

Actas

IV Jornadas sobre Programación y Lenguajes

Málaga, 11 al 12 de Noviembre de 2004

© Salvador Lucas
Ernesto Pimentel
María del Mar Gallardo

D.L.:
Imprime:

Prólogo

Este volumen recoge los trabajos aceptados para su presentación en las Cuartas Jornadas sobre Programación y Lenguajes (PROLE 2004) celebradas en Málaga los días 11 y 12 de Noviembre de 2004.

Estas cuartas jornadas continúan la serie que se inició en Almagro (Ciudad Real) en 2001, a la que le sucedieron las celebradas en El Escorial (Madrid) en 2002 y en Alicante en 2003. Las jornadas surgen como un lugar de reunión de los grupos españoles que trabajan en temas ligados a la programación y a los lenguajes de programación. Con la organización de esta serie de eventos, se pretende que los grupos españoles que trabajan en este ámbito puedan conocerse mejor, intercambiar experiencias y resultados de investigación y cooperar en el futuro. Desde su nacimiento, PROLE ha ido consolidándose cada año, con el objetivo de ser un referente nacional de los trabajos de calidad desarrollados en el área.

En el ámbito de PROLE son bienvenidos tanto los trabajos teóricos como los que desarrollan aspectos prácticos relativos a la especificación, diseño, implementación, análisis y verificación de programas y lenguajes de programación. Con más precisión, una lista no exhaustiva de los temas abordados en PROLE incluiría:

- Paradigmas de programación (concurrente, funcional, imperativo, lógico, visual, orientado a agentes, aspectos, objetos, ...) y su integración
- Especificación y lenguajes de especificación
- Sistemas de tipos
- Lenguajes y técnicas para nuevos modelos de programación (computación ADN, cuántica, genética, molecular, ...)
- Compilación; técnicas y herramientas de implementación de los lenguajes de programación
- Semántica y su aplicación al diseño, análisis, verificación, transformación o aprendizaje de programas
- Técnicas de análisis de programas
- Transformación y optimización de programas
- Estructuras de soporte al desarrollo e interconexión de programas (modularidad, programación genérica, lenguajes de marcas, WWW ...)

En esta ocasión, con el fin de aglutinar trabajos de diversa índole y en distinto grado de desarrollo, se solicitaron contribuciones en 5 cate-

gorías (trabajos originales, tutoriales, demostraciones de sistemas software, resúmenes de trabajos de alto nivel ya publicados y trabajos en progreso). De entre los trabajos recibidos, y después de un cuidadoso proceso de revisión, el comité de programa de PROLE 2004 ha seleccionado 22 para su presentación en las jornadas.

El programa científico de PROLE 2004 se ha completado con dos excelentes conferencias a cargo de investigadores de gran prestigio internacional: el Prof. Patrick Cousot, de la École Normale Supérieure de París, cuyos trabajos sobre la interpretación abstracta de programas han sido determinantes en los campos del análisis estático y la verificación; y el Prof. Zohar Manna, de la Universidad de Stanford, cuyas contribuciones en el ámbito de la verificación de sistemas reactivos son ampliamente reconocidas.

La celebración de un evento de las características de PROLE, con unas exigencias de calidad que se van incrementando en cada edición, no sería posible sin el trabajo desarrollado por los miembros del Comité de Programa y los revisores adicionales, que han contribuido a conformar un programa científico atractivo y de gran calidad.

Del mismo modo, las exigencias organizativas planteadas por la participación cada vez más numerosa y consolidada en las jornadas, no hubieran podido satisfacerse sin la dedicación y el entusiasmo de los miembros del Comité Organizador del Grupo de Ingeniería del Software de la Universidad de Málaga, a los que queremos agradecer el trabajo desarrollado en beneficio del éxito de las jornadas. También queremos agradecer el soporte recibido por las entidades patrocinadoras y colaboradoras.

Deseamos, finalmente, que la próxima edición de estas jornadas, que se celebrarán en Granada en el 2005 como parte del I Congreso Español De Informática (CEDI), mantenga la tendencia de estas últimas ediciones tanto en calidad científica como en participación.

Salvador Lucas
Presidente del Comité de Programa de PROLE 2004

María del Mar Gallardo
Ernesto Pimentel
Comité de organización de PROLE 2004

Comité de Programa de PROLE 2004

Presidente

Salvador Lucas U. Politécnica de Valencia

Miembros

María Alpuente	U. Politécnica de Valencia
Miquel Bertran	U. Ramon Llull
Francisco Bueno	U. Politécnica de Madrid
Javier Campos	U. de Zaragoza
Rafael Corchuelo	U. de Sevilla
Juan Manuel Cueva	U. de Oviedo
Francisco Durán	U. de Málaga
María del Mar Gallardo	U. de Málaga
Manuel Hermenegildo	U. Politécnica de Madrid
Pascual Julián	U. de Castilla-La Mancha
Francisco Javier López Fraguas	U. Complutense de Madrid
Paqui Lucio	U. del País Vasco
Narciso Martí Oliet	U. Complutense de Madrid
Carlos Martín Vide	U. Rovira i Virgili
José Meseguer	U. Illinois at Urbana-Champaign (EEUU)
Juan José Moreno Navarro	U. Politécnica de Madrid
Robert Nieuwenhuis	U. Politècnica de Catalunya
Fernando Orejas	U. Politècnica de Catalunya
Ricardo Peña	U. Complutense de Madrid
María José Ramírez	U. Politécnica de Valencia
Manuel Vilares	U. de Vigo

Revisores

Unai Arronategui	Jaime Lipton	Jaime Sánchez
Demis Ballis	Marisa Navarro	Clara Segura
Santiago Escobar	Joaquín Ezpeleta	Alicia Villanueva
Vicente Estruch	Luis Garmendia	

Comité de organización de PROLE 2004

Presidente

Ernesto Pimentel

U. de Málaga

Miembros

José F. Aldana

Pablo López

Enrique Alba

Francisco Luna

Mercedes Amor

Gabriel Luque

Carlos Canal

Jesús Martínez

Manuel Díaz

Pedro Merino

Almudena Díaz

Nathalie Moreno

Francisco Durán

Ismael Navas

Raúl Fernández-Santa Cruz

Antonio J. Nebro

Antonio J. Fernández

Mónica Pinto

Manuel Fernández Bertoa

Ana M. Roldán

Lidia Fuentes

Manuel Roldán

María del Mar Gallardo

María del Mar Roldán

Francisco Gutiérrez

José Raúl Romero

Daniel Jiménez

Antonio Vallecillo

Entidades Colaboradoras

Departamento de Lenguajes y Ciencias de la Computación

E.T.S.I. Informática

Universidad de Málaga

Federación de Asociaciones de Ingenieros de Informática

Parque Tecnológico de Andalucía

Ministerio de Educación y Ciencia

Unicaja

Microsoft

Intersystems

Procedimientos

Ingenia

Índice

Introducing constraints on sets of spatial objects <i>J. M. Almendros-Jiménez, A. Corral</i>	1
Una aproximación práctica al desarrollo de comprobadores de tipos basada en patrones de diseño <i>F. Ortín, L. Vinuesa, J. M. Cueva</i>	11
Agregando orden superior al lenguaje Java. Una nueva perspectiva para el diseño de métodos <i>J. Llberia Albarrán, S. I. Clérical Martínez</i>	27
A lightweight approach to program specialization <i>C. Ochoa, J. Silva, G. Vidal</i>	41
An input/output semantics for distributed program equivalence reasoning <i>M. Bertran, F. Babot, A. Climent</i>	55
Dealing denotationally with stream-based communication <i>M. Hidalgo-Herrero, Y. Ortega Mallén</i>	69
Analysing definitional trees <i>P. Julián Iranzo, C. Villamizar Lamus</i>	85
Una aproximación lógica a la verificación de propiedades de programas lógico-funcionales <i>J. M. Cleva, J. Leach, F. J. López-Fraguas</i>	89
Rewriting-engined Web Sites verification <i>M. Alpuente, D. Ballis, M. Falaschi</i>	93
A general natural rewriting strategy <i>S. Escobar, J. Meseguer, P. Thati</i>	97

A tutorial on specifying data structures in Maude <i>N. Martí-Oliet, M. Palomino, A. Verdejo</i>	101
Towards (constructor) normal forms for Maude within full Maude <i>F. Durán, S. Escobar, S. Lucas</i>	125
Invariant-based control of the execution of Maude specifications: The LTL case <i>F. Durán, M. Roldán</i>	137
A first-order functional language for reasoning about heap consumption <i>R. Peña, C. Segura</i>	153
Tagged systems: a framework for the specification of history-dependent properties <i>F. Rosa Velardo, D. de Frutos Escrig, C. Segura Díaz</i>	163
Proving modal properties of rewrite theories using Maude's metalevel <i>I. Pita, M. Palomino</i>	177
Un marco de trabajo para la construcción de herramientas de model checking <i>A. Carmona, M. M. Gallardo, J. Martínez, P. Merino</i>	191
Phrase similarity through approximate tree matching <i>F. J. Ribadas, M. A. Alonso, D. Cabrero</i>	201
Compiling thread-level parallel programs with a C-compiler <i>M. Forsell</i>	215
Elimination of local variables from definite logic programs <i>J. Álvarez, P. Lucio</i>	227
Unfolding-based improvements on fuzzy logic programs <i>P. Julián, G. Moreno, J. Penabad</i>	241
Integrated transformation strategies for integrated declarative programs <i>S. González, G. Moreno</i>	255

Introducing Constraints on Sets of Spatial Objects [★]

Jesús M. Almendros-Jiménez and Antonio Corral
email: {jalmen,acorral}@ual.es

Dpto. de Lenguajes y Computación. Universidad de Almería.

Abstract. In this paper we present a framework for handling of constraints on sets of spatial objects. For this spatial constraint system we formally define a set of spatial data types together with operations on them. We show how to formulate a spatial constraint satisfaction problem using these constraints, by means of several examples. We also study safety conditions in order to ensure that each spatial constraint satisfaction problem is equivalent to a solved form representing its set of solutions.

1 Introduction

The framework of constraint databases [5, 7] is based on the simple idea that a constraint can be seen as an extension of a relational tuple, called *generalized tuple*, or, vice versa, that a relational tuple can be interpreted as a conjunction of equality constraints. Each generalized tuple finitely represents a possibly infinite set of relational tuples, called *extension* of the generalized tuple, one for each solution of the constraint. Constraints are, for instance, a powerful mechanism for modeling *spatial* and *temporal* concepts, where often infinite information have to be represented. The basic idea is to see spatial objects as infinite sets of points and provide a finite representation of these infinite sets, with constraints over a dense domain.

As far as we know spatial constraint databases have been focused on *linear constraints* handling [9, 8, 3] in which spatial reasoning is based on *linear equations* and *inequations*. Most known spatial data types like point, line and region can be modeled with these frameworks, and distance-based operations can be included on it [2]. However, in our opinion, an interesting direction could be the study of other kinds of constraint systems, *specialized for any class of spatial data types*, and handling of *sets of spatial objects*.

In this paper, we present a new framework for handling of *constraints on sets of spatial objects*. For this spatial constraint system, we formally define a set of spatial data types together with operations on them. We show how to formulate an *spatial constraint satisfaction problem* using these constraints, by means of several examples. We also study *safety conditions* in order to ensure that each spatial constraint satisfaction problem is equivalent to a *solved form* representing its set of *solutions*.

[★] This work has been partially supported by the Spanish project of the Ministry of Science and Technology, “INDALOG” TIC2002-03968 under FEDER funds.

The structure of the paper is as follows. Section 2 presents an informal introduction to constraints on sets of spatial objects and presents a set of examples of constraint satisfaction problems using spatial data. Section 3 formally defines the set of spatial data types together with the set of operations defined on them. Section 4 formally defines a spatial constraint satisfaction problem and its solved form. Finally, section 5 concludes and presents future work.

2 Examples of Spatial Constraints

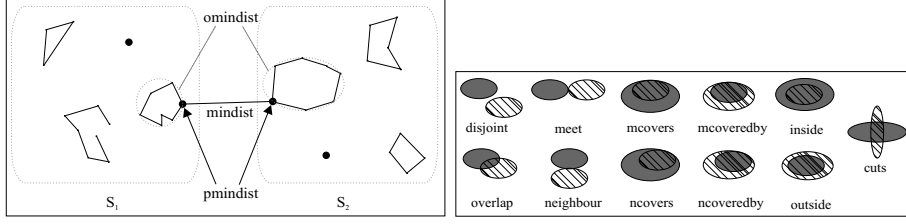
Basically, we handle sets of spatial objects where an spatial object can be a point, line, polygon or region. Points will be represented by means of real coordinates: $(1, 2)$ and $(x, 1)$; lines by means of an ordered sequence of points: $\text{line}[(1, 2), (4, 5), (9, 10)]$; and the same can be said for polygons and regions, where the last point is connected to the first one. Line, polygon and region are intended not contain more vertices.

With this representation we can describe sets of *closed points-based geometric figures*: $\{(x, 1), \text{line}[(1, 2), (4, 5), (9, 10)]\}$. On these spatial data types, a set of binary spatial operations is defined. We can classify them in five groups: equality and membership relations, set operations, metric operations, topological relations, and finally, structural operations. We are not interested in a complete set of operations, but enough for most usual spatial queries.

With respect to equality relations, spatial objects can be compared, for instance $(1, 2) = (2, 3)$, $(1, x) = (1, 9)$, $\text{line}[(1, 2), (10, y)] = \text{line}[(z, 2), (10, 12)]$. The first equality relation is false, but the other two equality relations define a relation on x , y and z , which can be seen as a constraint, that is, a requirement that states which combinations of values from the variable domains are admitted. In this case, x equal to 9, y equal to 12 and z equal to 1. This can be expressed, at the same time, as a constraint of the form $x = 9, y = 12, z = 1$.

With respect to the membership relations, we can express membership of points to a spatial object, for instance $(1, 6) \in \text{line}[(1, 2), (1, 9)]$, which is trivially true, but we can also add variables. However, we can also use the membership relation for expressing that a spatial object belongs to a set of spatial objects. For instance, $A \in \{\text{line}[(1, 2), (1, 9)], (9, 10)\}$. In this case, A ranges on two spatial objects, $\text{line}[(1, 2), (1, 9)]$ and $(9, 10)$. Combining membership relations we can build our first CSP using spatial data. For instance, $(1, x) \in A, A \in \{\text{line}[(1, 2), (1, 9)], (1, 10)\}$. The allowed set of values for x is then $[2, 9]$ and 10. This can be expressed by means of two constraints $x \in [2, 9]$ and $x = 10$. Usually, in constraint programming, a disjunction can be used for expressing both cases in a sole constraint: $x \in [2, 9], A = \text{line}[(1, 2), (1, 9)] \vee x = 10, A = (1, 10)$. This written form can be seen as a *solved form* or *answer* to the constraint satisfaction problem.

With respect to the set operations, we consider *union*, *intersection*, and *difference* of sets. We have two cases, for spatial objects and sets of spatial objects. For instance, $(1, 2) \cup \text{line}[(1, 2), (9, 10)]$ represents the set of points belonging to both spatial objects. Therefore, the constraint $(1, x) \in (1, 2) \cup \text{line}[(1, 2), (9, 10)]$ has a solved form $x = 2$. An accurate treatment needs the *difference*. When the difference involves two spatial objects, the result of performing such operation

Fig. 1. Metric and Topological Operations

can be a spatial object with holes. For this reason, we allow solved forms in our framework, including geometric figures like $\text{line}[(1, 2), (9, 10)] - (1, 2)$ representing a *closed points-based geometric figure with holes*. These holes are an arbitrary union of closed points-based geometric figures.

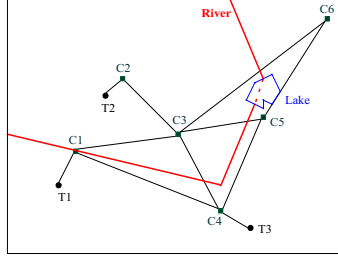
The third kind of operations are the metric ones (see figure 1). They are based on distances, and also handle spatial objects and sets of spatial objects. We can suppose a fixed distance $d(., .)$ for pairs of points: Euclidean, Manhattan, etc. The fixed distance is not relevant, and we can suppose any of them. In this case we have **mindist**, **maxdist**, **pmindist** and **pmaxdist**. The first two represent the minimum and maximum distance between two spatial objects. For instance $x = \text{mindist}(\text{line}[(1, 2), (1, 10)], \text{line}[(3, 2), (12, 15)])$, is a constraint with solved form $x = 3$. Distance-based queries have a particularity. They are *rigid* operations, that is, objects for which a minimum (maximum) distance is computed must be a fully defined object, avoiding the use of ranged variables in their description. For instance, it is forbidden to formulate a constraint satisfaction problem of the form $x = \text{mindist}((1, y), (1, 1)), y \in [1, 3]$. This is due to such a constraint satisfaction problem has not a solved form expressible with equality and membership relations. The solution of this CSP is $x = 0$ if $y = 1$ and otherwise the value of x depends on the value of y and the considered metric.

The operations **pmindist** and **pmaxdist** (resp. **omindist** and **omaxdist**) get the set of pairs of points (resp. objects) with the minimum (resp. maximum) distance. In the same case, $P \in \text{pmindist}(\text{line}[(1, 2), (1, 10)], \text{line}[(3, 2), (12, 15)])$, requires the variable P , a new kind of variable, to be a member of a set of pairs of spatial objects. This set represents the set of pairs of spatial objects which are at the minimum distance. The solved form will be $P \in \{ \{ < (1, 2), (3, 2) > \} \}$. Therefore, we have to handle in addition to set of objects, set of pairs of objects, using a couple of brackets as notation.

With respect to topological relations (see figure 1), they are usual binary topological relations on spatial objects and set of spatial objects. For instance, $A \in \{ (1, 4), (4, 5) \}$, $B \in \{ \text{line}[(1, 4), (1, 10)], (4, 5) \}$, $A \text{ inside } B$ has as solved form $A = (1, 4), B = \text{line}[(1, 4), (1, 10)] \vee A = (4, 5), B = (4, 5)$. Analogously, the constraint $P \in \{ (1, 4), (4, 5) \} \text{ inside } \{ \text{line}[(1, 4), (1, 10)], (4, 5) \}$ has as solved form $P \in \{ \{ < (1, 4), \text{line}[(1, 4), (1, 10)] >, < (4, 5), (4, 5) > \} \}$.

A special case is **rangeq**, which is a topological operation for range queries, such as the occurrence of a spatial object or a set of spatial objects that fall on a certain distance from a point or on a window. In order to handle this kind of queries, we consider as special case of spatial data types, the circle defined by a point and a given radius (**circle**) and a window given by two points (**window**).

Fig. 2. Spatial Example



For instance, $A \in \{(1,2), (9,10)\}$ `rangeq circle((0,0),3)` has as solved form $A = (1,2)$.

In addition, we have considered a set of structural operations in order to compute the size of spatial objects, the set of points and lines which conform the frontier of a spatial object (and a set of spatial objects). They are `setpoints`, `setsegments`, `area`, `length` and `perimeter`.

Finally, our constraint system can handle *negation*, but with restricted use. Equality and membership relations can be used in negative form $\neq$ and $\notin$, but variables used in them (which can be referred to distances, coordinates of points, spatial objects and pairs of spatial objects) must be also ranged in *at least one positive constraint* in order to formulate a constraint satisfaction problem. This restricted version of negation prevents anomalies in the use of negation such as infinite answers for a constraint satisfaction problem. For instance, $C = A \cap B, A \notin \{(1,2), (3,4)\}, B \notin \{(1,2)\}$ cannot be represented in solved form as a finite disjunction of equality and membership relations for C , A and B . The problem is that there is a *infinite and not finitely representable set of solutions* for C .

Therefore, and in order to ensure that *each constraint satisfaction problem has an equivalent solved form as a disjunction of equality and membership relations*, we require that the constraint satisfaction problem is formulated as follows. Each variable, representing a distance, coordinates of points, spatial objects or pair of spatial object, must be ranged in an interval of real numbers, spatial object, sets of spatial objects or sets of pairs of spatial objects, respectively. This (syntactic) condition of range restriction, called *safety condition*, and required over the constraint satisfaction problems, is usual in deductive and constraint databases [8, 1].

Now, we show an example of spatial data to explain the use of our framework for processing spatial queries as Constraint Satisfaction Problems (CSP's). In this example, we have spatial information (non-spatial data can also be managed (e.g. Name, Population, Length, etc.), but we are only interested in its spatial shape) about Cities (set of points), Tourist Resorts (set of points), Roads (set of lines), River (line) and Lake (polygon).

The figure 2 illustrates the spatial data, and assuming the work area in the window $(0, 0)$ for lower-left corner and $(80, 60)$ for the upper-right corner, the coordinates of the spatial components (expressed as constraints) are the following. Cities: $C1 = (16, 24)$, $C2 = (27, 41)$, $C3 = (40, 28)$, $C4 = (50, 10)$, $C5 = (60, 32)$,

$C6 = (75, 55)$; Tourist Resorts: $T1 = (12, 16)$, $T2 = (23, 37)$, $T3 = (57, 6)$; Road $\in \{\text{line}[C6, C5, C3], \text{line}[C6, C3, C1, T1], \text{line}[C5, C4, C1], \text{line}[T3, C4, C3, C2, T2]\}$; River = $\text{line}[(52, 60), (60, 41), (50, 16), (0, 28)]$; and Lake = $\text{polygon}[(64, 38), (62, 35), (60, 36), (60, 34), (56, 36), (58, 40), (62, 42)]$. Note that the road is expressed as a set of line segments due to we cannot handle graphs in our representation.

The following spatial queries (topological, distance-based and directional) are expressed as a CSP as follows, where capitalized names represent variables:

Query	CSP	Answer
(1) List the part of the river that passes through the lake	$\text{Parts} \in \text{setsegments}(\text{River})$ $\text{Through} = \text{Parts} \cap \text{Lake}$	$\text{Through} = \text{line}[(60, 41), (56, 35)]$ $\text{Parts} = \text{line}[(60, 41), (50, 16)]$
(2) Report the part of the road that touches the lake and its length	$\text{Parts} \in \text{setsegments}(\text{Road})$ $\text{Touches} = \text{Parts} \cap \text{Lake}$ $\text{Parts neighbor Lake}$ $l = \text{length}(\text{Touches})$	$\text{Touches} = \text{line}[(64, 36), (62, 35)]$ $\text{Parts} = \text{line}[C6, C5], l = 3.5$
(3) Assuming the river can supply water to cities that are within 10 km. Report all the cities that can use water from the river	$\text{City} \in \{C1, C2, C3, C4, C5, C6\}$ $d = \text{mindist}(\text{City}, \text{River})$ $d \in [0, 10]$	$d = 0, \text{City} = C1 \vee$ $d = 9, \text{City} = C3 \vee$ $d = 6, \text{City} = C4 \vee$ $d = 3, \text{City} = C5$
(4) Find tourist resorts that are within 7.5 km of a city	$\text{Near} \in \{T1, T2, T3\}$ rangeq $\text{circle}(\text{City}, 7.5)$	$\text{Near} = T2, \text{City} = C2$
(5) Which city is the closest to any tourist resort?	$\text{Closest} \in \text{pmindist}(\{C1, C2, C3, C4, C5, C6\}, \{T1, T2, T3\})$	$\text{Closest} \in \{\{< C2, T2 >\}\}$
(6) List the cities to the north of C3	$(x, y) = \text{City}, C3 = (u, v), y \in [u, 60]$	$(x, y) = C2, (u, v) = (40, 28) \vee$ $(x, y) = C5, (u, v) = (40, 28) \vee$ $(x, y) = C6, (u, v) = (40, 28)$
(7) Report bridges which are not in a city:	$\text{Intersection} = \text{Road} \cap \text{River}$ $\text{Intersection} \notin \{C1, C2, C3, C4, C5, C6\}$	$\text{Intersection} = (59, 43) \vee$ $\text{Intersection} = (56, 31) \vee$ $\text{Intersection} = (46, 27)$

3 Constraints on Sets of Spatial Objects

Now we formally present our spatial constraint system. Let $N\text{Var}$ be a set of numerical variables $x, y, \dots$, $O\text{Var}$ a set of variables for objects $A, B, \dots$, $P\text{Var}$ a set of variables for pairs of objects $P, Q, \dots$, and $\mathbb{R}$ the set of real numbers. Now, we have a data constructor for pairs of coordinates $(-, -) : N\text{Type} \times N\text{Type} \rightarrow \text{Point}$, where $N\text{Type} = N\text{Var} \cup \mathbb{R}$, a set of data constructors for spatial data types: $\emptyset : \rightarrow \text{SDT}$, $\text{line} : [\text{Point}] \rightarrow \text{Line}$, $\text{polygon} : [\text{Point}] \rightarrow \text{Polygon}$, and $\text{region} : [\text{Point}] \rightarrow \text{Region}$ where $\text{SDT} = O\text{Var} \cup \text{Point} \cup \text{Line} \cup \text{Polygon} \cup \text{Region}$. We have also a data constructor for pairs of spatial data types: $\langle -, - \rangle : \text{SDT} \times \text{SDT} \rightarrow \text{PairSDT}$, and $\text{PSDT} = P\text{Var} \cup \text{PairSDT}$. With respect to sets of spatial objects and pairs of spatial objects, we have a set of data constructors for sets of (pairs of) spatial objects: $\emptyset : \rightarrow \text{SetSDT}$, $\{ _ \} : \text{SDT} \times \text{SetSDT} \rightarrow \text{SetSDT}$, $\emptyset : \rightarrow \text{SetPSDT}$, and $\{ _ \} : \text{PSDT} \times \text{SetPSDT} \rightarrow \text{SetPSDT}$.

Finally, we have data constructors for a special kind of spatial data types, called **Window**, **Circle** such that $\text{window} : \text{Point} \times \text{Point} \rightarrow \text{Window}$, $\text{circle} : \text{Point} \times N\text{Type} \rightarrow \text{Circle}$ and $\text{Range} = \text{Window} \cup \text{Circle}$. In addition, **Interval** builds finite closed real intervals: $[-, -] : N\text{Type} \times N\text{Type} \rightarrow \text{Interval}$.

Now we present a set of operations over the defined types. Some operations are *partial*, that is, are not defined for some arguments. Assuming the symbol $\perp$ representing an special value called *undefined*, we can consider the following types. For the type **NType**, we have the binary operations $=, \neq: \mathbf{NType} \times \mathbf{NType} \rightarrow \mathbf{Boolean}$, and $\in, \notin: \mathbf{NType} \times \mathbf{Interval} \rightarrow \mathbf{Boolean}$. The formal semantics is as follows. Given a valuation μ of numerical variables into $\mathbb{R} \cup \{\perp\}$, then we denote by $\mu(\mathbf{n})$, where $\mathbf{n} \in \mathbf{NType}$, the value of $\mathbf{n}$ under the valuation μ as the real number (or undefined), defined as $\mu(n) = \mu(x)$ if $n \equiv x$ is a variable and $\mu(n) = n$, otherwise. Now, given $n_1, n_2, n_3 \in \mathbf{NType}$, and a valuation μ , we define $\perp = \perp$, $\perp \neq \perp$ are both false; $n_1 = n_2$ iff $\mu(n_1) = \mu(n_2)$; $n_1 \neq n_2$ iff $\mu(n_1) \neq \mu(n_2)$; $n_1 \in [n_2, n_3]$ iff $\mu(n_2) \leq \mu(n_1) \leq \mu(n_3)$; and finally, $n_1 \notin [n_2, n_3]$ iff $\mu(n_2) > \mu(n_1)$ or $\mu(n_1) > \mu(n_3)$.

$\perp$ represents undefined and therefore an incomparable value. We denote by $Val(\mathbf{NType})$ the set of valuations of numerical variables into $\mathbb{R} \cup \{\perp\}$. Now, the value of a spatial object O under a valuation $\mu \in Val(\mathbf{NType})$, denoted by $\mu(O)$, is a *closed point-based geometric figure* ($\mathcal{FIG}$), and it is defined as follows:

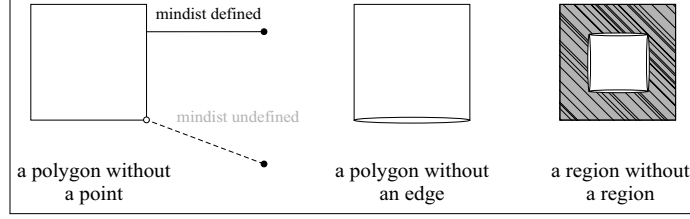
- $\mu((p_1, p_2)) =_{def} \{(\mu(p_1), \mu(p_2))\}$ if none of $\mu(p_i)$ are $\perp$; and $\emptyset$, otherwise.
- $\mu(line[(p_1, p_2), (p_3, p_4), \dots, (p_{n-1}, p_n)]) =_{def} \{(r, s) \mid r = \alpha \times \mu(p_{2k+1}) + (1 - \alpha) \times \mu(p_{2k+3}), s = \alpha \times \mu(p_{2k+2}) + (1 - \alpha) \times \mu(p_{2k+4}), \alpha \in [0, 1], 0 \leq k \leq n - 4/2, k \in \mathbb{N}, \alpha \in \mathbb{R}\}$, if neither of the $\mu(p_i)$ is $\perp$; and $\emptyset$, otherwise, where $n \geq 4$, $n \in \mathbb{N}$, where each $p_i \in \mathbf{NType}$. Analogously for polygon and region.

Spatial objects can be grouped in both sets of objects and sets of pairs of objects. The operations on sets of (pairs of) spatial objects are interpreted into *closed point-based geometric figures with holes* ($\mathcal{FH}$).

Definition 1 (Figures with Holes). *Given $\mathcal{FIG}$'s, F, H_i , a figure with holes has the form $F - \cup_{i \geq 0} H_i$ where $H_i \cap H_j = \emptyset$ if $i \neq j$ and $\cup_{i \geq 0} H_i \subseteq F$. The set $\cup_{i \geq 0} H_i$ is called the holes of the F . The set of figures with holes is $\mathcal{FH}$. We define the following concepts:*

- We denote by $\bar{\mathcal{G}}$ the figure obtained from $\mathcal{G}$, adding the topological frontier.
- We denote by $mindist(\mathcal{G}, \mathcal{G}')$ (resp. $maxdist(\mathcal{G}, \mathcal{G}')$) the minimum (resp. maximum) distance from $\mathcal{G}$ to $\mathcal{G}'$, defined as follows:
 $mindist(\mathcal{G}, \mathcal{G}') =_{def} \perp$ if $\min\{d((p, q), (r, s)) \mid (p, q) \in \mathcal{G}, (r, s) \in \mathcal{G}'\} < \min\{d((p, q), (r, s)) \mid (p, q) \in \bar{\mathcal{G}}, (r, s) \in \bar{\mathcal{G}}'\}$ and otherwise $mindist(\mathcal{G}, \mathcal{G}') =_{def} \min\{d((p, q), (r, s)) \mid (p, q) \in \mathcal{G}, (r, s) \in \mathcal{G}'\}$ and analogously $maxdist$;
- $\mathcal{L}(\mathcal{G})$ denotes the length of a line $\mathcal{G}$. $\mathcal{P}(\mathcal{G})$ denotes the size of the perimeter of a polygon or region $\mathcal{G}$. $\mathcal{A}(\mathcal{G})$ denotes the size of the area of a region $\mathcal{G}$. The three are defined as $\perp$ whenever the region has a hole. $\mathcal{PS}(\mathcal{G})$ denotes the set of points which defines $\mathcal{G}$. Finally, $\mathcal{F}(\mathcal{G})$ denotes the set of line segments which defines the frontier of $\mathcal{G}$.

An element of $\mathcal{FH}$ can have holes, which are in the topological frontier of the object (see figure 3). When the distance (minimum or maximum) from an element of $\mathcal{FH}$ to another element of $\mathcal{FH}$ is computed, it could be that the

Fig. 3. Closed Point-Based Geometric Figures with Holes

nearest points are in the holes. In this case, we consider both distances are undefined (see figure 3). The size of the elements of a figure (area, perimeter and length) is only defined whenever the figure has no holes. The elements of $\mathcal{FH}$ can be grouped into sets, denoted by $\mathcal{SFH}$, and sets of pairs of elements of $\mathcal{FH}$, denoted by $\mathcal{SPFH}$. Grouping $\mathcal{FH}$ into sets we can handle more complex spatial objects like graphs, among others. Sets of (pairs of) spatial objects can have variables representing spatial objects. Given a valuation Δ of variables of spatial objects into $\mathcal{FH}$ then: $\Delta(\mu(\{O_1, \dots, O_n\})) =_{def} \{\Delta(\mu(O_1)), \dots, \Delta(\mu(O_n))\}$, and $\Delta(\mu(\{< O_1, O_2 >, \dots, < O_{n-1}, O_n >\})) =_{def} \{(\Delta(\mu(O_1)), \Delta(\mu(O_2))), \dots, (\Delta(\mu(O_{n-1})), \Delta(\mu(O_n)))\}$ where each $O_i \in \text{SDT}$. Note that the semantics of a set of spatial objects is a set of $\mathcal{FH}$'s. We can also consider valuations of variables of pairs of spatial objects into pairs of $\mathcal{FH}$'s. We denote by $Val(\text{SDT})$ the set of valuations into $\mathcal{FH}$'s and $Val(\text{PSDT})$ the set of valuations into pairs of $\mathcal{FH}$'s.

With respect to set theory operations, we have the set: $\cup, \cap, - : \text{SDT} \times \text{SDT} \rightarrow \text{SetSDT}$, $\text{SetSDT} \times \text{SetSDT} \rightarrow \text{SetSDT}$, the semantics is the usual on set theory, that is, given valuations $\mu \in Val(\text{NType})$, $\Delta \in Val(\text{SDT})$, and $\Omega \in Val(\text{PSDT})$; $O_i \in \text{SDT}$ and $S_i \in \text{SetSDT}$; the set operations are defined as $\|O_1 \theta O_2\|_{(\mu, \Delta, \Omega)} = \|O_1\|_{(\mu, \Delta, \Omega)} \theta \|O_2\|_{(\mu, \Delta, \Omega)}$ and $\|S_1 \theta S_2\|_{(\mu, \Delta, \Omega)} = \|S_1\|_{(\mu, \Delta, \Omega)} \theta \|S_2\|_{(\mu, \Delta, \Omega)}$, where $\theta \in \{\cup, \cap, -\}$.

With respect to equality and membership operations we have the following set: $=, \neq : \text{SDT} \times \text{SDT} \rightarrow \text{Boolean}$, $\text{PSDT} \times \text{PSDT} \rightarrow \text{Boolean}$, and $\in, \notin : \text{Point} \times \text{SDT} \rightarrow \text{Boolean}$, $\text{SDT} \times \text{SetSDT} \rightarrow \text{Boolean}$, $\text{PSDT} \times \text{SetPSDT} \rightarrow \text{Boolean}$. The above operations are defined in terms of the set of points that the (set of) spatial objects define. For instance, $\|O_1 = O_2\|_{(\mu, \Delta, \Omega)} =_{def} \text{true}$ if $\Delta\mu(O_1) = \Delta\mu(O_2)$ and $\|(p_1, p_2) \in O\|_{(\mu, \Delta, \Omega)} =_{def} \text{true}$ if $\mu((p_1, p_2)) \subseteq \Delta\mu(O)$. Similarly with the other equality and membership operations.

With respect to metric operations, we have the following set: $\text{mindist}, \text{maxdist} : \text{SDT} \times \text{SDT} \rightarrow \mathbb{R}$, $\text{SetSDT} \times \text{SetSDT} \rightarrow \mathbb{R}$, $\text{pmindist}, \text{pmaxdist} : \text{SDT} \times \text{SDT} \rightarrow \text{PSDT}$, $\text{SetSDT} \times \text{SetSDT} \rightarrow \text{PSDT}$, and $\text{omindist}, \text{omaxdist} : \text{SetSDT} \times \text{SetSDT} \rightarrow \text{PSDT}$. Operations mindist (resp. maxdist) are undefined whenever the minimum (resp. maximum) distance of the $\mathcal{FH}$'s they represent is undefined. For instance, $\|\text{mindist}(O_1, O_2)\|_{(\mu, \Delta, \Omega)} =_{def} \min\{d((p_1, p_2), (q_1, q_2)) \mid (p_1, p_2) \in \|O_1\|_{(\mu, \Delta, \Omega)}, (q_1, q_2) \in \|O_2\|_{(\mu, \Delta, \Omega)}\}$ if $\text{mindist}(\|O_1\|_{(\mu, \Delta, \Omega)}, \|O_2\|_{(\mu, \Delta, \Omega)}) \neq \perp$, and $\perp$, otherwise; and $\|\text{mindist}(S_1, S_2)\|_{(\mu, \Delta, \Omega)} =_{def} \min\{d((p_1, p_2), (q_1, q_2)) \mid (p_1, p_2) \in \mathcal{G}_1, \mathcal{G}_1 \in \|S_1\|_{(\mu, \Delta, \Omega)}, (q_1, q_2) \in \mathcal{G}_2, \mathcal{G}_2 \in \|S_2\|_{(\mu, \Delta, \Omega)}\}$, where $O_i \in \text{SDT}$ and $S_i \in \text{SetSDT}$. Similarly with the other metric operations.

With respect to the topological ones, we have the following set: $\text{inside}, \text{outside}, \text{overlap}, \text{disjoint}, \text{meet}, \text{cuts}, \text{neighbor}, \text{mcovers}, \text{ncovers}, \text{mcover-}$

edby, ncoveredby : $\text{SDT} \times \text{SDT} \rightarrow \text{Boolean}$, $\text{SetSDT} \times \text{SetSDT} \rightarrow \text{SetPSDT}$, and $\text{rangeq} : \text{SetSDT} \times \text{Range} \rightarrow \text{SetSDT}$. For instance, $\llbracket O_1 \text{ inside } O_2 \rrbracket_{(\mu, \Delta, \Omega)} =_{\text{def}} \text{true}$ if $\llbracket O_1 \rrbracket_{(\mu, \Delta, \Omega)} \subseteq \llbracket O_2 \rrbracket_{(\mu, \Delta, \Omega)}$; and $\llbracket S_1 \text{ inside } S_2 \rrbracket_{(\mu, \Delta, \Omega)} =_{\text{def}} \{(\mathcal{G}_1, \mathcal{G}_2) \mid \mathcal{G}_1 \in \llbracket S_1 \rrbracket_{(\mu, \Delta, \Omega)}, \mathcal{G}_2 \in \llbracket S_2 \rrbracket_{(\mu, \Delta, \Omega)} \text{ and } \mathcal{G}_1 \subseteq \mathcal{G}_2\}$, where $O_i \in \text{SDT}$ and $S_i \in \text{SetSDT}$. Similarly with the other topological operations.

Finally, we have structural operations we have the set: $\text{area} : \text{Region} \rightarrow \mathbb{R}$, $\text{length} : \text{Line} \rightarrow \mathbb{R}$, $\text{perimeter} : \text{Polygon} \cup \text{Region} \rightarrow \mathbb{R}$, $\text{area}, \text{length}, \text{perimeter} : \text{SetSDT} \rightarrow \mathbb{R}$, and $\text{setpoints}, \text{setsegments} : \text{SDT} \rightarrow \text{SetSDT}$, $\text{SetSDT} \rightarrow \text{SetSDT}$. For instance, $\llbracket \text{area}(O) \rrbracket_{(\mu, \Delta, \Omega)} =_{\text{def}} \mathcal{A}(\llbracket O \rrbracket_{(\mu, \Delta, \Omega)})$; and $\llbracket \text{area}(S) \rrbracket_{(\mu, \Delta, \Omega)} =_{\text{def}} \sum_{\mathcal{G} \in \llbracket S \rrbracket_{(\mu, \Delta, \Omega)}, \mathcal{G} \text{ is a region}} \mathcal{A}(\mathcal{G})$. Similarly with the other structural operations.

4 Spatial Constraint Satisfaction Problem

In this section we define what is a spatial constraint satisfaction problem, by presenting its general form, a special class which represents the solved forms, and a set of conditions in order to ensure that each spatial constraint satisfaction problem is equivalent to a solved form.

Definition 2 (Spatial CSP).

A spatial constraint satisfaction problem (SCSP) is a conjunction of typed boolean operations over the types NType , SDT , PSDT , SetSDT , SetPSDT , Range and Interval of the form $\Gamma \equiv \varphi_1, \dots, \varphi_n$.

Definition 3 (Solution). A triple (μ, Δ, Ω) , where $\mu \in \text{Val}(\text{NType})$, $\Delta \in \text{Val}(\text{SDT})$ and $\Omega \in \text{Val}(\text{PSDT})$ is a solution of $\varphi_1, \dots, \varphi_n$ if each $\llbracket \varphi_i \rrbracket_{(\mu, \Delta, \Omega)}$ is equal to true.

A SCSP is called *satisfiable* whenever has at least a solution. Solutions are represented by means of solved SCSP, which are disjunctions of SCSP's in a particular form.

Definition 4 (Solved SCSP). A solved SCSP Π is a disjunction of spatial constraint satisfaction problems of the form $\Pi \equiv \bigvee_{i \geq 1} \Gamma_i$ where each Γ_i is of the form: $\Gamma_i \equiv \varphi_1, \dots, \varphi_n$, and each φ_j is a solved constraint of the form:

1. $x = n$ where $x \in \text{NVar}$ and $n \in \mathbb{R} \cup \text{Dom}(\Gamma_i)$
2. $x \in [n_1, n_2] - \bigcup_{i \geq 0} [h_i, h_{i+1}]$ where $x \in \text{NVar}$ and $n_1, n_2, h_i, h_{i+1} \in \mathbb{R} \cup \text{Dom}(\Gamma_i)$; in addition, $\Gamma_i(n_1) \leq \Gamma_i(h_i) \leq \Gamma_i(h_{i+1}) \leq \Gamma_i(n_2)$, for all i , and $[\Gamma_i(h_i), \Gamma_i(h_{i+1})] \cap [\Gamma_i(h_j), \Gamma_i(h_{j+1})] = \emptyset$ if $i \neq j$.
3. $(x, y) = (n_1, n_2)$, where $x, y \in \text{NVar}$, $n_1, n_2 \in \mathbb{R} \cup \text{Dom}(\Gamma_i)$
4. $(x, y) \in O$, where $x, y \in \text{NVar}$ and $O \in \mathcal{FH} \cup \text{Dom}(\Gamma_i)$
5. $A = O - \bigcup_{i \geq 0} H_i$ where $A \in \text{OVar}$ and $O, H_i \in \mathcal{FH} \cup \text{Dom}(\Gamma_i)$; in addition, $\bigcup_{i \geq 0} \Gamma_i(H_i) \subseteq \Gamma_i(O)$, and $\Gamma_i(H_i) \cap \Gamma_i(H_j) = \emptyset$ if $i \neq j$.
6. $A \in S$ where $A \in \text{OVar}$ and $\Gamma_i(S) \in \mathcal{SFH}$.
7. $P = Q$ where $P \in \text{PVar}$, where $Q \in \text{Dom}(\Gamma_i)$ or $Q \equiv < Q_1, Q_2 >$, $Q_1, Q_2 \in \mathcal{FH} \cup \text{Dom}(\Gamma_i)$
8. $P \in SP$ where $P \in \text{PVar}$ and $\Gamma_i(SP) \in \mathcal{SPFH}$.

where there exists at most one solved constraint for each variable x , A and P in each Γ_i , which is called domain of Γ_i , denoted by $\text{Dom}(\Gamma_i)$. $\Gamma_i(x)$ (resp. $\Gamma_i(A)$, $\Gamma_i(P)$) denote the set of solutions represented by the solved constraints in which occurs x (resp. A and P), and $\Gamma_i(S)$, $\Gamma_i(SP)$ the set of $\mathcal{SFH}$'s, respectively $\mathcal{SPFH}$'s obtained from them.

Solved forms assign values to distances, coordinates of points, variables of spatial objects, and variables of pairs of spatial objects. Each assignment can represent a finite or infinite set of solutions, and is defined as follows.

