e.g. 8 processors and R=4). As the data size grows (table 4), HPF performance is better that DIP in the cases where few processors are used or only task parallelism is carried out.
322 M. Diaz, B. Rubio, E. Soler, J. M. Troya

Table 3. Computational time (in milliseconds) for the NPB-FT problem

<table>
<thead>
<tr>
<th>Processors</th>
<th>HPF</th>
<th>DIP R=1</th>
<th>DIP R=2</th>
<th>DIP R=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>37.5</td>
<td>35.4</td>
<td>35.3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>33.4</td>
<td>25.8</td>
<td>20.7</td>
<td>22.6</td>
</tr>
<tr>
<td>16</td>
<td>43.5</td>
<td>37.3</td>
<td>19.8</td>
<td>15.0</td>
</tr>
</tbody>
</table>

Table 4. Computational time (in milliseconds) for the NPB-FT problem

<table>
<thead>
<tr>
<th>Processors</th>
<th>HPF</th>
<th>DIP R=1</th>
<th>DIP R=2</th>
<th>DIP R=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>330</td>
<td>388</td>
<td>480</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>230</td>
<td>207</td>
<td>211</td>
<td>278</td>
</tr>
<tr>
<td>16</td>
<td>233</td>
<td>118</td>
<td>113</td>
<td>148</td>
</tr>
</tbody>
</table>

4 Conclusions

A new pattern-based approach to integrate task and data parallelism has been proposed. The main advantage of this approach is to supply programmers with a concise, pattern-based, high level declarative way to describe the interaction among HPF tasks. The use of domains and the establishment of data and processor layouts at the coordination level allow pattern reusability and efficient implementations, respectively. Patterns allow a high level description of an application structure and the programmer can modify it by means of simple changes in the patterns. The approach also provides the programmer with implementation templates, so that a higher level of abstraction to manage communication and synchronization aspects and computational code reusability can be achieved. By means of some examples we have shown the expressiveness and suitability of the model. The evaluation of a prototype has also shown the efficiency of the approach.

References

Reasoning about Probabilistic and Nondeterministic Processes *

Diego Cazorla, Fernando Cuartero, Valentín Valero, Fernando L. Pelayo

Escuela Politécnica Superior de Albacete
Universidad de Castilla-La Mancha
02071 Albacete, Spain
{dcazorla, fernando, valentin, fpelayo}@info-ab.uclm.es

Abstract. In this paper we present an algebraic language for the specification of probabilistic and nondeterministic processes, PNAL, which is a probabilistic extension of finite EPL (EPLf) that maintains nondeterminism.

We have defined both an operational and a testing semantics for PNAL, as well as a denotational semantics and a proof system. We show that the equivalence between nondeterministic processes in EPLf is preserved, and we also prove that the testing semantics, the denotational semantics and the proof system are fully abstract.

1 Introduction

The fundamentals of process algebras were established in the eighties but only qualitative aspects of concurrent systems were considered in these pioneering works. Subsequently quantitative aspects have been proposed, and several extensions to classical process algebra models have been developed including time, priorities and probabilities.

Several models for probabilistic processes have been presented in [18, 7, 12, 3, 4, 15]. Nevertheless, the main idea in all these models consists of substituting a nondeterministic behaviour by a probabilistic one in which the different types of behaviours are quantified with probabilities. A precise knowledge of the probabilities associated with each behaviour is therefore required to define a process. However, in practice, this probability distribution could be completely or partially unknown.

It is also necessary therefore to provide operators that allow us to deal with nondeterminism, but, at the same time, taking into account the probabilistic aspects. In the process of modelling a concurrent system we sometimes find nondeterministic behaviours that can be quantified (for instance, the probability of a faulty communication channel failing), whereas on other occasions these nondeterministic behaviours cannot be quantified (for instance, relative speed between

* This work has been partially supported by the CICYT project “Performance Evaluation of Distributed Systems”, TIC2000-0701-c02-02.
two processes). This is why we need a language that allows us to model nondeterministic systems with or without quantification, depending on the nature of the system in question.

In line with these ideas, in [21] a probabilistic extension of CCS is presented. The language adds a probabilistic choice operator \( (\oplus_p) \) indexed with a probability \( p \in (0, 1) \), which models the probabilistic internal choice, whereas the original CCS choice (+) is maintained for representing nondeterminism. A testing semantics, based on De Nicola and Hennessy testing, is presented. This work is taken further in [10, 11], where these preorders are redefined and denotations characterization for may-testing and must-testing are presented.

In [13] a probabilistic extension of CSP (P CSP) that maintains nondeterminism is introduced. The model is provided with an operational semantics, and several denotational semantics are given, based on different kinds of observations (trace, broom and barb semantics), where semantic objects are trees with three kinds of nodes: probabilistic, nondeterministic and action nodes.