Definition 5 (Solutions of a Solved SCSP).

A solved SCSP Π defines a set of solutions, denoted by $\text{Sol}(\Pi)$, and defined as $\text{Sol}(\Pi) = \cup_{\Gamma \in \Pi} \text{Sol}(\Gamma)$, where each Γ defines a set of solutions which consists on triples (μ, Δ, Ω) , recursively defined as follows:

$$\begin{aligned} - \mu &=_{\text{def}} \cup_{x=n \in \Gamma} \{x/\mu(n)\} \cup_{(x \in [n_1, n_2]) \in \Gamma, r \in [\mu(n_1), \mu(n_2)]} \{x/r\} \\ &\quad \cup_{((x,y)=(n_1, n_2)) \in \Gamma} \{x/\mu(n_1), y/\mu(n_2)\} \\ &\quad \cup_{((x,y) \in O) \in \Gamma, (p,q) \in \Delta\mu(O)} \{x/p, y/q\} \\ - \Delta &=_{\text{def}} \cup_{A=O \in \Gamma} \{A/\Delta\mu(O)\} \cup_{(A \in S) \in \Gamma, \mathcal{G} \in \Delta\mu(S)} \{A/\mathcal{G}\} \\ - \Omega &=_{\text{def}} \cup_{P=Q \in \Gamma} \{P/\Omega\Delta\mu(Q)\} \cup_{(P \in PO) \in \Gamma, (\mathcal{G}_1, \mathcal{G}_2) \in \Omega\Delta\mu(PO)} \{P/(\mathcal{G}_1, \mathcal{G}_2)\} \end{aligned}$$

Finally, $\Gamma(x) =_{\text{def}} \mu(x)$, $\Gamma(A) =_{\text{def}} \Delta(A)$ and $\Gamma(P) =_{\text{def}} \Omega(P)$, $\Gamma(n) =_{\text{def}} n$ if $n \in \mathbb{R}$, $\Gamma(O) =_{\text{def}} O$ if $O \in \mathcal{FH}$, $\Gamma(P) =_{\text{def}} P$ if $P = (\mathcal{G}_1, \mathcal{G}_2)$ and $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{FH}$ for each $\Gamma \in \Pi$.

In order to ensure that any spatial constraint satisfaction problem is equivalent to a solved SCSP we have to require that the SCSP follows some syntactic conditions. We call *safe SCSP* to SCSP ensuring these conditions.

Definition 6 (Range Restricted Variables). We say that a variable α occurring on a SCSP Γ is range restricted if there exist (a) an equality constraint $\alpha = E$ or (b) membership constraint $\alpha \in E$ such that E is ground (without variables) or all the variables of E are ranged restricted.

We call that a variable is *strongly range restricted* considering only the case (a) in the above definition. Finally, we require that distance-based and structural operations are *rigid*.

Definition 7 (Rigid Constraints). A spatial constraint is rigid if whenever an operation involving $\Theta \in \{\text{mindist}, \text{maxdist}, \text{pmindist}, \text{pmaxdist}, \text{omindist}, \text{omaxdist}, \text{area}, \text{length}, \text{perimeter}\}$ is included in the form $\Theta(O_1, O_2)$ (or $\Theta(O)$) (resp. $\Theta(S_1, S_2)$ (or $\Theta(S)$); and O_1 and O_2 (or O) (resp. S_1 and S_2 (or S))) only contain strongly range restricted variables.

Definition 8 (Safe SCSP). A safe SCSP is a SCSP where all the variables are range restricted, and all the spatial constraints are rigid.

Theorem 1. Each satisfiable safe SCSP is equivalent to a solved SCSP.

5 Conclusions and Future Work

We have presented the basis of our constraint system. Our framework is closer to the research on spatial databases focused on sets of spatial objects [10, 4, 11, 6], which is a widely investigated area. Constraint programming can adopt from this research area some *efficient solving techniques* studied for the management of spatial objects. For instance, the sets of spatial objects can be approximated by means of *minimum bounded rectangles (MBR's)*, and stored on suitable data structures like *R-trees*. In addition, *constraint solver algorithms* can be based on *heuristics* studied for *branch and bound algorithms* suitable for solving of queries on sets of spatial objects.

References

1. J. M. Almendros-Jiménez and A. Becerra-Terón. A Safe Relational Calculus for Functional Logic Deductive Databases, Selected Papers of the WFLP'03. *Electronic Notes on Theoretical Computer Science*, 86(3), 2003.
2. A. Belussi, E. Bertino, and B. Catania. Manipulating Spatial Data in Constraint Databases. In *Proceeding of SSD'97 Conference*, LNCS 1262, pages 115–141. Springer, 1997.
3. A. Belussi, E. Bertino, and B. Catania. An Extended Algebra for Constraint Databases. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 10(5):686–705, 1998.
4. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest Pair Queries in Spatial Databases. In *Proceeding of ACM SIGMOD Conference*, pages 189–200. ACM Press, 2000.
5. G. M. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer, 2000.
6. D. Papadias, Y. Theodoridis, T. K. Sellis, and M. J. Egenhofer. Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees. In *Proceedings of ACM SIGMOD Conference*, pages 92–103, 1995.
7. P. Revesz. *Introduction to Constraint Databases*. Springer Verlag, 2002.
8. P. Z. Revesz. Safe Query Languages for Constraint Databases. *ACM TODS*, 23(1):58–99, 1998.
9. P. Rigaux, M. Scholl, L. Segoufin, and S. Grumbach. Building a Constraint-based Spatial Database System: Model, Languages, and Implementation. *Information Systems*, 28(6):563–595, 2003.
10. P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases: with application to GIS*. Morgan Kaufmann Publishers, 2001.
11. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proceedings of ACM SIGMOD Conference*, pages 71–79, 1995.

Una Aproximación Práctica al Desarrollo de Comprobadores de Tipos basada en Patrones de Diseño

Francisco Ortín, Luis Vinuesa, Juan Manuel Cueva

Universidad de Oviedo, Departamento de Informática, Calvo Sotelo s/n, 33007, Oviedo
{ortin, vinuesa, cueva}@uniovi.es

Abstract. Los sistemas de tipos son mayoritariamente empleados para asegurar que las distintas construcciones de un lenguaje de programación puedan ser ejecutadas sin la aparición de comportamientos erróneos en tiempo de ejecución. Se suelen emplear formalismos como el cálculo lambda con tipos para demostrar que los sistemas de tipos poseen determinadas características, pudiendo demostrarse que están bien definidos o que son seguros. Un comprobador de tipos es un algoritmo encargado de verificar si un programa cumple todas las reglas especificadas en el sistema de tipos del lenguaje en el que fue escrito. Aunque existen herramientas de descripción y procesamiento de sistemas de tipos, éstas no suelen ser empleadas en el desarrollo de procesadores de lenguajes comerciales por el conjunto de carencias que poseen.

Buscando la independencia del lenguaje de implementación, hemos desarrollado un diseño orientado a objetos encaminado a implementar comprobadores de tipos de cualquier lenguaje de programación, empleando para ello un conjunto de patrones de diseño. Elementos existentes en la mayoría de los sistemas de tipos, tales como expresión, constructor, equivalencia y coerción de tipos, o más avanzados como el polimorfismo paramétrico, han sido desarrollados con el diseño presentado. No es necesario que el desarrollador del procesador tenga conocimiento alguno de formalismos ni notaciones matemáticas, empleados frecuentemente en la teoría de tipos. El diseño es fácilmente comprensible, permitiendo así una sencilla depuración y extensión del comprobador y sistema de tipos. Adicionalmente se obtienen los beneficios de emplear determinados patrones de diseño orientados a objetos, altamente probados en el desarrollo de sistemas informáticos.

Palabras Clave. Sistemas de tipos, comprobador de tipos, patrones de diseño, orientación a objetos.

1 Introducción

Un tipo es una colección de objetos poseedores de una estructura similar; la teoría de tipos, en sus comienzos, fue una rama de las matemáticas y la lógica basada en clasificar objetos mediante tipos [1]. Con la aparición de los ordenadores y los lenguajes de programación, la teoría de tipos se ha constituido como una aplicación práctica del desarrollo de lenguajes de programación. Un sistema de tipos es un método dirigido por sintaxis para demostrar la ausencia de comportamientos erróneos de programas, clasificando las construcciones sintácticas en función del tipo de valor que pueden

procesar [2]. Así, los sistemas de tipos dividen los posibles valores en tipos, identificando ciertas construcciones del lenguaje como ilegales, en función de su tipo.

La comprobación de tipos es el análisis que detecta inconsistencias y anomalías semánticas en un programa, asegurándose de que las entidades se ajusten a su declaración y sean combinadas con otras entidades de tipos compatibles; el algoritmo que lleva a cabo este proceso se llama comprobador de tipos [3]. Un procesador de lenguaje acostumbra a implementar un comprobador de tipos en su fase de análisis semántico [4]. Los principales beneficios obtenidos en la comprobación de tipos efectuada en tiempo de compilación (estática) son:

- Detección temprana de errores. La comprobación estática de tipos reduce el número de errores que un programa puede generar en tiempo de ejecución. Éste es el beneficio más obvio ya que, gracias a la detección temprana de los errores, el programador podrá reparar éstos de un modo casi inmediato –y no cuando se esté ejecutando la aplicación, pudiendo incluso haber sido implantada.
- Abstracción. Otra ventaja de emplear tipos en los lenguajes de programación es que su uso fuerza al programador a dividir el problema en diversos tipos de módulos, de un modo disciplinado. Los tipos identifican la interfaz de los módulos (funciones, clases, paquetes o componentes) proporcionando una simplificación de los servicios que ofrece cada módulo; un tipo de contrato parcial entre los desarrolladores del módulo y sus clientes.
- Eficiencia. Una entidad de un programa declarada con un tipo específico indica información relativa a lo que se intenta hacer con ella. De este modo, al conocer el tipo de las construcciones del lenguaje, se podrá generar código de carácter más específico y eficiente que si no tuviésemos esa información. En los procesadores que no poseen tipos en tiempo de compilación se tiene que descubrir la ejecución específica en tiempo de ejecución, con la consecuente pérdida de eficiencia.
- Seguridad. Un sistema de tipos es seguro si es capaz de garantizar la integridad de las abstracciones de alto nivel ofrecidas al programador. Por tanto, un programa aceptado por un sistema de tipos seguro se comportará de un modo adecuado, sin generar errores de tipo en tiempo de ejecución.
- Legibilidad. Un tipo de una entidad (variable, objeto o función) transmite información acerca de lo que se intenta hacer con ella, constituyendo así un modo de documentación del código.

Aunque el uso tradicional de los sistemas de tipos es la validación semántica de los programas de un lenguaje, también se emplean en otros campos como la ingeniería inversa [5], pruebas de software [6], metadatos del web [7] o incluso en seguridad [8].

1.2 Herramientas de Sistemas de Tipos

Los sistemas de tipos son uno de los métodos formales dentro de la computación empleados para asegurar que un sistema se comporta adecuadamente en función de una especificación del comportamiento deseado. Su formalización requiere notaciones precisas, válidas para demostrar propiedades de los sistemas de tipos [3]. Los formalismos más empleados en la actualidad son los cálculos lambda con tipos y la semántica operacional [9]; en el caso de los comprobadores de tipos, lo más común es emplear gramáticas con atributos [10].

Sin embargo, aunque la formalización es necesaria para realizar estas demostraciones, también se requiere tener una infraestructura para la implementación de comprobadores de tipos basados en estos sistemas de tipos. La mayoría de las herramientas existentes se basan en estos formalismos: TinkerType utiliza cálculo lambda con tipos para demostrar propiedades de los sistemas de tipos, permitiendo implementar comprobadores de tipos en ML [11]; los métodos formales *ligeros*, tales como ESC/Java [12], SLAM [13] o SPIN [14], realizan análisis exhaustivos y verificación de programas para detectar errores que un compilador pasa por alto; las escasas herramientas de construcción de compiladores que ofrecen la generación de comprobadores de tipos están basadas en gramáticas con atributos [15] o modificaciones de éstas [16].

Aunque los sistemas de tipos y sus herramientas son utilizadas en distintas áreas dentro de la informática (como la investigación en lenguajes de programación), la mayoría de los compiladores comerciales no son desarrollados empleando estas herramientas [17]. Los principales motivos son:

- Los métodos formales están más enfocados a demostrar propiedades de los sistemas de tipos, más que a implementar comprobadores. Por ello, las notaciones matemáticas empleadas pueden ser desconocidas para el desarrollador del compilador.
- La salida de las herramientas está restringida a un lenguaje de programación. Si tomamos TinkerType como un ejemplo, los comprobadores generados están desarrollados en ML.
- Si se emplea una herramienta de generación de comprobadores de tipo a partir de un método formal, existirá un salto en la abstracción de la representación formal al código generado. De esta forma, el código producido es más difícil de depurar y comprender puesto que proviene de la traducción automática de otra notación.
- Las herramientas basadas en métodos formales no están inmersas en entornos de desarrollo comerciales. Puesto que la orientación a objetos se ha constituido como el principal paradigma del desarrollo de software comercial, la implementación de un comprobador de tipos empleando estas tecnologías se podrá beneficiar de sus aportaciones [18], de los patrones de diseños existentes [19] y de todos los componentes, herramientas y *frameworks* reutilizables, disponibles en el mercado.

Sin tratar de crear una nueva herramienta que supere las limitaciones mencionadas, definiremos un diseño de ingeniería del software capaz de implementar comprobadores de tipos cuyos sistemas de tipos puedan haberse demostrado correctos previamente —empleando en dicho caso métodos formales. Para ello, describiremos un conjunto de diseños orientados a objetos que empleen los beneficios de este paradigma, así como la utilización de patrones de diseño ampliamente conocidos. El diseño será independiente del lenguaje de programación, fácilmente comprensible y mantenible por el desarrollador del procesador del lenguaje, y cubrirá las características estáticas y dinámicas de los sistemas de tipos más comunes, permitiendo extender y adaptar éste en función de las necesidades del lenguaje a procesar.

El resto del documento está organizado de la siguiente manera. La sección 2 introduce el modo de modelar las expresiones de tipo. La equivalencia de expresiones de tipo es descrita en la sección 3, y en el siguiente apartado se modelan éstas reduciendo el consumo de memoria y procesador. Cómo desarrollar las propiedades de coer-

ción de tipos y polimorfismo paramétrico son abordados en las secciones 5 y 6 respectivamente. Finalmente se presentan las conclusiones y trabajo futuro.

2 Expresiones de Tipo

A la hora de implementar un comprobador de tipos, el procesador de lenguaje deberá representar internamente los tipos del lenguaje de algún modo. Cualquier representación, interna o externa, del tipo de cada construcción sintáctica del lenguaje se denomina expresión de tipo [4]. Un procesador de lenguaje deberá poseer una representación interna de las expresiones de tipo del lenguaje que procesa, para llevar a cabo, entre otras tareas, las comprobaciones de tipo oportunas.

Las expresiones de tipo se centran en la definición constructiva de un tipo: Un tipo es definido como un conjunto de tipos simples (también llamados predefinidos, básicos o primitivos), o bien como un tipo compuesto o construido formado a partir de otros tipos. Un tipo simple es un tipo atómico cuya estructura interna no puede ser modificada por el programador (`integer`, `float`, `boolean...`). Un tipo construido o compuesto es un tipo construido por el programador a partir de un constructor de tipos (`record`, `array`, `set`, `class...`) aplicado a otros tipos —básicos o construidos.

Un comprobador de tipos asocia (dinámica o estáticamente) un tipo a las distintas construcciones del lenguaje, calculando en todo momento el tipo de las expresiones, para posteriormente realizar las comprobaciones oportunas del análisis semántico. Este proceso de asociación de una expresión de tipo a las distintas construcciones del lenguaje se denomina inferencia de tipos [3]. Diversas características de los lenguajes de programación como la equivalencia y coerción de tipos, el polimorfismo, la genericidad y la sobrecarga hacen que la inferencia de tipos de un lenguaje se convierta en una tarea compleja. Un buen diseño de la representación interna de las expresiones de tipo del lenguaje será clave para poder inferir los tipos del modo más sencillo, extensible, mantenible y reutilizable.

2.1 Implementación de las Expresiones de Tipo

Cada compilador podrá representar internamente sus expresiones de tipo de un modo distinto en función de las características del lenguaje a procesar y del lenguaje de programación empleado en su implementación. Como primer ejemplo, las expresiones de tipo de un lenguaje que únicamente posea tipos simples, sería tan sencillo como utilizar una representación mediante variables enteras, enumeradas o caracteres. Sin embargo, estas representaciones no serían válidas cuando el lenguaje a procesar posea algún constructor de tipo —el constructor de tipo puntero, por ejemplo, puede dar lugar a infinitas expresiones de tipo.

Con la aparición de constructores de tipo se crean infinitas combinaciones a la hora de enumerar las posibles expresiones de tipo. El empleo de cadenas de caracteres para representar las expresiones de tipos sería una solución factible, utilizando algún lenguaje de representación de expresiones de tipo como el descrito por Alfred Aho [4]. No obstante, cada vez que queramos extraer un tipo que forme parte de otro tipo compuesto, deberíamos procesar la cadena de caracteres como si de un lenguaje se tratase, incurriendo en una excesiva e innecesaria complejidad computacional.

Una representación más sencilla y eficiente sería con estructuras de datos recursivas, mediante registros o uniones en los que los constructores de tipo empleen punteros recursivos. Esta estructura de tipos es más fácil de manipular que una cadena de caracteres y requiere una menor capacidad de cómputo a la hora de procesar una expresión de tipo compuesta.

La principal limitación de este tipo de estructura de datos es que, precisamente, sólo modela datos. En el desarrollo de un procesador de lenguaje será necesario asociar a estas estructuras un conjunto de rutinas tales como inferencia, equivalencia, coerción y comprobaciones de tipos para la fase de análisis semántico, o tamaño y representación de datos para la fase de generación de código. Existirán rutinas específicas para cada expresión de tipo, y otras comunes a todas ellas. Haciendo uso de la orientación a objetos, podremos asociar estas rutinas generales y específicas a los datos oportunos mediante el empleo de clases, obteniendo adicionalmente una mayor calidad en el diseño mediante el empleo de técnicas de encapsulamiento, abstracción, herencia y polimorfismo, reduciendo así la complejidad de la implementación de un sistema de tipos.

El problema de representar estructuras compuestas recursivamente de un modo jerárquico, aparece en diversos contextos dentro del campo de la computación. El patrón de diseño orientado a objetos *Composite* [19] ha sido utilizado para modelar y resolver este tipo de problemas. Permite crear estructuras compuestas recursivamente, tratando tanto los objetos simples como los compuestos de un modo uniforme, de forma que la utilización de un objeto compuesto y simple pueda realizarse sin conllevar distinción alguna. Su modelo estático se representa mediante el diagrama de clases de la Figura 1.

Los distintos elementos del modelo son:

- **Componente.** Clase típicamente abstracta que declara la interfaz de todos los elementos, independientemente de que sean simples o compuestos. Puede implementar en sus métodos un comportamiento por omisión, o declararlo como abstracto para que cada uno de los subtipos lo implementen.
- **Hoja.** Representa cada nodo hoja de la estructura jerárquica. Un nodo hoja no tienen ningún tipo “hijo”. Definirá en sus métodos las operaciones concretas para ese nodo específico.
- **Compuesto (*Composite*).** Modela aquellos nodos que se construyen como composición de otros nodos, almacenando referencias a sus nodos “hijo”. Implementa las operaciones en función de los hijos que posee, e incluso en función de los resultados de cada una de las operaciones de sus hijos.
- **Cliente.** Cualquier elemento software que utiliza el patrón de diseño presentado.

Siguiendo este patrón de diseño, cada expresión de tipo primitivo será una clase hoja y los tipos compuestos con cada constructor de tipo serán clases *Composite*. Las operaciones comunes a todos los tipos, propias de la fase de análisis semántico (equi-

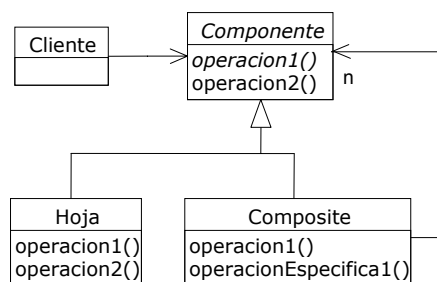


Fig. 1. Diagrama de clases del patrón de diseño *Composite*.

valencia, coerción, inferencia o unificación) y de generación de código (tamaño o representación), serán ubicadas en la clase componente. El hecho de ubicar estos métodos en la clase raíz de la jerarquía, implica que se pueda llevar a cabo un tratamiento uniforme de todos los tipos respecto a este conjunto de mensajes públicos, indistintamente de su estructura interna.

Si existe un comportamiento por omisión para la mayoría de las expresiones de tipos, será implementado al nivel de la clase Componente. Cada clase derivada redefinirá, si fuese necesario, la operación general definida en el componente para su caso específico —u obtendrá, en caso contrario, el comportamiento por defecto. Adicionalmente, cualquier clase derivada podrá definir métodos específicos propios de su expresión de tipo, sin necesidad de que éstos estén declarados en su clase base. Estos métodos definirán los mensajes específicos de la expresión de tipo que representa la clase, no teniendo sentido éstos para el resto de tipos.

El diagrama de clases de la Figura 2 muestra la estructura preliminar del diseño empleado para modelar las expresiones de tipo de un subconjunto del lenguaje C, siguiendo el patrón de diseño *Composite*.

La clase que juega el papel de componente en el patrón *Composite* es la clase abstracta `ExpresionTipo`. Esta posee el comportamiento general de todas las expresiones de tipo, alguno de ellos predefinido por omisión. Los métodos de ejemplo definidos son:

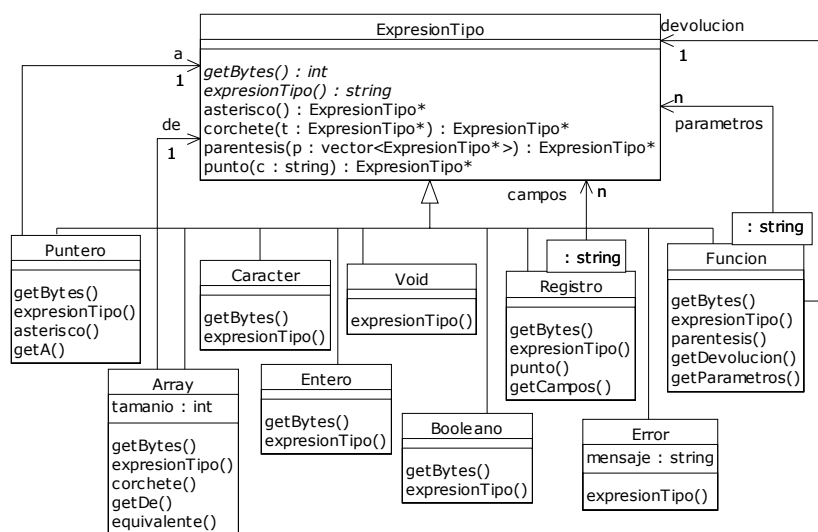


Fig. 2. Diagrama de clases empleado para modelar expresiones de tipo del lenguaje C.

- Métodos `asterisco`, `corchete`, `parentesis` y `punto`: Estos métodos tienen dos responsabilidades: calcular (inferir) el tipo resultante tras aplicar un operador del lenguaje, a partir de una expresión de tipo y, en algún caso, otras expresiones de tipo pasadas como parámetro, y realizar la comprobación de tipos; devolverán el tipo `Error` con una descripción del mismo, si existe un error de tipo. La implementación por omisión es devolver siempre un error, indicando así que dicha operación no está semánticamente definida para el tipo. Cada tipo que implemente este operador deberá redefinir el método relacionado. Por ejemplo, el

mente este operador deberá redefinir el método relacionado. Por ejemplo, el tipo `Puntero` implementa el método `asterisco` puesto que esta operación está permitida para este tipo; la expresión de tipo que devuelve (infiere) es el tipo al que apunta.

- Método `expresionTipo`: Devuelve una cadena de caracteres representativa de su expresión de tipo, siguiendo la notación utilizada en muchos compiladores [4]. Sus objetivos principales son facilitar las tareas de depuración y el ahorro de memoria (explicado en siguiente punto). El código C++ de la Figura 3 muestra un ejemplo de implementación de este método para la clase `Funcion`. Así, una función que reciba un puntero a un entero y un real y no devuelva nada, poseerá la expresión de tipo `"(Puntero(int),float)->void"`.
- Método `getBytes`: Es un mero ejemplo de cómo los tipos del lenguaje poseen funcionalidad propia de la fase de generación de código. Este método devuelve el tamaño en bytes necesario para albergar una variable de ese tipo.

```
string Funcion::expresionTipo() const {
    ostringstream o;
    if (parametros.size()) o<<"(";
    unsigned i=0;
    for (;i< parametros.size()-1;i++)
        o<< parametros[i]-> expresionTipo()<<', ';
    if (parametros.size())
        o<< parametros [i]-> expresionTipo()<<')';
    o<<"->"<<devolucion-> expresionTipo();
    return o.str(); }
```

Fig. 3. Implementación del método que calcula expresiones de tipo.

Nótese cómo los tipos compuestos poseen asociaciones al tipo base: un puntero requiere el tipo al que apunta (`a`); un *array* necesita el tipo que colecciona (`de`); un registro requiere una colección de sus campos (`campos`), cualificada por el nombre del campo (`string`); una función requiere una asociación al tipo que devuelve (`devolucion`), así como una colección cualificada por el nombre de cada uno de sus parámetros (`parametros`). El hecho de que todas las asociaciones estén dirigidas hacia la clase base de la jerarquía, hace que cada tipo compuesto pueda formarse con cualquier otro tipo –incluyendo él mismo– gracias al polimorfismo.

Otra de las ventajas de este diseño, es la facilidad con la que los tipos se pueden ir construyendo con las herramientas existentes de análisis, tales como Lex/Yacc, AntLR o JavaCC. Las expresiones de tipo se pueden construir dirigidas por la sintaxis, por medio de sus constructores. Inicialmente se crean los tipos básicos y éstos se emplean para construir posteriormente nuevos tipos compuestos en las rutinas semánticas de las herramientas de análisis.

2.2 Un Primer Ejemplo

Mostraremos un escenario de ejemplo de utilización del diseño mostrado para representar las expresiones de tipo de un subconjunto del lenguaje C. La parte léxica y sintáctica se ha especificado mediante las herramientas Lex y Yacc y el resto del procesador se ha implementado en C++ ISO/ANSI. Cuando una variable es definida (líneas 1 a la 9 de la Figura 4), se asocia en una tabla de símbolos su identificador con su tipo –un puntero de tipo `ExpresionTipo`. En estas rutinas semánticas, se crean los

tipos mediante la invocación del constructor apropiado, componiendo los tipos complejos conforme el Yacc vaya realizando las reducciones.

Las sentencias de una función son una repetición de expresiones. Cada expresión tiene que tener un tipo y éste se representa mediante un puntero a `ExpresionTipo`. En cada reducción será necesario inferir el tipo de la subexpresión, aplicando en cada caso el operador oportuno. Esta inferencia será procesada mediante el envío del mensaje oportuno al tipo de la subexpresión. Por ejemplo, la línea 14 de la Figura 4 obtiene el tipo puntero de la primera subexpresión `puntero`; en la siguiente reducción aparece el operador `*`, enviándose el mensaje asterisco, infiriéndose el tipo apuntado `Entero`; el siguiente paso es obtener de la tabla de símbolos el tipo `Array` de `vector`; a éste se le pasa el mensaje `corchete`, pasándole el tipo `Entero` previamente inferido; el tipo resultante de la expresión es `entero`.

A modo de ejemplo, la Figura 4 muestra una traza de las expresiones de tipos inferidas para cada sentencia del programa principal. En la parte izquierda se muestra el código fuente con su número de línea. Lo mostrado en la parte derecha es la cadena de caracteres devuelta por el envío del mensaje `expresionTipo`, a cada tipo inferido.

1:	<code>int vector[10];</code>	
2:	<code>int *puntero;</code>	
3:	<code>int **punteroDoble;</code>	
4:	<code>char **v[10];</code>	
5:	<code>char *w[10];</code>	
6:	<code>struct Fecha { int dia, mes, anio; };</code>	
7:	<code>struct Fecha fecha;</code>	
8:	<code>int *f(int, char*);</code>	
9:	<code>void p(int*);</code>	
10:	<code>int main() {</code>	
11:	<code> fecha;</code>	<code>Registro((dia x int)x</code>
12:		<code> (mes x int)x(anio x int))</code>
13:	<code> v;</code>	<code>Array(10, Puntero(Puntero(char)))</code>
14:	<code> vector[*puntero];</code>	<code>int</code>
15:	<code> **v[**punteroDoble];</code>	<code>char</code>
16:	<code> w[*f(3, w[1])];</code>	<code>Puntero(char)</code>
17:	<code> p(f(fecha.dia, w[2])); }</code>	<code>void</code>

Fig. 4. Trazo de inferencia de tipos en un programa de ejemplo.

3 Equivalencia de Expresiones de Tipo

Como hemos mencionado con anterioridad, el principal objetivo de un sistema de tipos es evitar la aparición de errores de tipo en tiempo de ejecución. En diversas construcciones de un lenguaje de programación (como la invocación a funciones o la asignación de una expresión a una variable) es necesario detectar la incorrecta utilización de expresiones que no posean tipos equivalentes. Por tanto, uno de los elementos a tener en cuenta cuando se diseña un sistema de tipos es determinar las condiciones que hacen que dos tipos sean equivalentes entre sí, es decir, que sean el mismo tipo. Existen diversas alternativas de equivalencia de tipos, siendo las dos principales clasificaciones las siguientes [20]:

- Equivalencia estructural. Dos tipos son estructuralmente equivalentes, únicamente si poseen la misma estructura, es decir, o son el mismo tipo básico, o bien están formadas aplicando el mismo constructor a tipos estructuralmente equivalentes.

Lenguajes tales como ML, Algol-68 y Modula-3 emplean un sistema de tipos basado en equivalencia estructural. C++ implementa equivalencia estructural, excepto para clases, registros y uniones.

- Equivalencia de nombres (nominal). Establece que todo tipo ha de tener un nombre único, considerando dos tipos equivalentes sólo si tienen el mismo nombre.

Lenguajes con equivalencia de nombres son Java y Ada.

El diseño de la segunda es más sencillo que el primero, puesto que se reduce a la comparación de los identificadores únicos de cada tipo. Sin embargo, la equivalencia estructural requiere un proceso de comparación recursiva de cada uno de los elementos que constituyen un tipo.

Utilizando el diseño presentado para las comprobaciones de tipos, la implementación de un algoritmo de equivalencia estructural se convierte en una tarea realmente sencilla. Puesto que todos los tipos han de poder ser comparados con cualquier otro tipo, se añadirá a la clase `ExpresionTipo` un método `equivalente` encargado de devolver si el tipo pasado como parámetro es equivalente al actual.

En el caso de los tipos simples, el método `equivalente` se limitará a comprobar si el parámetro es del mismo tipo que el de la clase —ésta será la implementación por omisión ubicada en la clase `ExpresionTipo`. En el caso de las expresiones de tipo compuestas es necesario comprobar que ambos han sido construidos con el mismo constructor de tipos, y además llevar a cabo una comparación recursiva de cada una de las expresiones de tipo empleadas para construir el tipo compuesto. Una implementación en C++ de estos dos ejemplos es la siguiente:

```
bool ExpresionTipo::equivalente(const ExpresionTipo *et) const {
    return typeid(*this)==typeid(*et); /* RTTI */ }
bool Registro::equivalente(const ExpresionTipo *te) const {
    const Struct *record=dynamic_cast<const Struct*>(te);
    if (!record) return false;
    if (campos.size()!=record->campos.size()) return false;
    map<string,TypeExpression*>::const_iterator it1,it2;
    for (it1=campos.begin(),it2=record->campos.begin();
         it1!=campos.end();++it1,++it2) {
        if (it1->first!=it2->first) return false;
        if (!it1->second->equivalente(it2->second)) return false;
    }
    return true; }
```

Fig. 5. Implementación de ejemplo de equivalencia estructural.

Empleando la información de tipos ofrecida por el C++ estándar (RTTI), la implementación por omisión devuelve la igualdad de la clase de la que las expresiones de tipo son instancia. En el caso del registro deberá cumplirse, además, que el número de campos coincida y que cada uno de ellos sean equivalentes entre sí.

4 Representación de Tipos mediante GADs

Del diagrama de clases utilizado para representar las expresiones de tipo (empleando el patrón *Composite*) se pueden crear las expresiones mediante árboles de objetos. Si la instanciación de objetos es llevada a cabo mediante una estructura de árbol, se duplica la representación de tipos para un identificador dado. Por ejemplo, en el árbol

asociado al identificador `f` del programa mostrado en la Figura 4, aparece dos veces el tipo `Entero` –para el tipo devuelto y para su primer parámetro. Esta duplicación de objetos se hace más latente conforme se van creando distintas expresiones de tipos para las diversas construcciones sintácticas del lenguaje, que en la mayoría de los programas reales supone cifras demasiado elevadas. El alto consumo de memoria y el tiempo excesivo requerido para crear y recorrer este número de estructuras de objetos hace necesario el rediseño de las expresiones de tipo.

Este tipo de problemas es resuelto de un modo sencillo por el patrón de diseño *Flyweight* [19]. Su modelo se basa en identificar objetos compartidos simultáneamente en múltiples contextos, representando el estado compartido en todos los estados aparte de los estados particulares de cada objeto. Conforme se cree un número elevado de objetos, la parte compartida a todos ellos (*Flyweight*) se representa mediante un único objeto; otras instancias de un tamaño más reducido representarán la información particular de cada uno de los objetos por separado.

La separación de los estados compartidos y particulares en nuestro caso es trivial. Ante un programa de entrada, un procesador del lenguaje reconocerá un conjunto elevado de símbolos, teniendo cada uno de ellos un tipo –al igual que toda expresión del lenguaje.

- La información relativa a un símbolo es particular. Una variable tendrá un desplazamiento en memoria, un ámbito, un tipo y un identificador. Cada símbolo deberá tener esta información particular a cada uno.
- Un tipo posee información compartida por todo símbolo y construcción sintáctica de este tipo: su tamaño, equivalencia, coerción, inferencia y representación a bajo nivel.

Por este motivo, separando los símbolos de los tipos de un lenguaje y reutilizando los segundos, obtendremos un mayor aprovechamiento de la memoria y, como veremos en la sección 4.1, un mayor rendimiento. El patrón de diseño *Flyweight* ofrece un modelo para no repetir la creación de, en nuestro caso, expresiones de tipo. Una factoría de tipos será la encargada de obtener y crear (si fuese necesario) los tipos del sistema, sin necesidad de ir creando uno distinto cada vez que se requiera un tipo.

A modo de resumen, los siguientes criterios son los empleados para conseguir controlar la creación de expresiones de tipo por medio de la tabla de tipos (Figura 6):

- El método `getTipo` de `TablaTipos` es la única forma de acceder a un tipo. Este método recibe un parámetro cadena de caracteres con la representación textual de la expresión de tipos, cuya notación fue mostrada con anterioridad en este artículo. Este método se encargará de localizar el tipo asociado a dicha expresión de tipo (creándolo si fuese necesario), controlando que éste no esté duplicado. Devolverá un puntero a `ExpresionTipo`.
- Prohibición de creación y destrucción de tipos. Para controlar la no duplicidad de los tipos, la construcción y destrucción de instancias de tipos será prohibida. En el lenguaje C++, este objetivo se consigue declarando los constructores y destructores de todos los tipos como protegidos y haciendo que la tabla de tipos sea una clase amiga de cada clase tipo. En otros lenguajes como Java, la implementación se puede conseguir con la utilización de paquetes.

En el caso de que el lenguaje requiera destrucción explícita de objetos (como

C++), la tabla de tipos será la encargada de destruir todos los tipos cuando ésta vaya a ser liberada.

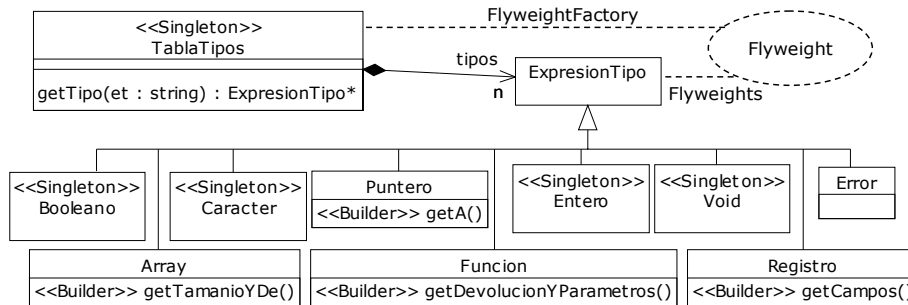


Fig. 6. Diagrama de clases de la tabla de tipos (Flyweight).

- Construcción heterogénea de expresiones de tipo. El proceso de crear el tipo apropiado a partir de la cadena que representa su expresión de tipo no es una tarea elemental. Es necesario analizar ésta y llamar a los constructores apropiados componiendo la estructura recursiva de objetos. Este proceso sólo se emplea en la construcción de las expresiones de tipo; la posterior utilización de las mismas se realizará mediante un puntero `ExpresionTipo`.

El proceso de creación de expresiones de tipos es llevado a cabo mediante la utilización de otro patrón de diseño denominado *Builder*, que permite separar la construcción de objetos complejos [19]. Las clases representativas de tipos compuestos poseerán un método de clase (*static*) encargado de obtener los tipos compuestos de esa expresión de tipo. Por ejemplo, el método `getDevolucionYParametros` de las funciones, analizará la cadena y devolverá los tipos de los que está compuesto. Este es un proceso mutuamente recursivo, es decir, estos métodos se apoyarán en el método `getTipo` que a su vez los ha invocado.

- Única instancia de los tipos simples. Para asegurar la creación única de los tipos simples –nótese que los compuestos no han de ser únicos, puesto que dependen de sus tipos agregados– se empleará un patrón de diseño *Singleton* que asegurará su unicidad [19]. La tabla de tipos también empleará este patrón de diseño.

Mediante el diseño presentado, las estructuras de objetos que representan los tipos del lenguaje dejarán de tener una estructura de árbol con instancias repetidas, pasando a estar representadas mediante Grafos Acíclicos Dirigidos (GAD).

4.1 Equivalencia Estructural mediante GADs

Puesto que la nueva estructura de objetos asegura la unicidad de las expresiones de tipo, la implementación de la equivalencia estructural de tipos (método `equivalente`) se simplifica en gran medida. No será necesario comprobar recursivamente los estados de los objetos `ExpresionTipo`, sino que nos apoyaremos en su identidad –en C++ se trata de comprobar direcciones de memoria en lugar de atributos, y en Java referencias en lugar del método `equals`. Dos expresiones de tipo serán equivalentes si son exactamente el mismo objeto. Este proceso requiere un tiempo de proceso ínfimo en relación con el necesario en las estructuras de árbol.

```
bool ExpresionTipo::equivalente(const ExpresionTipo *te) const {
    return this==te; }
```

Fig. 7. Nueva implementación de equivalencia estructural de tipos, mediante estructuras GAD.

5 Coerción de Tipos

Ante diversas construcciones sintácticas, los lenguajes de programación permiten realizar conversiones de los tipos inferidos de un modo implícito (realizadas por el procesador) o explícito (por el programador). Las conversiones implícitas reciben el nombre de coerción —en ocasiones también denominadas promoción. La mayoría de los lenguajes ofrecen coerción de tipos en aquellos contextos donde la conversión no supone pérdida alguna de información. La conversión implícita de números enteros donde se requiera un número real se da, por ejemplo, en los lenguajes C++, Pascal, Java y C#. En Modula-2 no existe coerción alguna.

Las conversiones implícitas entre tipos (coerciones) hacen que la definición de equivalencia se vea modificada en cierto modo. Si dos tipos no son equivalentes, puede ser que exista coerción entre ellos y, por tanto, sí sean finalmente equivalentes. Por tanto, crearemos un nuevo método en la jerarquía `igualQue`, cuyo código será el propio de la implementación de equivalencia estructural (el mostrado en la Figura 8). Otro método `coercion` devolverá, a partir del tipo pasado, la expresión de tipo resultante de la coerción del objeto implícito al tipo pasado como parámetro. Así, la nueva implementación del método `equivalente` pasará a ser:

```
const ExpresionTipo * ExpresionTipo::equivalente(const
ExpresionTipo *te) const {
    if (this->igualQue(te)) return this;
    return this->coercion(te); }
```

Fig. 8. Método `equivalente` teniendo en cuenta la coerción de tipos.

Gracias al encapsulamiento, la modificación de la implementación del método `equivalente` sin haber alterado su interfaz supone que el resto de la implementación del comprobador de tipos que hace uso de este método (como el operador de asignación o la invocación a una función) no sufra cambio alguno.

Las coerciones de tipos varían de un lenguaje a otro. Nuestro diseño hace que la modificación de las coerciones de un lenguaje a otro sea modelada de un modo sencillo. Así, el método de coerción se ubicará en la clase raíz de la jerarquía (`ExpresionTipo`) devolviendo siempre el valor nulo (0 en C++ o null en Java), indicando que no es posible hacer la coerción. Cada una de las expresiones de tipo deberá indicar en su método `coercion` aquellos tipos a los cuales puede promocionar. A modo de ejemplo, un valor lógico en C++ puede convertirse implícitamente a un entero, carácter o real. Adicionalmente, un array que colecciona elementos de tipo T promociona a un puntero que apunte a elementos de tipo T. Basándonos en este diseño, se pueden implementar estas conversiones en los métodos coerción de las clases `Booleano` y `Array`, tal y como se muestra en la Figura 9.

```
const ExpresionTipo *Booleano::coercion(const ExpresionTipo *te)
const {
    string ste=te->expresionTipo();
    if ( ste=="int" || ste=="char" || ste=="float" )
```

```

        return TablaTipos::getInstance().getTipo(ste);
    return 0;
}
const ExpresionTipo *Array::coercion(const ExpresionTipo *te)
const {
    const Puntero *puntero=dynamic_cast<const Puntero*>(te);
    if (puntero && de->igualQue(puntero->getA()) )
        return puntero;
    return 0; }

```

Fig. 9. Implementación de las coerciones de booleanos y arrays.

El resultado de este diseño es que, distribuyendo el código de coerción de tipos en las distintas clases, todo procesador de lenguaje realizará, si es necesario, una conversión implícita entre los tipos sin necesidad de programarlo explícitamente. La modificación de las reglas de promoción entre tipos es muy sencilla y directa. Nótese cómo el mecanismo de polimorfismo de los sistemas de tipos de los lenguajes con herencia es otra aplicación de este diseño –toda clase deberá permitir la coerción a sus superclases.

6 Polimorfismo

El polimorfismo (universal) es la propiedad de un lenguaje que hace que una parte de un programa pueda tener varios tipos [3]. Se produce cuando una función puede definirse sobre un conjunto determinado de tipos. Mediante el empleo de herencia, muchos lenguajes orientados a objetos ofrecen un polimorfismo restringido a una jerarquía de tipos, denominado polimorfismo de inclusión (o de subtipos) [21]. Los lenguajes orientados a objetos denominan al polimorfismo universal (también llamado paramétrico) como genericidad.

La principal aportación de los sistemas de tipos polimórficos es que añaden variables de tipo en las expresiones de tipo. Una variable de tipo representa cualquier aparición de una expresión de tipo como instancia de esta variable. Una variable de tipo sirve para representar cualquier tipo dentro de otra expresión de tipo. ML, Haskell y OCaml son lenguajes funcionales que ofrecen polimorfismo; C++ y Eiffel son lenguajes orientados a objetos que también lo implementan.

Un procesador de un lenguaje que ofrezca polimorfismo, deberá implementar un mecanismo de unificación: proceso de encontrar una sustitución para cada variable de tipo con otra expresión de tipo, acorde al empleo del operador o función polimórfico [22]. Si tenemos una función que recibe cualquier tipo y devuelve su dirección, ante la invocación de ésta con una variable entera, deberá ser capaz de inferir que dicha invocación posee el tipo puntero a entero. Esta inferencia se lleva a cabo mediante un algoritmo de unificación. A modo de ejemplo, la siguiente es una entrada válida para nuestro procesador:

```

1:      int vector[10];
2:      template <typename T> struct Registro { T field; };
3:      template <typename T> struct Lista {
4:          T uno;
5:          T* muchos; };
6:      template <typename T> T *f(T);
7:      template <typename T> T g(T,T);
8:      template <typename T1, typename T2> T2 *h(T1,T2);

```

```

9:      struct Registro<bool> registroBool;
10:     struct Lista<int> listaInt;
11:     int main() {
12:         f(3);                                Puntero(int)
13:         g(3, '3');                            int
14:         h(3.3, vector);                       Puntero(Array(10, int))
15:         f<int>(true);                          Puntero(int)

```

Fig. 10. Programa de entrada que utiliza polimorfismo.

El programa C++ anterior requiere un proceso de unificación en la inferencia de tipos. Por ejemplo, la línea 14 invoca a la función `h` con dos parámetros; es necesario inferir el tipo de ambos y unificar éstos a las variables de tipo `T1` y `T2`; una vez unificadas, se podrá saber que el tipo devuelto es un puntero a *array* de enteros (puntero a `T2`). Nótese cómo, además de unificación, las líneas 13 y 15 también emplean coerción de tipos.

En nuestro diseño, es necesario añadir un nuevo tipo de expresión de tipo: `VariableTipo`. Esta clase deberá tener un atributo que indique el nombre de la variable de tipo, necesario para la unificación. Adicionalmente, se añadirá un método `unificar` a todos los elementos de la jerarquía en el que se recibirá la expresión de tipo a unificar y la lista de sustituciones (asociación de variables de tipo a expresiones de tipo). La devolución de `unificar` es el tipo unificado.

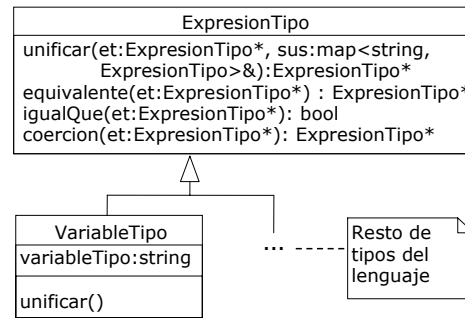


Fig. 11. Diagrama de clases que permite polimorfismo.

La implementación del método `unificar` de cada clase es sencilla:

- Los tipos simples, al no tener variables de tipo, devolverán una llamada al método `equivalente`. Ésta será la implementación por omisión de `ExpresionTipo`.
- Los tipos compuestos comprobarán que son el mismo constructor de tipo y llamarán al método `unificar` de cada uno de sus tipos base. La devolución será una expresión de tipo con el mismo constructor de tipo y las unificaciones de cada uno de sus tipos base.
- Finalmente, la variable de tipo buscará en las sustituciones (segundo parámetro) si su variable de tipo existe. En caso afirmativo devuelve la expresión de tipo asociada a ésta; si no, asigna el tipo a unificar (primer parámetro) a su variable de tipo y devuelve el mismo.

El comprobador de tipos invocará al método `unificar` cada vez que se llame a una función y cuando se trate de una unificación explícita por parte del programador (líneas 9, 10 y 15 de la Figura 9).

7 Conclusiones y Trabajo Futuro

En la actualidad existen diversos formalismos empleados para diseñar y verificar la validez de los sistemas de tipos de los lenguajes de programación. Las escasas herra-

mientas que utilizan dichas notaciones para implementar comprobadores de tipos poseen un conjunto de limitaciones como la dependencia de un lenguaje específico, dificultades en la depuración y carencias de interacción y reutilización de componentes software. Para paliar estos inconvenientes, hemos realizado un diseño de comprobadores de tipos para cualquier lenguaje de programación, aportando los siguientes beneficios:

1. No se emplean notaciones distintas a las utilizadas en desarrollo software. Aunque la descripción, validación e investigación relacionada con los sistemas de tipos es llevada a cabo a través de notaciones matemáticas, la implementación de un comprobador de tipos requiere codificar las reglas de inferencia y validez de los tipos, sin tener conocimientos de las notaciones que fueron empleadas para definir el sistema de tipos.
2. Independencia del lenguaje. Puesto que lo que hemos presentado es un diseño, éste podrá ser implementado en cualquier lenguaje de programación que soporte las abstracciones de clase, encapsulamiento, herencia y polimorfismo.
3. Comprensible, mantenible y reutilizable. La sencillez del diseño hace que sea fácil de depurar, modificar y extender. Si una vez desarrollado un comprobador de tipos para un lenguaje de programación fuese necesario implementar un comprobador para otro lenguaje, se podría reutilizar todo el código representante de aquellas partes comunes a ambos sistemas de tipos, introduciendo únicamente las nuevas características del segundo sistema de tipos.
4. El sistema puede ser empleado para implementar sistemas de tipos tanto estáticos como dinámicos. Determinados lenguajes realizan la comprobación de tipos total o parcialmente en tiempo de ejecución. Este diseño se puede emplear en ambos escenarios.
5. Independencia del lenguaje a procesar y de las herramientas. Este diseño para desarrollar comprobadores de tipos puede emplearse junto a cualquier tipo de herramientas de generación de compiladores. Adicionalmente, se basa en conceptos tales como expresión de tipo, tipo primitivo, constructor, equivalencia y coerción de tipos o polimorfismo; conceptos que no son propios de un paradigma concreto y que aparecen en sistemas de tipos de múltiples lenguajes.
6. Beneficios propios del empleo de patrones de diseño y orientación a objetos. La utilización de patrones de diseño orientados a objetos, altamente probados y revisados, hace que los diseños que los empleen correctamente adopten los beneficios de éstos [19].

El diseño aquí presentado ha sido empleado en el desarrollo de múltiples procesadores de lenguaje tales como la implementación de una máquina virtual [23] o un intérprete genérico independiente del lenguaje [24], así como su utilización a nivel docente e investigador [25]. El diseño se ha implementado en los lenguajes C++ y Java, junto con las herramientas Lex/Yacc, AntLR y JavaCC.

En la actualidad se está desarrollando una herramienta de generación de comprobadores de tipos capaz de, basándose en los diseños presentados, generar automáticamente comprobadores de tipos para lenguajes de programación (<http://www.di.uniovi.es/tys>).

Referencias

1. Whitehead, A. N., Russell, B.: Principia Mathematica. Cambridge University Press, Cambridge, MA (1910).
2. Pierce, B.C.: Types and Programming Languages. MIT Press (2002).
3. Cardelli, L.: Type Systems. The Computer Science and Engineering Handbook, CRC Press (2004).
4. Aho, A.V., Ullman, J.D., Sethi, R.: Compilers: Principles, Techniques, and Tools. Addison Wesley (1985).
5. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) (2001) 241–253.
6. Richardson, D.J.: TAOS: Testing with analysis and Oracle support. In: International Symposium on Software Testing and Analysis (1994) 138–153.
7. Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. ACM Transactions on Internet Technology (TOIT) 3(2) (2003) 117–148.
8. Abadi, M.: Security Protocols and Specifications. In: Foundations of Software Science and Computation Structures (FOSSACS) (1999) 1–13.
9. Pitts, A.M.: Operational Semantics and Program Equivalence. In: Applied Semantics, International Summer School (APPSEM). Lecture Notes in Computer Science, Tutorial, Vol. 2395, Springer-Verlag (2000) 378–412.
10. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2 (2). (1968) 127–145.
11. Levin, M.Y., Pierce, B.C.: TinkerType: a language for playing with formal systems. Journal of Functional Programming 13 (2) (2003) 295–316.
12. Rustan, K., Leino, M., Nelson, G., Saxe, J.B.: ESC/Java User's Manual. Technical Note 2000-002, Compaq Systems Research Center (2000).
13. Ball, T., Rajamani, S.K.: The SLAM Project: Debugging System Software via Static Analysis. In: Principles of Programming Languages (POPL) (2002) 1–3.
14. Holzmann, G.J., Holzmann, G.J.: The Spin Model Checker. Addison Wesley (2003).
15. Reps, T.W., Teitelbaum, T.: The Synthesizer Generator. In: Software Development Environments (SDE) (1984) 42–48.
16. Brand, M., et. al: The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In: Compiler Construction (CC), European Joint Conference on Theory and Practice of Software (ETAPS) (2001) 365–370.
17. Scott, M.L.: Programming Language Pragmatics. Morgan Kaufmann Publishers (2000).
18. Booch, G.: Object-Oriented Analysis and Design With Applications, 2nd Edition. Benjamin Cummings (1994).
29. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison Wesley (1995).
20. Welsh, J., Sneeringer, M.J., Hoare, C.A.R.: Ambiguities and Insecurities. Pascal, Software Practice and Experience 7 (6) (1977) 685–696.
21. Strachey, C.: Fundamental concepts in programming languages. Higher-Order and Symbolic Computation, Vol. 13 (2000) 11–49.
22. Milner, R.: A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences 17(3) (1978) 348–375.
23. Ortin, F., Cueva, J.M., Martínez, A.B.: The reflective nitrO abstract machine. ACM SIGPLAN Notices 38(6) (2003) 40–49.
24. Ortin, F., Cueva, J.M.: Dynamic Adaptation of Application Aspects. Elsevier Journal of Systems and Software 71(3) (2004) 229–243.
25. APLT, *Advanced Programming Languages and Technologies*, <http://www.di.uniovi.es/~labra/APLT>

Agregando orden superior al lenguaje Java. Una nueva perspectiva para el diseño de métodos

Josefina Llaberia Albarrán¹

Silvia Inés Clérici Martínez²

Facultad de Informática de Barcelona, Universidad Politécnica de Cataluña
Barcelona, España

¹e3445268@est.fib.upc.es, fllaberia@menta.net

²silvia@lsi.upc.es

Resumen FJava es una extensión del API de Java que el orden superior y la aplicación parcial de funciones típicas de lenguajes como Haskell. FJava representa una nueva alternativa para el diseño e implementación de los métodos de las clases, al incorporar a Java especificaciones algorítmicas de estilo funcional que son directamente traducibles a diseños orientados a objetos. En el paradigma de la orientación a objetos todo son objetos. Pero esta afirmación se refiere fundamentalmente a que todas las estructuras de datos lo son. Sin embargo los algoritmos no son tratados como objetos. FJava los convierte en objetos también, dando un paso más hacia el cierre del paradigma. Diseñadores o programadores haciendo uso exclusivo de la orientación a objetos disponen de las ventajas del paradigma funcional para el diseño de algoritmos, y las herramientas CASE habituales son capaces de generar automáticamente el código final.

1. Introducción

En la programación actual orientada a objetos [Mey99] podemos ver claramente la existencia de patrones algorítmicos que están siendo ignorados por los diseñadores, lo que obliga a los programadores a que escriban una y otra vez los mismos códigos. La programación de los métodos de las clases es básicamente imperativa, sin primitivas algorítmicas para los esquemas más frecuentes. Así, cada caso se resuelve de manera ad-hoc mediante asignaciones, ciclos y condicionales dificultando la verificación sistemática.

Además, los mecanismos de diseño de métodos como los diagramas de secuencia o colaboración [BRJ99] no bastan para describir el comportamiento esperado con un nivel de abstracción suficiente. Suelen complementarse con anotaciones en un lenguaje formal como OCL [OCL] o en lenguaje natural, y las herramientas CASE sólo generan parte del código.

¹ Este trabajo es el resultado de un proyecto final de carrera de la Facultad de Informática de Barcelona de la UPC [Lla04]. La participación en Prole'04 ha sido parcialmente financiada por la institución

Por el contrario, la programación en los lenguajes funcionales de tercera generación como Miranda [Mir][Tur86] o Haskell [Bir99][Has] se basa precisamente en el uso de pequeños patrones algorítmicos provistos por el lenguaje como funciones predefinidas para desarrollar algoritmos más o menos complejos.

Por ejemplo, si se desea obtener el mínimo de una clase ordenable en Haskell y en Java [Java][Sun98] se podría escribir para cada uno de los lenguajes un código como el siguiente,

(Haskell)

(Java)

```
foldl1  min lista  res = (Comparable) lista.next();

while(lista.hasNext()){

    elem = (Comparable) lista.next();

    if(res.compareTo(elem) == 1)

        res = (Comparable) elem;}
```

La función *foldl1* resume un esquema de programa aplicable cada vez que el resultado es un cálculo por acumulación que se obtiene aplicando una función de dos parámetros, por ejemplo la función suma para calcular un sumatorio. La función se va aplicando a cada uno de los elementos y al resultado acumulado de los restantes. En este caso el mínimo se obtiene empleando la función *min* (también predefinida) que devuelve el menor de sus dos parámetros.

El uso de este tipo de funciones predefinidas es la idea fundamental que se desea trasladar a Java. Los esquemas algorítmicos más potentes y útiles son los que permiten el paso de funciones como parámetros y el retorno de funciones como resultado, es decir el orden superior. El objetivo es que los programadores dispongan de las ventajas del paradigma funcional para el diseño de algoritmos sin necesidad de aprender el uso de un lenguaje funcional. Por ello FJava intenta no ser una simulación de Haskell en Java, como ocurre en cierta medida en otros enfoques.

En FJava los algoritmos son objetos. Han debido mirarse entonces bajo esta perspectiva y trasladar a Java junto con estas primitivas todo lo que implica el uso pleno del orden superior, es decir el paso de funciones sin evaluar, la aplicación parcial de funciones, la asimilación de métodos Java como funciones, cuestiones de transparencia referencial y de control de tipos.

En la sección 2 se presentan los trabajos más directamente relacionados. La sección 3 describe las principales facilidades de FJava y se ilustra su uso con algunos ejemplos. En la sección 4 se analiza la exportación de construcciones netamente funcionales al paradigma de la orientación a objetos, se justifican las principales decisiones de diseño e implementación y se presentan otras facilidades más avanzadas de FJava. La sección 5 presenta un caso de uso completo de FJava para el diseño y la generación de código mediante una herramienta CASE. La sección 6 resume las aportaciones del artículo y de FJava y discute posibles mejoras o extensiones.

2. Trabajos Relacionados

Martin Odersky y Philip Wadler presentaron en 1997 un nuevo lenguaje llamado Pizza [OW97]. Pizza es una extensión de Java que incorpora polimorfismo paramétrico basado en la genericidad, tipos algebraicos y funciones de orden superior. Está implementado mediante un traductor a Java, lo que lo hace muy completo al incorporarle un potente comprobador de tipos que FJava no proporciona.

La idea de Pizza es que los métodos forman parte de los objetos y por tanto pueden verse como objetos. La solución que plantea para la consecución del orden superior se basa en el uso de un tipo nuevo, diferente a los objetos y los primitivos Java, que parecen ser punteros a métodos. FJava propone una solución diferente, tratando los métodos como objetos realmente. Por otro lado, Pizza no parece dar soporte a la aplicación parcial, lo que en FJava se ha considerado un objetivo esencial para el diseño de algoritmos usando esquemas algorítmicos predefinidos. Los objetivos y motivaciones de Pizza parecen algo distintos y en algunos aspectos mucho más amplios que los de FJava. Pero creemos que la falta de aplicación parcial es una restricción importante que FJava consigue evitar.

En 2003 David R Naugler [Nau03] presenta un estilo de programación en Java que simula el del paradigma funcional de Haskell. A partir del uso de interfaces y clases internas Java se agrega a éste lenguaje funciones de orden superior. La aplicación parcial se logra a partir de definiciones de métodos para cada una de las posibilidades que ofrece una función curryficada. Esta forma de trabajo no convierte realmente los algoritmos en objetos, ya que siguen siendo métodos Java y la forma de definición de funciones de más de un parámetro es demasiado compleja. Además, tal y como afirma el autor, su propuesta representa un gran paso para un diseñador o programador Java, que necesitará aprender previamente el manejo de un lenguaje funcional para poder usar Java de forma funcional.

Existe también una publicación electrónica muy reciente [Bri04] sin referencias a haber sido publicada en ningún medio distinto a la página del autor. Parece hacer una mejora del trabajo de Naugler [Nau03] partiendo de ideas similares pero agregando las ventajas del uso de la genericidad en Java. Esta publicación parece el inicio de una propuesta para convertir Java en Haskell. Y es precisamente este enfoque el que hace que, nuevamente, sus soluciones no permitan el trabajo con algoritmos como objetos. El autor apunta en su trabajo los posibles problemas que plantea su solución como son el hecho de tener que definir un método estático por cada posible parámetro a ser instanciado para cada clase de función y que los métodos Java no son funciones. Cabe destacar que para el trabajo con listas también ha escogido la interfaz *Iterator* de Java, como se hace en FJava.

Finalmente, se ha encontrado, también en Internet, un software [Kat04] sin más referencias de publicación que la web de la empresa. Ese software aporta la ejecución de métodos pasados como argumento sin usar el API de reflexión. Es similar a lo aportado por FJava para el trabajo con métodos Java, aunque FJava no necesita de la distinción entre un evaluador para funciones parcialmente instanciadas o totalmente instanciadas, sino que es transparente al usuario. Parece soportar orden superior y curryficación pero no tiene un conjunto de funciones predefinidas equivalentes a las de Haskell. Al no tratarse de un software libre no ha sido posible encontrar información demasiado concreta sobre su implementación.

3. FJava

FJava se basa en la idea de que el concepto de objeto es lo suficientemente amplio como para permitir no sólo que las estructuras de datos puedan ser vistas como objetos sino también los algoritmos. En FJava los esquemas algorítmicos definen clases de funciones. Al declarar una clase para cada esquema sus instancias serán concreciones del algoritmo para determinados argumentos y valores. FJava está implementado mediante un package, también llamado FJava, que extiende el API predefinido con una colección de clases apropiadas para el manejo de funciones de orden superior similares a las de lenguajes funcionales como Haskell.

La clase principal es una clase abstracta llamada *FFunction* de la cual son herederas todas las clases de funciones predefinidas que provee FJava. El repertorio de clases *FFunction* predefinidas incluye versiones equivalentes de las funciones más usuales en Miranda o Haskell, y de los operadores Java de los tipos básicos.

Por otro lado, si es necesario, los métodos pueden ser vistos también como *FFunction*, para ello FJava proporciona la clase *MethodAsFFunction*.

Además, el package FJava proporciona un control de tipos en tiempo de ejecución para los objetos *FFunction* permitiendo la generación de excepciones de la clase *FFunctionException* si los tipos de los argumentos o resultados reales no se corresponden con los formales.

3.1 La Clase FFunction

Esta clase, que es el núcleo de FJava implementa una idea muy simple e intuitiva que es el punto de partida del enfoque: la visión de función como representante abstracto de una clase de objetos de características afines (métodos, funciones, operadores, esquemas algorítmicos). Un objeto función tiene un cierto número de argumentos que deben ser proporcionados en cierto orden, debe poderse aplicar sobre ellos, al ser aplicada producirá un resultado (o efecto), y ese resultado se obtiene aplicando una cierta regla de cálculo o algoritmo.

Los métodos de mayor interés de la clase *FFunction* son entonces los siguientes:

- El método *setArgument(Object o, int i)* instancia el objeto pasado como argumento en la posición de argumento indicada.
- El método *eval()* ejecuta la evaluación del algoritmo sobre los argumentos de la función sobre la que se ha invocado este método. Retorna un *Object* como resultado. Este método hace uso del método *algorithm()*.
- El método *algorithm()* es abstracto y debe ser definido por las clases herederas con el algoritmo a implementar por la función.

3.2 FFunctions Predefinidas

FJava incluye un repertorio básico similar, aunque un poco más restringido, que el del preludio de Haskell. Este conjunto puede ampliarse, y así se recomienda, con los esquemas propios del dominio de la aplicación que se esté desarrollando.

El listado de funciones predefinidas de FJava es el siguiente:

- FFunctions de operadores básicos Java. Son clases envolventes de operadores para los tipos básicos: *Sum*, *Subs*, *Prod*, *Div*, *IsMoreThan*, *IsLessThan*, *Not*, *And*, *Or*
- FFunctions polimórficas: *Id*, *Max2*, *Min2*, *Const*
- FFunctions de primer orden sobre colecciones de objetos: *Length*, *Head*, *Tail*, *Init*, *Last*, *Prefix*, *Postfix*, *Take*, *Drop*, *Reverse*, *Member*, *Index*, *Concat*, *Merge*, *Sort*, *Mkset*, *Repeat*
- FFunctions de orden superior sobre colecciones de objetos: *TakeWhile*, *DropWhile*, *Filter*, *Folder*, *Map*, *Zipwith* ó *Map2*, *Fnot*

3.3 Listas como Recorridos de Colecciones de Objetos

Las FFunctions equivalentes a las funciones sobre listas de Haskell o Miranda trabajan sobre recorridos de colecciones de objetos, ya sean estas vectores, listas, árboles o incluso ficheros o combinaciones de colecciones. La única condición es que la estructura ofrezca un iterador para su recorrido que será el equivalente al parámetro lista de Haskell. Por ejemplo, si en la aplicación existe un árbol de búsqueda referenciado por la variable ‘tree’, podemos obtener un iterador ‘it’ para recorrer sus elementos usando el método ‘*iterator*’ que tienen todas las colecciones Java.

En este caso los valores conseguidos durante el recorrido del árbol estarán ordenados ya que es el recorrido estándar ofrecido por el iterador de la clase *TreeSet* de Java (árbol de búsqueda en Java). Este iterador ‘it’ puede ser pasado como argumento a la FFunction que interese en un momento dado. En la sección siguiente se muestra un ejemplo de aplicación.

3.4 Orden superior y Aplicación Parcial

El último grupo de FFunctions de la sección 3.2. son esquemas algorítmicos de orden superior con soporte a la aplicación parcial que operan sobre colecciones de objetos. Por ejemplo *Map*, que es una clase heredera de la clase abstracta *FFunction*, representa el esquema algorítmico de aplicar una función o método, su primer argumento, a todos los elementos de una colección de objetos cuyo iterador es su segundo argumento. Tras crear un objeto de tipo *Map* deberán aplicarse sus argumentos para especializarlo en un algoritmo concreto. Por ejemplo, suponiendo que los elementos del árbol ‘tree’ del ejemplo anterior fueran enteros, una función *triple* que triplique valores enteros y el iterador ‘it’ podrían ser argumentos válidos. En este caso el nuevo objeto de tipo *Map* representa el algoritmo que triplica los valores obtenidos al recorrer el árbol ‘tree’.

FJava permite aplicación parcial, y de forma análoga a como se haría en Haskell, la función *triple* es una instancia de la clase *Prod* (clase representante del operador producto que tiene dos argumentos numéricos) parcialmente aplicada a un objeto *Integer* de valor 3 en su primer argumento. Al ser aplicada por *Map* a los elementos proporcionados por ‘it’ éstos serán el segundo argumento de *triple*.

Sin embargo, a diferencia de lo que ocurre en Haskell en que existe un orden fijo de aplicación, en FJava la aplicación de argumentos mediante el método *setArgument* requiere que se indique, además del objeto, la posición de argumento que ocupa de entre todos los posibles de la función (ver discusión en la sección 4.1).

La codificación en FJava para este ejemplo será la siguiente:

```
Map m = new Map(22);    Iterator it = tree.iterator();

FFunction triple = new Prod();

triple.setArgument(new Integer(3), 1);

m.setArgument(triple, 1).setArgument(it, 2);
```

Para la evaluación del algoritmo sólo hay que ejecutar su método *eval*.

```
Iterator res = (Iterator) m.eval();
```

La variable ‘res’ es un iterador creado por m que podría volver a usarse como argumento de otra FFunction. Por ejemplo de un *Filter* que se quede sólo con los múltiplos de 5 devolviendo un nuevo iterador, y así sucesivamente.

Nótese que por claridad de código se usan más variables de las requeridas. Realmente no es necesario dar un nombre a *triple*, tal como no lo es en Haskell donde se usaría directamente la función (*3). Un código más al estilo funcional sería el siguiente,

```
Map m = new Map(2);

Iterator res=(Iterator) m.setArgument(new
Prod()).setArgument(new Integer(3),
1),1).setArgument(tree.iterator(), 2).eval();
```

Una FFunction totalmente instanciada puede pasarse como parámetro de otra FFunction y de forma automática es substituida por el valor que representa, es decir, la evaluación de los parámetros funcionales no ha de ser explícitamente indicada.

3.5 Métodos Java como FFunctions

Los esquemas algorítmicos introducidos por FJava deben convivir con muchos métodos que han sido definidos dentro de clases del propio API de Java o del dominio de una aplicación. FJava permite convertir cualquier método Java en una FFunction en una sola línea de código. De este modo el método puede ser pasado como argumento o devuelto como resultado de cualquier FFunction. Para ello se usa la clase *MethodAsFFunction*, heredera de FFunction, desarrollada para crear objetos envolventes de métodos Java. Es una clase abstracta con tres subclases, una cada para tipo de método Java: constructor (*ConstructorMethodAsFFunction*), método de instancia (*InstanceMethodAsFFunction*) y método de clase o estático (*StaticMethodAsFFunction*).

A continuación se presenta un ejemplo sobre una pila de elementos de tipo *Saludo* cuyas instancias representan las formas de saludo en un idioma dado según la hora del día. Lo que se desea hacer es obtener la frase de buenos días (saludo matutino) de

² El argumento para el constructor de la clase Map con valor 2 indica que los elementos proporcionados por el iterador 'it' serán el segundo argumento de la función 'triple'. (Ver aclaración en la sección 4.1)

cada idioma. El método *matutino* de la clase *Saludo* no tiene ningún argumento y nos devuelve el saludo matutino del objeto sobre el que se ejecuta. La clase *Saludo* se encuentra en el package *idiomas*. Las instancias de *Saludo* para los idiomas español, inglés, alemán y japonés se encuentran guardadas en la variable ‘pila’ de tipo *java.util.Stack*.

```
String argTypes [] = {};
```

```
InstanceMethodAsFFunction fmatutino=new
InstanceMethodAsFFunction("idiomas.Saludo","matutino",a
rgTypes);
```

```
Map map = new Map(1);
```

```
Iterator res=(Iterator) map.setArgument(fmatutino,
1).setArgument(pila.iterator(), 2).eval();
```

La variable *res* contiene ahora un recorrido sobre los valores “Buenos días”, “Good morning”, “guten Morgen” y “ohayou gozaimasu”.

4. Decisiones de Diseño e Implementación

Como ya se ha dicho el principal objetivo de diseño en FJava ha sido trasladar a Java las ventajas de los lenguajes funcionales en cuanto a la potencia algorítmica de sus primitivas, pero hacerlo de manera totalmente consistente con el enfoque orientado a objetos y su metodología habitual de trabajo. El paradigma funcional ha sido utilizado solamente como fuente de ideas. En esta sección se analizan las características de los lenguajes funcionales más directamente relacionadas con el uso pleno del orden superior, en particular la aplicación parcial y cómo encontrar un equivalente en Java para ellas. Se analiza hasta que punto es posible o conveniente incorporar una cierta característica netamente funcional y las decisiones de diseño y de implementación que se tomaron para conseguirlo. Se detallan algo más algunas cuestiones de implementación esbozadas en la sección anterior y presentan otras facilidades más avanzadas.

4.1 Orden Superior y Aplicación Parcial

En los lenguajes funcionales el orden superior y la aplicación parcial están fuertemente relacionados. Las funciones son *curryficadas* y por lo tanto naturalmente aplicables parcialmente. Esto es así porque una función de *n* parámetros en realidad sólo tiene como argumento al primero de ellos, y al ser aplicada a éste devuelve una segunda función que tiene como argumento al segundo, y así sucesivamente. Es decir que toda función de más de un parámetro es de orden superior pues devuelve una función. Por otro lado la curryficación fija el orden de aplicación, si se le proporcionan *k* de sus *n* argumentos está implícito que estos corresponden a los *k* primeros parámetros. Por eso Haskell ofrece una función predefinida *flip* para invertir el orden de aplicación, pero sólo resulta útil para funciones de dos parámetros. En el

caso general, si se desea aplicar parcialmente una función de varios parámetros a cualquier combinación de ellos en orden arbitrario se ha de definir explícitamente una función equivalente en cada caso.

Al analizar cómo llevar a FJava el orden superior en toda su potencia se vió que no podía dejarse de lado la aplicación parcial, pero que fijar el orden de aplicación quitaba flexibilidad sin ofrecer a cambio una ventaja apreciable desde el punto de vista del usuario. El estilo de código en ambos paradigmas es radicalmente diferente y forzar un estilo demasiado funcional no agrega claridad, al contrario, como se muestra al final de la sección 3.4. Se decidió entonces que los objetos de tipo `FFunction` soportasen aplicación parcial en cualquiera de sus parámetros, independientemente del orden establecido en su signatura. Como ya se ha visto, en el método *set-argument* hay que indicar no sólo el argumento sino también qué número de argumento es.

La solución adoptada en [Nau03] y [Bri04] para la aplicación parcial (en ambos trabajos el orden de aplicación es fijo) es tener un método *eval* por cada número de parámetros. La adoptada en FJava es más simple y a la vez permite que la aplicación se haga en cualquier orden, se usa un sólo método *eval* sin parámetros. La idea base la da la currficación, una función al aplicarse devuelve una función idéntica a ella excepto en ese parámetro ya aplicado. Pero esto obliga a hacer clonables a todos los objetos `FFunction`.

Por otro lado la independencia en el orden de aplicación obliga también a considerar en las funciones de orden superior qué argumento o argumentos se deben aplicar en sus parámetros funcionales. Esa información es un parámetro del constructor de la clase y varía en cada caso. Como se vió en el ejemplo 3.4 el argumento para el constructor de la clase `Map` con valor 2 indica que los elementos recorridos por el iterador 'it' serán el segundo argumento de la función triple.

Si en vez de triple que es una función de 2 parámetros ya aplicada al primero, se hubiese utilizado por ejemplo la versión `FFunction` del ++, el argumento del constructor de clase debería haber sido 1. Si por el contrario la función tuviera más de un argumento aún sin aplicar, el parámetro del constructor diría cuál de ellos es el que se aplicaría al elemento. En ese caso el resultado de la aplicación sería una copia de la misma `FFunction` aplicada todavía parcialmente pero a un argumento más.

4.2 Transparencia Referencial en FJava. Funciones vs. Objetos Funcionales

En el paradigma funcional evaluación significa manipulación simbólica de una expresión y el resultado dependerá a lo sumo de la política de evaluación empleada. Si la expresión es la aplicación de una función el resultado será siempre el mismo si se la aplic a los mismos argumentos, y su evaluación no habrá producido ningún efecto adicional. Nada de esto se puede mantener para una `FFunction`. El hecho de que los métodos de una clase tengan visibilidad y acceso libre a todos los atributos declarados en ella, y el uso de técnicas como la definición de atributos estáticos dentro de las clases hace del "trabajo con efectos laterales" la manera natural de trabajar en Java. Es decir, la transparencia referencial es difícilmente trasladable a Java y además su filosofía no concuerda con la filosofía global del paradigma.

Sin embargo la clase `FFunction` provee tres métodos para el usuario avanzado de FJava que desee reducir los posibles efectos laterales en algún caso:

- *setArgumentTransp*: que permite la aplicación de funciones retornando una copia de la función original. El argumento se aplica sobre la copia retornada.
- *cleanArguments()*: elimina la lista de argumentos reales de una función
- *cleanArgument(int i)*: elimina el argumento real referenciado por la posición 'i'.

4.3 Esquemas para Colecciones de Objetos

Como ya se ha visto anteriormente las funciones que implementan algoritmos sobre colecciones de objetos en FJava usan la interfaz *Iterator* como tipo base de entrada de datos y salida de resultados. De este modo su aplicabilidad incluye no sólo colecciones que ofrezca el propio API de Java, sino cualquier estructura de datos recorrible por un iterador, como pueden ser las bases de datos del *package* JDBC de Java, ficheros, o cualquier combinación de estructuras que se implemente.

Además la visión de un iterador como lista abstracta resulta natural y en cierto sentido más consistente con la visión funcional (más libre de efectos laterales). Porque un iterador no es una estructura de almacenamiento sino de recorrido. Si el resultado de una función sobre colecciones de objetos es un iterador será responsabilidad del usuario diseñador o programador el guardar esos datos en una estructura adecuada para la aplicación que esté desarrollando. Por otro lado un iterador sólo puede recorrerse una vez. Si fuera necesario pasar como parámetro un mismo iterador a dos funciones distintas, o como argumentos distintos de una misma función, se debe usar la clase *IteratorClone* que ha sido creada para generar clonadores de iteradores. Conviene aclarar también que el efecto de aplicar una instancia de *Map* a una colección mediante su iterador, por más que no altere la estructura en sí no siempre es “inocuo” para los elementos, como sucede en los ejemplos anteriores y a veces justamente se pretende que no lo sea.

Si en el ejemplo de la sección 3.4 en lugar de enteros se tuviera un árbol de Alumnos a los que se desea incrementar su nota de expediente en 0.25 puntos, se tendría que el *Map* resultante de aplicar el método *incNota* sobre cada alumno dejaría modificados los elementos del árbol con el valor de nota incrementado. El método *incNota* se encuentra en la clase *Alumno* del *package* *academico* e incrementa la nota de un alumno en el valor de tipo *double* que se le pasa como argumento.

El código resultante sería el siguiente,

```
Map map = new Map(1);String argTypes[] = {"double"};

InstanceMethodAsFFunction f=new
InstanceMethodAsFFunction( "academico.Alumno" ,
"incNota", argTypes);

f.setArgument(0.25, 2);

map.setArgument(f,1).setArgument(tree.iterator(),
2).eval();
```

En cuanto al polimorfismo, las funciones de trabajo con colecciones se han implementado usando la clase *Object* como tipo de sus elementos. Esto supone código extra de conversiones de tipos que Java 1.5 tal vez nos podría evitar en

algunos casos. Esta versión de FJava se ha implementado cuando todavía estaba en su fase Beta de desarrollo.

4.4 Agregación de Nuevos Esquemas Algorítmicos

La clase `FFunction` proporciona los medios para crear esquemas algorítmicos propios que se adecuen a las necesidades del desarrollo de aplicaciones dentro de un dominio concreto. Para la creación de `FFunctions` propias sólo hay que crear una clase que extienda a la clase `FFunction`, programar el método abstracto `algorithm()` para el algoritmo que calcula la función, y el método también abstracto `createInstance()` para llamar al constructor de la clase con los parámetros que permitan crear un objeto clon.

4.5 Polimorfismo. Implementación del control de tipos

Java es un lenguaje fuertemente tipificado pero que no posee inferencia de tipos. FJava es por tanto también fuertemente tipificado. Cuando en FJava se trabaja con *Object* para simular el polimorfismo paramétrico se relaja el control de tipos, y al evaluar una función es necesario convertir los argumentos a los tipos esperados. Para poder controlar los posibles errores de conversiones de tipos entre los parámetros formales y reales de una función, incluyendo el de retorno, se ha dotado a FJava de un conjunto de clases para el manejo de todas las posibles excepciones relacionadas con el control de tipos. Es decir, el control de tipos en las funciones se hace en tiempo de ejecución y puede ser activado por el usuario de una clase para un objeto en concreto mediante el método `enableTypeChecking(String [] argTypes, String resType)` de la clase `FFunction`. El usuario deberá indicar los nombres de tipos de los parámetros y del resultado que deberán tener los argumentos reales.

5. Un Caso de Estudio con FJava

El siguiente ejemplo ilustra la diferencia de enfoque que introduce el uso de FJava en el diseño e implementación de sistemas de información desarrollados en un marco orientado a objetos.

Supongamos que una empresa de selección de personal desea seleccionar a los ‘n’ mejores candidatos de una base de datos para realizarles una entrevista. Cada candidato tiene un currículum CV y cada CV tiene un valor numérico calculable al que llamaremos nota. Además, se cuenta con un CV mínimo que el candidato ha de superar para ser entrevistado. Como resultado del análisis y posterior diseño se ha obtenido el diagrama de clases de la figura *Fig.1*. Los métodos involucrados en el algoritmo de selección son ‘seleccMejorCandidatos’ y ‘nota’. El comportamiento del primero de ellos se puede especificar así:

1. Calcula nota de cada CV para cada candidato
2. Descarta los que no superen la nota mínima
3. Ordena en orden descendente
4. Toma los ‘n’ primeros CV

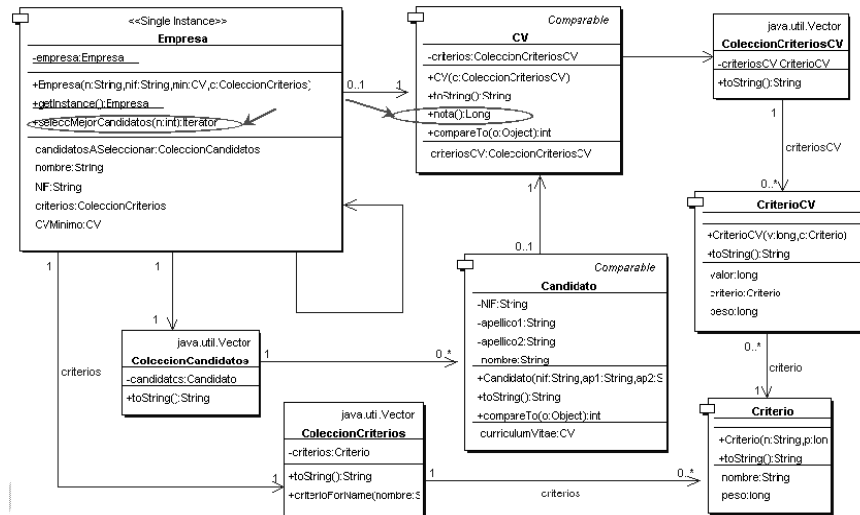


Fig. 1. Diagrama de clases del diseño.

El comportamiento de 'seleccMejorCandidatos' traducido a funciones de FJava es el diseño algorítmico siguiente:

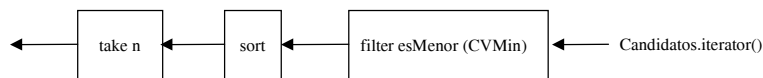


Fig. 2. Diagrama de especificación algorítmica

En el diseño orientado a objetos, para describir el comportamiento de los métodos se utilizan los diagramas de secuencia. Una solución en el estilo clásico quedaría expresada mediante el diagrama de la figura Fig.3. Ha sido editado utilizando Together [Tog], una herramienta CASE capaz de generar código Java. Como puede verse, la correspondencia del comportamiento especificado en lenguaje natural con el diseño en términos de bucles, condicionales y asignaciones no es inmediata. Por otro lado, el código generado tampoco es directamente ejecutable. Para llegar a una compilación satisfactoria se han debido hacer algunos retoques y comprobaciones.

En cambio, usando FJava la traducción de la especificación en lenguaje natural a la algorítmica y de esta al diseño orientado a objetos es prácticamente directa, al existir un objeto función por cada esquema algorítmico en dicha especificación. Sólo deben asignárseles los argumentos indicados en la especificación y evaluar la función principal (*Take* en este caso). La figura Fig.4 muestra el diagrama de secuencia resultante. Y el código, que fue generado automáticamente por la herramienta CASE, es directamente ejecutable.

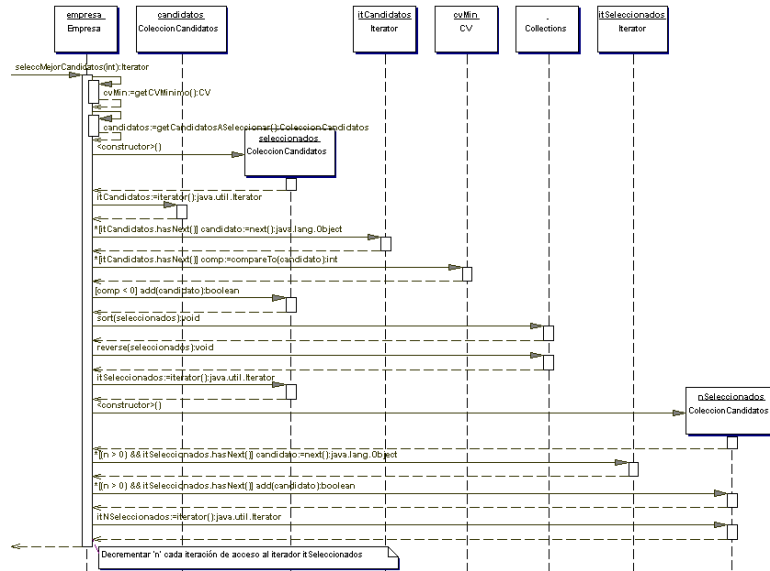


Fig. 3. Diagrama de secuencia usual

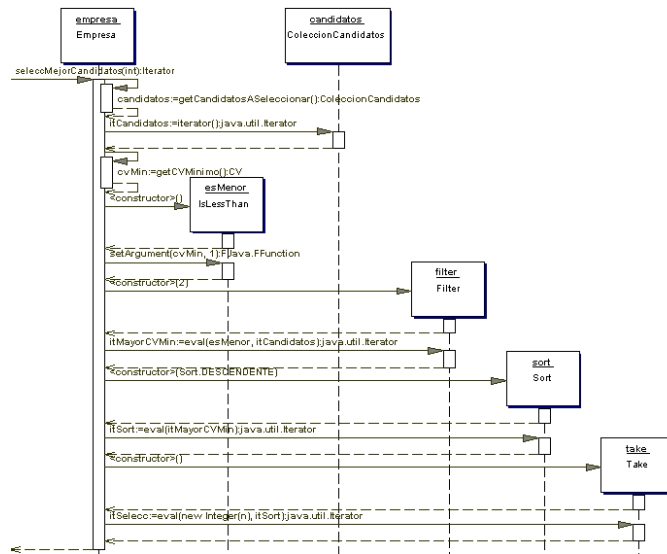


Fig. 4. Diagrama de secuencia usando FJava

```

ColeccionCandidatos candidatos =
this.getCandidatosASeleccionar();

java.util.Iterator itCandidatos=candidatos.iterator();

CV cvMin = this.getCVMinimo();

```

```

IsLessThan esMenor = new IsLessThan();

FFunction some_FFFunction=esMenor.setArgument(cvMin, 1);

Filter filter = new Filter(2);

Iterator itMayorCVMin = filter.eval(esMenor,
itCandidatos);

Sort sort = new Sort(Sort.DECENDENTE);

Iterator itSort = sort.eval(itMayorCVMin);

Take take = new Take();

Iterator itSelecc = take.eval(new Integer(n), itSort);

return itSelecc;

```

Nótese que no hay bucles, sólo creación de funciones, búsqueda de sus argumentos, aplicación de los mismos y evaluación de la función principal. La algorítmica de bajo nivel queda encerrada en cada una de la funciones predefinidas usadas.

6. Conclusiones

Se ha presentado una solución completa para poder utilizar en Java la potencia algorítmica de las primitivas de los lenguajes funcionales como Haskell. El enfoque es novedoso y totalmente consistente con el paradigma de la orientación a objetos ya que considera a los algoritmos también como objetos. La idea clave en FJava es la noción de función como representante abstracto de objetos en algún sentido afines como lo son las funciones, los métodos, los operadores y los esquemas algorítmicos. Se han discutido las características principales del paradigma funcional, en qué medida pueden compatibilizarse con las del paradigma de la orientación a objetos, y en particular trasladarse al lenguaje Java. Se ha presentado una comparación del enfoque con otros de objetivos similares y se han justificado las principales decisiones de diseño e implementación. Se ha ilustrado la utilización de FJava en el marco de un diseño orientado a objetos y su integración con las herramientas CASE habituales.

Las aportaciones de FJava son diversas.

- Es una solución de aplicación directa por estar implementado como un package.
- Al ser totalmente compatible con la forma habitual de programación en Java , sólo que de manera más abstracta, no exige un esfuerzo inicial de aprendizaje por parte del usuario más alto que el uso de cualquier otro paquete
- El diseño de un método puede usarse como especificación debido al alto nivel de abstracción ofrecido.
- El uso de herramientas CASE hace desaparecer la etapa de implementación en muchos casos.