Another probabilistic extension of CSP is presented in [14], where a denotational semantics of CSP is defined by applying the probabilistic powerdomain construction of Jones and Plotkin [9] over a directed-complete partial order. Probabilistic processes are considered to be probability distributions over processes of CSP.

Based on ACP, we can cite [1]. In that paper, a probabilistic version of ACP is presented leading to a language that combines probability and nondeterminism. An operational semantics is defined based on the alternating model [7]. In the construction of the term models they use a probabilistic bisimulation showing soundness and completeness of the term model with respect to the proposed axiom systems.

Different approaches, in the field of probabilistic automaton and Markov decision processes, are considered in [17, 16, 19] where the separation between nondeterministic and probabilistic behaviour is achieved by means of adversaries, schedulers or policies, which resolve the nondeterminism.

The specific aim of this paper is to add probabilistic choice to Hennessy's EPL_t [8], to show how the semantics of that can be defined, and to investigate the resulting process algebra.

The paper is structured as follows. Section 2 introduces our language. In Sections 3 and 4 an operational semantics and a testing semantics are presented. In Section 5 a denotational semantics is given by means of finite probabilistic acceptance trees and we prove that the denotational semantics is fully abstract with respect to the testing semantics. In section 6 a sound and complete proof system is presented. Finally, in section 7 we evaluate the work and compare it with other approaches.

2 PNAL language

The syntax of the language we present, PNAL, is based on EPL_t (EPL without recursion) [8] and provides three kinds of choice: classical internal choice, denoted
by $\oplus$, classical external choice, denoted by $+$, and a probabilistic choice labelled by a probability $p \otimes_p$.

Let $\Delta$ be a set of input actions, $\overline{\Delta} = \{ a | a \in \Delta \}$ their corresponding output actions, and $\mathcal{Act} = \Delta \cup \overline{\Delta}$. Furthermore we assume a set of process variables $Id$ ranged over by $X$ and $Y$. Terms of PNAL are defined by:

$$P ::= 0 \mid X \mid a.P \mid P \oplus P \mid P + P \mid P \otimes_p P \mid P\mid P \mid P \setminus A$$

where $a \in \mathcal{Act}$, $p \in (0, 1)$, $X \in Id$, and $A \subseteq \Delta$.

The classical operators are interpreted in the usual fashion and the probabilistic choice has a generative interpretation, i.e., $P \otimes_p Q$ behaves like $P$ with probability $p$, and like $Q$ with probability $1 - p$, this decision being made internally.

In order to deal with probabilities and nondeterminism we have to decide how to solve a situation where both choices appear. In the literature available we have, basically, two alternatives: in [1, 14, 21] probabilistic choices are solved first, whereas in [13] the opposite approach is taken. Note that this decision is not at all meaningless, because, depending on the approach adopted, some properties of classical process algebras will be preserved or not. For example, idempotency of internal choice is only maintained when internal choice is resolved first. Idempotency of external choice is never maintained, because if we take $P = a \otimes_{0.5} b$, we find that $P \parallel P$ increments the probability of $a$, for any model we consider.

In all these models we can identify two kinds of nondeterminism: nondeterminism introduced by internal choice (syntactic nondeterminism), and nondeterminism introduced by external choice and parallel composition between processes prefixed by the same action. Let us consider the following example (using PNAL syntax):

$$(a.P \mid (a.Q \otimes_p b)) \oplus R$$

If we consider Lowe’s approach [13], we first resolve internal choice and later (considering the left hand side process) probabilistic choice; at this point we find a nondeterministic behaviour introduced by the process $a.P[a.Q$, and the resolution of this nondeterminism is delayed after the execution of action $a$. But, in Lowe’s words (sic) “it is not enough to resolve all nondeterminism in a given state before the probabilistic choices; we have to resolve all nondeterminism before any probabilistic choice ... can we find a model that is (faithful to our intuition)? Such a model would need to ensure that all nondeterministic branching occurs before any other branching, but it is far from obvious how to write down the semantic equations so as to achieve this”. This is still, indeed, an open problem.

If we consider the approach taken by Morgan et al. [14], probabilistic choice is resolved first, and later on both kinds of nondeterminism can be resolved. Moreover, in that model we find the opposite behaviour, i.e., every probabilistic choice must be resolved before any internal choice. But, in the model in question, probabilistic choice is distributed through all the other operators, so this aim can be achieved easily.
Taking into account both models [13, 14], and the study of asynchronous probabilistic parallel composition introduced in [5] (where “bundle probabilistic transition systems” are introduced), we consider that the more suitable approach is to resolve probabilistic choice first, and then resolve any kind of nondeterminism later on. As we will see in the following sections, this decision leads us to a model that shares some characteristics with the Morgan et al. [14] model (e.g., distribution properties of probabilistic choice).

3 Operational semantics

The operational semantics is defined in the usual way, by using a labelled transition system. In order to understand these rules we require a preliminary definition:

Definition 1 (Probabilistic stability)
We define in an inductive way the probabilistic stability predicate

i) $O$ and $a.P$ are probabilistically stable.

ii) $P \oplus Q, P + Q, P | Q$ are probabilistically stable iff both $P$ and $Q$ are.

iii) $P \setminus A$ is probabilistically stable iff $P$ is.

We use $P \downarrow$ to denote that $P$ is probabilistically stable.

Transition rules are divided into 3 tables. Table 1 defines the probabilistic choice operator behaviour, by using probabilistic transitions, like $P \rightarrow_p Q$, which means that $P$ may evolve to $Q$ with probability $p$ immediately. Rules P1a and P1b indicate that process $P \oplus Q$ behaves like $P$ with probability $p$ and like $Q$ with probability $1 - p$. The remaining rules establish the precedence of the probabilistic choice with respect to the other operators. Processes $P$ and $Q$ are supposed to be probabilistically independent processes in rules P2c, P3c, and P4c, like in [1, 7, 21].

The behaviour of the remaining operators is presented in Tables 2 and 3, where we assume that processes are now probabilistically stable. The second table defines the rules for unlabelled transitions, i.e., those representing an internal evolution, $P \rightarrow Q$, with the usual meaning that $P$ may evolve internally to $Q$ without performing any visible action. Rules IIa and IIb show how the internal choice operator behaves. Rules I2a, I2b, I3a and I3b indicate that internal choice distributes over the external choice and parallel composition. Rule I3c indicates that an unbarred action may synchronize with its homologous barred action and the obtained process evolves internally. Finally, rule I4 shows the operational behaviour for the restriction operator.

The third table defines the inference rules for transitions labelled with actions in $\text{Act}$, $P \xrightarrow{a} Q$, with the usual meaning that $P$ can perform $a$ and move to $Q$.

Definition 2 The operational semantics of PNAL is defined as the multiset of transitions we can derive by using the rules in Tables 1–3.

Throughout the paper we will use $\rightarrow$ instead of $\rightarrow_p$, $\rightarrow$ and $\xrightarrow{a}$ in order to represent any of these transitions.
4 Testing semantics

A test is just a process which additionally may perform a special action $w$ to report a successful state. A purely probabilistic process $P$ will pass a test $T$ with a certain probability $p$. But, when dealing with nondeterministic processes, i.e., including internal choices, we will have to compute a set of probabilities to pass each test, in order to capture all the possibilities to do that.

As usual, in order to relate two processes using the testing semantics, we take one of them as a specification and the other one as its implementation. We will say that a process is a good implementation of a specification if the range of probabilities with which the implementation passes every test is included in the range of probabilities with which the specification passes the same test.
We use $P \parallel T$ to stand for $(P(T)) \setminus \text{Act.}$ $P \parallel T$ is a process that can only perform probabilistic transitions $\rightarrow_p$, unlabelled transitions $\rightarrow$ or $w$-transitions $\rightarrow_w$. These processes are called test processes and will be represented by $S$, $S'$, etc.

Given a process $P$ and a test $T$, $\phi_{P,T}$ will represent the set of probabilities for accepting $T$ considering each possible evolution of $P \parallel T$. In order to define $\phi_{P,T}$ we need some preliminary definitions, following similar ideas to those presented in [21].

**Definition 3** A *resolution* $R$ of $P \parallel T$ is a maximal subtree of $P \parallel T$ such that, whenever $S \ni S_1$, $S \ni S_2$, we have $S_1 = S_2$, where $\ni$ represents either $\rightarrow_p$, $\rightarrow$ or $\rightarrow_w$.

Therefore, every nondeterministic node in a resolution has a unique output arc.

**Definition 4** A *computation* $C$ of a test process $S$ is a maximal sequence of transitions, $S \ni S_1 \ldots S_{n-1} \ni S_n \ldots$ where $\ni$ represents either $\rightarrow_p$, $\rightarrow$ or $\rightarrow_w$.

We will use the following notation: $\mathcal{R}(S)$ denotes the set of resolutions of $S$, $\mathcal{C}(R)$ denotes the set of computations of resolution $R$. $Pr(C)$ denotes the probability to reach a successful state along the computation $C$, and $Pr(R)$ denotes the probability to reach a successful state in the resolution $R$.

**Definition 5** A computation $C = S \ni S_1 \cdots S_{n-1} \ni S_n \cdots S_n$ is a successful computation if $S_n \rightarrow_w$ and $S_k \not\rightarrow_w$ for all $k < n$. We use $\hat{\mathcal{C}}(R)$ to denote the set of successful computations of $R$. We may compute the probability for $C$ to be a successful computation as follows:

\[
\begin{align*}
Pr(S_n \rightarrow_w S_{n+1}) &= 1 \text{ (success)} \\
Pr(S_i \rightarrow C) &= Pr(C) \\
Pr(S_i \rightarrow_p C) &= p \cdot Pr(C)
\end{align*}
\]

And the probability to get a success in a resolution: $Pr(R) = \sum_{C \in \hat{\mathcal{C}}(R)} Pr(C)$

**Definition 6** Given a process $P$ and a test $T$, we say that $P$ may pass $T$ with the set of probabilities $\phi_{P,T} = \{ Pr(R) \mid R \in \mathcal{R}(P \parallel T) \}$.