- El código generado es totalmente plano sin bucles ni muchas ramas condicionales. La semántica del algoritmo es más clara y fácil de verificar.
- La posibilidad de agregar nuevos esquemas algorítmicos permite extender los esquemas básicos a otros más complejos o de propósito más específico.

Sin embargo el código FJava generado a partir de una especificación algorítmica no es del todo eficiente respecto de algoritmos análogos escritos al estilo imperativo del Java tradicional. Esto se debe a que normalmente en una especificación no se tiene en cuenta la optimización en el número de recorridos de las colecciones de objetos implicadas. Sin embargo, analizando las estructuras de diseño y código generado parece posible la creación de un optimizador de código FJava.

Una posible continuación del trabajo sería la exploración de esa línea. La evaluación perezosa es una mejora de eficiencia que también resulta interesante considerar. Pero habría que analizar cuidadosamente qué implicaciones tendría emplear esta forma de evaluación sólo para algunas funciones al integrarse éstas en un contexto impaciente. Por último lo más inmediato sería analizar qué simplificaciones o mejoras se podrían introducir en la implementación de FJava con las facilidades de la nueva versión de Java 1.5

7. Referencias

- [Bir99]: Richard Bird. Introducción a la programación funcional usando Haskell. Prentice-Hall. Segunda edición. 1999.
- [Bri04]: Björn Bringert. Higher Order Java. Internet: <http://www.dtek.chalmers.se/~d00bring/hoj>. Mayo 2004.
- [BRJ99]: Grady Booch, James Rumbaugh e Ivar Jacobson. El lenguaje Unificado de Modelado. Addison-Wesley. 1999.
- [Has]: The Haskell Home Page. <http://www.haskell.org>.
- [Java]: Reference API Specifications of Java. <http://java.sun.com/reference/api/index.html>.
- [Kat04]: Kataba Software. <http://www.kataba.com/func/Func.asp>. Mayo 2004.
- [Lla04]: Josefina Llaberia Albarrán. Agregando orden superior al lenguaje Java. PFC Ingeniería Informática. Facultad de Informática de Barcelona UPC Bajo la dirección de Silvia Cléricki Dto.LSI UPC Julio 2004.
- [Mey99]: Bertrand Meyer. Construcción de Software Orientado a Objetos. Prentice-Hall. Segunda edición. 1999.
- [Mir]: The Miranda Programming Language. <http://www.engin.umd.umich.edu/CIS/course.des/cis400/miranda/miranda.html>
- [Nau03]: David R Naugler (Southeast Missouri State University). Functional programming in Java. Consortium for Computing Sciences in Colleges. 2003.
- [OCL]: UML 2.0 OCL Final Adopted specification of Object Constraint Language <http://www.omg.org/cgi-bin/doc?ptc/2003-10-14>
- [OW97]: Martin Odersky (University of Karlsruhe) y Philip Wadler (University of Glasgow). Pizza into Java: Translating theory into practice. The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Enero 1997.
- [Sun98]: Sun Microsystems. Programación de Java.. Enterprise Services August 1998, Revision B.
- [Tog]: Together 4.2. TogetherSoft Corporation. 2000. <http://www.borland.com/together/>
- [Tur86]: An overview of Miranda. D.A. Turner. SIGPLAN Notices, vol 21, no 12, pp 158-166. December 1986.

A Lightweight Approach to Program Specialization^{*}

Claudio Ochoa, Josep Silva, and Germán Vidal

DSIC, Tech. University of Valencia, Camino de Vera s/n, E-46022 Valencia, Spain.
{cochoa, jsilva, gvidal}@dsic.upv.es

Abstract. Within the imperative programming paradigm, program slicing has been widely used as a basis to solve many software engineering problems, like debugging, testing, differencing, specialization, and merging. In this work, we present a lightweight approach to program specialization of lazy functional logic programs which is based on dynamic slicing. The kind of specialization performed by our approach cannot be achieved with other related techniques like partial evaluation.

1 Introduction

Program slicing is a method for decomposing programs by analyzing their data and control flow. It was first proposed as a debugging tool to allow a better understanding of the portion of code which revealed an error. Since this concept was originally introduced by Weiser [24]—in the context of imperative programs—it has been successfully applied to a wide variety of software engineering tasks (e.g., program understanding, debugging, testing, differencing, specialization, merging). Although it is not so popular in the declarative programming community, several slicing techniques for declarative programs have also been developed during the last decade (see, e.g., [4, 14, 16–18, 20, 23]).

Basically, a *program slice* consists of those program statements which are (potentially) related with the values computed at some program point and/or variable, referred to as a *slicing criterion*. Program slices are usually computed from a *program dependence graph* [3, 12] that makes explicit both the data and control dependences for each operation in a program. Program dependences can be traversed backwards and forwards—from the slicing criterion—giving rise to so-called *backward* and *forward* slices, respectively. Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program’s input is provided or not. More detailed information on program slicing can be found in [10, 21].

In this work, we propose a lightweight approach to program specialization in the context of *lazy* functional logic languages which is based on dynamic slicing. Modern functional logic languages like Curry [6, 8] and Toy [15] combine the most important features of functional *and* logic languages, e.g., lazy evaluation and non-determinism (see [5] for a survey). Our aim in this paper is similar to that

^{*} This work has been partially supported by CICYT TIC 2001-2705-C03-01, by the Generalitat Valenciana GRUPOS03/025, by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054, and by the MCYT HU2003-0003.

$P ::= D_1 \dots D_m$		
$D ::= f(x_1, \dots, x_n) = e$		
$e ::= x$	(variable)	
$c(x_1, \dots, x_n)$	(constructor call)	$\mathcal{X} = \{x, y, z, \dots\}$ (variables)
$f(x_1, \dots, x_n)$	(function call)	$\mathcal{C} = \{a, b, c, \dots\}$ (constructors)
$\text{let } x = e_1 \text{ in } e_2$	(let binding)	$\mathcal{F} = \{f, g, h, \dots\}$ (functions)
$e_1 \text{ or } e_2$	(disjunction)	
$\text{case } x \text{ of } \{\overline{p_n \rightarrow e_n}\}$	(rigid case)	
$\text{fcase } x \text{ of } \{\overline{p_n \rightarrow e_n}\}$	(flexible case)	
$p ::= c(x_1, \dots, x_n)$	(flat pattern)	

Fig. 1. Syntax for normalized flat programs

of Reps and Turnidge [17], who designed a program specialization method for *strict* functional programs based on *static* slicing. However, in contrast to [17], we consider *lazy* functional logic programs and our technique is based on *dynamic* slicing. We consider dynamic slicing since it is simpler and more accurate than static slicing (i.e., it has been shown that dynamic slices can be considerably smaller than static slices [11, 22]). Informally speaking, specialization proceeds as follows: first, the user identifies a *representative* set of slicing criteria; then, dynamic slicing is used to compute a program slice for each slicing criterion; finally, the specialized program is built from the union of the computed slices. This is a lightweight approach since dynamic slicing is simpler than static slicing (as in [17]). Furthermore, a dynamic slicing technique is already available for the considered language [16]. Unfortunately, [16] defines a program slice as a set of *program positions* rather than as an executable program, since this is sufficient for debugging (the aim of [16]). Therefore, we also introduce an appropriate technique to extract executable slices from a set of program positions.

This paper is organized as follows. In the next section, we recall a simple lazy functional logic language. Section 3 presents an informal introduction to program specialization by dynamic slicing. Section 4 formalizes an instrumented semantics which also stores the location of each reduced expression. Section 5 defines the main concepts involved in dynamic slicing and, finally, Section 6 concludes and points out several directions for further research.

2 The Language

We consider *flat* programs [7] in this work, a convenient standard representation for functional logic programs which makes explicit the pattern matching strategy by the use of case expressions. The flat representation constitutes the kernel of modern declarative multi-paradigm languages like Curry [6, 8] and Toy [15]. In addition, we assume that flat programs are *normalized*, i.e., *let* constructs are used to ensure that the arguments of functions and constructors are always variables. As in [13], this is essential to express *sharing* without the use of complex graph structures. A simple normalization algorithm can be found in [1]; basically, this algorithm introduces one new *let* construct for each non-variable argument of a function or constructor call, e.g., $f(e)$ is transformed into “*let* $x = e$ *in* $f(x)$ ”.

The syntax of normalized flat programs is shown in Fig. 1, where $\overline{o_n}$ denotes the *sequence of objects* $o_1, \dots, o_n$. A program P consists of a sequence of function definitions D such that the left-hand side has pairwise different variable arguments. The right-hand side is an expression $e \in \text{Exp}$ composed by variables, data constructors, function calls, let bindings where the local variable x is only visible in e_1 and e_2 , disjunctions (e.g., to represent set-valued functions), and case expressions. In general, a case expression has the form:

$$(f)\text{case } x \text{ of } \{c_1(\overline{x_{n_1}}) \rightarrow e_1; \dots; c_k(\overline{x_{n_k}}) \rightarrow e_k\}$$

where $(f)\text{case}$ stands for either $f\text{case}$ or case , x is a variable, $c_1, \dots, c_k$ are different constructors, and $e_1, \dots, e_k$ are expressions. The *pattern variables* $\overline{x_{n_i}}$ are locally introduced and bind the corresponding variables of the subexpression e_i . The difference between case and $f\text{case}$ shows up when the argument x evaluates to a free variable: case suspends whereas $f\text{case}$ non-deterministically binds this variable to the pattern in a branch of the case expression.

Extra variables are those variables in a rule which do not occur in the left-hand side. They are intended to be instantiated by flexible case expressions. We assume that each extra variable x is explicitly introduced by a let binding of the form “*let* $x = x$ in e ”. We also call such variables *logical variables*.

3 Overview of the Specialization Process

In this section, we present an informal overview of the specialization process based on dynamic slicing. Similarly to Reps and Turnidge [17], we use an example inspired in a functional version of the well-known Unix word-count utility, and specialize it to *only* count characters. This is a kind of specialization that cannot be achieved by standard partial evaluation [17]. Consider the program shown in Fig. 2 where data structures are built from:¹

```
data Nat      = Z | S Nat      data Pairs = Pair Nat Nat
data Letter = A | B | CR      data String = Nil | Cons Letter String
```

As it is common, we use “`[]`” and “`:`” as a shorthand for `Nil` and `Cons`. Observe that we consider a maximally simplified alphabet for simplicity.

The program in Fig. 2 includes a function, `lineCharCount`, to count the number of lines and characters in a string (by using two accumulators, `lc` and `cc`, to build up the line and character counts as it travels down the string `str`). Function `printLineCount` (resp. `printCharCount`) is used to only print the number of lines (resp. characters) in a string. Function `ite` is a simple conditional while `eq`—not shown in Fig. 2—is an equality test on letters. For example, the execution of “`lineCharCount [letter, CR]`” returns the following three results:

```
Pair (S Z)      (S (S Z))      if letter reduces to A
Pair (S Z)      (S (S Z))      if letter reduces to B
Pair (S (S Z))  (S (S Z))      if letter reduces to CR
```

since function `letter` non-deterministically outputs `A`, `B` or `CR`.

¹ In the examples, for readability, we omit some of the brackets and write function applications as in Curry. Moreover, in this section, we consider programs that are not normalized for clarity.

```

printLineCount t = fcase t of {Pair x y -> printNat x}
printCharCount t = fcase t of {Pair x y -> printNat y}
printNat n = fcase n of {Z   -> 0;
                        S m -> 1 + printNat m}

lineCharCount str = lcc str Z Z
lcc str lc cc = fcase str of {[]      -> Pair lc cc;
                             (s:ss) -> ite (eq s CR)
                                           (lcc ss (S lc) (S cc))
                                           (lcc ss lc (S cc))    }

letter = A or B or CR
ite x y z = fcase x of {True -> y; False -> z}

```

Fig. 2. Example program `lineCharCount`

Now, our aim is the specialization of this program in order to extract those statements which are needed to compute *the number of characters* in a string. For this, we first consider the selection of appropriate slicing criteria and, then, the extraction of the corresponding slice (the *specialized* program).

From Tracing to Slicing. In the literature of dynamic slicing for imperative programs, the slicing criterion usually identifies a concrete point of the execution history, e.g., $\langle n = 2, 8^1, x \rangle$, where $n = 2$ is the input for the program, 8^1 denotes the *first* occurrence of statement 8 in the execution trace, and x is the variable we are interested in (see [21]).

In our functional logic setting, slicing criteria can also be determined by inspecting the traces of the relevant program executions. For this purpose, we consider a tracing tool [2] which relies on the computation of *redex trails* [19]. In order to use this tracer, a distinguished function `main` (with no arguments) should be added to start the computation. Since we are only interested in counting characters, an appropriate definition of `main` could be as follows:

```
main = printCharCount (lineCharCount [letter, CR])
```

Here, the tracer initially shows:

```

main -> 2
      -> 2
      -> 2

```

which means that there are three non-deterministic computations starting from `main` and, moreover, all of them give the same result, 2, since `printCharCount` only demands the computation of the number of characters in the input list.

The user can now select any of these values and the system will show a top-level trace of the associated computation. For example, the trace associated to the first computation—where `letter` is evaluated to `A`—is as follows:

```

2 = main
2 = printCharCount (Pair _ (S (S Z)))
2 = printNat      (S (S Z))
2 = (+)           1 1
2 = 2

```

where “_” denotes an argument whose evaluation is not needed in the computation. Each row shows a pair “ $val = exp$ ” where exp is an expression and val is its value. Note that function arguments appear *fully evaluated w.r.t. the considered computation*² in order to ease the understanding of the trace. Now, we select the argument of `printCharCount` in order to see the corresponding subtrace:

```

2          = printCharCount (Pair _ (S (S Z)))
Pair _ (S (S Z)) = lineCharCount [A, CR]
Pair _ (S (S Z)) = lcc [A, CR] _ Z
Pair _ (S (S Z)) = ite False _ (Pair _ (S (S Z)))
Pair _ (S (S Z)) = lcc [CR] _ (S Z)
Pair _ (S (S Z)) = ite True (Pair _ (S (S Z))) _
Pair _ (S (S Z)) = lcc [] _ (S (S Z))
Pair _ (S (S Z)) = Pair _ (S (S Z))

```

From this subtrace, we can already identify the interested call to `lineCharCount`. However, this information is not generally enough to determine a slicing criterion. Following [16], a slicing criterion is defined as a tuple of the form $\langle f(\overline{v_n}), v, l, \pi \rangle$, where $f(\overline{v_n})$ is a function call with fully evaluated arguments, v is its value (within a particular computation), l is the occurrence of a function call $f(\overline{v_n})$ that outputs v in the considered computation, and π is a pattern that determines the interesting part of the computed value (see below). The connection with the tracing tool should be clear: given a (sub)trace of the form

$$\begin{aligned}
val_1 &= f_1(v_1^1, \dots, v_m^1) \\
&\dots \\
val_k &= f_k(v_1^k, \dots, v_m^k)
\end{aligned}$$

a slicing criterion has the form $\langle f_i(v_1^i, \dots, v_m^i), val_i, l, \pi \rangle$. Therefore, the user can easily determine a slicing criterion from the trace of a computation. For example,

$$\langle \text{lineCharCount } [A, CR], \text{Pair } _ (S (S Z)), 1, \text{Pair } \perp \top \rangle$$

is a valid slicing criterion w.r.t. the considered computation. In the slicing pattern (the fourth component), $\top$ means that the computation of the corresponding subexpression is of interest while $\perp$ means that it can be ignored.

Specialization based on slicing. Given a slicing criterion, [16] presents a dynamic slicing technique that extracts the associated program slice from the *redex trail* of the computation. Redex trails [19] are directed graphs which record copies of all values and redexes (*reducible expressions*) of a computation, with a backward link from each reduct to the parent redex that created it. Note that the same data structure—the redex trail—is used for both tracing [2] and slicing [16]. For instance, Fig. 3 shows the redex trail of the previous computation for `main`. Basically, redex trails contain three types of arrows:

Successor arrows: There is a successor arrow, denoted by a solid arrow, from each redex to its reduct (e.g., from `main` to `printCharCount` in Fig. 3).

Argument arrows: Arguments of function and constructor calls are denoted by a pointer to the corresponding subexpression. If the evaluation of an argument

² I.e., as much as needed in the *complete* computation under consideration.

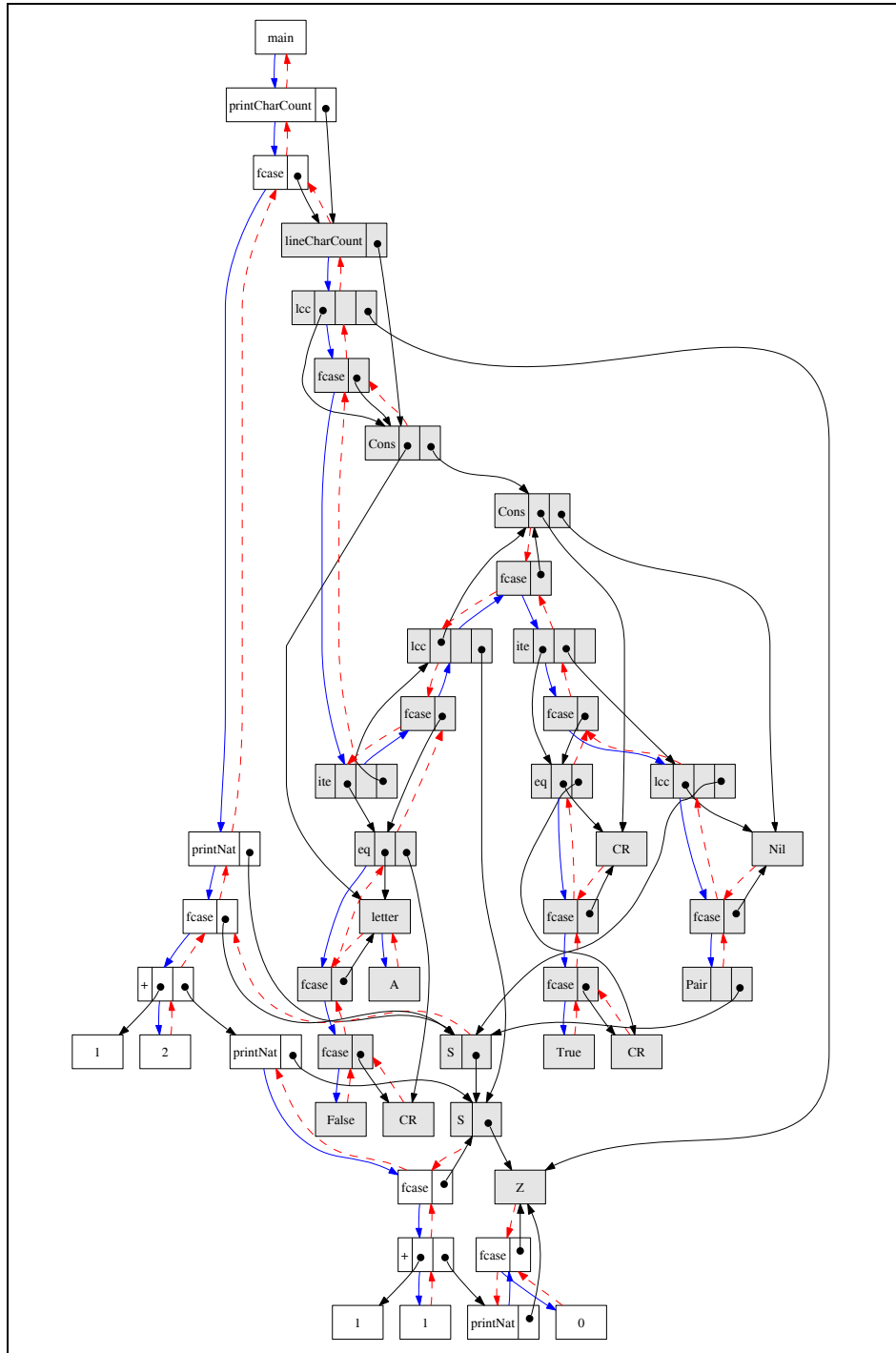


Fig. 3. Redex trail for program `lineCharCount`

```

lineCharCount str = lcc str ? Z
lcc str lc cc = fcase str of {[]      -> Pair ? cc;
                             (s:ss) -> ite (eq s CR)
                                           (lcc ss ? (S cc))
                                           (lcc ss ? (S cc)) }

letter = A or ?
ite x y z = fcase x of {True -> y; False -> z}
eq x y = fcase x of {A  -> fcase y of {CR -> False};
                    CR -> fcase y of {CR -> True } }

```

Fig. 4. Specialization of program `lineCharCount`

is not required in a computation, we have a null pointer (e.g., the second argument of `lcc` in Fig. 3).

Parent arrows: They are denoted by dashed arrows and point either to the redex from which an expression results—i.e., the inverse of the successor arrow—like, e.g., from `printCharCount` to `main` in Fig. 3, or to the expression who *demand*ed its evaluation, e.g., from `lineCharCount` to `fcase`.

In order to extract a program slice from the computed redex trail, one should first identify the node of the graph which is associated to the slicing criterion. In our example, the node associated to the slicing criterion

$(\text{lineCharCount } [A, CR], \text{Pair } _ (S (S Z)), 1, \text{Pair } \perp \top)$

is the topmost shadowed node of Fig. 3, since it records the first call to function `lineCharCount` that evaluates to $(\text{Pair } _ (S (S Z)))$ and whose argument evaluates to the list $[A, CR]$. Then, we determine the nodes which are reachable from this node according to the slicing pattern $(\text{Pair } \perp \top)$. The set of reachable nodes are also shadowed in Fig. 3. Finally, the set of *program positions* (i.e., the locations in the source program, see Def. 1) of the expressions in the shadowed nodes is returned as the computed slice.

This notion of slice is sufficient for debugging, where we are interested in highlighting those expressions which are related to the slicing criterion. In order to specialize programs, however, we should produce *executable* program slices. For this purpose, we introduce an algorithm that extracts a program slice from the original program and the computed set of program positions. For instance, the program slice associated to the shadowed nodes of Fig. 3 is shown in Fig. 4, where the distinguished symbol “?” is used to point out that some subexpression is missing due to the slicing process.

Clearly, producing a specialized program for a concrete computation is not always appropriate. For instance, in the slice above, function `letter` can only be evaluated to `A` but not to `B` or `CR`. To overcome this problem, [17] considers *static* slicing where no input data are provided and, thus, the extracted slices preserve the semantics of the original program for any input data. Our lightweight approach is based on dynamic slicing and, thus, a *representative* set of slicing criteria—covering the interesting computations—should be determined in order to obtain program slices which are general enough.

Rule	Heap	Control	Stack	Fun	Pos
varcons	$\Gamma[x \mapsto_{(h,w')} t]$	x	S	g	w
	$\Rightarrow \Gamma[x \mapsto_{(h,w')} t]$	t	S	h	w'
varexp	$\Gamma[x \mapsto_{(h,w')} e]$	x	S	g	w
	$\Rightarrow \Gamma[x \mapsto_{(h,w')} e]$	e	$x : S$	h	w'
val	Γ	v	$x : S$	g	w
	$\Rightarrow \Gamma[x \mapsto_{(g,w)} v]$	v	S	g	w
fun	Γ	$f(\overline{x_n})$	S	g	w
	$\Rightarrow \Gamma$	$\rho(e)$	S	f	Λ
let	Γ	$let\ x = e_1\ in\ e_2$	S	g	w
	$\Rightarrow \Gamma[y \mapsto_{(g,w.1)} \rho(e_1)]$	$\rho(e_2)$	S	g	$w.2$
or	Γ	$e_1\ or\ e_2$	S	g	w
	$\Rightarrow \Gamma$	e_i	S	g	$w.i$
case	Γ	$(f)\ case\ x\ of\ \{\overline{p_k \rightarrow e_k}\}$	S	g	w
	$\Rightarrow \Gamma$	x	$((f)\{\overline{p_k \rightarrow e_k}\}, g, w) : S$	g	$w.1$
select	Γ	$c(\overline{y_n})$	$((f)\{\overline{p_k \rightarrow e_k}\}, h, w') : S$	g	w
	$\Rightarrow \Gamma$	$\rho(e_i)$	S	h	$w'.2.i$
guess	$\Gamma[y \mapsto_{(g,w)} y]$	y	$(f\{\overline{p_k \rightarrow e_k}\}, h, w') : S$	g	w
	$\Rightarrow \Gamma[y \mapsto_{(-,-)} \rho(p_i),$ $\overline{y_n} \mapsto_{(-,-)} \overline{y_n}]$	$\rho(e_i)$	S	h	$w'.2.i$
where in varcons : t is constructor-rooted varexp : e is not constructor-rooted and $e \neq x$ val : v is constructor-rooted or a variable with $\Gamma[v] = v$ fun : $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n} \mapsto \overline{x_n}\}$ let : $\rho = \{x \mapsto y\}$ and y is fresh or : $i \in \{1, 2\}$ select : $i \in \{1, \dots, k\}$, $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n} \mapsto \overline{y_n}\}$ guess : $i \in \{1, \dots, k\}$, $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n} \mapsto \overline{y_n}\}$, and $\overline{y_n}$ are fresh					

Fig. 5. Small-step instrumented semantics

4 Instrumented Semantics

Now, we present an instrumented version of the small-step operational semantics for functional logic programs of [1] that also computes *program positions*.

The instrumented operational semantics is shown in Fig. 5. We distinguish two components in this semantics: the first component (columns *Heap*, *Control*, and *Stack*) defines the standard semantics introduced in [1]; the second component (columns *Fun* and *Pos*) is used to store *program positions* [16], i.e., a function name and a position (within this function) for the current expression in the control. The semantics obeys the following naming conventions:

$$\Gamma, \Delta \in \text{Heap} :: \mathcal{X} \rightarrow \text{Exp} \quad v \in \text{Value} ::= x \mid c(\overline{v_n})$$

A *heap* is a partial mapping from variables to expressions. Each mapping $x \mapsto_{(g,w)} e$ is labeled with the program position (g, w) of expression e (see below). The *empty heap* is denoted by $[\]$. The value associated to variable x in heap Γ

is denoted by $\Gamma[x]$. $\Gamma[x \mapsto_{(g,w)} e]$ denotes a heap with $\Gamma[x] = e$, i.e., we use this notation either as a condition on a heap Γ or as a modification of Γ . In a heap Γ , a logical variable x is represented by a circular binding of the form $\Gamma[x] = x$. A *stack* (a list of variable names and case alternatives where the empty stack is denoted by $[]$) is used to represent the current context. A *value* is a constructor-rooted term or a logical variable (w.r.t. the associated heap).

A configuration of the semantics is a tuple $\langle \Gamma, e, S, g, w \rangle$ where $\Gamma \in \text{Heap}$ is the current heap, $e \in \text{Exp}$ is the expression to be evaluated (called the control), S is the stack, and g and w denote the program position of expression e . *Program positions* uniquely determine the location, in the program, of each reduced expression. This notion is formalized as follows:

Definition 1 (program position). *Positions are represented by a sequence of natural numbers, where Λ denotes the empty sequence (i.e., the root position). They are used to address subexpressions of an expression viewed as a tree:*

$$\begin{aligned} e|_{\Lambda} &= e && (\text{for all } e \in \text{Exp}) \\ c(\overline{x_n})|_{i.w} &= x_i && (i \in \{1, \dots, n\}) \\ f(\overline{x_n})|_{i.w} &= x_i && (i \in \{1, \dots, n\}) \\ e_1 \text{ or } e_2|_{i.w} &= e_i|_w && (i \in \{1, 2\}) \\ (f) \text{ case } x \text{ of } \{\overline{p_n \rightarrow e_n}\}|_{1.w} &= x && \text{let } x = e \text{ in } e'|_{1.w} = e|_w \\ (f) \text{ case } x \text{ of } \{\overline{p_n \rightarrow e_n}\}|_{2.i.w} &= e_i|_w && (i \in \{1, \dots, n\}) \text{ let } x = e \text{ in } e'|_{2.w} = e'|_w \end{aligned}$$

Given a program P , we let $\text{Pos}(P)$ denote the set of all program positions in P . A program position is a pair $(g, w) \in \text{Pos}(P)$ that addresses the subexpression $e|_w$ in the right-hand side of the definition, $g(\overline{x_n}) = e$, of function g in P .

Note that not all subexpressions are addressed by program positions (this is the case, e.g., of the patterns in a case expression). Indeed, only those expressions which are relevant for the slicing process need to be addressed. A brief explanation for each rule of the extended semantics follows:

(**varcons**) It is used to evaluate a variable x which is bound to a constructor-rooted term t in the heap. Trivially, it returns t as a result of the evaluation. In the derived configuration, the current program position is updated with the position of the expression in the heap, (h, w') , which was previously introduced by rules **val** or **let** (see below).

(**varexp** and **val**) In order to evaluate a variable x that is bound to an expression e —which is not a value—rule **varexp** starts a subcomputation for e and adds variable x to the stack. As in rule **varcons**, the derived configuration is updated with the program position of the expression in the heap. Once a value is eventually computed, rule **val** updates the value of x in the heap. The associated program position for the binding is then updated with the program position (g, w) of the computed value.

(**fun**) This rule performs a simple unfolding; we assume that the considered program P is a global parameter of the calculus. Trivially, the program position is reset to (f, Λ) since the control of the derived configuration is the complete right-hand side of function f .

(**let**) In order to reduce a let construct, this rule adds the binding to the heap and proceeds with the evaluation of the expression. The local variable is renamed with a fresh name to avoid variable name clashes. The binding $(x \mapsto_{(g,w.1)} e_1)$

introduced in the heap is labeled with the program position of e_1 . This is necessary to recover the program position of the binding in rules **varcons** and **varexp**. The program position in the new configuration is updated to $(g, w.2)$ in order to address the expression e_2 of the *let* construct.

(**or**) This rule *non-deterministically* evaluates a disjunction by either evaluating the first or the second argument. Clearly, the program position of the derived configuration is $(g, w.i)$ where $i \in \{1, 2\}$ refers to the selected disjunct.

(**case**, **select**, and **guess**) Rule **case** initiates the evaluation of a case expression by evaluating the case argument and pushing the alternatives $(f)\{\overline{p_k} \rightarrow \overline{e_k}\}$ (together with the current program position) on top of the stack. If a constructor-rooted term is produced, rule **select** is used to select the appropriate branch and continue with the evaluation of this branch. If a logical variable is produced and the case expression on the stack is flexible (i.e., of the form $f\{\overline{p_k} \rightarrow \overline{e_k}\}$), rule **guess** non-deterministically chooses one alternative and continues with the evaluation of this branch; moreover, the heap is updated with the binding of the logical variable to the corresponding pattern. In both rules, **select** and **guess**, the program position of the case expression (stored in the stack by rule **case**) is used to properly set the program position of the derived configuration.

Trivially, the instrumented semantics is a conservative extension of the original small-step semantics of [1], since the last two columns of the calculus impose no restriction on the application of the standard component of the semantics (columns *Heap*, *Control*, and *Stack*).

In order to perform computations, we construct an *initial configuration* and (non-deterministically) apply the rules of Fig. 5 until a *final configuration* is reached. An initial configuration has the form $\langle [], \text{main}, [], -, - \rangle$, where $(-, -)$ denotes a null program position (since there is no call to **main** from the right-hand side of any function definition). A final configuration has the form: $\langle \Delta, v, [], g, w \rangle$, where v is a value, Δ is a heap containing the computed bindings, and (g, w) is a program position. We denote by $\Longrightarrow^*$ the reflexive and transitive closure of $\Longrightarrow$. A derivation $C \Longrightarrow^* C'$ is *complete* if C is an initial configuration and C' is a final configuration.

As an example, consider functions **eq** and **letter** of program **lineCharCount** again, together with the initial call **(eq letter CR)**. The normalized flat program is shown in Fig. 6, where the program positions of some selected expressions are displayed in the right column. A complete computation—the one in which the non-deterministic function **letter** is evaluated to **A**—with the rules of Fig. 5 is shown in Fig. 7, where each computation step is labeled with the applied rule.

5 Computing Specialized Programs

We now formalize the computation of specialized programs. Following [16], a dynamic slice is originally defined as a set of program positions:

Definition 2 (dynamic slice). *Let P be a program. A dynamic slice for P is a set $\mathcal{W}$ of program positions such that $\mathcal{W} \subseteq \text{Pos}(P)$.*

Although we are interested in computing executable slices, the above notion of slice is still useful in order to easily compute the differences between several

main = let x1 = letter	[let:(main,1),letter:(main,1)]
in let x2 = CR in eq x1 x2	[let:(main,2),CR:(main,2.1)]
letter = A or (B or CR)	[A:(letter,1),B:(letter,2.1),CR:(letter,2.2)]
eq x y = fcase x of	[fcase:(eq,1),x:(eq,1)]
{A -> fcase y of	[fcase:(eq,2.1),y:(eq,2.1.1)]
{A -> True;	[True:(eq,2.1.2.1)]
B -> False;	[False:(eq,2.1.2.2)]
CR -> False}	[False:(eq,2.1.2.3)]
B -> fcase y of {A -> False; B -> True; CR -> False}	
CR -> fcase y of {A -> False; B -> False; CR -> True} }	

Fig. 6. Functions `letter` and `eq` with program positions

slices—i.e., the intersection of the corresponding program positions—as well as the merging of slices—i.e., the union of the corresponding program positions. In particular, specialized programs are computed w.r.t. a set of relevant slicing criteria and, thus, they will be obtained from the original program and the *union* of the computed sets of program positions (see below).

As discussed in Section 3, an appropriate slicing criterion in our setting is given by a tuple $\langle f(\overline{pv}_n), pv, l, \pi \rangle$. Formally,

Definition 3 (slicing criterion). Let P be a program and $(C_0 \Longrightarrow^* C_m)$ a complete derivation. A slicing criterion for P w.r.t. $(C_0 \Longrightarrow^* C_m)$ is a tuple $\langle f(\overline{pv}_n), pv, l, \pi \rangle$ such that $f(\overline{pv}_n) \in FCalls$ is a function call whose arguments are fully evaluated w.r.t. $(C_0 \Longrightarrow^* C_m)$, $pv \in PValue$ is a partial value, $l > 0$ is a natural number, and $\pi \in Pat$ is a slicing pattern. The domains $FCalls$ (fully evaluated calls) and $PValue$ (partial values) obey the following syntax:

$$FCalls ::= f(pv_1, \dots, pv_n) \quad pv \in PValue ::= _ \mid x \mid c(pv_1, \dots, pv_k)$$

where $f \in \mathcal{F}$ is a defined function symbol (arity $n \geq 0$), $c \in \mathcal{C}$ is a constructor symbol (arity $k \geq 0$), and “ $_$ ” is a special symbol to denote any non-evaluated expression. The domain Pat of slicing patterns is defined as follows:

$$\pi \in Pat ::= \perp \mid \top \mid c(\pi_1, \dots, \pi_k)$$

where $c \in \mathcal{C}$ is a constructor symbol, $k \geq 0$, $\perp$ denotes a subexpression of the value whose computation is not relevant and $\top$ a subexpression which is relevant.

In order to compute a slice, we should first identify the configuration that is associated to a slicing criterion, which we call *SC-configuration*. Intuitively, the SC-configuration associated to a slicing criterion $\langle f(\overline{pv}_n), pv, l, \pi \rangle$ is the l -th configuration, C_i , of a computation that fulfills the following conditions: (i) the control of C_i is $f(\overline{x}_n)$, (ii) the variables $\overline{x}_n$ are bound in the considered computation to expressions which are “more defined” than $\overline{pv}_n$ (i.e., which are equal to $\overline{pv}_n$ except that some occurrences of “ $_$ ” are replaced by an arbitrary expression), and (iii) the subcomputation that starts from C_i ends with a configuration that contains a value in the control which is “more defined” than pv . The formal definition can be found in [16].

Now, we should determine those configurations that *contribute* to the evaluation of a slicing criterion. Roughly speaking, these contributing configurations

```

=>fun  ⟨[], main, [], -, -⟩
=>let  ⟨[x1 ↦(main,1) letter], let x2 = CR in eq x1 x2, [], main, 1⟩
=>let  ⟨[x1 ↦(main,1) letter, x2 ↦(main,2.1) CR], eq x1 x2, [], main, 2.2⟩
=>fun  ⟨[x1 ↦(main,1) letter, x2 ↦(main,2.1) CR],
      fcase x1 of {A → fcase...; B → fcase...; CR → fcase...}, [], eq, 1⟩
=>case  ⟨[x1 ↦(main,1) letter, x2 ↦(main,2.1) CR], x1,
      [(f{A → fcase...; B → fcase...; CR → fcase...}, (eq, 1))], eq, 1⟩
=>varexp  ⟨[x1 ↦(main,1) letter, x2 ↦(main,2.1) CR], letter,
      [x1, (f{A → fcase...; B → fcase...; CR → fcase...}, (eq, 1))], main, 1⟩
=>fun  ⟨[x1 ↦(main,1) letter, x2 ↦(main,2.1) CR], A or (B or CR),
      [x1, (f{A → fcase...; B → fcase...; CR → fcase...}, (eq, 1))], letter, 1⟩
=>or  ⟨[x1 ↦(main,1) letter, x2 ↦(main,2.1) CR], A,
      [x1, (f{A → fcase...; B → fcase...; CR → fcase...}, (eq, 1))], letter, 1⟩
=>val  ⟨[x1 ↦(letter,1) A, x2 ↦(main,2.1) CR], A,
      [(f{A → fcase...; B → fcase...; CR → fcase...}, (eq, 1))], letter, 1⟩
=>select  ⟨[x1 ↦(letter,1) A, x2 ↦(main,2.1) CR],
      fcase x2 of {A → True; B → False; CR → False}, [], eq, 2.1⟩
=>case  ⟨[x1 ↦(letter,1) A, x2 ↦(main,2.1) CR], x2,
      [(f{A → True; B → False; CR → False}, (eq, 2.1))], eq, 2.1.1⟩
=>varcons  ⟨[x1 ↦(letter,1) A, x2 ↦(main,2.1) CR], CR,
      [(f{A → True; B → False; CR → False}, (eq, 2.1))], main, 2.1⟩
=>select  ⟨[x1 ↦(letter,1) A, x2 ↦(main,2.1) CR], False, [], eq, 2.1.2.3⟩

```

Fig. 7. An example computation with the instrumented semantics

are obtained as follows: First, we collect all configurations in the subcomputation from the SC-configuration to a configuration with a value in the control. Once we reach the final configuration of the subcomputation, we also collect those configurations which are needed to compute the inner subterms of the value *according to pattern* π . A formal algorithm can be found in [16].

Given a slicing criterion, dynamic slices can be trivially obtained from the program positions of the SC-configuration as well as the contributing configurations [16]. Now, we introduce a novel method to extract an *executable* program slice from the computed program positions.

Definition 4 (executable slice). Let P be a program and $\mathcal{W}$ an associated dynamic slice. The corresponding executable slice, $P_{\mathcal{W}}$, is obtained as follows:

$$P_{\mathcal{W}} = \{f(\overline{x_n}) = \llbracket e \rrbracket_{\mathcal{Q}}^A \mid (f, A) \in \mathcal{W} \text{ and } \mathcal{Q} = \{p \mid (f, p) \in \mathcal{W}\} \}$$

where $\llbracket e \rrbracket_{\mathcal{Q}}^p = ?$ if $p \notin \mathcal{Q}$ and, otherwise (i.e., $p \in \mathcal{Q}$), it is defined inductively as follows:

$$\begin{aligned}
\llbracket x \rrbracket_{\mathcal{Q}}^p &= x \\
\llbracket \varphi(x_1, \dots, x_n) \rrbracket_{\mathcal{Q}}^p &= \varphi(x_1, \dots, x_n) \\
\llbracket \text{let } x = e' \text{ in } e \rrbracket_{\mathcal{Q}}^p &= \text{let } x = \llbracket e' \rrbracket_{\mathcal{Q}}^{p,1} \text{ in } \llbracket e \rrbracket_{\mathcal{Q}}^{p,2} \\
\llbracket e_1 \text{ or } e_2 \rrbracket_{\mathcal{Q}}^p &= \llbracket e_1 \rrbracket_{\mathcal{Q}}^{p,1} \text{ or } \llbracket e_2 \rrbracket_{\mathcal{Q}}^{p,2} \\
\llbracket (f) \text{case } x \text{ of } \{\overline{p_k \rightarrow e_k}\} \rrbracket_{\mathcal{Q}}^p &= (f) \text{case } x \text{ of } \{p_k \rightarrow \llbracket e_k \rrbracket_{\mathcal{Q}}^{p,2,k}\}
\end{aligned}$$

where $\varphi \in (\mathcal{C} \cup \mathcal{F})$ is either a constructor or a defined function symbol.

The slice computed in this way would be trivially executable by considering “?” as a fresh 0-ary constructor symbol (since it is not used in the considered computation). However, from an implementation point of view, this is not a good decision since we should also extend all program types in order to include “?”. Fortunately, in our functional *logic* setting, there is a simpler solution: we consider “?” as a fresh existential variable (i.e., like the anonymous variable of Prolog). This is very easy to implement in Curry by just adding a **where**-declaration to each function with some fresh variables. For instance,

```
lineCharCount str = lcc str ? Z
```

is replaced by

```
lineCharCount str = lcc str x Z where x free
```

Now, the specialization process should be clear. First, we compute dynamic slices—sets of program positions—for the selected slicing criteria. Then, an executable slice is built for the union of the computed dynamic slices. Moreover, there are some post-processing simplifications that can be added in order to reduce the program size and improve its readability:

$$\begin{aligned} \text{let } x = ? \text{ in } e &\implies \rho(e) && \text{if } \rho = \{x \mapsto ?\} \\ (f)\text{case } x \text{ of } \{\overline{p_k \rightarrow e_k}\} &\implies (f)\text{case } x \text{ of } \{\overline{p'_m \rightarrow e'_m}\} && \text{if } \{\overline{p'_m \rightarrow e'_m}\} = \{p_i \rightarrow e_i \mid e_i \neq ?, i = 1, \dots, k\} \end{aligned}$$

More powerful post-processing simplifications could be added but they require a form of *amorphous* slicing [9], which is out of the scope of this work.

In [16], we prove that the computed slices are correct and minimal. From these results, it would be easy to prove that our executable slices (according to Def. 4) are also correct and minimal specializations of the original program w.r.t. the considered slicing criteria.

6 Conclusions and Future Work

In this work, we have presented a lightweight approach to program specialization of lazy functional logic programs which is based on dynamic slicing. Our method obtains a kind of specialization that cannot be achieved with other related techniques like partial evaluation.

As a future work, we plan to extend the current method in order to perform more aggressive simplifications (i.e., a form of amorphous slicing). Another promising topic for future work is the definition of program specialization based on *static* slicing rather than on *dynamic* slicing. In particular, it would be interesting to establish the strengths and weaknesses of each approach.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 2004. To appear.

2. B. Braßel, M. Hanus, F. Huch, and G. Vidal. A Semantics for Tracing Declarative Multi-Paradigm Programs. In *Proc. of PPDP'04*, pages 179–190. ACM Press, 2004.
3. J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM TOPLAS*, 9(3):319–349, 1987.
4. V. Gouranton. Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing. In *Proc. of SAS'98*, pages 115–133, 1998.
5. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
6. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of POPL'97*, pages 80–93. ACM, 1997.
7. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
8. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: <http://www.informatik.uni-kiel.de/~curry/>.
9. M. Harman and S. Danicic. Amorphous Program Slicing. In *Proc. of the 5th Int'l Workshop on Program Comprehension*. IEEE Computer Society Press, 1997.
10. M. Harman and R.M. Hierons. An Overview of Program Slicing. *Software Focus*, 2(3):85–92, 2001.
11. T. Hoffner, M. Kamkar, and P. Fritzson. Evaluation of Program Slicing Tools. In *Proc. of ADEBUG'95*, pages 51–69. IRISA-CNRS, 1995.
12. D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimization. In *Proc. of POPL'81*, pages 207–218. ACM, 1981.
13. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of POPL'93*, pages 144–154. ACM Press, 1993.
14. M. Leuschel and M.H. Sørensen. Redundant Argument Filtering of Logic Programs. In *Proc. of LOPSTR'96*, pages 83–103. LNCS 1207 83–103, 1996.
15. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of the 10th Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
16. C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of PEPM'04*, pages 123–134. ACM Press, 2004.
17. T. Reps and T. Turnidge. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, pages 409–429. Springer LNCS 1110, 1996.
18. S. Schoenig and M. Ducasse. A Backward Slicing Algorithm for Prolog. In *Proc. of SAS'96*, pages 317–331. Springer LNCS 1145, 1996.
19. J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *Proc. of PLILP'97*, pages 291–308. Springer LNCS 1292, 1997.
20. G. Szilagy, T. Gyimothy, and J. Maluszynski. Static and Dynamic Slicing of Constraint Logic Programs. *J. Automated Software Engineering*, 9(1):41–65, 2002.
21. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.
22. G.A. Venkatesh. Experimental Results from Dynamic Slicing of C Programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–217, 1995.
23. G. Vidal. Forward Slicing of Multi-Paradigm Declarative Programs Based on Partial Evaluation. In *Proc. of LOPSTR 2002*, pages 219–237. Springer LNCS 2664, 2003.
24. M.D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.

An Input/Output Semantics for Distributed Program Equivalence Reasoning

Miquel Bertran, Francesc Babot, and August Climent

Informàtica La Salle, Universitat Ramon Llull, Barcelona
{miqbe,fbabot,augc}@salleURL.edu

Abstract. Some notions and deduction rules needed in formal equivalence reasoning with distributed imperative programs, with synchronous communications, are introduced. Starting at the semantic framework of Manna and Pnueli, and working in their notation SPL, the notions of input/output behavior and equivalence are defined as extensions. The former records the values traversing channels, in addition to the usual state variable values. Input/output equivalence is weaker than congruence but preserves the input/output relation, or function, of modular procedures, a slight modification of the modules of Manna and Pnueli. With this equivalence, a novel set of SPL laws for practical communication elimination has been made possible. A summary of these laws, as well as an outline of an equivalence proof of a pipelined processor software model, are included as applications.

1 Introduction

Imperative languages with explicit parallelism and synchronous communication statements provide an intuitive, explicit, and complete framework to express distributed programs with perspective and clarity. OCCAM [1–3] and the *simple programming language* SPL of Manna and Pnueli [4, 5] are well known representatives of these imperative notations.

Equivalence reasoning is heavily used for numbers, matrices, and other fields. However, for imperative programs, although being still a very intuitive verification activity, it has not been explored much in the framework of concurrent and distributed programs. It could be applied complementing other proof methods such as model checking [6–8] and interactive verification [9, 10]. It is based on the application of a set of laws, which depends on a notion of equivalence and on the fairness assumptions [4].

A set of laws for OCCAM was given in [11]. Rather than communication elimination, the focus there was to obtain normal forms and to define the semantics of the notation. Some laws for SPL are given in [4], which also covers an SPL semantics based on fair transition systems (FTS), but communication elimination laws for SPL are not given there. The first set of relations suitable for communication elimination in SPL was given and proved sound in [12], showing the necessity of avoiding strong fairness. A correctness communication elimination proof of a distributed fast Fourier transform was outlined there as well. A

formal correctness proof of a pipelined processor model has been reported in [13]. In these works, the notion of equivalence was assimilated there to congruence, a very strong equivalence. This had the drawback of limiting the formulation of most communication elimination laws as unidirectional refinements. Clearly, the need of working with a weaker equivalence, where all laws could be formulated with it, avoiding the asymmetry of refinement relations, was outstanding.

The paper introduces a notion of equivalence, *input/output equivalence* which is weaker than congruence but strong enough to preserve functionality of the programs and to lead to laws for communication elimination. It is justified in a semantics which extends the Manna-Pnueli framework as the main contribution of the paper, where each statement denotes a set of *input/output behaviors*. They correspond to an extension of the notion of *reduced behavior* given in [4]. In order to capture the complete input/output relation, the former adds to the latter a recording of the values traversing synchronous channels, in addition to the usual data state variables. This reflects the fact that values may be input or output via channels, as well as via proper variables. In this context, the grounding work with streams introduced in [14] is related, but in our concrete imperative program context we needed a new model where both channels and variables are explicit. The work on compatibility of components [15] is also related.

An important and necessary ingredient of equivalence reasoning is the possibility of substituting procedure reference statements of two equivalent procedures. A result giving the conditions for the validity of such substitutions is also given and its proof is outlined. This establishes the necessary base theory for formal io-equivalence reasoning with distributed programs. Communication elimination laws for io-equivalence, together with application of the results to pipelined processor verification are also overviewed in the paper.

2 Programming Notation

2.1 Syntax of the Basic Notation

Programs will be expressed in a reduced version of SPL, which is general enough to express any practical program. Its syntax is presented now. The basic statements are **Skip**, **Nil**, **Stop**, the assignment $u := e$, send $\alpha \Leftarrow e$, and receive $\alpha \Rightarrow u$. We limit our work to synchronous channels α , which will be referred to as *channels*. In them both the sender and the receiver wait for each other before exchanging a value and continuing execution. The two communication statements will be referred to more simply as *communications*. The skip statement involves a transition in the underlying fair transition system, but without any effect on the data variables. The nil statement makes its pre and post control locations equivalent, involving no transition. The stop statement has neither the transition nor the label equivalence relation. Both channels and variables are declared globally before their usage. The rest of the notation is defined recursively.

Concatenation is n -ary: $[S_1; \dots; S_n]$. The iterations are **[while c do S]**, where c is a boolean expression, and **[loop forever do S]**, which is defined as

[**while true do** S]. The *cooperation* statement is also n -ary: $[S_1 || \dots || S_n]$. Its substatements S_j are the *top parallel statements* of the cooperation statement, which is the *minimal common ancestor* of them. It will be assumed throughout the paper that the S_j 's are *disjoint*, in the sense that they only share read variables, and that they communicate values through synchronous channels only.

The *regular selection* and the *communication selection* statements are non-deterministic and have, respectively, the forms $[b_1, S_1 \text{ or } \dots \text{ or } b_n, S_n]$ and $[b_1, c_1, S_1 \text{ or } \dots \text{ or } b_n, c_n, S_n]$, where the b_i 's are boolean expressions referred to as *boolean guards*, and the c_i 's are synchronous communication statements referred to as *communication guards*.

2.2 Modular Procedures

This notion was introduced in [12] combining the notions of SPL *module* [4, 16] and *procedure*. As modules, modular procedures can be composed in parallel, but may be invoked by *procedure reference* statements, which make explicit the names of all the interface channels and variables. Common variables are prohibited. The notation $r ::= P(p)$ will be used for a procedure reference, where r and p stand for the result and parameter lists of the interface, and P is the procedure name. Modular procedures will be referred to more simply as *procedures*.

An example of procedure is given below. Its *procedure reference* stands at the left, and the *procedure body* at the right.

$$(r, cr) ::= Pc(p, cp) :: \left[\begin{array}{l} \text{out } r : \text{integer} \\ \text{out } cr : \text{channel of integer} \\ \text{external in } p : \text{integer} \\ \text{external in } cp : \text{channel of integer} \\ \text{local } a1, a2 : \text{integer} \\ \text{local } c : \text{channel of integer} \\ \left[\begin{array}{l} cp \Rightarrow a1; \\ a1 := a1 + p; \\ c \Leftarrow a1; \\ \text{skip} \end{array} \right] || \left[\begin{array}{l} cp \Rightarrow a2; \\ c \Rightarrow r; \\ a2 := r + a2; \\ cr \Leftarrow a2 \end{array} \right] \end{array} \right]$$

The exact meaning of modes **out** and **external in** is not important here, since processes are disjoint and communication is point to point and half-duplex. The following semantics is possible since no common variables or channels are allowed.

Semantics of the reference statement It is unchanged with the *replacement* of the reference by the body of the referred procedure, with a renaming of variables and channels when necessary. It has to be consistent with the reverse operation of *encapsulation* of a part of a program within a procedure.

The set $\mathcal{O}$ of observed variables of a procedure Contains all proper variables in the interface, i.e. in the result and parameter lists, and an auxiliary *channel variable* for each channel in the interface. The set $\mathcal{O}$ is also referred to as *interface set*.

A channel variable records, as a triplet, the values passed at each communication event, an integer mark reflecting their order for each channel, and an input/output mark (i, o). When the communication event is internal, a dot replaces either of these marks. For the above procedure, this set is $\mathcal{O} : \{r, p, cr, cp\}$, where cr and cp are the auxiliary variables associated to each respective channel.

2.3 Basic Notions for the Semantics

The semantics follows the style of Manna and Pnueli, based on fair transition systems (FTS), [4, 5]. It is available in [12]. We assume that the reader is familiar with the following notions taken from these books: *computation*, *reduced behavior*, *observed variable*, *program context*, and *congruence* of two statements: $S_1 \approx_{\mathcal{O}} S_2$. Some extended notions, needed later in the paper are introduced next.

An *input/output computation* (*io-computation*) records the changes of both the variables and the channels of the procedure body during an execution. It has a row for each change and a column for each variable or channel. It is an extension of the notion of computation, adding to it a column for each channel. Whereas a computation is a sequence of states only, an io-computation is a sequence of states where the values crossing channels are also recorded. Groups of computations will be represented as *schemas*, which have *value variables*. Computations have just values (integers, booleans, etc...). A possible *io-computation schema* of the procedure above is the following

		r	p	cr	cp	$a1$	$a2$	c
0	<i>initial</i>	x	$p1$	x_T	x_T	x	x	x_T
1	$cp \Rightarrow a1$	x	$p1$	x_T	$cp1, 1, i$	$cp1$	x	x_T
2	$cp \Rightarrow a2$	x	$p1$	x_T	$cp2, 2, i$	$cp1$	$cp2$	x_T
3	$a1 := a1 + p$	x	$p1$	x_T	$cp2, 2, i$	$cp1 + p1$	$cp2$	x_T
4	$c \Leftarrow a1 c \Rightarrow r$	$cp1 + p1$	$p1$	x_T	$cp2, 2, i$	$cp1 + p1$	$cp2$	$cp1 + p1, 1, .$
5	$a2 := r + a2$	$cp1 + p1$	$p1$	x_T	$cp2, 2, i$	$cp1 + p1$	$cp1 + p1 + cp2$	$cp1 + p1, 1, .$
6	$cr \Leftarrow a2$	$cp1 + p1$	$p1$	$cp1 + p1 + cp2, 1, o$	$cp2, 2, i$	$cp1 + p1$	$cp1 + p1 + cp2$	$cp1 + p1, 1, .$

A triplet (*value*, *count*, *i/o indication*) is associated to each new value of a channel variable. x denotes any value and x_T any triplet. $p1$, $cp1$, $cp2$, etc ... are value variables, whereas $a1$, $a2$, r , and p are program variables. cp , cr , and c are auxiliary channel variables. Giving integer values to $p1$, $cp1$, and $cp2$, specific io-computations would be obtained. Leaving aside the initial row, each row corresponds to the state resulting from the transition of the second column statement. The transition of row 4 is the joint transition of the synchronous communication over channel c . All computations are of infinite length. Thus the last row corresponds to a *terminal* state, repeating itself implicitly by idle transition firings. A computation schema could be obtained from the above by deleting the cr , cp , c , and the two left columns. Then deleting, as in [4], any row which equals its predecessor but not all of its successors.

3 Input/Output Behaviors

3.1 Basic Notions

An input/output behavior is a procedure execution trace seen from its outside.

Definition 1 (Input/Output behavior of a procedure) *An input/output behavior of a procedure, also referred to as io-behavior, is the result of deleting from an io-computation all columns of variables not belonging to $\mathcal{O}$, and then deleting any row which equals its predecessor but not all of its successors.*

The condition in the last deletion is necessary since the infinite implicit repetitions of the last row should not be deleted. Due to event counters, consecutive events are not deleted when their values are equal. This should be so since they may correspond to two inputs of the procedure function. Thus all channel events are represented by at least one row.

An io-behavior has one row for each value change of a result variable $v \in \mathcal{O}$. A parameter variable never changes its value, unless it is also a result. Input and output channel variables exhibit value changes. Next io-behavior schema results from the io-computation schema above.

	r	p	cr	cp
0	x	$p1$	$x\tau$	$x\tau$
1	x	$p1$	$x\tau$	$cp1, 1, i$
2	x	$p1$	$x\tau$	$cp2, 2, i$
4	$cp1 + p1$	$p1$	$x\tau$	$cp2, 2, i$
6	$cp1 + p1$	$p1$	$cp1 + p1 + cp2, 1, o$	$cp2, 2, i$

Rows 3 and 5 have been deleted since they are equal to their predecessors 2 and 4 respectively. Suppose now that $cp1 = cp2$, then row 2 would not be deleted due to the value of the counter field of the cp column.

Definition 2 (Component of an io-behavior) *An io-behavior component is the list of values, a column, corresponding to a variable of $\mathcal{O}$. But any value in the list which equals its predecessor but not all of its successors is deleted. There are both proper and channel variable components.*

Definition 3 (Equivalence of io-behaviors) *Two io-behaviors are equivalent when they share the same interface set, and the two components of the same variable of both are equal.*

The order of value changes among different components of the io-behavior is lost, but not the order within the same component. Equivalence only requires equality of homologous component lists.

3.2 Composition of io-behaviors

Sequential composition The io-behaviors of a sequential composition are formed by *post-coupling* an io-behavior of its second statement to an io-behavior of the first. In the binary composition $[r1 := P_1(p1)] ; [r2 := P_2(p2)]$, in general, $\mathcal{O}_1 \neq \mathcal{O}_2$, and $\mathcal{O}_1 \cap \mathcal{O}_2$ may or may not be empty. Given schema b_1 of P_1 , the schema b_2 of P_2 depends on the values of the last row of b_1 . The schema $b_{1;2}$ of $P_1; P_2$ corresponding to b_1 and b_2 has as many components as variables in $\mathcal{O}_1 \cup \mathcal{O}_2$. It is formed by *post-coupling* b_2 after b_1 . Informally, the components of b_2 go after the ones of b_1 , but certain values of the first row of b_2 have to equal their homologous ones in the last row of b_1 . More specifically, the post-coupling is done as follows:

1. Proper variable components. Cases:

- (a) $v \in \mathcal{O}_1 \cap \mathcal{O}_2$: b_2 has to be such that the values of the first positions (*initial* row) of the components of such v 's equal the last value of their homologous components of b_1 . Such a b_2 can always be found, since corresponding values share the same type, when v is a parameter of P_2 , and undefined values (x) can be coerced to any value, when v is a result of P_2 .
 - (b) $v \notin \mathcal{O}_1 \cap \mathcal{O}_2$ and $v \in \mathcal{O}_1$: The last values of such v components of b_1 are propagated into the future, over the b_2 selected as in (a).
 - (c) $v \notin \mathcal{O}_1 \cap \mathcal{O}_2$ and $v \in \mathcal{O}_2$: The first values of such v components of the b_2 above are propagated into the past, over b_1 .
2. Channel variable components. Cases:
- (a) $c \in \mathcal{O}_1 \cap \mathcal{O}_2$: Any undefined initial triplets of such c components of b_2 are set equal to the last triplets of their homologous components of b_1 . The counts of the rest of the triplets are increased accordingly.
 - (b) $c \notin \mathcal{O}_1 \cap \mathcal{O}_2$ and $c \in \mathcal{O}_1$: The last triplets of such c components of b_1 are repeated into the b_2 portion, in other words into the future.
 - (c) $c \notin \mathcal{O}_1 \cap \mathcal{O}_2$ and $c \in \mathcal{O}_2$: The b_1 portions of such c components are filled in with undefined triplets.

Example In the composition $[cr1, r := P_1(cp1, p)] ; [cr2, p := P_2(cp2, r)]$, a result variable r of the first is a parameter of the second, and a parameter p of the first is a result of the second. The respective interface sets are $\mathcal{O}_1 : \{cr1, r, cp1, p\}$ and $\mathcal{O}_2 : \{cr2, p, cp2, r\}$. No channel is shared. In addition, $\mathcal{O}_1 \cap \mathcal{O}_2 = \{r, p\}$ and we assume that $\mathcal{O} : \mathcal{O}_1 \cup \mathcal{O}_2 : \{cr1, r, cp1, p, cr2, cp2\}$. The following schemas are meant to be the io-behaviors b_1 of P_1 , and b_2 of P_2 , and their post-coupling $b_1; b_2$

b_1					
		$cr1$	r	$cp1$	p
0	<i>initial</i>	x_T	x	x_T	x_2
1	$cp1 \Rightarrow v_{11}$	x_T	x	$x_3, 1, i$	x_2
2	$cr1 \Leftarrow v_{21}$	$x_4, 1, o$	x	$x_3, 1, i$	x_2
3	$r := v_{31}$	$x_4, 1, o$	x_5	$x_3, 1, i$	x_2

b_2					
		$cr2$	p	$cp2$	r
0	<i>initial</i>	x_T	x	x_T	y_2
1	$p := v_{12}$	x_T	y_3	x_T	y_2
2	$cp2 \Rightarrow v_{22}$	x_T	y_3	$y_4, 1, i$	y_2
3	$cr2 \Leftarrow v_{32}$	$y_5, 1, o$	y_3	$y_4, 1, i$	y_2

$b_1; b_2$							
		$cr1$	r	$cp1$	p	$cr2$	$cp2$
0	<i>initial</i>	x_T	x	x_T	x_2	x_T	x_T
1	$cp1 \Rightarrow v_{11}$	x_T	x	$x_3, 1, i$	x_2	x_T	x_T
2	$cr1 \Leftarrow v_{21}$	$x_4, 1, o$	x	$x_3, 1, i$	x_2	x_T	x_T
3	$r := v_{31} ; \text{initial}$	$x_4, 1, o$	$x_5 (= y_2)$	$x_3, 1, i$	$x_2 (= x)$	x_T	x_T
4	$p := v_{12}$	$x_4, 1, o$	x_5	$x_3, 1, i$	y_3	x_T	x_T
5	$cp2 \Rightarrow v_{22}$	$x_4, 1, o$	x_5	$x_3, 1, i$	y_3	x_T	$y_4, 1, i$
6	$cr2 \Leftarrow v_{32}$	$x_4, 1, o$	x_5	$x_3, 1, i$	y_3	$y_5, 1, o$	$y_4, 1, i$

Within $b_1; b_2$, the y_2 of b_2 becomes x_5 , and the x of b_2 is coerced to x_2 . The v_{ij} 's are program variables.

Parallel Composition The io-behaviors of parallel compositions are formed by *side-coupling* io-behaviors of their component statements. In the parallel composition $[r1 := P_1(p1)] \parallel [r2 := P_2(p2)]$, in general $\mathcal{O}_1 \neq \mathcal{O}_2$ and $\mathcal{O}_1 \cap \mathcal{O}_2$ may or may not be empty. Given schemas b_1 of P_1 and b_2 of P_2 , the io-behavior schema $b_{1||2}$ of $P_1 \parallel P_2$ corresponding to b_1 and b_2 has as many components as

variables in $\mathcal{P} : (\mathcal{O}_1 \cup \mathcal{O}_2) - \mathcal{O}_I$ where $\mathcal{O}_I \subseteq (\mathcal{O}_1 \cup \mathcal{O}_2)$ is the set of proper and channel variables declared as internal, non-observable, in the composition. As such, these channels give rise to internal communication events. We assume disjointness of P_1 and P_2 and deadlock-freeness of their parallel composition. In this work, deadlock-freeness of P means *internal* deadlock-freeness, disregarding interaction with any environment where P may be embedded.

The io-behavior $b_{1||2}$, resulting from side-coupling, is constructed as follows:

1. Selection of matching behaviors. Since P_1 and P_2 are disjoint, the selection of b_1 and b_2 , the io-behaviors of P_1 and P_2 respectively, is determined by the internal channels. They are chosen so that the value components of the two triplets, one in each io-behavior, giving rise to each internal communication event are equal. This will be always possible since we assume deadlock-freeness, which in our context means that any internal communication in P_1 has a matching communication in P_2 , and vice versa. Furthermore, under this assumption, the counts of corresponding triplets can also be made equal, and one of them will have an i mark and the other one an o mark, but not necessarily always in the same side (io-behavior). We say that such corresponding triplets are *matching*.
2. Construction of the intermediate form $\bar{b}_{1||2}$.
 - (a) Its number of components equals the number of variables in $\mathcal{O}_1 \cup \mathcal{O}_2$.
 - (b) Its rows are separated in sublists by its internal communication event rows, constructed first as follows: their variable components are filled in with the values of the corresponding variables of the matching rows of the two io-behaviors, with the exception of the internal communication event triplet, which will be constructed with the value and the count of the two matching triplets. A dot will be placed in its third component, replacing the i and the o . This, again, assumes deadlock freeness.
 - (c) The rows of the sublists of b_1 and b_2 , separated by communication event rows, are interleaved in $\bar{b}_{1||2}$. Any interleaving is possible.
3. $b_{1||2}$ is constructed by deleting the components of $\bar{b}_{1||2}$ not in $\mathcal{P}$, and any row of the result which equals its predecessor but not all of its successors.

Example Consider the composition $[r1, cr1 := S(p1, cp1)] || [r2, cp1 := A(p2, cr1)]$ of two disjoint processes, with set of internal channels $\mathcal{O}_I : \{cr1, cp1\}$. The interface sets of S and A are $\mathcal{O}_s : \{r1, cr1, p1, cp1\}$ and $\mathcal{O} : \{r2, cp1, p2, cr1\}$. Assume that the interface set of the composition is $\mathcal{P} : (\mathcal{O}_s \cup \mathcal{O}) - \mathcal{O}_I : \{r1, p1, r2, p2\}$. The following schemas correspond to the io-behaviors b_s of S , b_a of A , and of their side-coupling intermediate form $\bar{b}_{s||a}$.

b_s					b_a				
		$r1$	$p1$	$cr1$	$cp1$			$cr1$	$cp1$
0	initial	x	x_2	x_T	x_T	0	initial	x_T	x_T
1	$cr1 \Leftarrow v_{21}$	x	x_2	$x_3, 1, o$	x_T	1	$r2 := v_{12}$	x_T	x_T
2	$cp1 \Rightarrow v_{11}$	x	x_2	$x_3, 1, o$	$x_4, 1, i$	2	$cr1 \Rightarrow v_{22}$	$y_4, 1, i$	x_T
3	$r1 := v_{31}$	x_5	x_2	$x_3, 1, o$	$x_4, 1, i$	3	$cp1 \Leftarrow v_{32}$	$y_4, 1, i$	$y_5, 1, o$

$\bar{b}_{s a}$									
		$r1$	$p1$	$cr1$	$cp1$	$r2$	$p2$		
0	initial	x	x_2	x_T	x_T	x	y_2		
1	$r2 := v_{12}$	x	x_2	x_T	x_T	y_3	y_2		
2	$cr1 \Leftarrow v_{21} cr1 \Rightarrow v_{22}$	x	x_2	$x_3 (= y_4), 1, \cdot$	x_T	y_3	y_2		
3	$cp1 \Rightarrow v_{11} cp1 \Leftarrow v_{32}$	x	x_2	$x_3, 1, \cdot$	$x_4 (= y_5), 1, \cdot$	y_3	y_2		
4	$r1 := v_{31}$	x_5	x_2	$x_3, 1, \cdot$	$x_4, 1, \cdot$	y_3	y_2		

The rows of matching internal communications, over channels $cr1$ and $cp1$, and of internal communication events have been isolated.

3.3 Selection Composition

For selection composition, $[b_1, c_1; [r1 := P_1(p1)] \text{ or } \dots \text{ or } b_n, c_n; [rn := P_n(pn)]]$, the set of io-behaviors is the *union* of the io-behaviors contributed by each of its alternatives A_k . Each of them contributes with the subset of the io-behaviors of $[c_k; [rk := P_k(pk)]]$ whose first row satisfies boolean condition b_k .

Let $\mathcal{O}_k$ be the interface set of procedure P_k . Then the interface set $\mathcal{O}_{A_k}$ of alternative A_k is given by $\mathcal{O}_k \cup \text{var}(c_k) \cup \text{chan}(c_k) \cup \text{var}(b_k)$; where $\text{var}(e)$ is the set of variables of expression e , $\text{chan}(c)$ is the singleton set containing the channel variable of c . The interface set of the above selection may be any set $\mathcal{P}$ such that $\mathcal{P} \subseteq \bigcup_{k=1}^n \mathcal{O}_{A_k}$. All this is consistent with non-determinism.

4 Input/Output Equivalence

4.1 The Notion

Definition 4 (Io-equivalent procedures) *Two procedures P_1 and P_2 are input/output equivalent with respect to their common interface set $\mathcal{O}$, written $P_1 =_{\mathcal{O}} P_2$, when any io-behavior of any of them is equivalent to an io-behavior of the other.*

Io-equivalence is weaker than congruence. Congruent procedures are always equivalent but not vice versa. The relative order of value changes in distinct components is neglected in io-equivalence. Therefore, substitution of a reference to a procedure by a reference to another procedure, io-equivalent to the first, may introduce deadlock. The following is a procedure resulting from Pc of subsection 2.2 after elimination of internal channel c . It has the same interface set.

$$(r, cr) ::= Pnc(p, cp) :: \begin{bmatrix} [cp \Rightarrow a1 \mid cp \Rightarrow a2]; \\ r := a1 + p; a2 := r + a2; cr \Leftarrow a2 \end{bmatrix}$$

The reader may verify that each io-behavior of Pc is an io-behavior of Pnc and vice versa. Therefore, they are congruent $Pc \approx_{\mathcal{O}} Pnc$. Consider also the two procedures

$$(r1, r2) ::= P1(cp1, cp2) :: [cp1 \Rightarrow r1; cp2 \Rightarrow r2]$$

$$(r1, r2) ::= P2(cp1, cp2) :: [cp2 \Rightarrow r2; cp1 \Rightarrow r1]$$

with the same interface set $\mathcal{O}$. Now $P1 \not\approx_{\mathcal{O}} P2$, but $P1 =_{\mathcal{O}} P2$. If $P1$ is parallel to a process which always offers an output via channel $cp1$ before offering another output via $cp2$, and we replace $P1$ by $P2$ in that program, deadlock is introduced.

4.2 Substitution rules

Substitution of reference statements to io-equivalent procedures is an essential step of equivalence reasoning. The three first lemmas, concerning concatenation, cooperation and selection, are given in preparation for the general rule. Only the post-concatenation case is treated, the other case would be carried out similarly.

Lemma 1 (Substitution in concatenation) *Let $S; [r := A(p)]$ be deadlock-free. Then, if $[r := A(p)] =_{\mathcal{O}} [r := B(p)]$, the equivalence $S; [r := A(p)] =_{\mathcal{P}} S; [r := B(p)]$ holds, where $\mathcal{P} \subseteq \mathcal{O} \cup \mathcal{O}_S$, and $\mathcal{O} = \mathcal{O}_A = \mathcal{O}_B$*

The justification is in the full version of this paper, available from the authors.

We study now the preservation of equivalence in a substitution within a co-operation, parallelism, statement. Let $S||[r := A(p)]$ have internal channel set I . If $\mathcal{O}$ is the interface set of $[r := A(p)]$, and $\mathcal{O}_I$ is the set of channel variables corresponding to I , then $\mathcal{O}_I \subseteq \mathcal{O}$. Channel variables in $\mathcal{O} - \mathcal{O}_I$ correspond to external channels of $S||[r := A(p)]$. Similarly, $\mathcal{O}_I \subseteq \mathcal{O}_S$, where $\mathcal{O}_S$ is the interface set of S . Also, channel variables in $\mathcal{O}_S - \mathcal{O}_I$ correspond to external channels of $S||[r := A(p)]$. Actually, the interface set $\mathcal{P}$ of the parallel composition is such that $\mathcal{P} \subseteq (\mathcal{O}_S \cup \mathcal{O}) - \mathcal{O}_I$, since there may be proper variables which are not declared as external.

Lemma 2 (Substitution in parallelism) *Let $S||[r := A(p)]$ be deadlock-free, and $r := A(p)$ be disjoint with S . Let also $[r := A(p)] =_{\mathcal{O}} [r := B(p)]$, and $S||[r := B(p)]$ be deadlock-free. Then $[S||[r := A(p)]] =_{\mathcal{P}} [S||[r := B(p)]]$, where $\mathcal{P} \subseteq (\mathcal{O}_S \cup \mathcal{O}) - \mathcal{O}_I$.*

Justification We will show that, in the construction of io-behaviors of $S||A$ and $S||B$, the steps of side-coupling, given in subsection 3.2, can be followed so that equivalent io-behaviors result from the two statements. Deadlock-freeness is required since it has been assumed in the construction.

1. Selection of matching behaviors. Since b_S is identical for both statements and $A =_{\mathcal{O}} B$, any io-behavior b_A matching b_S can be replaced by an equivalent b_B , which will also be matching b_S .
2. As a consequence, the intermediate forms $\bar{b}_{S||A}$ and $\bar{b}_{S||B}$ will be equivalent with respect to set $\mathcal{O}_S \cup \mathcal{O}$.
3. Hence, any construction starting from either $\bar{b}_{S||A}$ or $\bar{b}_{S||B}$ will give identical results, in particular the one defined in step 3, with the $\mathcal{P}$ defined above.

Therefore, any io-behavior of $S||A$ can be interpreted as an equivalent io-behavior of $S||B$ and vice versa. $\square$

Lemma 3 (Substitution in selection) *Let $[g, [r := A(p)] \text{ or } R]$ be deadlock-free, where R stands for the rest of the selection statement, and g is a boolean guard for a selection or both a boolean and a communication guard for a communications selection. Let also $[r := A(p)] =_{\mathcal{O}} [r := B(p)]$. Then $[g, [r := A(p)] \text{ or } R] =_{\mathcal{P}} [g, [r := B(p)] \text{ or } R]$, where $\mathcal{P} \subseteq (\mathcal{O}_R \cup \mathcal{O})$.*

Justification Let $\mathcal{I}_R$, $\mathcal{I}_{Ag}$, and $\mathcal{I}_{Bg}$ denote the sets of io-behaviors of R , the alternative of A and the alternative of B , respectively, and $\mathcal{I}_A$ and $\mathcal{I}_B$ be the sets of io-behaviors of A and B , respectively. If the statement is a regular selection, then $\mathcal{I}_{Ag}$ and $\mathcal{I}_{Bg}$ are formed with the io-behaviors of $\mathcal{I}_A$ and $\mathcal{I}_B$, respectively, whose first rows satisfy the boolean guard of g . However, if the statement is a communications selection, each io-behavior of $\mathcal{I}_{Ag}$ is obtained by post-coupling an io-behavior of the $\mathcal{I}_{Ag}$ of the last case to the io-behavior of to the send or receive statement in the guard g . The same is true for $\mathcal{I}_{Bg}$.

The set of io-behaviors of a selection statement is the union of the sets of io-behaviors of each of its alternatives. Hence $\mathcal{I}_R \cup \mathcal{I}_{Ag}$ and $\mathcal{I}_R \cup \mathcal{I}_{Bg}$ are the sets of io-behaviors of the l.h.s. and the r.h.s., respectively.

Therefore, for any io-behavior b_l of the l.h.s. there is an equivalent io-behavior b_r in the r.h.s. and vice versa. This is so in the case that the io-behavior belongs to $\mathcal{I}_R$ since this set is included in both sides and equal io-behaviors are io-equivalent as well. In the remaining case, where the io-behavior is in $\mathcal{I}_{Ag}$ or in $\mathcal{I}_{Bg}$ the truth follows from the fact that $[r := A(p)] =_{\mathcal{O}} [r := B(p)]$ and the guard g is the same in both sides. $\square$

The following result is needed for the organization of proofs around the procedures of a distributed program, making proof decomposition possible.

Lemma 4 (Equivalence deduction by procedure substitution) *Let $P[-]$ be a loop free program context, $P[r := A(p)]$ be deadlock-free, and $r := A(p)$ be disjoint with all its parallel substatements in $P[r := A(p)]$. Then, if $[r := A(p)] =_{\mathcal{O}} [r := B(p)]$, and $P[r := B(p)]$ is deadlock-free, the equivalence $P[r := B(p)] =_{\mathcal{P}} P[r := A(p)]$ holds for any $\mathcal{P}$.*

Justification The result follows from lemmas 1, 2, and 3. The minimal ancestor of A in $P[r := A(p)]$, or in $P[r := B(p)]$, is either a concatenation, a co-operation, or a selection. Then, equivalence between these ancestor statements follows from one of the three lemmas, and the laws to be overviewed in next section. The same reasoning can now be applied recursively to the ancestors of these ancestors until $P[r := A(p)]$ and $P[r := B(p)]$ are reached. $\square$

5 Laws for Input/Output Equivalence

The simple intuitive laws are available in [12], where it is shown that many of them do not hold when strong fairness is assumed. Some of them are $\mathbf{Nil}; S \approx S$, $S; \mathbf{Skip} \approx S$, $S || \mathbf{Skip} \approx S$. Both sequential and parallel composition are associative. The latter is both associative and commutative.

For non-deterministic selections, let all the statements S_k where $k = 1, \dots, m$ be disjoint with statement $\tilde{S}$, b_i be boolean expressions disjoint with $\tilde{S}$, and c_i be communication operations, then

$$\begin{aligned} [b_1, c_1; S_1 \text{ or } \dots \text{ or } b_m, c_m; S_m] || \tilde{S} &\approx b_1, [[c_1; S_1] || \tilde{S}] \text{ or } \dots \text{ or } b_m, [[c_m; S_m] || \tilde{S}] \\ [b_1, c_1; S_1 \text{ or } \dots \text{ or } b_m, c_m; S_m]; T &\approx b_1, c_1; [S_1; T] \text{ or } \dots \text{ or } b_m, c_m; [S_m; T] \\ H; [b_1, c_1; A_1 \text{ or } \dots \text{ or } b_m, c_m; A_m] &\approx [bp(b_1, H), H; c_1; A_1 \text{ or } \dots \text{ or } bp(b_m, H), H; c_m; A_m] \end{aligned}$$

where $bp(b, S)$ denotes the backward propagation of assertion b over statement S , as presented in [5]. The last three laws also hold for regular selection. Symbolically, these new laws are obtained by setting to nil all communication guards c_i . Both selection and communications selection are idempotent, for io-equivalent alternatives. The form of the above laws suggests the following result.

Lemma 5 (Top selection form) *Any loop-free statement S , whose parallel substatements are disjoint can be transformed into an equivalent selection all of whose selection substatements are substatements of selections.*

Its justification is available in the full form of the paper, available from the authors.

Top selection forms have terminal alternatives which are selection-free and loop-free. Their substatements are concatenations, parallelisms and basic substatements only. We will refer to these statements as *bounded communication* (BC) statements, since the number of communication events generated by their execution is finite. They are also loop-free.

Communication elimination from BC statements The laws to be given below allow the elimination of communications from BC statements. For each BC statement S we define a set C of *internal* channels, whose communications have to be eliminated. The rest of the channels involved in S are *external* in the sense that communication statements over these channels never match with other communications in S . The following are two intuitive communication elimination laws.

$$[\alpha \Leftarrow e \parallel \alpha \Rightarrow u] \approx [u := e]$$

$$[H^l; \alpha \Leftarrow e; T^l] \parallel [H^r; \alpha \Rightarrow u; T^r] \approx [H^l \parallel H^r; u := e; [T^l \parallel T^r]]$$

where H^l and H^r do not contain communication substatements over channels in C . As shown in [12], no bounded number of communication elimination laws suffices for the elimination of communications from BC statements in a single reduction. The following scheme

$$\left[\begin{array}{c} H_k^l; \\ [G_k^l \parallel P_k^l]; \\ T_k^l \end{array} \right] \parallel \left[\begin{array}{c} H_k^r; \\ [G_k^r \parallel P_k^r]; \\ T_k^r \end{array} \right] =_{\mathcal{O}} \left[\begin{array}{c} [H_k^l \parallel H_k^r]; \\ [G_k \parallel P_k^l \parallel P_k^r]; \\ [T_k^l \parallel T_k^r] \end{array} \right]$$

defines an unbounded set of laws when we identify it with

$$[G_{k+1}^l \parallel G_{k+1}^r] =_{\mathcal{O}} G_{k+1}$$

since then, the statements G_k^l , G_k^r , and G_k are defined recursively for $k = 1, 2, \dots$ together with the initial condition statements G_0^l and G_0^r being matching communication statements $\alpha \Leftarrow e$, and $\alpha \Rightarrow u$, and G_0 standing for $u := e$. There is a law for any finite integer k . The former two laws are special cases for $k = 0, 1$ and for some nil substatements. The laws hold for io-equivalence only.

The law is applied as a reduction from left to right, in order to eliminate any matching pair of communication statements in a single reduction. Observe, also, that some substatements are parallel in one side but not in the other. This disordering may introduce deadlock. Therefore, a set of suitable applicability conditions have to be checked for each law.

6 Applications to Verification

Distributed program simplification (DPS) This is a proof procedure carried out in our interactive prover, applying the laws given above. The first step is carried out by a *communication elimination* reduction algorithm, which applies automatically the laws presented in last section. When the algorithm terminates successfully, there is a guarantee that the original statement is deadlock-free. The resulting io-equivalent form has parallelism between disjoint substatements but no internal communication statements.

The next step, *parallelism to concatenation transformation*, is carried out interactively by applying permutation laws for transforming the parallel compositions of disjoint processes to io-equivalent sequential forms. A sequential program io-equivalent to the initial one is obtained. A third step is *redundant variable elimination*, which is also done interactively. State-vector reduction comes with the last step only. We refer to the three steps as *distributed program simplification* (DPS).

Communication elimination from non-BC statements There exist more than one way to extend the elimination to non-BC statements, where communications appear within indefinite loops. We will center only in the following very common structure: $S = [S_1 || \dots || S_m]$, where the S_k 's are of the form $S_k = \text{loop forever do } B_k$. The B_k 's are BC statements. Since they have communication statements and appear within indefinite iterations, the whole statement is non-BC.

Assume that we unfold n_k times the loop of each top substatement S_k , thus obtaining the statement $[B_1^{n_1}; S_1 || \dots || B_m^{n_m}; S_m]$, where the $B_k^{n_k}$'s stand for the concatenation of n_k copies of B_k : $B_k; \dots; B_k$.

Assume that we succeed, by DPS in reducing S to $B; E$, where B has no internal communication but the ending statement E is non-BC and may have both parallelism and inner communication. Assume also that $B; E$ is also reduced by DPS to $B; B; E$. Then, as a consequence of finite induction, $S =_{\mathcal{O}} [B^n; E]$ for any finite integer n , where B^n is inner parallelism and communication free. In the frequent case where the first elimination yields $B; S$, i.e. $E = S$, then $S =_{\mathcal{O}} \text{loop forever do } B$ and the right hand side statement has no inner communication. In many practical systems this occurs already for $n_k = 1$; $k = 1 \dots m$.

Global structure of an equivalence proof of a pipelined processor model A brief account of a verification of a DLX-like [17] processor model will illustrate the utility of the results reported in this work. The proof establishes io-equivalence between a program with two hierarchical levels of parallelism and internal communication, modeling the pipeline processor, and a sequential program. The parallel program has four processes connected in pipeline, at its first level. Some of those processes are defined with procedures which encapsulate a second level of parallelism. The size of the parallel program is too large to be shown in this paper, a detailed account has been reported in [13]. The sequential program, shown io-equivalent to it by the proof, captures the essential behavior of the pipelined processor software model. It is the following:

$$reg ::= VNCycle(reg, mem) :: \left[\begin{array}{l} \text{for } k := 1..n \text{ do} \\ \quad \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1; \\ reg(ir.rd) := alures(ir.func, reg(ir.rs1), reg(ir.rs2)) \end{array} \right] \end{array} \right]$$

As this program makes explicit, the processor interprets programs with ALU register to register instructions only. The instruction register is ir . The destination and source register indexes are $ir.rd$, $ir.rs1$ and $ir.rs2$. Procedure $alures$ gives the result of the ALU operation selected by $ir.func$. Integer n is the length of the program in mem . The proof establishes the io-equivalence

$$[reg ::= VNCycle(reg, mem)] =_O [reg ::= Pipeline2(reg, mem)]$$

where $Pipeline2$ corresponds to the pipelined model with two levels of parallelism. This result is proved with three DPS proofs. Although the pipelined structure has more stages, we illustrate the partitioning of the proof only with the instruction decode (ID) and the execution (EX) stages. The two are modeled as modular procedures. Actually, we have two io-equivalent ones for each stage, one without and another one with inner parallelism. With two DPS proofs the equivalences

$$\begin{aligned} [reg, coutID ::= ID_Seq(reg, cinID)] &=_O [reg, coutID ::= ID_Par(reg, cinID)] \\ [coutEX ::= EX_Seq(cinEX)] &=_O [coutEX ::= EX_Par(cinEX)] \end{aligned}$$

are established. The $cout$ and cin names denote the lists of output and input channels respectively. Procedure $Pipeline2$ above has references to the procedures with inner parallelism, $Pipeline2 : Pipeline[ID_Par, EX_Par]$. $Pipeline$ is a procedure with two holes for the references to the ID and EX procedures.

An auxiliary procedure $Pipeline1$ is defined as $Pipeline$ with the references to the sequential procedures, $Pipeline1 : Pipeline[ID_Seq, EX_Seq]$. It has one level of parallelism. Then, the equivalence

$$[reg ::= VNCycle(reg, mem)] =_O [reg ::= Pipeline1(reg, mem)]$$

is established with a DPS proof. Finally, $Pipeline1 =_O Pipeline2$ by the substitution rules, and assuming that the deadlock-freeness conditions of lemma 4 hold. Then the global result follows, since the success of the communication elimination algorithm guarantees deadlock-freeness of $Pipeline1$. Deadlock-freeness of $Pipeline2$ follows from deadlock-freeness of $Pipeline1$ and conservation of the order of the external communication offers of the parallel and sequential versions of the EX and ID procedures.

7 Conclusions and Future Work

A novel semantics for distributed programs has been presented as an extension of the SPL semantics of Manna and Pnueli. Auxiliary variables recording the list of values crossing channels have been added to the state variables of computations and reduced behaviors. A general formulation of input/output equivalence of procedures, integrating values communicated through both variables and synchronous channels, has been made possible in this new framework. As an application, a new set of laws for SPL and the new equivalence, and a formal equivalence proof of a pipelined processor software model have been summarized. Although, other equivalence proofs for distributed programs have been carried

out already in our interactive prover, this line effort should continue for other classes of distributed programs. The soundness of the laws for io-equivalence has been proved in other works. However, completeness has not been studied yet, and should be studied in the future.

Acknowledgements

We thank the encouragement received during the last years from Zohar Manna, Bernd Finkbeiner, and Tomas Uribe.