Now we may introduce the test preorder over processes, by using $\phi_{P,T}$.

**Definition 7** We define the test preorder as follows:

\[
P \sqsubseteq_{\text{test}} Q \text{ if } \forall T \text{ s.t. } \phi_{P,T} \neq \{0\}, \text{ then } \phi_{Q,T} \neq \{0\}, \text{ and } \sup \{ \phi_{P,T} \} \geq \sup \{ \phi_{Q,T} \}, \text{ and } \inf \{ \phi_{P,T} \} \leq \inf \{ \phi_{Q,T} \}
\]

This preorder impose that whenever $Q$ cannot accept a test $T$, $P$ cannot accept it either. This has been introduced because, according to the definition of test preorder given in [10], the processes $P = a \oplus c$ and $Q = a$ are related under the
Reasoning about Probabilistic and Nondeterministic Processes

331

test preorder, i.e., $P \subseteq_{\text{test}} Q$. On the contrary in Hennessy's testing $P \may c.w$ but $Q \not\may c.w$ so $P \not\subseteq_{\text{test}} Q$. According to our definition both situations are captured.

The test relation has the following interpretation: $P \subseteq_{\text{test}} Q$ if $Q$ is a good implementation of the specification $P$. The corresponding equivalence is defined as follows:

**Definition 8** $P \equiv_{\text{test}} Q$ if and only if $P \subseteq_{\text{test}} Q$ and $Q \subseteq_{\text{test}} P$

5 Denotational semantics

The aim of this section is to provide our language with a fully abstract denotational semantics with respect to the testing semantics. Our domain will be a set of finite probabilistic acceptance trees, $\text{fPAT}$. Let us informally describe the structure of these trees:

- We will consider trees with two kinds of nodes: probabilistic ($\otimes$), and nondeterministic ($\bullet$).
- The root node is a probabilistic one and probabilistic and nondeterministic nodes appear in an alternative way.
- Nondeterministic nodes are labelled with an acceptance set (as defined in [8]), while probabilistic nodes are unlabelled.
- Branches leaving a probabilistic node are labelled with probabilities $p_i$, with $\sum_i p_i = 1$. Every branch leaving a nondeterministic node is labelled with one action in the acceptance set, but there are not two branches labelled with the same action.
- Terminal nodes are always nondeterministic, and are labelled with the acceptance set $\{\emptyset\}$.

We can see that these trees are quite similar to those used in [8], but now we have to extend these trees in order to include the probabilistic information associated with every reachable state. Let us introduce now acceptance sets and probabilistic acceptance sets.

**Definition 9** An acceptance set $\mathcal{A}$ is a subset of $\mathcal{P}(\text{Act})$. Let $\mathcal{D}(\mathcal{A})$ be the domain of acceptance sets for a given set of actions $\text{Act}$. We say that $(p, \mathcal{A})$ is a "probability-acceptance set pair" if $(p, \mathcal{A}) \in [0,1] \times \mathcal{D}(\mathcal{A})$.

Let $\mathcal{A}$ be an acceptance set. We define $\mathcal{A}(\mathcal{A})$, the set of actions belonging to $\mathcal{A}$, as follows:

$$\mathcal{A}(\mathcal{A}) = \{a \in \text{Act} | \exists B \in \mathcal{A} \text{ s.t. } a \in B\}$$

**Definition 10** A probabilistic acceptance set, $\mathcal{P}\mathcal{A}$, is defined as follows:

$$\mathcal{P}\mathcal{A} = \{(p, \mathcal{A}_i) \in [0,1] \times \mathcal{D}(\mathcal{A}) | \sum_i p_i = 1 \land \mathcal{A}_i \not\equiv \mathcal{A}_j \text{ if } i \not\equiv j\}$$

The domain of probabilistic acceptance sets is represented by $\mathcal{D}(\mathcal{P}\mathcal{A})$. 
Definition 11 We define the probability distribution, induced by a probabilistic acceptance set, as the function \( \pi : D(\mathcal{A}) \times D(\mathcal{PA}) \rightarrow [0,1] \) given by
\[
\pi(p, \mathcal{PA}) = \begin{cases} 
p & \text{if } (p, \mathcal{A}) \in \mathcal{PA} \\
0 & \text{otherwise} \end{cases}
\]

Definition 12 Let \( \mathcal{PA} \) be a probabilistic acceptance set. We define the set of acceptance sets belonging to \( \mathcal{PA} \) as follows:
\[
\text{Accept}(\mathcal{PA}) = \{ \mathcal{A} \in D(\mathcal{A}) \mid \exists p \text{ s.t. } (p, \mathcal{A}) \in \mathcal{PA} \}
\]

We will use the following notation:
\[
\overline{C} = \{ \overline{a} \mid a \in C \}
\]
and \( \mathcal{A} \setminus C = \{ B - (C \cup \overline{C}) \mid B \in \mathcal{A} \} \).