References

1. INMOS-Limited: Occam Programming Manual. Prentice Hall (1985)
2. INMOS-Limited: Occam 2 Reference Manual. Prentice Hall (1988)
3. Jones, G.: Programming in Occam. Prentice Hall (1987)
4. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Specification. Springer (1991)
5. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems. Safety. Springer (1995)
6. Holtzmann, G.: Design and Validation of Computer Protocols. Prentice Hall (1991)
7. McMillan, K., Dill, D.: Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic (1993)
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press (1999)
9. Bjørner, N., Browne, A., M. Colón, B.F., Manna, Z., Sipma, H., Uribe, T.: Verifying Temporal Properties of Reactive Systems. A Step Tutorial. In: Formal Methods in System Design. (2000) 227–270
10. Kaufmann, M., Moore, J.S.: An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. IEEE Transactions on Software Engineering **23** (1997) 203–213
11. Roscoe, A., Hoare, C.: The laws of OCCAM programming. Theoretical Computer Science **60** (1988) 177–229
12. Bertran, M., Babot, F., Climent, A., Nicolau, M.: Communication and Parallelism Introduction and Elimination in Imperative Concurrent Programs. In Cousot, P., ed.: Static Analysis. 8th International Symposium, SAS 2001. Volume 2126 of LNCS., Paris, France, Springer (2001) 20–39
13. Babot, F., Bertran, M., Riera, J., Puig, R., Climent, A.: Mechanized Equivalence Proofs of Pipelined Processor Software Models. In: Actas de las III Jornadas de Programación y Lenguajes, Alicante, Universitat d’Alacant (2003) 91–104
14. Broy, M.: A logical basis for component-based systems engineering. In Broy, M., Steinbruggen, R., eds.: Calculational System Design., IOS Press (1999)
15. de Alfaro, L.: Game Models for Open Systems. In: International Symposium on Verification (Theory and Practice). Volume 2772 of LNCS., Springer (2003)
16. Finkbeiner, B., Manna, Z., Sipma, H.: Deductive Verification of Modular Systems. In: In Compositionality: The Significant Difference, COMPOS’97. Volume 1536 of LNCS., Springer (1998) 239–275
17. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Mateo, California (1990)

Dealing denotationally with stream-based communication[★]

Mercedes Hidalgo-Herrero¹ and Yolanda Ortega-Mallén²

¹ Dept. Didáctica de las Matemáticas
Facultad de Educación, Universidad Complutense de Madrid.
`mhidalgo@edu.ucm.es`

² Dept. Sistemas Informáticos y Programación
Facultad CC.Matemáticas, Universidad Complutense de Madrid.
`yolanda@sip.ucm.es`

Abstract. We define a denotational semantics for a kernel-calculus of the parallel functional language Eden. Continuations is our choice to deal with side-effects (process creation and communication) in a lazy context. Besides, the calculus include streams for communication, and their modelization by a denotational semantics is not direct because a stream may be infinite.

1 Introduction

Assuming that parallelism and distribution are efficiency improvements in programming, the main goal for designing Eden [LOP04] was to profit from both of them in the functional paradigm. Eden extends the functional language Haskell with constructs for defining explicit processes, so that the Eden programmer controls —from a higher level of abstraction— the process granularity, the data distribution, and the process topology. This circumstance is endorsed by the fact that the programmer has not to worry about synchronization tasks.

The language Eden comprises two layers: the functional level, or computational model, and the processes level, or coordination model. The computational model is the lazy functional language Haskell, while the coordination level includes the following features:

Process abstractions: expressions that define the general behaviour of a process in a purely functional way.

Process creations: applications of some process abstraction to a particular group of expressions that produces the creation of a new process to compute the result of the application.

Interprocess communications: these are asynchronous and implicit, since the programmer does not need to specify the message passing. Communications in Eden are not restricted to the transmission of a single value, processes can communicate values in a stream-like manner.

[★] Work partially supported by the Spanish project TIC2003-01000

Eden also includes some constructs to model reactive systems:

Dynamic creation of channels: without this possibility communications are only hierarchical, i.e. from parent to child and viceversa. Dynamic channels allow the creation of more complex communication topologies [PRS02].

Non-determinism: in order to model communications from many-to-one Eden introduces a predefined process, `merge`, whose inputs are several streams while its output is just one stream; the latter is the non-deterministic merge of the elements of the former.

The introduction of parallelism leads to a certain loss of laziness:

- Processes are eagerly created even if the output of the new process has not still been demanded.
- Communication is not realized without demand; when a process is created, it is initiated the evaluation of the expressions which will yield the values to be communicated through its channels.

The evaluation of an expression finishes when a weak head normal form (whnf) is reached. However, when this value has to be communicated and it is not a λ -abstraction, it will be evaluated to normal form. The head of a stream is strict, so that it will be evaluated to obtain a communicable value. However, the whole stream evaluation is lazy, allowing in this way the existence of potentially infinite streams.

Our aim in this work is to define a formal semantics to model the main characteristics of Eden. The semantics should consider the functional side of Eden as well as the parallelism and the distribution of the computation. In order to be used by a programmer, it is not necessary to include operational details. Consequently, we consider that it is more suitable to define a denotational semantics. Nevertheless, the chosen denotational model is not a direct denotational semantics, but a continuations denotational model [Ten76] where the semantic value of an expression is a function that transforms a state into another state. This decision is motivated by the wish of modelling together the laziness of the computational kernel of Eden and the *side-effects* resulting from process creations and communications. This kind of semantics has been defined before to express the meaning of laziness in [Jos89]; in that work, the author takes also into account side-effects, like the printing of results. In the case of Eden, the computed value produced by a process creation is the same as if the corresponding functional application was evaluated. However, apart from obtaining the corresponding value, a new process is created, and this creation together with the underlying communications are considered as side-effects. In the denotational model that we propose here a continuation will take into account these side-effects.

Thanks to the stream-based communication in Eden, processes can be defined whose behaviour is similar to introducing continuously characters with a keyboard. However, the modelization of streams by a denotational semantics is not direct because a stream may be infinite and the denotational value for such a stream could not be computed. However, in the model of continuations the

semantic value for an expression is a function that is well defined although its computational effect may be infinite.

We are interested in observing three different aspects of an Eden program:

- Functionality: the final value computed.
- Parallelism: the system topology, i.e. the existing processes and connections among them, and the interactions generated by the computation.
- Distribution: the degree of work duplication and speculative computation.

According to the purposes listed above, the model will allow us to capture the degree of speculative computation, which depends on the number of available processors, the scheduling decisions and the speed of basic instructions. Our model embodies two bounds, a minimal, in which only the necessary expressions are evaluated, and a maximal, where every expression is evaluated.

The same semantic model that we develop in this paper was first used in [HO03] where it was applied to a kernel of Eden that included a simple form of non-determinism but not streams. Besides, the same model was used for the languages GPH and pH, thus obtaining a common framework to compare three different ways of introducing parallelism in the functional language Haskell.

For the definition of the formal semantics we will consider an Eden simplification consisting of a lazy λ -calculus extended with explicit process creation and lists. The calculus is presented in Section 2, while Section 3 is devoted to the definition of the denotational semantics: semantic domains and semantic functions. Finally, we will show our conclusions and outline future works in Section 4.

2 The Calculus

The syntax of our calculus is detailed in Figure 1. The simple calculus that we consider here embodies Eden’s essentials: a basic λ -calculus with variables, functional abstractions and applications and local declarations, extended with process creations (parallel application $\#$) and lists for modelling streams.³

The effect of evaluating an expression like $E_1 \# E_2$ is the creation of a new process and of two channels communicating parent and child. The child will receive from its parent the result of evaluating E_2 , and then it will send the value corresponding to the application $E_1 E_2$. Notice that communication is asynchronous and implicit: there are not communication primitives such as **send** or **receive**. When the argument, E_2 , for the parallel application is a list the corresponding communication channel is a stream. The same rule applies for the result of $E_1 E_2$. The evaluation of a stream is not completely lazy: the constructor and the head must be obtained. In this way, the evaluation is element-wise.

From the point of view of the obtained final value, a parallel application is equivalent to a functional application. However, an operational observation of the computation shows that they are qualitatively different:

³ Non-determinism is not considered here because its combination with streams would make the semantic definitions more cumbersome.

Terms	Restricted terms	
$E ::= x$	x	variable
$\lambda x.E$	$\lambda x.E$	λ -abstraction
$E_1 E_2$	$x_1 x_2$	application
$E_1 \# E_2$	$x_1 \# x_2$	parallel application
$\text{let } \{x_i = E_i\}_{i=1}^n \text{ in } E$	$\text{let } \{x_i = E_i\}_{i=1}^n \text{ in } x$	local declaration
$\Pi[x_1 : x_2].E_1 \parallel E_2$	$\Pi[x_1 : x_2].E_1 \parallel E_2$	pattern matching
L	L	list
$L ::= \text{nil}$	nil	empty list
$[E_1 : E_2]$	$[x_1 : x_2]$	non-empty list

Fig. 1. Syntax

- The evaluation of a functional application argument is lazy, whereas in the case of a parallel application, i.e. the input of the new process, is eager.
- With respect to the application, the free variables of the abstraction are evaluated only under demand. However, when evaluating a parallel application there are two possibilities: either to evaluate them only if it is strictly necessary, or to generate an initial environment for the new process by previously evaluating the needed variables.

In order to facilitate the implementation in a distributed setting, Eden processes are independent entities, each one with its own memory containing all the information needed to evaluate its main output. Therefore, when a process is created it is associated to some initial environment where all the free variables of the abstraction, i.e. the body of the process, are defined. This initial environment may be built following two different alternatives: (a) Every variable is associated to a final value, or (b) the variables may be bound to expressions which are still unevaluated. The latter gives rise to a potential duplication of work, whereas the former assures that each variable is only evaluated once. In our case we have chosen the first option and, consequently, when a copy of variables takes place all of them must be already evaluated, that is, all the dependencies of free variables must be previously evaluated.

Local declaration of variables makes possible the parallel execution of several processes. The explicit parallelism, which is introduced by the $\#$ -expressions, is mostly exploited when some of the local variables are bound to parallel applications (*top-level*), because these processes will be created without other previous demand, i.e. speculatively.

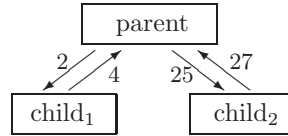
The next example illustrates the two situations where processes are created.⁴

Example 1 *Process creation*

```

let
  x = y#25, y = \z.(z + 2)
in y#2

```



⁴ In order to make the examples more understandable, we have considered natural numbers as syntactic sugar for the λ -expressions representing them.

This expression gives rise to the creation of two processes: the first child ($child_1 \equiv y\#2$) is created under demand, whereas the second ($child_2 \equiv y\#25$) is developed speculatively. Depending on the evaluation conditions, the second process will produce its result value or not.

□

We introduce a special form of abstraction ($\Pi[x_1 : x_2].E_1 \parallel E_2$) for *lists*. This construction just makes a pattern matching when applied to a list:

- Empty list: the evaluation goes on with E_1 .
- Non-empty list: E_2 is evaluated after the substitution of the formal parameters by the actual arguments.

Notice that this application is strict in its argument in order to distinguish between an empty list and a non-empty list. Consequently, we consider three kinds of application: lazy (functional), eager (parallel), and strict (pattern matching).

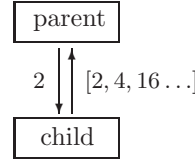
The evaluation of a list (not a stream) is lazy, and we consider that it is evaluated when its constructor (whnf) has been obtained.

The following example shows the creation of an infinite stream.

Example 2 *Infinite streams.*

The following expression gives place to two processes, one generates a stream where each element is the square of its precedent; while the other sums all the elements of that stream.

```
let s = λz.Π[y1 : y2].z[(s (z + y1) y2)
in s 0 (let p = \x.[x : (p (x * x))]
in (p # 2))
```



The output of the parent process, i.e. the sum of the stream produced by the child, will never be obtained because of the infiniteness of the stream, although the partial sums are obtained in z , the argument for s .

□

2.1 Normalization

Before defining the meaning of an expression of the given calculus, a process of normalization [Lau93] is carried out. The normalization introduces variables and local declarations in order to transform into variables both subexpressions of an application, as well as the body of a local declaration. Both the head and the tail of a non-empty list become variables too. With these transformations every subexpression gets shared, and, therefore, it is evaluated only once. This is an example of the evaluation based on the *full laziness*. Moreover, the fact that the components of applications are variables simplifies the semantic functions because it is not necessary to introduce fresh variables to model laziness. The restricted syntax was also included in the Figure 1.

Once we have described our calculus informally, we give its formal definition using a continuations-based denotational model.

3 A denotational semantics

The definition of the denotational semantics is done in three stages: definition of the semantic domains, definition of the evaluation function, and definition of the needed auxiliary functions.

3.1 The semantic domains

The semantic domains of our denotational model appear defined in the Figure 2.

$c \in \mathbf{Cont}$	$= \mathbf{State} \rightarrow \mathbf{State}$	continuations
$\kappa \in \mathbf{ECont}$	$= \mathbf{EVal} \rightarrow \mathbf{Cont}$	expression continuations
$s \in \mathbf{State}$	$= \mathbf{Env} \times \mathbf{SChan}$	states
$\rho \in \mathbf{Env}$	$= \mathbf{Id} \rightarrow (\mathbf{Val} + \{\text{undefined}\})$	environments
$v \in \mathbf{Val}$	$= \mathbf{EVal} + (\mathbf{IdProc} \times \mathbf{Clo}) + \{\text{not_ready}\}$	values
$\varepsilon \in \mathbf{EVal}$	$= (\mathbf{Abs} \times \mathbf{Ids}) + (\mathbf{Id} \times \mathbf{Id}) + \{\text{nil}\}$	expressed values
$\alpha \in \mathbf{Abs}$	$= \mathbf{Id} \rightarrow \mathbf{Clo}$	abstraction values
$\nu \in \mathbf{Clo}$	$= \mathbf{IdProc} \rightarrow \mathbf{ECont} \rightarrow \mathbf{Cont}$	closures
$\text{sch} \in \mathbf{SChan}$	$= \mathcal{P}_f(\mathbf{Chan})$	sets of channels
$ch \in \mathbf{Chan}$	$= \mathbf{IdProc} \times$ $((\mathbf{Id} + \{\text{closed}\}) \times \mathbf{SCVal}^{<>}) \times$ $\mathbf{IdProc}$	channels
$cv \in \mathbf{CVal}$	$= \mathbf{Abs} + \mathbf{CList}$	communicable values
$\sigma \in \mathbf{CList}$	$= \{\text{nil}\} + (\mathbf{CVal} \times \mathbf{CList})$	communicable lists
$\text{scv} \in \mathbf{SCVal}$	$= \mathbf{CVal}^+$	sequences
$\text{scw} \in \mathbf{SCVal}^{<>}$	$= \mathbf{SCVal} + \{<>\}$	
$I \in \mathbf{Ids}$	$= \mathcal{P}_f(\mathbf{Id})$	sets of identifiers
$p, q \in \mathbf{IdProc}$		process identifiers

Fig. 2. Semantic domains

Continuations. A continuation is a function which contains the details of the remainder of the program, that is, if an expression is inside a context, the information about this context is gathered in the continuation. In order to execute all the tasks derived from the context, a continuation is a *state transformer*, i.e. the function is applied to a state that is transformed according to the context information in the continuation. We can distinguish two kinds of continuations:

Command continuations: the computation of an imperative program is reflected in the transformations on the state caused by the execution of its instructions, but there is not a returned value. A command continuation behaves similarly, that is, it takes an state and modifies it without any information about returned values.

Expression continuations: the evaluation of a functional program yields a value. An expression continuation transforms the state taking into account this

returned value; that is, it takes the value as its argument and then behaves like a command continuation.

In our model we use both types of continuations, so that a *continuation*, $c \in \mathbf{Cont}$, transforms a state into another state and an *expression continuation*, $\kappa \in \mathbf{ECont}$, takes a value, obtained from the evaluation of an expression, and yields a continuation.

The state. Usually, a *state* is a dynamic entity representing the place where the program evaluation makes irreversible changes; they are considered irreversible because it would be very expensive to keep copies of the whole state in order to be able to return to previous versions after leaving a block. On the other hand, the *environment* is defined locally for each program block, so that its local references disappear after leaving each block. Environments are much smaller than the state and its copy is therefore feasible. This separation between environment and state is usual in the imperative languages, or in those languages that include imperative instructions —such as variable assignment—. However, in a functional language, where each variable is bound only once to a value and this association is never broken, it is more suitable to rename the variables and use only one combination state/environment, considering only one environment for the whole program instead of partial environments for each block.

Besides an environment, a state, $s \in \mathbf{State}$, includes a set of channels, $sch \in \mathbf{SChan}$, representing the actual process topology by defining a graph whose nodes are the processes and the edges are labelled by the values which have been communicated through the channels.

As usual, an *environment*, $\rho \in \mathbf{Env}$, bounds identifiers in $\mathbf{Id}$ to values; the value `undefined` indicates that the identifier has never been used.

Values. There are three possible evaluation states of a variable:

- (a) It has been completely evaluated: the variable is associated to the value which has been returned after evaluating the previously associated expression.
- (b) It has not been demanded yet: the identifier is associated to a closure.
- (c) Its evaluation has been demanded but not finished yet: the evaluation will go wrong if this variable is demanded again (*self-reference*).

In the first case, the identifier is bound to an *expressed value*, $\varepsilon \in \mathbf{EVal}$, or final value that includes *abstraction values*, $\alpha \in \mathbf{Abs}$. However, each denotational abstraction has associated a set of identifiers which contains the free variables of the syntactic abstraction.⁵ Why do we need these free variables? Because everything that is *copied* must have been evaluated previously.

The domain $\mathbf{EVal}$ includes the denotational value `nil` corresponding to the empty list and the domain of non-empty lazy lists. In normalized expressions (see Section 2.1) every subexpression is shared, and we do not want to lose this property in this denotational framework. Therefore, non-empty lists are

⁵ Although this set could be computed from the denotational value of the abstraction, that task would be much more complex.

represented by a pair of identifiers, $\mathbf{Id} \times \mathbf{Id}$: the former is bound to the value of the head, while the latter is associated to the value of the tail.

A *closure*, $\nu \in \mathbf{Clo}$, is associated to a process identifier corresponding to the process which introduced the variable in the environment. This process will be the parent in the case that the evaluation of the variable generates a new process. However, a closure does not need this information for generating its own offspring; in this case it is necessary to know in which process the closure is being evaluated. This information will be given to the closure via an argument in the semantic function. Notice that this process identifier is not partitioned the environment into different processes, and that the copy of variables is only virtual, that is, they are shared instead of being copied explicitly.

Example 3 *Process identifiers and closures.*

Let us observe the environment $\{x_1 \mapsto \lambda y.\nu_1, x_2 \mapsto \langle p, \nu_2 \rangle, x_3 \mapsto \langle p, \mathcal{E} \llbracket x_1 \# x_2 \rrbracket \rangle\}$. The ‘owner’ of x_3 is p . However, the evaluation of the closure associated to x_3 implies the creation of a new process, q ; so that the application $x_1 x_2$ will not be evaluated inside p , but inside q . We keep the identifier p because x_2 has to be evaluated inside p and the process creations caused by the evaluation of x_2 will be offspring of p , but if $x_1 x_2$ generates a process it will be a child of q . $\square$

A closure is only meaningful once the corresponding process identifier has been provided. However, it is still necessary to establish its context; for this purpose the closure has to receive an expression continuation. In this way, the closure becomes a state transformer.

When a variable is bound to the special value `not_ready` that means that the variable is being evaluated. If after applying an initial continuation to a state, the final state contains a variable bound to `not_ready`, then a self-reference—direct or indirect—has been detected.

Streams. The reason for introducing lists in our calculus is to model communication stream-channels. A non-stream channel is closed once it has sent its unique value; by contrast, a stream is kept open until the value `nil` is communicated. Thus, our denotational model needs some mechanism to express the channel state, that is either open or closed:

Open: the capability of going on communicating values through the channel is modelled by introducing in the denotational value an identifier, or variable, which corresponds, in the environment, to the list-stream obtained (or to the closure which, after being evaluated, will give place to the list-stream).

Closed: the identifier is replaced by the special value `closed`.

From these explanations it follows that the representation of a *channel*, $ch \in \mathbf{Chan}$, is a 4-tuple, which contains the information mentioned above. Besides, as infinite streams may exist, the value which is sent through a channel is a sequence of values, $scw \in \mathbf{SCVal}^{<>}$, which is built with communicable values, $cv \in \mathbf{CVal}$; at first, this sequence is empty, and represented by $<>$.

Communicable values include abstractions (for non-stream channels). In the communication channel this single value will be represented by a stream with

only one element. In case of a stream-channel, the value will be the whole sequence of sent values. These values belong to the domain **CList** of communicable lists: each of these may be either the empty list or a list whose head is a communicable value.

3.2 The evaluation function

We have already mentioned the necessity of indicating the process where an expression is going to be evaluated, that is, the parent of the potential new processes. Consequently, the type of the semantic function for the evaluation of expressions is the following:

$$\mathcal{E} :: \mathbf{Exp} \rightarrow \mathbf{IdProc} \rightarrow \mathbf{ECont} \rightarrow \mathbf{Cont}.$$

The expression continuation contains the information needed to use the value returned by the evaluation of the expression. Thus accumulating the effects of evaluating the given expression to those of the expression continuation.

The meaning of a ‘program’ E corresponds to the evaluation of the special variable *main* in an initial environment with only the binding $main \mapsto \langle p_{main}, \mathcal{E} \llbracket E \rrbracket \rangle$, where the p_{main} stands for the main process in the state.

The definition of the evaluation function appears in the Figure 3, where we use the operator $\oplus_e$ to extend/update environments, such as in $s \oplus_e \{x \mapsto \langle p, \nu \rangle\}$, $\oplus_{ch}$ in the case it is an extension/update of the set of channels of a state, and $\oplus$ when both components of the state are extended/updated. Besides, the functions *newld* and *newldProc* return a fresh variable and a fresh process identifier, respectively; the function *card* calculates the cardinal of a set.

Basic λ -calculus evaluation. The evaluation of an *identifier* “forces” the evaluation of the associated value in the given environment. We define the function *force* in the Figure 4 and it will be explained later on in the Section 3.3.

For the evaluation of a λ -*abstraction* the corresponding expressed value has to be built: the denotational abstraction—a function that takes an identifier and returns a closure—together with the set of free variables (**fv**) of the syntactic abstraction. Afterwards, the given expression continuation is applied to this semantic value.

Since *functional application* in the calculus is lazy, the evaluation of its argument is delayed, and it will only take place if it is demanded. In order to model this behaviour, the given expression continuation κ is substituted by κ' , i.e. the expression continuation for evaluating the variable corresponding to the abstraction, x_1 . In this way, once the value of the abstraction has been obtained, α , it is applied to the argument variable x_2 , and the resulting closure is evaluated with the expression continuation κ .

Parallel application evaluation. The evaluation of a *parallel application* implies the creation of a process with two communication channels: one ‘input’ from parent to child and one ‘output’ from child to parent. Each channel has a variable associated whose evaluation will give place to the (first) value to be

$$\begin{aligned}
\mathcal{E} \llbracket x \rrbracket p \kappa &= \text{force } x \kappa \\
\mathcal{E} \llbracket \lambda x. E \rrbracket p \kappa &= \kappa \langle \lambda x. \mathcal{E} \llbracket E \rrbracket, \text{fv}(\lambda x. E) \rangle \\
\mathcal{E} \llbracket x_1 x_2 \rrbracket p \kappa &= \mathcal{E} \llbracket x_1 \rrbracket p \kappa' \\
&\quad \text{where } \kappa' = \lambda \varepsilon. \lambda s. \text{case } \varepsilon \text{ of} \\
&\quad \quad \langle \alpha, I \rangle \in \mathbf{Abs} \times \mathbf{Ids} \longrightarrow (\alpha x_2) p \kappa s \\
&\quad \quad \text{otherwise} \longrightarrow \text{wrong } s \\
&\quad \text{endcase} \\
&\dots\dots\dots \\
\mathcal{E} \llbracket x_1 \# x_2 \rrbracket p \kappa &= \text{forceFV } x_1 \kappa' \quad (1) \\
&\quad \text{where } \kappa' = \lambda \varepsilon. \lambda s. \text{case } \varepsilon \text{ of} \quad (2) \\
&\quad \quad \langle \alpha, I \rangle \in \mathbf{Abs} \times \mathbf{Ids} \longrightarrow \text{forceFV } o \kappa'' s' \quad (3) \\
&\quad \quad \text{where } q = \text{newldProc } s \\
&\quad \quad \quad \{i, o\} = 2\text{Newld } s \\
&\quad \quad \quad s' = s \oplus \langle \{o \mapsto \langle q, (\alpha i) \rangle, i \mapsto \langle p, \mathcal{E} \llbracket x_2 \rrbracket \rangle\}, \\
&\quad \quad \quad \quad \{ \langle p, i, <>, q \rangle, \langle q, o, <>, p \rangle \} \rangle \\
&\quad \quad \quad \boxed{\begin{array}{l} \kappa''_{\min} = \lambda \varepsilon'. \lambda s''. \kappa \varepsilon' s'' = \kappa \\ (4) \text{ and } (5) \\ \kappa''_{\max} = \lambda \varepsilon'. \lambda s''. \kappa \varepsilon' s_o \\ \quad \text{where } s_i = \text{forceFV } i (\text{kstr } i) s'' \\ \quad \quad s_o = \text{forceFV } o (\text{kstr } o) s_i \end{array}} \\
&\quad \quad \text{otherwise} \longrightarrow \text{wrong } s \\
&\quad \text{endcase} \\
&\dots\dots\dots \\
\mathcal{E} \llbracket \text{let } \{x_i = E_i\}_n \text{ in } x \rrbracket p \kappa &= \lambda \langle \rho, \text{sch} \rangle. \mathcal{E} \llbracket x \rrbracket p \kappa' s' \\
&\quad \text{where } \{y_1, \dots, y_n\} = \text{newld } n \rho \\
&\quad \quad s' = s \oplus_e \{y_i \mapsto \langle p, \mathcal{E} \llbracket E_i[y_1/x_1, \dots, y_n/x_n] \rrbracket \rangle \mid 1 \leq i \leq n\} \\
&\quad \quad \boxed{\begin{array}{l} \kappa'_{\min} = \lambda \varepsilon. \lambda \langle \rho, \text{sch} \rangle. \kappa \varepsilon s'' \\ \quad \text{where } I = \{y_i \mid E_i \equiv x_1^i \# x_2^i \wedge (\rho y_i) \in \mathbf{IdProc} \times \mathbf{Clo}\} \\ \quad \quad m = \text{card } I \\ \quad \quad \{q_1, \dots, q_m\} = \text{newldProc } m \langle \rho, \text{sch} \rangle \\ \quad \quad s'' = s \oplus_{ch} \{ \langle p, \text{closed}, <>, q_j \rangle, \langle q_j, \text{closed}, <>, p \rangle \mid 1 \leq j \leq m \} \\ \kappa'_{\max} = \lambda \varepsilon. \lambda s. \kappa \varepsilon s_f \\ \quad \text{where } I = \{y_i \mid E_i \equiv x_1^i \# x_2^i \wedge 1 \leq i \leq n\} \\ \quad \quad s_f = \text{mforce } I s \end{array}} \\
&\dots\dots\dots \\
\mathcal{E} \llbracket \Pi[y_1 : y_2]. E_1 \llbracket E_2 \rrbracket \rrbracket p \kappa &= \\
&= \kappa \langle \lambda x. \lambda p'. \lambda \kappa'. \lambda s. \text{case } (\rho_a x) \text{ of} \\
&\quad \text{nil} \longrightarrow \mathcal{E} \llbracket E_1 \rrbracket p \kappa' s_a \\
&\quad \langle z_1, z_2 \rangle \longrightarrow \mathcal{E} \llbracket E_2[z_1/y_1, z_2/y_2] \rrbracket p \kappa' s_a \\
&\quad \text{otherwise} \longrightarrow \text{wrong } s_a \\
&\quad \text{endcase}, \\
&\quad \text{fv}(\Pi[y_1 : y_2]. E_1 \llbracket E_2 \rrbracket) \rangle \\
&\quad \text{where } s_a = \langle \rho_a, \text{sch}_a \rangle = \text{force } x \text{ id}_\kappa s \\
&\dots\dots\dots \\
\mathcal{E} \llbracket \text{nil} \rrbracket p \kappa &= \kappa \text{ nil} \\
&\dots\dots\dots \\
\mathcal{E} \llbracket [x_1 : x_2] \rrbracket p \kappa &= \kappa \langle x_1, x_2 \rangle
\end{aligned}$$

Fig. 3. Evaluation function $\mathcal{E}$

communicated. In the following we designate by i the variable corresponding to the input channel, while o is for the output.

The evaluation of a parallel application goes through the following ‘stages’:

- (a) To force the evaluation of the variable x_1 corresponding to the abstraction as well as all its free dependencies.
- (b) The context of x_1 is gathered in the expression continuation κ' , which creates two new channels communicating the parent p and the new child q , one suspended in i and the second waiting in o . Both of them are created with the initial value $\langle \rangle$.
- (c) To evaluate the application which is bound to the variable o . This evaluation yields a value that is communicated from the new process to its parent. All these tasks are carried out by the function **forceFV** (see Section 3.3).
- (d) In the case of a *maximal semantics*, after evaluating the application, if the channel is a stream, i.e. the identifier is bound to a list, then it must be evaluated completely. Again, the function **forceFV** is invoked, and using the expression continuation **kstr** x , each component of the stream is forced, free dependencies as well.
- (e) Also in the case of a *maximal semantics*, both channels have to be evaluated completely. Therefore the expression continuation forces i as well as o .

Local declaration evaluation. Before evaluating the body of a *local declaration of variables* it is compulsory to introduce the declared variables into the environment. A renaming is necessary to avoid name clashes. For each x_i ($1 \leq i \leq n$), a fresh variable y_i is introduced, which is associated to the closure $\mathcal{E} \llbracket E_i[y_j/x_j] \rrbracket$. Each of these closures is labelled by p , the identifier corresponding to the process where the evaluation takes place. The different levels of speculation are achieved by the new expression continuation κ' . To model a minimal semantics, κ' must create the top-level processes encountered in the local declaration, but the values of their channels are not demanded. In the case of a maximal semantics, every local variable which was bound originally to a parallel application is forced; and this causes the creation of new processes and channels.

List evaluation. The evaluation function $\mathcal{E}$ for *pattern matching* is similar to that of the functional abstraction. The only difference, due to the strictness of a pattern-matching, is the way of building the abstraction: first of all, the argument must be forced, and then the evaluation proceeds according to the form obtained: if the value is an empty list, the returned closure will be determined by the expression E_1 ; whereas in the case of a non-empty list, the corresponding closure is built with E_2 . Any other case is considered erroneous, and the continuation **wrong** is returned.

The evaluation of a *list* (empty or non-empty) applies the expression continuation to the corresponding denotational value.

In the next section we define and explain the auxiliary semantic functions that have been used for the evaluation function $\mathcal{E}$.

3.3 Auxiliary semantic functions

In this section we give the definition of those semantic functions that have occurred in the definition of $\mathcal{E}$. Some of them are used to *force* the demanded variables, while others deal with *streams* and compose the sequences of the communication values.

Forcing the evaluation. The function *force* just decides how the demanded variable must be forced, since in the case of a communication variable (which appears in a channel definition) the function *forceFV* is used to propagate the forcing to the free dependencies. The function *forceS* (*simple force*) compels the evaluation of an identifier. The context where this evaluation takes place is included in the expression continuation, and the result of this forcing is a continuation. There are three possibilities:

- (a) The identifier is bound to an expressed value: just apply the remainder of the program —expression continuation κ — to that value.
- (b) The identifier is bound to a closure which must be evaluated in the appropriate process. While the identifier is being evaluated it is bound to the value `not_ready`, once the value is obtained, it is bound to the variable —this association is carried out by the expression continuation κ' —, and the initial expression continuation, κ , is applied to that value.
- (c) Otherwise, the variable is `undefined` (has never been declared) or `not_ready` (it is being evaluated, i.e. it is a self-reference). In both cases it is an error interpreted by the continuation `wrong`.

The main details derived from the evaluation of streams appear in the function *forceFV*. After having forced a variable three cases are possible. At each case one must distinguish whether the variable is for communication or not. First of all the variable is forced. Thereafter, the free variables are evaluated.

- (a) If the obtained value is an abstraction: its free dependencies are also demanded, and thus evaluated recursively. If it is the case of a communication variable then the abstraction is communicated and the channel is closed. Finally, the expression continuation is applied to the value which has been obtained.
- (b) If the obtained value is `nil`: the expression continuation is applied to `nil` and to the state passed as argument. In the case of a communication the channel is closed.
- (c) Otherwise, the obtained value is a non-empty list: if the variable does not correspond to a communication then the whole list must be evaluated to normal form; thus, its head and its tail are forced, propagating the demand to their free dependencies. However, in the case of a communication, the list is a stream and the head is forced —together with all its free dependencies— in order to be communicated, and the communication identifier in the channel is substituted by a fresh one which is bound to the tail of the stream.

The function *mforce* (*multiple forcing*) forces a set of identifiers. Therefore, its definition invokes repeatedly the function *force*.

<pre> force :: Id → ECont → Cont force x κ = λ⟨ρ, sch⟩.case ⟨p, x, cv q⟩ ∈ sch of true → forceFV x κ false → forceS x κ endcase </pre>	<pre> forceS :: Id → ECont → Cont forceS x κ = λ⟨ρ, sch⟩.case (ρ x) of ε ∈ EVal → κ ε ⟨ρ, sch⟩ ⟨p, ν⟩ ∈ (IdProc × Clo) → ν p κ' s'' where s = ⟨ρ, sch⟩ κ' = λε'.λs'. κ ε' (s' ⊕_e {x ↦ ε'}) s'' = s ⊕_e {x ↦ not_ready} otherwise → wrong ⟨ρ, sch⟩ endcase </pre>
---	---

.....

```

forceFV :: Id → ECont → Cont
forceFV x κ = forceS x κ'
  where κ' = λε.λ⟨ρ, sch⟩.case ε of
    (a) ⟨α, I⟩ ∈ Abs × Ids → κ ε s'_f
      where s_f = ⟨ρ_f, sch_f⟩ = mforceFV I s
            s'_f = case ⟨p, x, scw, q⟩ ∈ sch_f of
              false → s_f
              true  → case ⟨p, x, scw, q⟩ of
                ⟨p, x, <>, q⟩ → s ⊕ch {⟨p, closed, <α>, q⟩}
                ⟨p, x, scv, q⟩ → s_f ⊕ch {⟨p, closed, scv ++ α, q⟩}
              endcase
            endcase
    (b) nil → case ⟨p, x, scw, q⟩ ∈ sch of
      false → κ nil s
      true  → κ ε s'
      where s' = case ⟨p, x, scw, q⟩ ∈ sch of
        ⟨p, x, <>, q⟩ → s ⊕ch {⟨p, closed, <nil>, q⟩}
        ⟨p, x, scv, q⟩ → s ⊕ch {⟨p, closed, scv ++ nil, q⟩}
      endcase
    endcase
    (c) ⟨x_h, x_t⟩ ∈ Id × Id → case ⟨p, x, scw, q⟩ ∈ sch of
      false → κ ε s_f
      where s_f = mforceFV {x_h, x_t} s
      true  → κ ⟨x_h, y⟩ s'_h
      where s_h = ⟨ρ_h, sch_h⟩ = forceFV x_h id_κ s
            s'_h = case cve_h of
              cv_h ∈ CVal → case ⟨p, x, scw', q⟩ ∈ sch_h of
                ⟨p, x, <>, q⟩ → s_h ⊕ {⟨y ↦ ⟨p, E [x_t]⟩, x ↦ ⟨x_h, y⟩⟩,
                                   {⟨p, y, cv_h, q⟩}}
                ⟨p, x, scv, q⟩ → s_h ⊕ {⟨y ↦ ⟨p, E [x_t]⟩, x ↦ ⟨x_h, y⟩⟩,
                                   {⟨p, y, scv ++ cv_h, q⟩}}
              endcase
            otherwise → wrong s_h
            endcase
      y = newld s_h
      cve_h = comval x_h s_h
    endcase
  endcase
  s = ⟨ρ, sch⟩

```

Fig. 4. Auxiliary semantic functions for forcing

Auxiliary functions for streams. The function $++$ adds a new item in the sequence of values that have been communicated through a stream-channel. The function comval (*composition of values*) ‘composes’ the value to be communicated, whereas the function kstr (expression continuation of stream) continues with the evaluation of a stream. Their definition is as follows:

$\text{comval} :: \text{Id} \rightarrow \text{State} \rightarrow \text{CValE}$ $\text{comval } x \ s = \text{case } (\rho x) \text{ of}$ $\quad \langle \alpha, I \rangle \longrightarrow \alpha$ $\quad \text{nil} \longrightarrow \text{nil}$ $\quad \langle x_1, x_2 \rangle \longrightarrow \langle (\text{comval } x_1 \ s), (\text{comval } x_2 \ s) \rangle$ endcase	$\text{kstr} :: \text{Id} \rightarrow \text{ECont}$ $\text{kstr } x = \lambda \varepsilon. \lambda s. \text{case } (\rho x) \text{ of}$ $\quad \langle \alpha, I \rangle \longrightarrow \{s\}$ $\quad \text{nil} \longrightarrow \{s\}$ $\quad \langle x_h, x_t \rangle \longrightarrow \text{forceFV } x_h \ (\text{kstr } x_t) \ s$ endcase
---	--

The value communicated through a stream may be a list—which is then evaluated to normal form—and to add this list to the sequence of the channel it has first to be ‘composed’, that is, it has to be transformed into a list of communicable values. This task is carried out by the function comval . Initially, the list to be composed belongs to the domain of generic list $\text{CValE} = \text{Abs} + \text{CList} + \{\text{nil}\} + (\text{CValE} \times \text{CValE})$ which, in particular, contains the domain CList . However, this list may be ‘amorphous’—it may not finish with the empty list—and we do not allow to communicate a list which is not well-formed.

Finally, during process creation a special expression continuation kstr is used: if the variable is bound to a non-empty list, the head and the tail are forced; otherwise, the function is like the identity expression continuation.

Example 4 *The denotational value of a stream.*

Let us consider the following expression:

$$\text{let } x_0 = [x_1 : x_2], x_1 = x_3 \ x_3, x_2 = \text{nil}, x_3 = \backslash x.x, x_4 = x_3 \# x_0 \text{ in } x_4$$

where the evaluation of x_4 demands the creation of a process for evaluating $x_3 \ x_0$. The input for this process is a stream because x_0 evaluates to a list.

The initial environment is:

$$\rho_0 = \{x_i \mapsto \text{undefined}\} \oplus_e$$

$$\{ \text{main} \mapsto \langle p_{\text{main}}, \mathcal{E} [\text{let } x_0 = [x_1 : x_2], x_1 = x_3 \ x_3, x_2 = \text{nil}, x_3 = \backslash x.x, x_4 = x_3 \# x_0 \text{ in } x_4] \rangle \}$$

And the final state obtained from $s_0 = \langle \rho_0, \emptyset \rangle$ and $\kappa_0 = id_\kappa$:

(a) $\mathcal{E}_{\text{min}} [\text{main}] p_{\text{main}} \kappa_0 s_0 =$

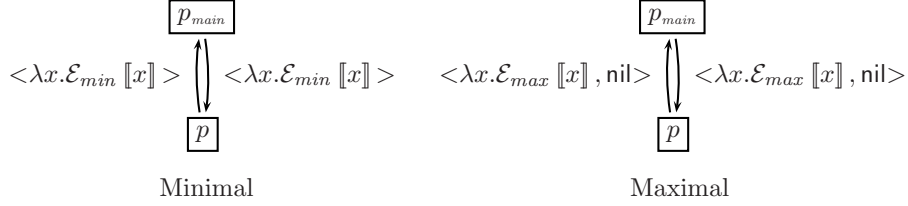
$$\begin{aligned} & \langle \{x_0 \mapsto \langle x_1, x_2 \rangle, x_1 \mapsto \langle \lambda x. \mathcal{E}_{\text{min}} [x], \emptyset \rangle, x_2 \mapsto \langle p_{\text{main}}, \mathcal{E}_{\text{min}} [\text{nil}] \rangle, \\ & \quad x_3 \mapsto \langle \lambda x. \mathcal{E}_{\text{min}} [x], \emptyset \rangle, x_4 \mapsto \langle x_1, c_1 \rangle, \text{main} \mapsto \langle x_1, c_1 \rangle, \\ & \quad c_0 \mapsto \langle p_{\text{main}}, \mathcal{E}_{\text{min}} [x_2] \rangle, c_1 \mapsto \langle p, \mathcal{E}_{\text{min}} [c_0] \rangle, o \mapsto \langle x_1, c_1 \rangle, i \mapsto \langle x_1, c_0 \rangle \}, \\ & \quad \{ \langle p_{\text{main}}, c_0, \langle \lambda x. \mathcal{E}_{\text{min}} [x] \rangle, p \rangle, \langle p, c_1, \langle \lambda x. \mathcal{E}_{\text{min}} [x] \rangle, p_{\text{main}} \rangle \} \rangle \end{aligned}$$

(b) $\mathcal{E}_{\text{max}} [\text{main}] p_{\text{main}} \kappa_0 s_0 =$

$$\begin{aligned} & \langle \{x_0 \mapsto \langle x_1, x_2 \rangle, x_1 \mapsto \langle \lambda x. \mathcal{E}_{\text{max}} [x], \emptyset \rangle, x_2 \mapsto \text{nil}, \\ & \quad x_3 \mapsto \langle \lambda x. \mathcal{E}_{\text{max}} [x], \emptyset \rangle, x_4 \mapsto \langle x_1, c_1 \rangle, \text{main} \mapsto \langle x_1, c_1 \rangle, \\ & \quad c_0 \mapsto \text{nil}, c_1 \mapsto \text{nil}, o \mapsto \langle x_1, c_1 \rangle, i \mapsto \langle x_1, c_0 \rangle \}, \\ & \quad \{ \langle p_{\text{main}}, \text{closed}, \langle \lambda x. \mathcal{E}_{\text{max}} [x], \text{nil} \rangle, p \rangle, \langle p, \text{closed}, \langle \lambda x. \mathcal{E}_{\text{max}} [x], \text{nil} \rangle, p_{\text{main}} \rangle \} \rangle \end{aligned}$$

From these final states we conclude that the values bound to *main* are the same in both cases, but in the case of the minimal semantics, the channels have been left open because only the heads have been demanded, while in the maximal semantics they have been wholly evaluated and closed. Moreover, in the maximal semantics all the variables, for instance x_2 , have been evaluated.

Graphically, the process topologies are the following:



□

4 Conclusions and future work

Some colleagues may argue that the denotational semantics presented here has not a very high level of abstraction because continuations are too ‘operational’. And they are right. Denotational semantics with continuations were defined in the seventies [Sto77, Ten76]. In our case, the choice of a denotational model based on continuations —instead of a direct denotational semantics— has allowed us to express the combination of a lazy computational kernel with the *side-effects* produced by the coordination layer, i.e. process creations and underlying communications. In fact, a continuation takes into account these side-effects produced during the evaluation of some expression. In short, we have defined a formal model for a lazy λ -calculus that is suitable to describe parallel processes.

Using the defined semantics, we have been able to associate each program with two denotations: a *minimal* one, which represents an almost completely lazy evaluation —processes are created eagerly but their bodies are evaluated only if they are demanded— and a *maximal* one, where laziness is restricted to functional applications. Consequently, the set of all the possible resulting states is between the minimal and the maximal denotations.

Our denotational model also allows to extract the degree of parallelism and the amount of speculative computation. In the final state, the nodes of the graph corresponding to the set of channels are the processes which have been created in the system during the evaluation of the main expression. Other degrees of parallelism can be obtained by modifying the expression continuation for $\#$ -expressions and local declarations; these degrees would be greater than the minimal one and smaller than the maximal one. One might, for instance, demand the evaluation of the output but not of the input of a process, or to evaluate only some subset of the parallel applications in a local declaration depending on the number of available processors. In order to get information of the speculative computation, we must analyze the edges of the system: if the edge from a child to its parent is labelled with $\langle \rangle$ then the child (together with all its offspring) is a speculative process because its output has not been used for the main result.

Although this denotational model is suitable for studying the equivalence between process systems, it has also some limitations because its abstraction level does not allow, for instance, to observe work duplication.

Regarding the future, the defined semantics can be used to prove the correctness of the transformations defined for Eden in [PPRS00]. Besides, we want to extend the calculus with dynamic channels.

We are working on the formal relationship between the operational semantics in [HO02] and the denotational one of this paper. This relationship will be based on the process topology which can be obtained from both semantics. Although in the operational model the topology cannot be built from the final system, this information can be obtained from the whole computation as we have just mentioned. In the denotational semantics this structure is contained in the channels set of the state. We want to prove that if the initial system is the same then the structure obtained from the operational computation is equivalent to the one obtained from the channels set of the final state in the denotational semantics.