Besides the definitions above, we will need two additional operators borrowed from \[8\]: operators \( c \) and \( u \). Due to the lack of space, we will not give here their formal definitions. Instead, we will remember their informal meaning; \( c \) is the convex closure of a set, and \( u \) is the pointwise union of sets of sets.

We may now define (see Table 4) the probabilistic acceptance set of each process at the top level, \( \mathcal{PA}(P) \). Parallel composition will be studied separately because extra explanation is needed.

\[
\begin{align*}
\mathcal{PA}(0) &= \{(1, \{\emptyset\})\} \\
\mathcal{PA}(a.) &= \{(1, \{a\})\} \\
\mathcal{PA}(t_1 \otimes t_2) &= \{(p, \mathcal{A}) \mid \mathcal{A} \in \text{Accept}(\mathcal{PA}(t_1)) \lor \mathcal{A} \in \text{Accept}(\mathcal{PA}(t_2)), \\
p &= q \cdot \pi(\mathcal{A}, \mathcal{PA}(t_1)) + (1 - q) \cdot \pi(\mathcal{A}, \mathcal{PA}(t_2)) \} \\
\mathcal{PA}(t_1 \oplus t_2) &= \{(p, \mathcal{A}) \mid \exists B \in \text{Accept}(\mathcal{PA}(t_1)), \exists C \in \text{Accept}(\mathcal{PA}(t_2)) \text{ s.t. } \mathcal{A} = c(B \cup C), \\
p &= \sum_{\mathcal{A} = c(B \cup C)} \pi(B, \mathcal{PA}(t_1)) \cdot \pi(C, \mathcal{PA}(t_2)) \} \\
\mathcal{PA}(t_1 + t_2) &= \{(p, \mathcal{A}) \mid \exists B \in \text{Accept}(\mathcal{PA}(t_1)), \exists C \in \text{Accept}(\mathcal{PA}(t_2)) \text{ s.t. } \mathcal{A} = B \cup C, \\
p &= \sum_{\mathcal{A} = B \cup C} \pi(B, \mathcal{PA}(t_1)) \cdot \pi(C, \mathcal{PA}(t_2)) \} \\
\mathcal{PA}(t_1 \setminus C) &= \{(p, \mathcal{A}) \mid \mathcal{A} = B \setminus C \land p = \sum_{\mathcal{A} = B \setminus C} \pi(t, B) \}
\end{align*}
\]

Table 4. Probabilistic acceptance set of a semantic process

The probabilistic acceptance set defined allows us to distinguish between the different reachable states, and also provides us with the probability associated with every state.

Let us now introduce the semantic process \( t/\langle \mathcal{A}, a \rangle \), which models the behaviour of \( t \), once it has executed the action \( a \) at the state given by \( \mathcal{A} \). We define \( t/\langle \mathcal{A}, a \rangle \) as follows:
- If $t$ is a process such that either $a \not\in A(A)$ or $A \not\in \text{Accept}(\mathcal{PA}(t))$ then $t/(A,a)$ is not defined.
- If $t = a.t_1$, the only reachable state is $\{\{a\}\}$ with probability 1. Then $t/(\{\{a\}\},a) = t_1$.
- $t = t_1 \oplus t_2$. We must distinguish several subcases:
  - $A \in \text{Accept}(\mathcal{PA}(t_1))$ and $A \not\in \text{Accept}(\mathcal{PA}(t_2))$. In this case only $t_1$ can execute the action $a$, so $t/(A,a) = t_1/(A,a)$.
  - $A \not\in \text{Accept}(\mathcal{PA}(t_1))$ and $A \in \text{Accept}(\mathcal{PA}(t_2))$. It is the symmetric case: $t/(A,a) = t_2/(A,a)$.
  - $A \not\in \text{Accept}(\mathcal{PA}(t_1))$ and $A \not\in \text{Accept}(\mathcal{PA}(t_2))$. Now $a$ can be made either by $t_1$ or $t_2$, then: $t/(A,a) = t_1/(A,a) \oplus t_2/(A,a)$.

Therefore, as $A$ may be obtained from several different sets $\mathcal{B}$ and $\mathcal{C}$, we have:

$$t/(A,a) = \bigoplus_{a \in A(\mathcal{B}) \land a \not\in A(\mathcal{C})} t_1/(B,a) \oplus \bigoplus_{a \not\in A(\mathcal{B}) \land a \in A(\mathcal{C})} t_2/(C,a) \oplus \bigoplus_{a \in A(\mathcal{B}) \land a \in A(\mathcal{C})} t_1/(B,a) \oplus t_2/(C,a)$$

- $t = t_1 + t_2$. We have that every state $A$ is built as $A = \mathcal{B} \cup \mathcal{C}$, where $\mathcal{B} \in \text{Accept}(\mathcal{PA}(t_1))$ and $\mathcal{C} \in \text{Accept}(\mathcal{PA}(t_2))$, and we can repeat the same case distinction as in the internal choice, thus obtaining exactly the same expression for $t/(A,a)$.
- $t \setminus \mathcal{C}$. If $a \in \mathcal{C}$ then $(t \setminus \mathcal{C})/(A,a)$ is not defined. If $a \not\in \mathcal{C}$ then:

$$t/(\mathcal{C},A) = \bigotimes_{A \in B(\mathcal{C})} \frac{\pi(t,B)}{\pi(t,\mathcal{C},A)} (t/(B,a)) \setminus \mathcal{C}$$

Once we have defined the probabilistic acceptance sets of a process, not only for its first action, but all over its possible executions, we are interested in all the possible sequences of actions it can perform. Let us now introduce some notation for that. As each probabilistic node in the tree representing $t$ can be reached by following a sequence of pairs (acceptance set, action), due to the alternating nature of these trees, we may identify these nodes by $t(s)$, where
\[ s = (A_1, a_1, \ldots, A_k, a_k) \] is the sequence leading us to the node. We can use a similar notation for syntactical terms. Then, \((P/(\mathcal{A}, a))/(\mathcal{B}, b)\) will be represented shortly by \(P/(\mathcal{A}, a, \mathcal{B}, b)\).

**Parallel composition.** The definition of the function over \(\text{fPAT}\) corresponding to \(\parallel\) is quite difficult, and a considerable amount of notation is needed. We will introduce intuitively the concepts related to this operator. Its formal definition is included in the extended version of this paper [2], available via world wide web.

Let us consider the processes \(P_1 = a,(c \oplus_0 d)e \oplus_0 b, P_2 = \pi d\) and \(P = P_1 | P_2\). In figure 1 we can see the \(\text{fPAT}\) of \(P_1, P_2\) and the first level of \(P_1 | P_2\).

![Fig. 1. \(\text{fPAT}\) of \(P_1, P_2\) and first level of \(P_1 | P_2\)](image)

In the first level, \(P\) may perform the actions that \(P_1\) or \(P_2\) may perform at their first level. Besides, in case of communication between \(P_1\) and \(P_2\), some actions of the following level in \(P_1\) and \(P_2\) are promoted to the first one in \(P\). We have to consider all the possibilities, i.e., every action may or may not communicate with its homologous barred action, and this introduced some kind of nondeterminism.

In this example, actions \(a\) and \(\overline{a}\) may communicate, so second level action, like \(c\) and \(d\) may be performed at the first level. Let us consider that \(a\) and \(\overline{a}\) communicate, and the probabilistic choice after action \(a\) in \(P_1\) is resolved following the branch labelled with probability 0.4. In this case, with probability \(0.3 \times 0.4 = 0.12\), the semantic process \(P\) reaches a nondeterministic acceptance set \(\mathcal{A}\) such that \(\mathcal{A}(\mathcal{A}) = \{a, \overline{a}, c, d\}\). In case of communication, only actions \(c\) and \(d\) may be performed. Nondeterminism is introduced in \(\mathcal{A}\) considering the convex closure of the union of sets \(\{c, d\}\) and \(\{a, \overline{a}, c, d\}\).
Definition 13 We define the sequences of $P$, denoted by $\text{Seq}(P)$, as the set of all the sequences of pairs (acceptance set, action) that we may obtain from $[P]_{\text{PAT}}$.

Definition 14 We say that two processes are equivalent according to the denotational semantics, which will be denoted by $[P]_{\text{PAT}} = [Q]_{\text{PAT}}$, iff:

- $\text{Seq}(P) = \text{Seq}(Q)$, and
- For every sequence $s$, $\mathcal{PA}(P)(s) = \mathcal{PA}(Q)(s)$.

We now state that this equivalence relation between probabilistic processes is an extension of that presented in [8] for nondeterministic processes.

Theorem 1 Let $P, Q \in \text{EPL}_{\text{f}}$ be two processes.

$$[P]_{\text{PAT}} = [Q]_{\text{PAT}} \text{ if and only if } [P]_{\text{PAT}} = [Q]_{\text{PAT}}$$

Also, we have that both the testing semantics and the denotational semantics identify the same processes, i.e., the denotational semantics is fully abstract with respect to the testing semantics.