References

- [HO02] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters (World Scientific Publishing Company)*, 12(2):211–228, 2002.
- [HO03] M. Hidalgo-Herrero and Y. Ortega-Mallén. Continuation semantics for parallel Haskell dialects. In *Proc. of the 1st Asian Symposium on Programming Languages and Systems*, pages 303–321. LNCS 2895, Springer, 2003.
- [Jos89] M. B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68:105–111, 1989.
- [Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages, POPL'93*, pages 144–154. ACM Press, 1993.
- [LOP04] R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *Journal of Functional Programming*, 2004. (To appear).
- [PPRS00] C. Pareja, R. Peña, F. Rubio, and C. Segura. Optimizing Eden by Transformation. In *Trends in Functional Programming (Selected papers of the 2nd Scottish Functional Programming Workshop)*, volume 2, pages 13–26. Intellect, 2000.
- [PRS02] R. Peña, F. Rubio, and C. Segura. Deriving non-hierarchical process topologies. In *Trends in Functional Programming (Selected papers of the 3rd Scottish Functional Programming Workshop)*, volume 3, pages 51–62. Intellect, 2002.
- [Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.
- [Ten76] R. D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453, August 1976.

Analysing Definitional Trees [★]

Pascual Julián Iranzo and Christian Villamizar Lamus

Dep. de Informática, Univ. de Castilla-La Mancha, Ciudad Real, Spain.

Pascual.Julian@uclm.es cvillami@inf-cr.uclm.es

1 Introduction

This paper summarizes the work presented in [7], where we investigate how an analysis of definitional trees can improve the quality of the Prolog code generated by high level implementations of (needed) narrowing.

2 Definitional Trees and Narrowing Implementations into Prolog

As it is well known, Needed Narrowing (NN) [1] is the standard operational mechanism of functional logic languages. NN relies on the notion of a definitional tree. A definitional tree is a tree structure whose leaves¹ contain (variants of) all the rules defining the function and each inner node is a pattern², where the inductive position is surrounded by a box. Inductive positions point out the order in which the arguments of a function call are evaluated.

Example 1. Given the rules defining the function $f/3$

$$R_1 : f(a, b, X) \rightarrow r_1, \quad R_2 : f(b, a, c) \rightarrow r_2, \quad R_3 : f(c, b, X) \rightarrow r_3.$$

the definitional tree of f is the one depicted in Fig. 1(a).

We are mainly interested in inductively sequential programs, i.e., programs where every defined function has an associated definitional tree.

In the last decade, a great effort has been done to provide the integrated languages with high level implementations of NN into Prolog [3, 5, 8]. These implementation systems are based on a two-phase transformation procedure that consists of: (i) an algorithm that obtains a suitable representation structure for the definitional trees associated with a functional logic program; and (ii) an algorithm that takes the above representation of definitional trees as an input parameter and visits its nodes generating a Prolog clause for each visited node. For languages with a NN semantics, a concrete instance of this translation procedure is given in [5]. When we apply that algorithm to the standard definitional tree of function f in Example 1, we obtain the following set of Prolog clauses:

[★] Supported by CICYT under grant TIC 2001-2705-C03-01.

¹ For the sake of simplicity, we omit the right hand side of the rules in the leaves of the graphical representation of a definitional tree.

² A *pattern* is a function symbol applied to a tuple of constructor terms.

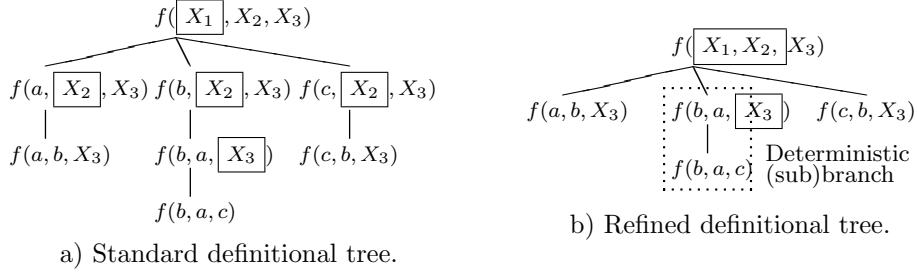


Fig. 1. Definitional trees for the function “ f ” of Example 1

```
% Clause for the root node: it exploits the first inductive position
f(X1, X2, X3, H) :- hnf(X1, HX1), f_1(HX1, X2, X3, H).
% Clauses for the remaining nodes:
f_1(a, X2, X3, H) :- hnf(X2, HX2), f_1_a_2(HX2, X3, H).
f_1_a_2(b, X3, H) :- hnf(r1, H).
f_1(b, X2, X3, H) :- hnf(X2, HX2), f_1_b_2(HX2, X3, H).
f_1_b_2(a, X3, H) :- hnf(X3, HX3), f_1_b_2_a_3(HX3, H).
f_1_b_2_a_3(c, H) :- hnf(r2, H).
f_1(c, X2, X3, H) :- hnf(X2, HX2), f_1_c_2(HX2, X3, H).
f_1_c_2(b, X3, H) :- hnf(r3, H).
```

where $\text{hnf}(T, H)$ is a predicate that is true when H is the head normal form³ (hnf) of a term T . The execution of this clauses conforms with the NN strategy.

3 A Refined Representation of Definitional Trees

The main idea of the refinement is as follows: when a pattern has several inductive positions, exploit them altogether. Therefore we need a criterion to detect inductive positions. We use the concept of uniformly demanded position (of [8]). A variable position of a pattern is uniformly demanded if and only if a constructor symbol appears at the corresponding position of each left hand side of a rule subsumed by the pattern. It is possible to prove that every inductive position of a pattern is uniformly demanded by the rules subsumed by that pattern. This give us a syntactic criterion to detect if a variable position of a pattern is an inductive position or not and, therefore, a guideline to build a refined definitional tree. We propose the following algorithm, given a pattern: (i) select a tuple of uniformly demanded positions for that pattern; fix them as inductive positions of the branch node and generate the corresponding child nodes. (ii) If the pattern doesn't have uniformly demanded positions and it is a variant of a left hand side of a rule, generate a leaf node and go on with its brothers. (iii) Otherwise, it is impossible to build the definitional tree, so return a fail condition.

The generic pattern $f(X_1, X_2, X_3)$ of the function f in Example 1 has two inductive positions and our algorithm produces the refined definitional tree de-

³ In our context, a term t is a *head normal form* if t is a variable or it is headed by a constructor symbol.

picted in Fig. 1(b). This new representation reduces the number of nodes. Note also that, there is only one refined definitional tree for each inductively sequential function.

4 Improving Narrowing Implementations into Prolog

4.1 Translation Based on Refined Definitional Trees

It is easy to adapt the translation algorithm that appears in [5] to use our refined representation of definitional trees as input. If we apply this slightly different algorithm to the refined definitional tree of Fig. 1(b), we obtain the following set of clauses, where the inductive positions 1 and 2 are exploited altogether:

```
% Clause for the root node:
f(X1, X2, X3, H) :- hnf(X1, HX1), hnf(X2, HX2), f_1_2(HX1, HX2, X3, H).
% Clauses for the remaining nodes:
f_1_2(a, b, X3, H) :- hnf(r1, H).
f_1_2(b, a, X3, H) :- hnf(X3, HX3), f_1_2_b_a(HX3, H).
f_1_2_b_a(c, H) :- hnf(r2, H).
f_1_2(c, b, X3, H) :- hnf(r3, H).
```

We have cut the number of clauses with regard to the standard representation into Prolog (of the rules defining function f) presented in Section 2.

4.2 Selective Unfolding Transformations

The analysis of definitional trees provides further opportunities. For instance, we can take notice that the definitional tree of function f in Example 1 has a “deterministic” (sub)branch, that is, a (sub)branch whose nodes have only one child (see Fig. 1(b)). This knowledge can be used as a heuristic guide for applying determinate unfolding transformation steps selectively. For our running example, the clauses:

```
f_1_2(b, a, X3, H) :- hnf(X3, HX3), f_1_2_b_a(HX3, H).
f_1_2_b_a(c, H) :- hnf(r2, H).
```

can be merged into: $f_1_2(b, a, X3, H) :- hnf(X3, c), hnf(r2, H)$. By applying a safe unfolding transformation on determinate atom calls⁴ [4].

This selective unfolding transformation can be easily integrated inside the compilation procedure described in [5]. It suffices to introduce an additional case in order to treat deterministic (sub)branches. Roughly speaking, the new case in the algorithm of [5] can be understood as follows. If there exists a deterministic (sub)branch visit its nodes in descending order forcing that the evaluation (to hnf) of the subterms at the inductive position o of a term be the flat constructor at position o of the child node. Proceed in this way until: (i) a non deterministic node is reached; or (ii) a leaf node is reached and, in this case, evaluate the right hand side of the rule to its hnf and stop the translation.

⁴ That is, an atom that matches exactly one clause head in the Prolog code, such as the atom call $f_1_2_b_a(HX3, H)$.

5 Experiments

We have made some experiments to verify the effectiveness of our proposals. We compared the Prolog code obtained by the `curry2prolog` compiler of PAKCS [2] (an implementation of the multi-paradigm declarative language Curry [6]) with the code generated by our techniques. For our first translation technique, the one using the refined representation of definitional trees, the results of the experiments show an average speedup of 9.2% and an improvement in memory usage of 7.9% for the benchmark programs and terms (goals) considered. Finally, regarding the second translation technique, the one which implements selective unfolding transformations, we obtain an average speedup of 8.8% and an improvement in memory usage of 12.3%. We note that our translation techniques can be applied jointly or separately. In the first case their effects are accumulative.

6 Conclusions

In this paper we have introduced a refined representation of definitional trees that eliminates the indeterminism in the selection of definitional trees in the context of the NN strategy. We have defined two translation techniques based on the analysis of (refined) definitional trees. Our work shows that there is a potential for the improvement of actual (needed) narrowing implementation systems, since we obtain valuable improvements of execution time and memory allocation when our translation techniques are relevant. On the other hand, our simple translation techniques are able to eliminate some *ad hoc* artifices in actual implementations of (needed) narrowing into Prolog, providing a systematic and efficient translation mechanism.

References

1. Antoy et al. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, July 2000.
2. Antoy et al. Pakcs 1.5: The Portland Aachen Kiel Curry System User Manual. Available at URL: <http://www.informatik.uni-kiel.de/pakcs/>, 2003.
3. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into prolog. In *Proc. of FroCoS'00*, pages 171–185. Springer LNCS 1794, 2000.
4. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, pages 88–98. ACM, 1993.
5. M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. of LOPSTR'95*, pages 252–266. Springer LNCS 1048, 1995.
6. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at <http://www-i2.informatik.rwth-aachen.de/hanus/curry>, 1999.
7. Julián-Iranzo et al. Analysing definitional trees: Looking for determinism. In *Proc. of FLOPS'04*, pages 55–69. Springer LNCS 2998, 2004.
8. Loogen et al. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of PLILP'93*, pages 184–200. Springer LNCS 714, 1993.

Una aproximación lógica a la verificación de propiedades de programas lógico-funcionales

José Miguel Cleva, Javier Leach, Francisco J. López-Fraguas

Departamento de Sistemas Informáticos y Programación.
Universidad Complutense de Madrid
[jcleva, leach, fraguas]@sip.ucm.es

Resumen Este trabajo es un resumen de [2,3], donde se aborda el tema de la verificación de propiedades de programas lógico-funcionales desde un punto de vista lógico. Para ello se parte de una semántica bien conocida para este tipo de lenguajes como es la proporcionada por el marco lógico *CRWL*, y se asocia a cada *CRWL*-programa un programa lógico que servirá como base para la demostración de las propiedades del programa original. Se analizan algunos inconvenientes que se encuentran en la práctica con este enfoque, y se proponen un par de refinamientos que mejoran la efectividad de la aproximación.

1. Introducción

En los lenguajes de programación lógico-funcional como Curry [5] y Toy [6], existe la posibilidad de definir funciones perezosas mediante sistemas de reescritura no confluente y no terminante lo que, en particular, impide poder llevar a cabo un razonamiento puramente ecuacional sobre dichos sistemas.

La lógica *CRWL* (Constructor based ReWriting Logic) [4] proporciona una lógica alternativa para este tipo de lenguajes. A partir de ella, la semántica de un programa viene dada por una relación de reducción $e \rightarrow t$ entre expresiones y términos construidos parciales.

En este trabajo buscamos una base lógica para probar propiedades de programas *CRWL*. Las propiedades de interés son aquellas válidas en el modelo inicial que se corresponde generalmente bien con el modelo “pretendido” del programa.

La idea básica es asociar a cada programa *CRWL* un programa lógico (por tanto una teoría de primer orden) que exprese la relación $e \rightarrow t$. Así, intentaremos probar propiedades en esta teoría de primer orden por medio de mecanismos de deducción lógica.

2. La lógica *CRWL* y su traducción como programa lógico

Consideramos una signatura $\Sigma = DC_\Sigma \cup FS_\Sigma$, donde $DC_\Sigma = \bigcup_{n \in \mathbb{N}} DC_\Sigma^n$ es un conjunto de símbolos de constructor y $FS_\Sigma = \bigcup_{n \in \mathbb{N}} FS_\Sigma^n$ es un conjunto de símbolos de función. También se considera un conjunto numerable de variables

$\mathcal{V}$. Para poder razonar sobre expresiones parciales, se considera la extensión de la signatura $\Sigma_\perp$ con una nueva constante $\perp$. Se denominan *c-términos* aquellos que sólo usan constructores o variables y *c-términos parciales* ($CTerm_\perp$) si además se considera $\perp$. Los c-términos parciales sin variables se denominan *términos cerrados*. Un programa $CRWL$ será un conjunto finito de reglas de la forma $f(\vec{t}) \rightarrow e$ donde f es un símbolo de función, $\vec{t}$ es una tupla de c-términos donde cada variable aparece una sola vez (i.e, lineal), y e es una expresión cualquiera.

Las reglas del cálculo $CRWL$ que hemos considerado son:

$$\begin{array}{ll} \text{(BT)} \frac{}{e \rightarrow \perp} & \text{(DC)} \frac{e_1 \rightarrow t_1, \dots, e_n \rightarrow t_n}{c(e_1, \dots, e_n) \rightarrow c(t_1, \dots, t_n)} \quad c \in DC^n, t_i \in CTerm_\perp \\ \text{(FR)} \frac{e_1 \rightarrow t_1, \dots, e_n \rightarrow t_n \quad e \rightarrow t}{f(e_1, \dots, e_n) \rightarrow t} & \text{si } t \neq \perp, f(t_1, \dots, t_n) \rightarrow e \in [P]_\perp \end{array}$$

donde el conjunto $[P]_\perp$ en **(FR)** está formado por todas las instancias $(l \rightarrow r)\sigma$ de reglas de P , donde $l \rightarrow r \in P$ y σ reemplaza variables por c-términos parciales. Este cálculo es una simplificación del original [4], pero esencialmente equivalente. Usamos $P \vdash_{CRWL} e \rightarrow t$ para indicar la $CRWL$ -derivabilidad de $e \rightarrow t$ a partir del programa P . Como ejemplo, las reglas $coin \rightarrow 0$, $coin \rightarrow s(0)$, $0 + Y \rightarrow Y$, $s(X) + Y \rightarrow s(X + Y)$, $double(X) \rightarrow X + X$ forman un $CRWL$ -programa no confluyente, para el que se tiene $P \vdash_{CRWL} double(coin) \rightarrow 0$ y $P \vdash_{CRWL} double(coin) \rightarrow s(s(0))$, pero $P \not\vdash_{CRWL} double(coin) \rightarrow s(0)$, lo que refleja una semántica de *call time choice* para el indeterminismo [4].

Sea P un programa $CRWL$ con signatura $\Sigma = DC \cup FS$. El programa lógico P_L asociado se obtiene mediante la transformación directa a cláusulas de las reglas de $CRWL$ para cada constructor y regla de función del programa P . Se comprueba que la reducciones $e \rightarrow t$ válidas en el programa lógico P_L son las mismas que las que se deducen por medio de $CRWL$ para el programa original.

Las propiedades que consideramos son fórmulas φ de primer orden sobre la relación de reducción $e \rightarrow t$. Consideramos las siguientes teorías en lógica de primer orden: $T_{P_L} = \{\varphi \mid P_L \models \varphi\}$, $T_{Comp(P_L)} = \{\varphi \mid Comp(P_L) \models \varphi\}$ y $T_{M_P} = \{\varphi \mid M_{P_L} \models \varphi\}$ donde $Comp(P_L)$ es la completión de Clark de P_L y M_{P_L} es el modelo mínimo de Herbrand de P_L . Las propiedades que nos interesan son aquellas que son válidas en el modelo mínimo de P_L , es decir, las que pertenecen a T_{M_P} . Como M_{P_L} es modelo tanto del programa lógico como de su completión, tenemos la cadena de inclusiones: $T_{P_L} \subseteq T_{Comp(P_L)} \subseteq T_{M_P}$. Así, podemos usar el programa lógico y su completión para obtener propiedades de M_{P_L} mediante deducción en lógica de primer orden. Por ejemplo, en el programa anterior se tiene $P_L \models coin \rightarrow 0$ y $Comp(P_L) \models coin \not\rightarrow s(0)$. De todos modos estas dos aproximaciones no permiten demostrar muchas propiedades interesantes de M_{P_L} que son de naturaleza inductiva, como sucede en el ejemplo anterior con $\forall X, T.X \rightarrow T \Rightarrow X + 0 \rightarrow T$. Por ello extendemos la completión del programa lógico con los axiomas de inducción estructural. Llamando a esta extensión $CompInd(P_L)$ y $T_{CompInd(P_L)} = \{\varphi \mid CompInd(P_L) \models \varphi\}$, obtenemos la siguiente cadena de inclusiones (estrictas, como no es difícil comprobar): $T_{P_L} \subset T_{Comp(P_L)} \subset T_{CompInd(P_L)} \subset T_{M_P}$

Hemos trasladado esta aproximación a diversos sistemas de demostración como ITP [1], LPTP [8] e Isabelle [7] y hemos analizado las distintas dificultades que se plantean a la hora de verificar algunas de las propiedades que hemos considerado. Muchos de los problemas detectados son comunes en todas las herramientas consideradas. En primer lugar, cuando se tratan propiedades sobre reducciones inalcanzables como, por ejemplo, $double(coin) \not\rightarrow s(0)$, se debe explorar todo el espacio de posibles reducciones que se pueden dar a partir de una determinada expresión, esto hace que las pruebas crezcan de una forma considerable. Por otra parte surgen problemas dependientes del sistema que se considera, ya que al traducir las distintas teorías a los sistemas en los que demostrar las propiedades, se produce en muchos casos una pérdida de automatismo en las pruebas que se realizan.

Para resolver alguno de estos problemas hemos considerado $CRWL'$ una versión de $CRWL$ en la que se elimina la regla **BT**. Esta nueva versión tiene como ventaja que se generan menos reducciones que en la versión original y además hace que el razonamiento sobre términos sea determinista, lo que, en particular, podría ser interesante ya que podría permitir una separación de las partes determinista y no determinista del programa. Se comprueba que si consideramos reducciones a términos totales, ambos cálculos son equivalentes.

3. Axiomatización de la derivabilidad

Como dijimos anteriormente en la cadena de inclusiones de las distintas teorías, las inclusiones son estrictas. En particular, se tiene $T_{CompInd(P_L)} \subsetneq T_{M_P}$. Consideremos la función con la única regla $\text{loop} \rightarrow \text{loop}$. Es fácil ver que $\text{loop} \not\rightarrow 0$ es válido en M_{Loop_L} , pero $CompInd(Loop_L) \not\models \text{loop} \not\rightarrow 0$.

Para demostrar propiedades como la anterior, una posibilidad es poder razonar sobre las derivaciones que se obtienen en $CRWL$. De este modo vamos a asociar al programa $CRWL$ original un programa lógico $Der(P)$ que defina la relación de derivabilidad por medio de una relación ternaria $d \vdash e \rightarrow t$, donde d representa términos de primer orden para las $CRWL$ -derivaciones. Consideraremos tanto la completación como la completación inductiva de $Der(P)$, teniendo en cuenta que, en este caso, hay dos tipos de construcciones: expresiones y derivaciones, por lo que podremos razonar por inducción sobre las derivaciones.

El siguiente resultado relaciona $Der(P)$ con el cálculo original.

Proposición 1 *Para cualquier programa $CRWL$ P , expresión e y término t : $Der(P) \models \exists D.D \vdash e \rightarrow t \Leftrightarrow P \vdash_{CRWL} e \rightarrow t$ y $Comp(Der(P)) \models \nexists D.D \vdash e \rightarrow t \Rightarrow P \not\vdash_{CRWL} e \rightarrow t$*

Si consideramos la traducción $\hat{\varphi}$ de una fórmula φ de P_L donde cada aparición de $e \rightarrow t$ se sustituye por $\exists D.D \vdash e \rightarrow t$, se obtiene el siguiente resultado:

Proposición 2 *Sea φ una fórmula de primer orden sobre la relación $\rightarrow$, entonces $M_{P_L} \models \varphi \Leftrightarrow M_{Der(P)} \models \hat{\varphi}$*

Análogamente se tiene el resultado para $Comp(P_L)$ y $Comp(Der(P))$. Esto quiere decir que incorporar las derivaciones no añade nada nuevo al programa lógico ni a su compleción, la diferencia se encuentra cuando consideramos la compleción inductiva ya que utilizando ahora un razonamiento inductivo sobre las derivaciones, se puede demostrar la propiedad $loop \nrightarrow 0$.

4. Conclusiones y trabajo futuro

En estos trabajos hemos dado una primera aproximación a la verificación de propiedades de lenguajes lógico-funcionales. Nos hemos basado en el marco *CRWL* y, para estos programas, hemos considerado aquellas propiedades que son válidas en el modelo inicial. Para probar estas propiedades hemos dado una traducción del programa original a un programa lógico a partir del cual demostrar propiedades válidas en el modelo mínimo del mismo por medio de mecanismos de deducción en lógica de primer orden. Además, hemos trasladado esta aproximación a diversos sistemas de demostración automática, analizando las limitaciones prácticas de este enfoque. Hemos mejorado la eficiencia por medio de dos refinamientos: *CRWL'* y la axiomatización de la derivabilidad que permite probar más propiedades mediante deducción en lógica de primer orden.

Como trabajo futuro y desde el punto de vista práctico, intentaremos trasladar este enfoque a otros sistemas de demostración automática y, además, extenderemos el repertorio de ejemplos considerando otras propiedades no triviales. Desde el punto de vista teórico, queremos mejorar la aproximación en dos sentidos, considerando lógica de primer orden con géneros ordenados y considerando distintas extensiones de *CRWL* que incorporen orden superior o fallo.

Referencias

1. M. Clavel. *The ITP tool*. Proc. of the 1st Workshop on Logic and Language, Kronos, 55–62, 2001. Sistema en <http://www.ucm.es/info/dsip/clavel/itp>.
2. J.M. Cleva, J. Leach, F.J. López-Fraguas. *A logical approach to the verification of functional-logic programs*. Proc. of the 13th International Workshop of Functional and Logic Programming, RWTH Aachen Technical Report, pp. 33–48, 2004.
3. J.M. Cleva, J. Leach, F.J. López-Fraguas. *A logic programming approach to the verification of functional-logic programs*. Proc. Principles and Practice of Declarative Programming (PPDP'04), ACM Press, pp. 9–19, 2004.
4. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic*. Journal of Logic Programming 40(1), pp. 47–87, 1999.
5. M. Hanus (ed.), *Curry: an Integrated Functional Logic Language*, Version 0.8, April 15, 2003. <http://www-i2.informatik.uni-kiel.de/~curry/>.
6. F.J. López-Fraguas, J. Sánchez-Hernández. *TCY: A Multiparadigm Declarative System*. Proc. RTA'99, Springer LNCS 1631, pp. 244–247, 1999.
7. T. Nipkow, L.C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer LNCS 2283, 2002.
8. R.F. Stærk. *The theoretical foundations of LPTP (A logic program theorem prover)*. Journal of Logic Programming 36, pp. 241–269, 1998.

Rewriting-engined Web Sites Verification^{*}

M. Alpuente¹, D. Ballis², and M. Falaschi²

¹ DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, Apdo. 22012, 46071 Valencia, Spain. Email: `alpuente@dsic.upv.es`.

² Dip. Matematica e Informatica, Via delle Scienze 206, 33100 Udine, Italy. Email: `{demis,falaschi}@dimi.uniud.it`.

Abstract. VERDI is a system for the automated verification of Web sites which can be used to specify integrity conditions for a given Web site, and then automatically check whether these conditions are actually fulfilled. It provides a rule-based, formal specification language which allows us to define syntactic/semantic properties of the Web site as well as a verification facility which computes the requirements not fulfilled by the Web site, and helps to repair the errors by finding out incomplete/missing Web pages.

1 Introduction

The increasing complexity of Web sites calls for tools which are able to aid Web designers in the construction and maintenance of Web sites. Systematic, formal approaches can bring many benefits to quality Web sites development, giving support for automated Web site verification [3]. In [4] a formal, declarative verification algorithm is proposed, which checks a particular class of integrity constraints concerning the Web site's structure (syntactic properties), but not the contents (semantic properties) of a given instance of the site. The framework XLINKIT [3] allows one to check the consistency of distributed, heterogeneous documents as well as to fix the (possibly) inconsistent information. Its specification language is a restricted form of first order logic combined with Xpath expressions [9]. This paper presents the prototype VERDI (Verification and Rewriting for Debugging Internet sites) which provides a rule-based language for the specification and the verification of syntactic as well as semantic properties on collections of XML/XHTML documents. The prototype is based on the framework we proposed in [1, 2] and enjoys the effectiveness of rule-based computation.

We use rewriting-based technology both to specify the required properties and to formalize a verification technique, which is able to check them.

^{*} This work has been partially supported by MCYT under grants TIC2001-2705-C03-01, HU2003-0003, by Generalitat Valenciana under grant GR03/025 and by ICT for EU-India Cross Cultural Dissemination Project under grant ALA/95/23/2003/077-054.

<pre> <members> <member status="professor"> <name> mario </name> <surname> rossi </surname> </member> <member status="technician"> <name> franca </name> <surname> bianchi </surname> </member> </members> </pre>	<pre> members(member(status(professor), name(mario), surname(rossi)), member(status(technician), name(franca), surname(bianchi))) </pre>
---	--

Fig. 1. An XML document and its corresponding encoding as a ground term.

2 Denotation of Web Sites

In our framework, a *Web page* is either an XML[8] or an XHTML[7] document. Since Web pages are provided with a tree-like structure, they can be straightforwardly translated into ordinary terms of a given term algebra as shown in Figure 1. Note that XML/XHTML tag attributes can be considered as common tagged elements, and hence translated in the same way. Therefore, *Web sites* can be represented as finite sets of (ground) terms.

3 Web specification language

Web specifications formalize conditions to be fulfilled by a given Web site. Roughly speaking, a Web specification is a finite set of rules of the form $l \rightarrow r$, where l and r are terms. Some symbols in the right-hand sides of the rules may be marked by means of the symbol $\sharp$. Marking information of a given rule r is used to select the subset of the Web site in which we want to check the condition formalized by r . Intuitively, the interpretation of a rule $l \rightarrow r$ w.r.t. a Web site W is as follows: if (an instance of) l is recognized in W , also (an instance of) r must be recognized in W .

In the following we present an example of Web specification.

Example 1. Consider the following Web specification, which models some required properties of a research group Web site containing information about group members affiliation, scientific publications and personal data.

$$\begin{aligned}
 \text{hpage}(\text{status}(\text{professor})) &\rightarrow \sharp \text{hpage}(\sharp \text{status}(\sharp \text{professor}), \text{teaching}) \\
 \text{member}(\text{name}(X), \text{surname}(Y)) &\rightarrow \sharp \text{hpage}(\text{name}(X), \sharp \text{surname}(\sharp Y), \text{status}) \\
 \text{pubs}(\text{pub}(\text{name}(X), \text{surname}(Y))) &\rightarrow \sharp \text{member}(\text{name}(X), \sharp \text{surname}(\sharp Y))
 \end{aligned}$$

The first rule states that, when a home page of a professor is recognized, then that page must also include some teaching information. Here, for instance, marks are used to check the condition only on professor home pages. The second rule formalizes the following property: if there is a Web page containing a member

list, then for each member, a home page exists containing the name, the surname and the status of this member. Finally, the third rule specifies that, whenever there exists a Web page containing information about scientific publications, each author of a publication should be a member of the research group.

4 The verification technique

Our verification technique allows us to verify a Web site w.r.t. a given Web specification in order to detect incomplete and/or missing Web pages. Moreover, by analyzing the requirements not fulfilled by the Web site, we are also able to find out the missing information which is needed to repair the Web site. Since reasoning on the Web calls for formal methods specifically fitting the Web context, we developed a novel, rewriting-based technique called *partial rewriting* [1], in which the traditional pattern matching mechanism is replaced by tree *simulation* [5] in order to provide a much more suitable mechanism for recognizing patterns inside semistructured documents, which is (i) independent of the order of the tagged elements contained in a Web page and (ii) able to efficiently extract the partial information of a Web page we need to check.

Basically our verification technique works in two steps. Given a Web site W and a Web specification I , we first compute a set of requirements (that is, information that should be present in the Web site) by generating the set of all possible Web pages that can be derived from W via I by partial rewriting. Then, we check whether the computed requirements are satisfied by W using simulation and marking information. Requirements which are not fulfilled allow us to detect missing or incomplete Web pages and provide the information which is necessary to fix the Web site.

Example 2. Consider the Web specification I of Example 1 and the following Web site W :

```
W = {(1) members(status(professor),member(name(mario),surname(rossi)),
               member(status(technician),name(franca),surname(bianchi)),
               member(status(student),name(giulio),surname(verdi))),
      (2) hpage(name(mario),surname(rossi),phone(333),status(professor),
               hobbies(hobby(reading),hobby(gardening))),
      (3) hpage(name(franca),surname(bianchi),status(technician),phone(555)),
      (4) hpage(name(anna),surname(blu),status(professor),phone(444),
               teaching(course(algebra))),
      (5) pubs(pub(name(mario),surname(rossi),title(blahblah1),year(2003)),
               pub(name(anna),surname(blu),title(blahblah2),year(2002)))}
```

Then, by running the VERDI system on Web site W and Web specification I , we compute the following set of requirements

```
{(a)#hpage(name(mario),#surname(#rossi),status),
 (b)#hpage(name(franca),#surname(#bianchi),status),
 (c)#hpage(name(giulio),#surname(#verdi),status),
 (d)#hpage(#status(#professor),teaching),
 (e)#member(name(mario),#surname(#rossi)),(f)#member(name(anna),#surname(#blu)),
 (g)#hpage(name(anna),#surname(#blu),status)}
```

During the analysis of the set of requirements, our system tries to recognize the structure and the contents of each requirement inside W , yielding the following outcomes: (i) requirement (c) is missing in Web site W , (ii) Web page (1) is incomplete w.r.t requirement (f), and (ii) Web page (2) is incomplete w.r.t. requirement (d).

Informally, the outputs tell us that (i) the home page of **Giulio Verdi** is missing in W , (ii) **Anna Blu** should be a member of the research group and (iii) professor **Mario Rossi** should add some teaching information to his personal home page.

5 Implementation

The basic methodology presented so far has been implemented in the prototype VERDI (VERification and Rewriting for Debugging Internet sites), which is written in DrScheme v205 [6] and is publicly available together with a set of tests at <http://www.dimi.uniud.it/~demis/#software>.

The implementation consists of about 80 function definitions (approximately 1000 lines of source code). VERDI includes a parser for semistructured expressions (i.e. XML/XHTML documents) and Web specifications, and several modules implementing the user interface, the partial rewriting mechanism and the verification technique. The system allows the user to load a Web site consisting of a finite set of semistructured expressions together with a Web specification. Additionally, he/she can inspect the loaded data and finally check the Web pages w.r.t the Web site specification. The user interface is guided by textual menus, which are (hopefully) self-explaining. We tested the system on several real Web site examples which can be found at the URL address mentioned above. In each considered test case, we were able to detect the errors (i.e. missing and incomplete Web pages) efficiently.

References

1. M. Alpuente, D. Ballis, and M. Falaschi. A Rewriting-based Framework for Web Sites Verification. In *Proc. of RULE'04*. ENTCS, Elsevier, 2004. To appear.
2. M. Alpuente, D. Ballis, and M. Falaschi. Verdi: An Automated Tool for Web Sites Verification. In *Proc. of JELIA'04*. LNCS 3229, pp. 726–729, 2004.
3. L. Capra, W. Emmerich, A. Finkelstein, and C. Nentwich. xlinkit: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2002.
4. M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Verifying Integrity Constraints on Web Sites. In *Proc. of IJCAI'99*, vol. 2, pp. 614–619. Morgan Kaufmann, 1999.
5. M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. of IEEE FOCS'95*, pp. 453–462, 1995.
6. PLT. DrScheme web site. Available at: <http://www.drscheme.org>.
7. World Wide Web Consortium (W3C). Extensible HyperText Markup Language (XHTML), 2000. Available at: <http://www.w3.org>.
8. World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0, second edition, 1999. Available at: <http://www.w3.org>.
9. World Wide Web Consortium (W3C). XML Path Language (XPath), 1999. Available at: <http://www.w3.org>.

A General Natural Rewriting Strategy

Santiago Escobar¹, José Meseguer², and Prasanna Thati³

¹ Universidad Politécnica de Valencia, Spain
`sescobar@dsic.upv.es`

² University of Illinois at Urbana-Champaign, USA
`meseguer@cs.uiuc.edu`

³ Carnegie Mellon University, USA
`thati@cs.cmu.edu`

Abstract. We define an efficient rewriting strategy for general term rewriting systems. Several strategies have been proposed over the last two decades for rewriting, the most efficient of all being the *natural rewriting strategy* of Escobar. All the strategies so far, including natural rewriting, assume that the given term rewriting system is left-linear and constructor-based. Although these restrictions are reasonable for some functional programming languages, they limit the expressive power of equational programming languages, and they preclude certain applications of rewriting to equational theorem proving and to languages combining equational and logic programming. In [5], we proposed a conservative generalization of the natural rewriting strategy that does not require the previous assumptions for the rules and we established its soundness and completeness.

1 Introduction

A challenging problem in modern programming languages is the discovery of sound and complete evaluation strategies which are: (i) optimal w.r.t. some efficiency criterion, (ii) easily implementable, and (iii) applicable for a large class of programs. This was first addressed in a seminal paper by Huet and Levy [10], where the *strongly needed reduction* strategy was proposed. Several refinements of this strategy have been proposed over the last two decades, the most significant ones being Sekar and Ramakrishnan's *parallel needed reduction* [11], and Antoy, Echahed and Hanus' *(weakly) outermost-needed rewriting* [1]. Recently, (weakly) outermost-needed rewriting has been improved by Escobar by means of the *natural rewriting strategy* [6,7]. Natural rewriting is based on a suitable refinement of the demandedness notion associated to (weakly) outermost-needed rewriting.

However, a typical assumption of the above rewriting strategies, including the natural rewriting strategy, is that the rewrite rules are left-linear and constructor-based. These restrictions are reasonable for some functional programming languages, but they limit the expressive power of equational languages such as OBJ [9], CafeOBJ [8], ASF+SDF [4], and Maude [3], where non-linear

left-hand sides are perfectly acceptable. This extra generality is also necessary for applications of rewriting to equational theorem proving, and to languages combining equational and logic programming, since in both cases assuming left-linearity is too restrictive. Furthermore, for rewrite systems whose semantics is not equational but is instead rewriting logic based, such as rewrite rules in ELAN [2], or **Maude** system modules, the constructor-based assumption is unreasonable and almost never holds.

In summary, generalizing natural rewriting to *general* rewriting systems will extend the scope of applicability of the strategy to more expressive equational languages and to rewriting logic based languages, and will open up a much wider range of applications. In the following, we give the reader an intuitive example of how the generalized natural rewriting strategy works.

Example 1. Consider the following TRS for proving equality of arithmetic expressions built using division ($\div$), modulus or remainder ($\%$), and subtraction ($-$) operations on natural numbers.

- | | |
|---|---|
| (1) $0 \div s(N) \rightarrow 0$ | (5) $M - 0 \rightarrow M$ |
| (2) $s(M) \div s(N) \rightarrow s((M-N) \div s(N))$ | (6) $s(M) - s(N) \rightarrow M-N$ |
| (3) $M \% s(N) \rightarrow (M-s(N)) \% s(N)$ | (7) $X \approx X \rightarrow \text{True}$ |
| (4) $(0 - s(M)) \% s(N) \rightarrow N - M$ | |

Note that this TRS is not left-linear because of rule (7) and it is not constructor-based because of rule (4). Therefore, it is outside the scope of all the strategies mentioned above. Furthermore, note that the TRS is neither terminating nor confluent due to rule (3).

Consider the term $t_1 = 10! \div 0$. If we only had rules (1), (2), (5) and (6), the natural rewriting strategy of [6] would be applicable and no reductions on t_1 would be performed, since t_1 is a head-normal form. In contrast, other strategies such as outermost-needed rewriting [1] would force⁴ the evaluation of the computationally expensive subterm $10!$. Hence, we would like to generalize natural rewriting to a version that enjoys this optimality and that can also handle non-left-linear and non-constructor-based rules such as (7) and (4).

Further, consider the term $t_2 = 10! \% (1-1) \approx 10! \% 0$. We would like the generalized natural rewriting strategy to perform only the optimal computation:

$$\begin{aligned} 10! \% (\underline{s(0)-s(0)}) &\approx 10! \% 0 \rightarrow 10! \% (\underline{0-0}) \approx 10! \% 0 \\ &\rightarrow \underline{10! \% 0} \approx \underline{10! \% 0} \rightarrow \text{True} \end{aligned}$$

that avoids unnecessary reduction of the subterm $10! \% 0$ at the final rewrite step and avoids also reductions on any term $10!$.

In [5], we proposed a conservative generalization of the demandedness notion of [6,7] that drops the assumptions that the rewrite rules are left-linear and constructor-based, while retaining soundness and completeness w.r.t. head-normal forms. In the following, we provide some useful insights about this generalization and refer the reader to [5] for further details.

⁴ See [6, Example 21] for a formal justification of this fact.

2 Generalized Natural Rewriting

We are interested in a lazy strategy that, to the extent possible, performs only those reductions that are essential for reaching head-normal forms, which is consistent with a lazy behavior of functional programs. We adopt the approach of computing a *demanded* set of redexes in t such that *at least* one of the redexes in the demanded set has to be reduced before *any* rule can be applied at the root position in t ; this is a common idea in lazy evaluation strategies for programming languages.

The basic approach is that if a term t is not a head-normal form, then we know that after a (possibly empty) sequence of rewrites at positions below the root, a rule $l \rightarrow r$ can be applied at the root. Thus, we compare the term t with each left-hand side l in the TRS and obtain the *least general context* between t and l , i.e., the maximal common parts between the terms t and l , in order to bound what we called *disagreeing positions* between t and l . In [5], we defined an operator $\mathcal{Pos}_{\neq}(T)$ for a set of terms T that returns the set of disagreeing positions between all the terms in T . Also, we defined an operator $DP_l(t)$ that returns the set of disagreeing positions between t and l and which uses $\mathcal{Pos}_{\neq}(T)$ in order to deal with non-linear variables in l .

Example 2. Consider the left-hand side $l_7 = \mathbf{X} \approx \mathbf{X}$ of the rule (7) and the term $t_2 = 10! \% (1-1) \approx 10! \% 0$ of Example 1. The least general context of l_7 and t_2 is $s = \mathbf{W} \approx \mathbf{Y}$. Now, while computing $DP_{l_7}(t_2)$, we obtain the set of disagreeing positions between the subterms in t_2 corresponding to the non-linear variable $\mathbf{X}$ in l_7 , i.e. the set $\mathcal{Pos}_{\neq}(10! \% (1-1), 10! \% 0) = \{2\}$. And thus we conclude that $DP_{l_7}(t_2) = \{1, 2\} \cdot \{2\} = \{1.2, 2.2\}$.

Moreover, in [5], we filtered some disagreeing positions using the notion that t is *failing w.r.t.* the left-hand side l , denoted by $l \blacktriangleleft t$. We say t is failing w.r.t. l if there is no rewrite sequence starting from t such that the first reduction in the sequence at the root position uses the rule $l \rightarrow r$. This notion is ultimately related to the notion of constructor clash between a term and a left-hand side, which is also common in lazy evaluation strategies for programming languages.

Example 3. Consider the terms $t = 10! \% 0$ and $l_3 = \mathbf{M} \% \mathbf{s}(\mathbf{N})$ from Example 1. We have that $l_3 \blacktriangleleft t$ because the position $2 \in DP_{l_3}(t)$ has two different symbols at l_3 and t and no possible reduction can make them equal; similarly $l_4 \blacktriangleleft t$. Now, consider the terms $t' = \mathbf{s}(\mathbf{Z}) \approx 0$ and $l_7 = \mathbf{X} \approx \mathbf{X}$, again from Example 1. In this case, which considers non-linear variables in l_7 , we have $l_7 \blacktriangleleft t'$, since no reduction can make positions 1 and 2 in $DP_{l_7}(t')$ equal.

Then, in [5], we collected all the demandedness information obtained from the left-hand sides that are not failing and built a set of *demanded redexes*, denoted by $DR(t)$, which recursively calls itself on the demanded positions obtained by each $DP_l(t)$, for all l in the TRS, and their positions above them. However, a further refinement was included in [5], since not all the demanded positions are inspected by $DR(t)$, but only those positions which are the *most frequently demanded positions* and which cover all the rules in the TRS.

Example 4. Continuing all the previous Examples, consider the computation of the set $DR(t_2)$. We have that $DP_{l_7}(t_2) = \{1.2, 2.2\}$ for the left-hand side l_7 of the rule (7). Then $DR(t_2)$ calls itself recursively on positions 1.2, 2.2, and their positions above them, i.e., positions 1 and 2. Position 2.2 is rooted by a constructor symbol and no rule is applicable, so $DR(t_{2|2.2}) = \emptyset$. By Example 3, subterm at position 2 is failing w.r.t. the TRS, so $DR(t_{2|2}) = \emptyset$. Position 1.2 is already a redex, so we say $DR(t_{2|1.2}) = \{\langle A, (6) \rangle\}$. Now, consider the subterm $t_{2|1} = 10! \% (1-1)$ and the left-hand sides of the rules (3) and (4). We have that $DP_{l_3}(t_{2|1}) = \{2\}$ and $DP_{l_4}(t_{2|1}) = \{1, 2\}$. However, the set $P = \{2\}$ covers $DP_{l_3}(t)$ and $DP_{l_4}(t)$ and then, position 2 is considered as the most frequently demanded position for $t_{2|1}$ in the TRS. Then, $DR(t_{2|1})$ will call recursively to $DR(t_{2|1.2})$, which was computed before, and finally we conclude $DR(t_{2|1}) = \{\langle 2, (6) \rangle\}$ and, obviously, $DR(t_2) = \{\langle 1.2, (6) \rangle\}$. That is, we have avoided any unnecessary reduction, following the optimal sequence of Example 1.

References

1. S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming ALP'92*, volume 632 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, Berlin, 1992.
2. P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
4. A. van Deursen, J. Heering, and P. Klint. *Language Prototyping: An Algebraic Specification Approach*. World Scientific, 1996.
5. S. Escobar, J. Meseguer, and P. Thati. Natural rewriting for general term rewriting systems. In *Pre-proceedings of 14th International Workshop on Logic-based Program Synthesis and Transformation, LOPSTR'04*, 2004. To appear.
6. S. Escobar. Refining weakly outermost-needed rewriting and narrowing. In D. Miller, editor, *Proc. of 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'03*, pages 113–123. ACM Press, New York, 2003.
7. S. Escobar. Implementing natural rewriting and narrowing efficiently. In Yuki Yoshi Kameyama and Peter J. Stuckey, editors, *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, volume 2998 of *Lecture Notes in Computer Science*, pages 147–162. Springer-Verlag, Berlin, 2004.
8. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
9. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.
10. G. Huet and J.-J. Lévy. Computations in Orthogonal Term Rewriting Systems, Part I + II. In *Computational logic: Essays in honour of J. Alan Robinson*, pages 395–414 and 415–443. The MIT Press, Cambridge, MA, 1992.
11. R.C. Sekar and I.V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. *Information and Computation*, 104(1):78–109, 1993.