Theorem 2 Let $P, Q \in \text{PNAL}_{\text{f}}$ be two processes, then

$$[P]_{\text{PAT}} = [Q]_{\text{PAT}} \text{ if and only if } P \equiv_{\text{eal}} Q$$

6 Proof system

In this section we present a proof system, which is sound and complete with respect to the denotational semantics. This proof system is based on that of EPL$_{\text{f}}$, by including the axioms related with the probabilistic choice operator. Nevertheless, some of the axioms of EPL$_{\text{f}}$ are not preserved in the general case. For instance, the internal and the external choice now fail to be idempotent, and distributivity of the internal over the external choice (and vice versa) is not maintained. However, these axioms can be used when we are working with a special kind of processes (probabilistically stable processes) and they are very useful when proving completeness.

Table 5 contains the set of axioms related to the following operators: 0, prefix and the three choices. Axioms in Table 5 show that probabilistic choice is commutative (P1), associative in a probabilistic way (P2), and idempotent (P3). These axioms allow us to generalize probabilistic choice to an arbitrary number of arguments. Thus we can write $\bigotimes[i=1^n][p_i]P_i$ to denote a probabilistic choice among $n$ processes, each one with probability $p_i > 0$. Other rules say that probabilistic choice is propagated to the following level when both processes are prefixed by the same action (P4), and that the probabilistic choice distributes over the internal and the external choice (P11, P12, P13, P14).

Let us observe that the axioms maintained from EPL$_{\text{f}}$ are: the internal and the external choice are commutative and associative (I1, I2, E1, E2), the external
choice has a zero (E3), and, finally, the internal and the external choice between processes prefixed with the same action are propagated to the following level (B3,E1). The rest of axioms (those with the symbol $\downarrow$), do not hold in the general case, but if we restrict $P$, $Q$ and $R$ to a kind of processes, which we call probabilistically stable processes, they become true. These axioms are: idempotency of the internal and the external choice (I4, E4), and distributivity of the internal choice over the external choice and vice versa (IE3, IE4).

| P1 | $P \circ P \equiv P \circ Q \equiv Q \circ P$ | E1 | $P + Q \equiv P + P$ |
| P2 | $(P \circ Q) \circ Q \equiv P \circ Q \circ (Q \circ R)$ | E2 | $P + (Q + R) \equiv (P + Q) + R$ |
| P3 | $P \circ P \equiv P$ | E3 | $P + 0 \equiv P$ |
| P4 | $aP \circ aQ \equiv a(P \circ aQ)$ | IE1 | $aP + aQ \equiv aP \circ aQ$ |
| P11 | $P \equiv [Q \circ P \equiv (P \equiv Q) \circ P \equiv P \equiv Q \circ P]$ | IE4 | $P + (Q \equiv R) \equiv (P \equiv Q) \equiv (P \equiv R)$ |
| P12 | $P \equiv (Q \equiv R) \equiv (P \equiv Q) \equiv (P \equiv R)$ | IE4 | $P + (Q \equiv R) \equiv (P \equiv Q) + (P \equiv R)$ |
| P13 | $aP \equiv aQ \equiv a(P \equiv Q)$ | |

**Table 5.** Axioms for $\circ$, prefix and choice operators

We may also apply, over probabilistically stable processes, the derived equations given in [8], which are useful to prove completeness. Finally, in Table 6 we can see the axioms for restriction and parallel composition, which, as usual, are derived operators. In rule PaE1, the function $\operatorname{comm}(P, Q)$ (defined in [8]) is false if there is not any possibility of communication between $P$ and $Q$.

Note that axioms Pa1 and PaE1, which are introduced in [8], hold only for processes in $\operatorname{EPL}_f$. Applying repeatedly P4, P11, P12, P13, and PaP1 (distribution axioms) every process in the form $P|Q$ may be transformed into $\bigotimes_{i,j}[p_i \cdot q_j][P_i|Q_j]$, where $P_i, Q_j \in \operatorname{EPL}_f$.

| RC1 | $(P + Q)^C = P^C + Q^C$ | RC1 | $(P \oplus Q)^C = P^C \oplus Q^C$ |
| Re1 | $P \circ C = 0$ if $C \equiv 0$ otherwise | Re1 | $0 \circ C = 0$ |
| PaP1 | $P \circ Q = P^C \circ Q^C$ | PaP1 | $(P \oplus Q) = (P^C \oplus Q^C)$ |
| Pa01 | $0P = P$ | Pa01 | $0(P \oplus Q) = (P^C \oplus Q^C)$ |
| PaE1 | if $P|Q$ are $\sum_{i \in I} P_i$, $i \in I$, $\sum_{j \in J} Q_j$, $j \in J$ | then where $\operatorname{int}(P|Q) = \bigoplus \{P_i|Q_i, i \in I\}$ |
| P|Q | $\bigotimes_{i \in I} P_i \cdot \bigotimes_{j \in J} Q_j$ | otherwise $\operatorname{ext}(P|Q) = \bigoplus \{P_i|Q_i, i \in I\} + \bigoplus \{Q_j, j \in J\}$ |

**Table 6.** Axioms for restriction and parallel composition

**Theorem 3** Axioms are sound with respect to the denotational semantics.
As usual, to prove the completeness, we look for the adequate normal forms. Essentially, these normal forms represent the different ways a process has to complete its execution. This leads us to a generalized probabilistic choice at the top, followed by a generalized internal choice among a set of states, which is followed by a generalized prefixed external choice among the actions in this set, whose continuations are also in normal form.

**Definition 15 (Normal forms)**

- Process $O$ is in normal form.
- if $A_i$ is a convex set and for every $a \in A(A_i)$ there is a normal form $n(A_i; a)$, then

$$\bigotimes_i^n \bigoplus_{a \in A} a \cdot n(A_i; a)$$

is a normal form.

These normal forms share some characteristics of classical models (CSP, EPL), as well as some characteristics of other probabilistic models [4]. Thus, the continuations after performing the same action in different probabilistic branches do not need to be identical (usual in probabilistic models), and the acceptance sets $A_i$ must be convex as usual in classical models.

**Lemma 1** Every term in PNAL can be transformed, by using the given proof system, into another equivalent term in normal form.

**Theorem 4** Let $P, Q \in PNAL$ be two processes, then $[P]_{PAT} = [Q]_{PAT}$ if and only if $P \equiv Q$.

Finally, by using theorems 2 and 4, we can relate the proof system with the testing semantics as follows:

**Corollary 1** Let $P, Q \in PNAL$ be two processes, then $P \equiv_{test} Q$ if and only if $P \equiv Q$.

7 Conclusions and comparisons

In this paper we have presented an algebraic theory of probabilistic and nondeterministic processes which is a probabilistic extension of Hennessy’s theory presented in [8]. An operational and a testing semantics have been introduced, as well as a denotational semantics and a proof system. The testing semantics, the denotational semantics and the proof system are shown to be fully abstract.

To our knowledge, this paper is the first that presents a general framework of probabilistic and nondeterministic processes based on Hennessy’s model. Similar approaches have been presented, based on different classical process algebras. Thus, in [1] an operational semantics, a probabilistic bisimulation and a proof system are introduced for a probabilistic extension of ACP. In [14], a denotational
semantics and a proof system are introduced for a probabilistic extension of CSP. In [13], only an operational and a denotational semantics are defined for another probabilistic version of CSP which does not include a hiding operator. Finally, in [21, 10] a probabilistic extension of CCS is presented, and an operational semantics, a testing semantics and a denotational semantics are defined.

In order to deal with probabilism and nondeterminism we have chosen an approach that considers that a probabilistic choice is resolved before an internal choice. Thus, the transition system we obtain is very similar to “bundle probabilistic transition systems” (BPTS) presented in [3], which are the converse of the simple model of [17, 16]. In our case (as in BPTS), a set of nondeterministic alternatives is chosen with a certain probability, while in [17, 16], an action is chosen nondeterministically and then, for each action, a distribution of probabilities over the successor states is given (reactive probabilistic choice).

A similar operational semantics is defined in [21], but in this case some of the premises of the rules include negative conditions over transitions, which is problematic (see [6]). In our model we have avoided this problem by introducing the “probabilistic stable” predicate, which is defined syntactically.

The decision of resolving probabilistic choices first leads us to a model in which some of the properties of classical process algebras are not maintained. For instance, the internal choice now fails to be idempotent, as the following example shows. Let \( P \) be the process \( a.0 \oplus b.0 \), we then have:

\[
\mathcal{TA}(P) = \{(0.5, \{\{a\}\}), (0.5, \{\{b\}\})\} \\
\mathcal{TA}(P \oplus P) = \{(0.25, \{\{a\}\}), (0.25, \{\{b\}\}), (0.5, \{\{a\}, \{b\}, \{a, b\}\})\}
\]

This is, however, a known consequence of introducing probabilities and nondeterminism by taking the approach of first resolving the probabilistic choices. We encounter the same problem in some of the aforementioned papers [1, 14, 21], and also in [20], where a probabilistic extension of CCS is presented by introducing intervals of probabilities that allow us to specify processes with uncertain information.

The normal forms are defined in a very natural way. They consist in a generalized probabilistic choice at the top, followed by a generalized internal choice between a set of states, which is followed by a generalized prefixed external choice between the actions in this set, whose continuations are also in normal form. But, as we could see in Section 6, distribution axioms allow us to define an alternative normal form that consists in a probabilistic choice followed by a non-probabilistic process in EPLf’s normal form, i.e., each probabilistic choice must be resolved before any internal choice. The same conclusion is obtained in [14] for a probabilistic extension of CSP.

Finally, the proof system we have obtained is very similar to Andova’s proof system [1]. One noteworthy feature is that, in her case, the proof system is sound and complete with respect to probabilistic bisimulation, while in our case it is sound and complete with respect to the testing semantics.
References