A tutorial on specifying data structures in Maude^{*}

Narciso Martí-Oliet, Miguel Palomino, and Alberto Verdejo

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
`narciso,miguelpt,alberto@sip.ucm.es`

Abstract. This tutorial describes the equational specification of a series of typical data structures in Maude. We start with the well-known stacks, queues, and lists, to continue with binary and search trees. Not only are the simple versions considered but also advanced ones such as AVL and 2-3-4 trees. The operator attributes available in Maude allow the specification of data based on constructors that satisfy some equational properties, like concatenation of lists which is associative and has the empty list as identity, as opposed to the free constructors available in other functional programming languages. Moreover, the expressive version of equational logic in which Maude is based, namely membership equational logic, allows the faithful specification of types whose data are defined not only by means of constructors, but also by the satisfaction of additional properties, like sorted lists or search trees. In the second part of the paper we describe the use of an inductive theorem prover, ITP, which itself is developed and integrated in Maude by means of the powerful metalevel and metalanguage features offered by the latter, to prove properties of the data structures. This is work in progress because the ITP is still under development and, as soon as the data gets a bit complex, the proof of their properties gets even more complex.

Keywords: Data structures, algebraic specification, membership equational logic, Maude, inductive theorem proving.

1 Introduction

Maude is a declarative language and system based on rewriting logic [6, 5]. Even though both the language and the system keep being improved, they reached maturity with the public release of version 2 in the summer of 2003. Since then, Maude is being used throughout the world in teaching and research, being specially useful for the specification and prototyping of logical systems, programming languages, and computational systems in general. However, there is still a lack of common libraries that could be shared and reused.

This tutorial tries to contribute to fill this gap by providing a library of typical data structures specified in Maude. This is accomplished by updating

^{*} Research partially supported by MCyT Spanish projects *MIDAS* (TIC2003-01000) and *MELODIAS* (TIC2002-01167).

and considerably extending the set of specifications that were available in the tutorial distributed with version 1 of Maude [4]. More specifically, we start by describing well-known versions of basic datatypes such as stacks, queues, and lists; we then continue with several versions of trees, including binary, general, and search trees; we do not consider only the simple versions, but also advanced ones such as AVL, 2-3-4, and red-black trees; finally, we describe an abstract version of priority queues and a more concrete one based on leftist trees.

For all of these specifications we do not need to consider rewriting logic in its full generality, but just its equational sublogic, namely, membership equational logic [11] (from the Maude language point of view, all of our specifications are *functional modules*). A very important point is that the expressivity of this equational logic allows the faithful specification of types whose data are defined not only by means of constructors, but also by the satisfaction of additional properties, like sorted lists, search trees, balanced trees, etc. We will see along the paper how this is accomplished by means of membership assertions that equationally characterize the properties satisfied by the corresponding data.

All the datatypes that we consider are generic, that is, they are constructions on top of other datatypes that appear as parameters in the construction. Therefore, our specifications are *parameterized* and, for this reason, we use Full Maude [6], which provides powerful mechanisms for parameterization based on *theories* that describe the requirements that a data type must satisfy for the construction to make sense. For example, lists can be constructed on top of any data whatsoever, but sorted lists only make sense for data that have a total order; for a binary operation to be a total order, several properties have to be satisfied, which are written in the corresponding parameter theory as equations.

We assume some knowledge about the data structures that are specified. There are many textbooks that describe well-known imperative and object-oriented implementations [9, 2, 16]. Less known, but very useful for our purposes, are implementations in functional programming languages such as ML or Haskell [12, 15, 14]; in some cases, our equations are very similar to the ones given in such texts. On the other hand, we do not assume much knowledge about Maude and thus we describe the main features as they come out along with the specifications.

As mentioned before, here we do not consider at all rule-based programming in Maude. For an introduction to those features, we refer the interested reader to the paper [13].

All the code in this paper and more can be found in the web page [10].

2 Review of main features

2.1 Functional modules

A functional module in Maude corresponds to an equational theory in membership equational logic.

Both the logic and the language are typed, and *types* are declared by means of the keywords `sort` or `sorts`. Then each operator, introduced by means of the

keyword **op**, has to be declared together with the sorts of its arguments and the sort of its result. There is an inclusion relation between types, which is described by means of **subsort** declarations. Operators can be *overloaded*.

With typed variables (that can either be declared separately, or used on-the-fly annotated with the corresponding sort) and operators, we can build terms in the usual way. A given term can have many different sorts, because of the subsorting and overloading. Under some easy-to-satisfy requirements, a term has a least sort. Terms are used to form

- *membership assertions* $t : s$ (introduced with keyword **mb**), stating that the term t has sort s , and
- *equations* $t = t'$ (introduced with keyword **eq**), stating that the meaning of terms t and t' is the same.

Both memberships and equations can be conditional, with respective keywords **cmb** and **ceq**. Conditions are formed by a conjunction (written $\wedge$) of equations and memberships.

Computation in a functional module is done by using the equations as simplification rules from left to right until a canonical form is found. For this to be meaningful, the variables in the righthand side of an equation have to be included among those in the lefthand side (a generalization is provided by means of matching equations in conditions, as we will see later); moreover, the set of equations must be terminating and confluent. This guarantees that all terms will simplify to a unique canonical form [1].

Some equations, like commutativity, are not terminating, but nonetheless they are supported by means of *operator attributes*, so that Maude performs simplification modulo the equational theories provided by such attributes, that can be associativity, commutativity, identity, and idempotence. Properties such as termination and confluence should be understood in this more general context of simplification modulo some equational theories.

Modules can be imported in different modes. The most important one is **protecting** that asserts that all the information in the imported module does not change because of the importation; more specifically, different data in the imported module are not identified in the importing module, and no new data are added to the imported sorts. When this is not case, the importation mode can be **including**.

2.2 Parameterization

As we have already mentioned, parameterized datatypes use *theories* to specify the requirements that the parameter must satisfy. A (functional) theory is also a membership equational specification but since its equations are not used for equational simplification, they need not satisfy any requirement about variables in the righthand side, confluence, or termination.

The simplest theory is the one requiring just the existence of a sort, as follows:

```
(fth TRIV is
  sort Elt .
endfth)
```

This theory is used as requirement for the parameter of parameterized data types such as stacks, queues, lists, multisets, sets, and binary trees.

A more complex theory is the following, requiring a (strict) total order over elements of a given sort. Notice the *new* variable *E2* in the righthand side of the first conditional equation. This makes this equation non-executable, as stated by the attribute **nonexec** next to the equation.

```
(fth TOSET< is
  protecting BOOL .
  sort Elt .
  ops _<_ _>_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq E1 < E1 = false .
  ceq E1 < E3 = true if E1 < E2 and E2 < E3 [nonexec] .
  ceq E1 < E2 or E2 < E1 = true if E1 /= E2 .
  eq E1 > E2 = E2 < E1 .
endfth)
```

This theory imports in **protecting** mode the predefined module **BOOL** of Boolean values, meaning that the Boolean values are not disturbed in any way.

Theories are used in a parameterized module as in the following example:

```
(fmod LIST(X :: TRIV) is ... endfm)
```

where $X :: \text{TRIV}$ denotes that *X* is the label of the formal parameter, and that it must be instantiated with modules satisfying the requirement expressed by the theory **TRIV**. The way to express this instantiation is by means of *views*. A view shows how a particular module satisfies a theory, by mapping sorts and operations in the theory to sorts and operations (or, more generally, terms) in the target module, in such a way that the induced translations on equations and membership axioms are provable in the module. In general, this requires theorem proving that is not done by the system, but is instead delegated to the ITP tool (see Section 4). However, in many simple cases the proof of obligations associated to views are completely obvious, as for example in the following view from the theory **TRIV** to the predefined module **NAT** of natural numbers, where, since **TRIV** has no equations, no proof obligations are generated.

```
(view Nat from TRIV to NAT is
  sort Elt to Nat .
endv)
```

Then, the module expression **LIST(Nat)** denotes the instantiation of the parameterized module **LIST(X :: TRIV)** by means of the above view **Nat**. Views can also go from theories to theories, as we will see later in Section 3.4. For more information on parameterization and how it is implemented in the Maude system, the reader is referred to [7,6].

3 Data Structures

3.1 Stacks

We begin our series of datatype specifications with stacks. Since stacks can be built over any datatype, the requirement theory is **TRIV**. The main sort is **Stack(X)**; notice that its name makes explicit the label of the parameter. In this way, when the module is instantiated with a view, like for example **Nat** above (from **TRIV** to **NAT**), the sort name is also instantiated becoming **Stack(Nat)**, which makes clear that the data are stacks of natural numbers.

The sorts and operations of the theory are used in the body of the parameterized module, but sorts are qualified with the label of the formal parameter; thus in this case the parameter sort **Elt** becomes **X@Elt** in the **STACK** parameterized module. In this way, although not needed in our examples, one can have several different parameters satisfying the same parameter theory.

The only subtle point in a stack specification is that the **top** operation is partial, because it is not defined on the empty stack. In our specification, we use a subsort **NeStack(X)** of non-empty stacks to handle this situation. Then, **push** becomes a *constructor* of non-empty stacks, while both **empty** and **pop** (the latter via subsorting) are *constructors* of stacks; notice the **ctor** attribute of those operators, indicating that they are constructors. Now, the **top** and **pop** operations are defined as total with domain **NeStack(X)**.

Finally, all modules import implicitly the predefined **BOOL** module, and therefore we can use the sort **Bool** and the Boolean values **true** and **false** when necessary.

```
(fmod STACK(X :: TRIV) is
  sorts NeStack(X) Stack(X) .
  subsort NeStack(X) < Stack(X) .
  op empty : -> Stack(X) [ctor] .
  op push : X@Elt Stack(X) -> NeStack(X) [ctor] .
  op pop : NeStack(X) -> Stack(X) .
  op top : NeStack(X) -> X@Elt .
  op isEmpty : Stack(X) -> Bool .
  var S : Stack(X) .   var E : X@Elt .
  eq pop(push(E,S)) = S .
  eq top(push(E,S)) = E .
  eq isEmpty(empty) = true .
  eq isEmpty(push(E,S)) = false .
endfm)
```

The following view maps the theory **TRIV** to the predefined module **INT** of integers, and is then used in an example of term reduction, invoked with the Maude command **red**.

```
(view Int from TRIV to INT is
  sort Elt to Int .
endv)
```

```
Maude> (red in STACK(Int) : top(push(4,push(5,empty))) .)
result NzNat : 4
```

Notice that Maude computes the least sort of the result. In the following sections, for lack of space, we do not show reduction examples, but they can be found in [10].

3.2 Queues

The specification for queues is very similar to the one for stacks and therefore it is not included here. It is available in [10].

3.3 Lists

We are going to specify lists in two ways, by using different sets of constructors in each case. In both cases, lists are parameterized with respect to the TRIV theory.

The first version uses the two standard free constructors that can be found in many functional programming languages: the empty list *nil*, here denoted `[]`, and the *cons* operation that adds an element to the beginning of a list, here denoted with the mixfix syntax `_:_`. As usual, *head* and *tail* are the selectors associated to this constructor. Since they are not defined on the empty list, we avoid their partiality in the same way as we have done for stacks (see Section 3.1) by means of a subsort *NeList* of non-empty lists. The remaining operations on lists (defined as usual by structural induction on the two constructors) concatenate two lists, calculate the length of a list, and reverse a list; the second one of those has a result of sort *Nat* that comes from the imported (in *protecting* mode) predefined module *NAT*.

Due to parsing restrictions, some characters (`[ ] { } ,`) have to be preceded by a backquote “escape” character ``` when declaring them in Full Maude.

```
(fmod LIST(X :: TRIV) is
  protecting NAT .
  sorts NeList(X) List(X) .
  subsort NeList(X) < List(X) .
  op '[' : -> List(X) [ctor] .
  op _:_ : X@Elt List(X) -> NeList(X) [ctor] .
  op tail : NeList(X) -> List(X) .
  op head : NeList(X) -> X@Elt .
  op _+_ : List(X) List(X) -> List(X) .
  op length : List(X) -> Nat .
  op rev : List(X) -> List(X) .
  var E : X@Elt .   var N : Nat .   vars L L' : List(X) .
  eq tail(E : L) = L .
  eq head(E : L) = E .
  eq [] ++ L = L .
```

```

eq (E : L) ++ L' = E : (L ++ L') .
eq length([]) = 0 .
eq length(E : L) = 1 + length(L) .
eq rev([]) = [] .
eq rev(E : L) = rev(L) ++ (E : []) .
endfm)

```

Another way to generate lists is to begin with the empty list and the singleton lists, and then use the concatenation operation to get bigger lists. However, concatenation cannot be a free list constructor, because it satisfies an associativity equation. This equation is not declared directly as such, but as an operator attribute `assoc`. Moreover, there is also an identity relationship of concatenation with respect to the empty list, which is expressed by means of the attribute `id: []`. It is very convenient here to use empty juxtaposition syntax (declared as `__` in the following module) for the concatenation operator as constructor; in this way, the list of integers `1 : 2 : 3 : []` in the previous notation now becomes simply `1 2 3`.

Notice how the singleton lists are identified with the corresponding elements (by means of a subsort declaration `X@Elt < NeList(X)`) and also how the concatenation operator is subsort overloaded, having one declaration for non-empty lists and another one for lists, both with the same attributes. There are two more possibilities of concatenation overloading (`NeList List -> NeList` and `List NeList -> NeList`) but they are unnecessary in this case because of the attribute for identity. Finally, notice that operator attributes in overloaded operations have to coincide, even though, by reading alone the second declaration for concatenation, it may sound a bit strange to say that the empty list is an identity for an operation only defined on non-empty lists. When there are many overloaded declarations for an operator, it is possible to use the operator attribute `ditto` to implicitly repeat the attributes without having to write all of them explicitly again.

Because of the operator attributes, congruence classes over which simplification takes place are computed modulo associativity and identity. Therefore, we only need two equations to completely specify the behavior of defined operations like `length` and `rev`; the singleton case is included in the `E L` case by instantiating the variable `L` with the constant `nil` and applying the equational attribute for identity. Notice that in this way, even though we have changed the set of list constructors, we are not changing so much (except for the notation) the style of the definitions by structural induction of the remaining operations; the case `E L` corresponds to the *cons* case in the previous specification.

```

(fmod LIST-CONCAT(X :: TRIV) is
  protecting NAT .
  sorts NeList(X) List(X) .
  subsorts X@Elt < NeList(X) < List(X) .
  op '[' : -> List(X) [ctor] .
  op __ : List(X) List(X) -> List(X) [ctor assoc id: '['] .
  op __ : NeList(X) NeList(X) -> NeList(X) [ctor assoc id: '['] .

```

```

op tail : NeList(X) -> List(X) .
op head : NeList(X) -> X@Elt .
op length : List(X) -> Nat .
op rev : List(X) -> List(X) .
var E : X@Elt .    var L : List(X) .
eq tail(E L) = L .
eq head(E L) = E .
eq length(nil) = 0 .
eq length(E L) = 1 + length(L) .
eq rev(nil) = nil .
eq rev(E L) = rev(L) E .
endfm)

```

3.4 Ordered lists

In order-sorted equational specifications, subsorts must be defined by means of constructors, but it is not possible to have a subsort of ordered lists, for example, defined by a property over lists; a more expressive formalism is needed. *Membership equational logic* allows subsort definition by means of conditions involving equations and/or sort predicates. In this example we use this technique to define a subsort **OrdList**, containing *ordered lists*,¹ of the sort **List** of lists, which is imported from the module **LIST** in Section 3.3. Notice the three (conditional) membership axioms defining the sort **OrdList**: the empty and singleton lists are always ordered, and a longer list is ordered when the first element is less than or equal to the second, and the list without the first element is also ordered.

Parameterized ordered lists need a stronger requirement than **TRIV**, because we need a total order over the elements to be ordered. The theory **TOSET<** that we saw in Section 2.2 requires a strict total order on the elements; since repetitions do not give any trouble for sorting a list, we can have a $_ \leq _$ operation in the requirement. We can also import theories (in **including** mode), and thus we can define our theory as follows:

```

(fth TOSET<= is
  including TOSET< .
  op _<=_ : Elt Elt -> Bool .
  vars X Y : Elt .
  eq X <= Y = X < Y or X == Y .
endfth)

```

The parameterized module for ordered lists is going to import the parameterized list module. However, note that we want lists for a totally ordered set, instead of lists over any set; therefore, we *partially instantiate* **LIST** with a view from the theory **TRIV** to the theory **TOSET<=**

```

(view Toset from TRIV to TOSET<= is
  sort Elt to Elt .
endv)

```

¹ We prefer “ordered list” over “sorted list” because “sort” is already used to refer to types in this context.

and we are still left with a parameterized module and corresponding dependent sorts, but now with respect to the `TOSET<=` requirement. This is the reason justifying the notation `LIST(Toset)(X)` in the `protecting` importation below, as well as `NeList(Toset)(X)` and `List(Toset)(X)` in the names of the imported sorts.

As part of this module for ordered lists, we also define several well-known sorting operations: `insertion-sort`, `quicksort`, and `mergesort` (the following code only includes the first one, while the other two can be found in the more complete version available in [10]). Each of them uses appropriate auxiliary operations whose behavior is the expected one; for example, `insertion-sort` recursively sorts the list without the first element, and then calls `insert-list`, which inserts the missing element in the correct position.

The important point is that we are able to give finer typing to all these sorting operations than the usual typing in other algebraic specification frameworks or functional programming languages. Thus, `insertion-sort` is declared as an operation from `List(Toset)(X)` to `OrdList(X)`, instead of the much less informative typing from `List(Toset)(X)` to `List(Toset)(X)`. The same applies to each of the auxiliary operations. Also, a function that requires its input argument to be an ordered list can now be defined as a *total* function, whereas in less expressive typing formalisms it would have to be either *partial*, or to be defined with exceptional behavior on the erroneous arguments.

```
(fmod ORD-LIST(X :: TOSET<=) is
  protecting LIST(Toset)(X) .
  sorts OrdList(X) NeOrdList(X) .
  subsorts NeOrdList(X) < OrdList(X) NeList(Toset)(X) < List(Toset)(X) .
  op insertion-sort : List(Toset)(X) -> OrdList(X) .
  op insert-list : OrdList(X) X@Elt -> OrdList(X) .
  vars N M : X@Elt .          vars L L' : List(Toset)(X) .
  vars OL OL' : OrdList(X) .   var NEOL : NeOrdList(X) .
  mb [] : OrdList(X) .
  mb (N : []) : NeOrdList(X) .
  cmb (N : NEOL) : NeOrdList(X) if N <= head(NEOL) .
  eq insertion-sort([]) = [] .
  eq insertion-sort(N : L) = insert-list(insertion-sort(L), N) .
  eq insert-list([], M) = M : [] .
  ceq insert-list(N : OL, M) = M : N : OL if M <= N .
  ceq insert-list(N : OL, M) = N : insert-list(OL, M) if M > N .
endfm)
```

3.5 Multisets

In the same way as associativity and identity for concatenation provide structural axioms for lists (or strings), we can specify multisets by considering a union constructor (written again with empty juxtaposition syntax) that satisfies associativity, commutativity (because now order between elements does not matter), and identity structural axioms, all declared as attributes. Singleton multisets are

identified with elements, as we also did for lists, by declaring `X@Elt` as a subsort of `Mset(X)`.

```
(fmod MULTISSET(X :: TRIV) is
  protecting NAT .
  sort Mset(X) .
  subsort X@Elt < Mset(X) .
  op empty : -> Mset(X) [ctor] .
  op _ : Mset(X) Mset(X) -> Mset(X) [ctor assoc comm id: empty] .
  op size : Mset(X) -> Nat .
  op mult : X@Elt Mset(X) -> Nat .
  op delete : X@Elt Mset(X) -> Mset(X) .
  op is-in : X@Elt Mset(X) -> Bool .
  vars E E' : X@Elt .   var S : Mset(X) .
  eq size(empty) = 0 .
  eq size(E S) = 1 + size(S) .
  eq mult(E, empty) = 0 .
  eq mult(E, E S) = 1 + mult(E, S) .
  ceq mult(E, E' S) = mult(E, S) if E /= E' .
  eq delete(E, empty) = empty .
  eq delete(E, E S) = delete(E, S) .
  ceq delete(E, E' S) = E' delete(E, S) if E /= E' .
  eq is-in(E, S) = mult(E, S) > 0 .
endfm)
```

3.6 Sets

Analogously to obtaining multisets from lists by adding commutativity, one can get sets from multisets by adding idempotence. In the current version of Maude, the operator attributes for associativity and idempotence are not compatible, but this is easily solved by adding an operator attribute for associativity and an explicit equation for idempotence. Since the specification of sets is very similar to the previous one for multisets, we refer to [10].

3.7 Binary trees

Binary trees (parameterized with respect to `TRIV`) are built with two free constructors: the empty tree, denoted `empty`, and an operation that puts an element as root above two given trees, its left and right children, denoted `_[_]_`. The three selectors associated to this constructor (`root`, `left`, and `right`) only make sense for non-empty trees, that belong to the corresponding subsort.

The operation that calculates the depth (or height) of a binary tree calls a `max` operation on natural numbers in the recursive non-empty case to obtain the maximum of two such numbers. Since this operation is not provided in the predefined module `NAT`, we import a module `NAT-MAX` that has previously added the `max` operation to `NAT`.

Finally, we also have three operations that calculate the standard binary tree traversals, with `List(X)` as value sort (this is the reason this module imports the

LIST module). Since all of them have the same rank, they are declared together by means of the keyword `ops`.

```
(fmod BIN-TREE(X :: TRIV) is
  protecting LIST(X) .    protecting NAT-MAX .
  sorts NeBinTree(X) BinTree(X) .
  subsort NeBinTree(X) < BinTree(X) .
  op empty : -> BinTree(X) [ctor] .
  op _'[_]_ : BinTree(X) X@Elt BinTree(X) -> NeBinTree(X) [ctor] .
  ops left right : NeBinTree(X) -> BinTree(X) .
  op root : NeBinTree(X) -> X@Elt .
  op depth : BinTree(X) -> Nat .
  ops preorder inorder postorder : BinTree(X) -> List(X) .
  var E : X@Elt .    vars L R : BinTree(X) .    vars NEL NER : NeBinTree(X) .
  eq left(L [E] R) = L .
  eq right(L [E] R) = R .
  eq root(L [E] R) = E .
  eq depth(empty) = 0 .
  eq depth(L [E] R) = 1 + max(depth(L), depth(R)) .
  eq preorder(empty) = [] .
  eq preorder(L [E] R) = E : (preorder(L) ++ preorder(R)) .
  eq inorder(empty) = [] .
  eq inorder(L [E] R) = inorder(L) ++ (E : inorder(R)) .
  eq postorder(empty) = [] .
  eq postorder(L [E] R) = postorder(L) ++ (postorder(R) ++ (E : [])) .
endfm)
```

3.8 General trees

General trees can have a variable number of children for each node. One can specify them by using an auxiliary datatype of *forests* that behave like lists of trees. Since otherwise the parameterization and specification techniques do not differ from the ones we have already described before, we do not include the specification here, and instead refer again to [10].

3.9 Binary search trees

This example is similar in philosophy to the one for ordered lists, but is more complex. We specify a subsort of *(binary) search trees* by using several (conditional) membership axioms over terms of the sort `BinTree` of binary trees defined in Section 3.7.

Although we allowed repeated elements in an ordered list, this should not be the case in a search tree, where all nodes must contain different values. A binary search tree is either the empty binary tree or a non-empty binary tree such that all elements in the left child are smaller than the element in the root, all elements in the right child are bigger than it, and both the left and right children are also binary search trees. This is checked by means of auxiliary operations that calculate the minimum and maximum element in a non-empty

search tree, and that are also useful when deleting an element. Again, the most important point is that membership equational logic allows us both to define the corresponding subsort by means of membership assertions (we consider five cases in the specification below) and to assign typings in the best possible way to all the operations defined for this datatype.

Although we could parameterize binary search trees just with respect to a total order given by the theory $\text{TOSET}<$, we specify here the version of search trees containing in the nodes pairs formed by a key and its associated contents, so that we think of search trees as dictionaries. The search tree structure is with respect to a strict total order on keys, but contents can be over an arbitrary sort. When we insert a pair $\langle K, C \rangle$ and the key K already appears in the tree in a pair $\langle K, C' \rangle$, insertion takes place by combining the contents C' and C . This combination can be replacing the first with the second, just forgetting the second, addition if the trees are used to implement multisets and the C s represent multiplicities, etc. Therefore, as part of the requirement parameter theory for the contents we will have an associative binary operation **combine** on the sort **Contents**. Notice how the two parameter theories on which the specification depends are separated by `|`.

In principle, a pair construction should be enough to put together the information that we have just described, but we will import the module for search trees in the specifications of more complex data structures that happen to be particular cases of search trees, such as AVL and red-black trees. In those cases, it is important to add new information in the nodes: for the AVL trees, one needs the depth of the tree hanging in each node, while for red-black trees one needs the appropriate node color. Therefore, taking this into account, it is important to define a datatype that is *extensible*, and we have considered *records* for this, defined in the module **RECORD**. A record is defined as a collection (with an associative and commutative union operator denoted by `_.`) of pairs consisting of a field name and an associated value. After importing the binary trees instantiated with a **Record** view, we define the fields for keys (with syntax `key: _`) and for contents (with syntax `contents: _`), together with corresponding projection operations that extract the appropriate values from the record. Notice how these operations, as well as the operations on trees, can be applied to records that have more fields, unknown yet at this time, by using a variable **Rec** that takes care of “the rest of the record.”

This construction of adding fields to records is modifying the data in the sort **Record**, which are the data in the nodes of the trees, and thus the trees themselves. For this reason, we have imported the module **BIN-TREE** (after instantiating it with the view **Record**) in **including** mode.

In addition to operations for insertion and deletion, we have a **lookup** operation that returns the contents associated to a given key, when the key appears in the tree. However, this last operation is partial,^f because it is not defined when the key does not appear in the tree; this error state cannot be handled by means of a subsort, because the partiality condition depends on the concrete values of the arguments. We have used a *partial operator declaration* by stating that

the result of the `lookup` operator is in the *kind* `[Y@Contents]` associated to the sort `Y@Contents` (notice the square brackets around the sort name to denote the corresponding kind). One can think of a kind as an error “supersort” where, in addition to correct well-formed terms, there are undefined or error terms. Instead of having at the level of kinds just a term that does not reduce, we have declared a special value `not-found` of kind `[Y@Contents]` which is returned by the `lookup` operation when the key does not appear in the tree.

These operations are specified as usual, by structural induction, and in the non-empty case by comparing the given key K with the key in the root of the tree and distinguishing the three cases according to whether K is smaller than, equal to, or bigger than the root key.

```
(fth CONTENTS is
  sort Contents .
  op combine : Contents Contents -> Contents [assoc] .
endfth)

(fmod RECORD is
  sorts Record .
  op null : -> Record [ctor] .
  op _',_ : Record Record -> Record [ctor assoc comm id: null] .
endfm)

(view Record from TRIV to RECORD is
  sort Elt to Record .
endv)

(fmod SEARCH-TREE(X :: TOSET< | Y :: CONTENTS) is
  including BIN-TREE(Record) .
  sorts SearchTree(X | Y) NeSearchTree(X | Y) .
  subsorts NeSearchTree(X | Y) < SearchTree(X | Y) < BinTree(Record) .
  subsort NeSearchTree(X | Y) < NeBinTree(Record) .

  --- Construction of the record used as node in binary search trees.
  op key:_ : X@Elt -> Record [ctor] .
  op contents:_ : Y@Contents -> Record [ctor] .
  op not-found : -> [Y@Contents] .
  op key : Record -> X@Elt .
  op contents : Record -> Y@Contents .
  var Rec : Record .   var K : X@Elt .   var C : Y@Contents .
  eq key(Rec,key: K) = K .
  eq contents(Rec,contents: C) = C .

  --- Operations for binary search trees.
  op insert : SearchTree(X | Y) X@Elt Y@Contents -> SearchTree(X | Y) .
  op lookup : SearchTree(X | Y) X@Elt -> [Y@Contents] .
  op delete : SearchTree(X | Y) X@Elt -> SearchTree(X | Y) .
  ops min max : NeSearchTree(X | Y) -> Record .
  var Rec' : Record .           vars L R : SearchTree(X | Y) .
```

```

vars L' R' : NeSearchTree(X | Y) .   var C' : Y@Contents .

mb empty : SearchTree(X | Y) .
mb empty [Rec] empty : NeSearchTree(X | Y) .
cmb L' [Rec] empty : NeSearchTree(X | Y) if key(max(L')) < key(Rec) .
cmb empty [Rec] R' : NeSearchTree(X | Y) if key(Rec) < key(min(R')) .
cmb L' [Rec] R' : NeSearchTree(X | Y)
    if key(max(L')) < key(Rec) and key(Rec) < key(min(R')) .

eq insert(empty, K, C) = empty [key: K,contents: C] empty .
eq insert(L [Rec,key: K,contents: C] R, K, C') =
    L [Rec,key: K, contents: combine(C, C')] R .
ceq insert(L [Rec] R, K, C) = insert(L, K, C) [Rec] R if K < key(Rec) .
ceq insert(L [Rec] R, K, C) = L [Rec] insert(R, K, C) if key(Rec) < K .

eq lookup(empty, K) = not-found .
eq lookup(L [Rec,key: K,contents: C] R, K) = C .
ceq lookup(L [Rec] R, K) = lookup(L, K) if K < key(Rec) .
ceq lookup(L [Rec] R, K) = lookup(R, K) if key(Rec) < K .

eq delete(empty, K) = empty .
ceq delete(L [Rec] R, K) = delete(L, K) [Rec] R if K < key(Rec) .
ceq delete(L [Rec] R, K) = L [Rec] delete(R, K) if key(Rec) < K .
eq delete(empty [Rec,key: K,contents: C] R, K) = R .
eq delete(L [Rec,key: K,contents: C] empty, K) = L .
eq delete(L' [Rec,key: K,contents: C] R', K) =
    L' [min(R')] delete(R', key(min(R'))) .

eq min(empty [Rec] R) = Rec .
eq min(L' [Rec] R) = min(L') .
eq max(L [Rec] empty) = Rec .
eq max(L [Rec] R') = max(R') .
endfm

```

3.10 AVL trees

It is well-known that in order to have better efficiency on search trees one has to keep them balanced. One nice solution to this problem is provided by AVL trees; these are binary search trees satisfying the additional constraint in each node that the difference between the depth of both children is at most one. This constraint guarantees that the depth of the tree is always logarithmic with respect to the number of nodes, thus obtaining a logarithmic cost for the operations of search, lookup, insertion and deletion, assuming that the last two are implemented in such a way that they keep the properties of the balanced tree. As we have already anticipated in Section 3.9, it is convenient to have in each node as additional data the depth of the tree having this node as root, so that comparing the depths of children to check the balance property of the AVL trees becomes very quick. This is accomplished by importing the module

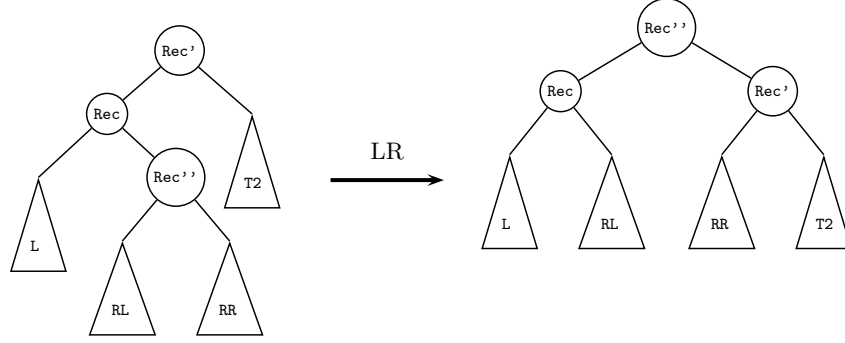


Fig. 1. Left-right rotation in an AVL tree.

SEARCH-TREE of search trees and adding a **depth** field to the record, together with the corresponding projection.

The sort **AVL** of AVL trees is a subsort of the sort **SearchTree** of search trees, defined by means of additional membership assertions (recall that we have already used this technique to define **SearchTree** as a subsort of **BinTree** in Section 3.9); in the specification below, just two memberships are enough, one for the empty tree and the other for non-empty AVL trees. Notice the use of the *symmetric difference* operator **sd** on natural numbers; the result of this operation applied to two natural numbers is the result of subtracting the smallest from the biggest of the two.

For lookup we use the same operation as for search trees, imported from the module **SEARCH-TREE**; on the other hand, insertion and deletion have to be re-defined so that they keep the AVL properties. They work as in the general case, by comparing the given key with the one in the root, but the final result is built by means of an auxiliary **join** operation that checks that the difference between the depths of the two children is less than one, using again the symmetric difference operator **sd**; when this is not the case, appropriate *rotation* operations are invoked. It is enough to have a left rotation **lRotate** and a right rotation **rRotate**. This is quite similar to the typical imperative or object-oriented versions of these operations [9, 2, 16]. For example, the second equation for **lRotate** is illustrated in Figure 1. In the specification below we do not show the equations for **deleteAVL** and **rRotate**, which can be found in [10].

```
(fmod AVL(X :: TOSET< | Y :: CONTENTS) is
  including SEARCH-TREE(X | Y) .
  --- We add a new field to the node's record.
  op depth:_ : Nat -> Record [ctor] .
  op depth : Record -> Nat .
  var N : Nat .   vars Rec Rec' Rec'' : Record .
  eq depth(Rec,depth: N) = N .
```

```

--- AVL trees memberships and operations.
sorts NeAVL(X | Y) AVL(X | Y) .
subsorts NeAVL(X | Y) < AVL(X | Y) < SearchTree(X | Y) .
subsorts NeAVL(X | Y) < NeSearchTree(X | Y) .
vars L R RL RR T1 T2 : AVL(X | Y) .

mb empty : AVL(X | Y) .
cmb L [Rec] R : NeAVL(X | Y) if L [Rec] R : SearchTree(X | Y) /\
    sd(depth(L),depth(R)) <= 1 /\
    1 + max(depth(L),depth(R)) = depth(Rec) .

op insertAVL : X@Elt Y@Contents AVL(X | Y) -> NeAVL(X | Y) .
op deleteAVL : X@Elt AVL(X | Y) -> AVL(X | Y) .
op depthAVL : AVL(X | Y) -> Nat .
op buildAVL : AVL(X | Y) Record AVL(X | Y) -> AVL(X | Y) .
op join : AVL(X | Y) Record AVL(X | Y) -> AVL(X | Y) .
op lRotate : AVL(X | Y) Record AVL(X | Y) -> AVL(X | Y) .
op rRotate : AVL(X | Y) Record AVL(X | Y) -> AVL(X | Y) .
vars K K' : X@Elt . vars C C' : Y@Contents .

eq insertAVL(K, C, empty) =
    buildAVL(empty,(depth: 0,key: K, contents: C),empty) .
eq insertAVL(K, C, L [Rec,key: K, contents: C'] R) =
    L [Rec,key: K, contents: combine(C,C')] R .
ceq insertAVL(K, C, L [Rec] R) = join(insertAVL(K,C,L), Rec, R)
    if K < key(Rec) .
ceq insertAVL(K, C, L [Rec] R) = join(L, Rec, insertAVL(K,C,R))
    if key(Rec) < K .

eq depthAVL(empty) = 0 .
eq depthAVL(L [Rec,depth: N] R) = N .

eq buildAVL(T1, (Rec',depth: N), T2) =
    T1 [Rec',depth: (max(depthAVL(T1),depthAVL(T2)) + 1)] T2 .

ceq join(T1, Rec, T2) = buildAVL(T1, Rec, T2)
    if sd(depthAVL(T1),depthAVL(T2)) <= 1 .
ceq join(T1, Rec, T2) = lRotate(T1, Rec, T2)
    if depthAVL(T1) = depthAVL(T2) + 2 .
ceq join(T1, Rec, T2) = rRotate(T1, Rec, T2)
    if depthAVL(T1) + 2 = depthAVL(T2) .

ceq lRotate(L [Rec] R,Rec',T2) = buildAVL(L,Rec,buildAVL(R,Rec',T2))
    if depthAVL(L) >= depthAVL(R) .
ceq lRotate(L [Rec] (RL [Rec''] RR), Rec', T2) =
    buildAVL(buildAVL(L, Rec, RL), Rec'', buildAVL(RR, Rec', T2))
    if depthAVL(L) < depthAVL(RL [Rec''] RR) .
endfm)

```

3.11 2-3-4 trees

Other solutions to the problem of keeping balanced search trees are provided by *2-3 trees*, which are not treated here, and *2-3-4 trees*, whose specification we consider in this section. This kind of search trees generalizes binary search trees to a version of general trees of degree 4, so that a non-leaf node can have 2, 3 or 4 children. The number of values in the node depends on the number of children; for example, there are two different values (let us call N1 the smallest of the two, and N2 the greatest) in the node when it has three children. Moreover, the values in the children are well organized with respect to the values in the node; in the same example, all the values in the first child must be smaller than N1, all the values in the second child must be bigger than N1 and smaller than N2, and all the values in the third child must be bigger than N2. Furthermore, the children must have exactly the same depth, and recursively they have to satisfy the same properties. As expected, all of these properties can be stated by means of memberships assertions.

Since these trees need a different set of constructors, they have no direct relationship to binary search trees. Also, in order to simplify the presentation we just parameterize the specification with respect to the theory $\text{TOSET}<$, that is, we consider only values in the nodes, instead of keys and associated values as we did in previous sections.

The specification of the search operation is immediate. Although the main ideas of insertion are quite simple, the details of the implementation become much lengthier than expected, requiring several auxiliary operations and several equations to treat the different cases arising from combining the different constructors. Even worse is the implementation of deletion, which needs a *zillion* of equations to deal with all possible cases. No wonder most textbooks avoid presenting these details and leave them as a challenging programming project!

Here we only show the memberships that specify 2-3-4 trees; the remaining details are available in [10].

```
sort Ne234Tree?(T) 234Tree?(T) Ne234Tree(T) 234Tree(T) .
subsort Ne234Tree?(T) < 234Tree?(T) .
subsort Ne234Tree(T) < 234Tree(T) < 234Tree?(T) .
subsort Ne234Tree(T) < Ne234Tree?(T) .

op empty234 : -> 234Tree?(T) [ctor] .
op _['_']_ : 234Tree?(T) T@Elt 234Tree?(T) -> Ne234Tree?(T) [ctor] .
op _<_>_<_>_ : 234Tree?(T) T@Elt ... -> Ne234Tree?(T) [ctor] .
op _['_']_['_']_['_']_['_']_ : 234Tree?(T) T@Elt ... -> Ne234Tree?(T) [ctor] .
vars N N1 N2 N3 : T@Elt .
var 234TL 234TLM 234TC 234TRM 234TR : 234Tree(T) .

mb empty234 : 234Tree(T) .
cmb 234TL [ N ] 234TR : Ne234Tree(T) if greaterKey(N,234TL) /\
    smallerKey(N,234TR) /\ depth(234TL) = depth(234TR) .
cmb 234TL < N1 > 234TC < N2 > 234TR : Ne234Tree(T)
    if N1 < N2 /\ greaterKey(N1,234TL) /\ smallerKey(N1,234TC) /\
```

```

        greaterKey(N2,234TC) /\ smallerKey(N2,234TR) /\
        depth(234TL) = depth(234TC) /\ depth(234TC) = depth(234TR) .
cmb 234TL { N1 } 234TLM { N2 } 234TRM { N3 } 234TR : Ne234Tree(T)
    if N1 < N2 /\ N2 < N3 /\ greaterKey(N1,234TL) /\
        smallerKey(N1,234TLM) /\ greaterKey(N2,234TLM) /\
        smallerKey(N2,234TRM) /\ greaterKey(N3,234TRM) /\
        smallerKey(N3,234TR) /\ depth(234TL) = depth(234TLM) /\
        depth(234TL) = depth(234TRM) /\ depth(234TL) = depth(234TR) .

```

3.12 Red-black trees

Yet another solution to the problem of keeping search trees balanced are *red-black* search trees. These are standard binary search trees that satisfy several additional constraints that are related to a color (hence the name!) that can be associated to each node (in some presentations, to the edges). One can think of red-black trees as a binary representation of 2-3-4 search trees, and this provides helpful intuition.

Since the color is additional information in each node, we make again use of the record construction described in Section 3.9. Once more, memberships allow a faithful specification of all the constraints. All details can be found in [10].

3.13 Priority queues

The abstract specification of priority queues is very simple. It is parameterized with respect to the theory $\text{TOSET} \leq$, because we allow repetitions and the priority is identified with each value. We have as constructors the constant **empty** and **insert**, that adds a new element to the priority queue. However, these constructors are not free because the order of insertion does not matter, the priority being the information that determines the actual order in the queue. This is made explicit in a “commutativity” equation for the **insert** operator, but this is not a standard commutativity equation for a binary operator with both arguments of the same sort, and thus it cannot be expressed as a **comm** attribute; in any case, it is not terminating, and therefore it has been stated as *non-executable* by means of the **nonexec** attribute (which is associated to the equation, and not to the operator).

We consider the version of priority queues in which the first element is the minimum. Both **findMin** and **deleteMin** are easily specified as total operations on non-empty priority queues by structural induction and in the second case by comparing the priorities of two elements.

Even though this is a very abstract specification, it is directly executable after instantiating appropriately the parameter.

```

(fmod PRIORITY-QUEUE(X :: TOSET<=) is
  sort NePQueue(X) PQueue(X) .
  subsort NePQueue(X) < PQueue(X) .
  op empty : -> PQueue(X) [ctor] .
  op insert : PQueue(X) X@Elt -> NePQueue(X) [ctor] .

```

```

op deleteMin : NePQueue(X) -> PQueue(X) .
op findMin : NePQueue(X) -> X@Elt .
op isEmpty : PQueue(X) -> Bool .
var PQ : PQueue(X) . vars E F : X@Elt .
eq insert(insert(PQ,E),F) = insert(insert(PQ,F),E) [nonexec] .
eq deleteMin(insert(empty,E)) = empty .
ceq deleteMin(insert(insert(PQ,E),F)) =
  insert(deleteMin(insert(PQ,E)),F) if findMin(insert(PQ,E)) <= F .
ceq deleteMin(insert(insert(PQ,E),F)) =
  insert(PQ,E) if findMin(insert(PQ,E)) > F .
eq findMin(insert(empty,E)) = E .
ceq findMin(insert(insert(PQ,E),F)) =
  findMin(insert(PQ,E)) if findMin(insert(PQ,E)) <= F .
ceq findMin(insert(insert(PQ,E),F)) =
  F if findMin(insert(PQ,E)) > F .
eq isEmpty(empty) = true .
eq isEmpty(insert(PQ,E)) = false .
endfm)

```

3.14 Leftist trees

Among the many different data structures implementing priority queues the most efficient are *heaps*, which can be defined (in the case of *min heaps*) as binary trees satisfying the additional constraints that the value in each node is smaller than (or equal to) the values in its children and moreover the tree is complete. If we forget the latter requirement and instead assign to each node a *rank* (also known as *minimum depth*) defined as the length of the rightmost path to a leaf, and require that the root of each left child has a rank bigger than or equal to the rank of the corresponding right child (that can be empty), we get the trees known as *leftist trees*. These trees implement priority queues with the same efficiency as standard heaps, and have the additional property that two leftist trees can be merged to obtain a leftist tree containing all elements in the two given trees in logarithmic time with respect to the total number of nodes.

As usual, the sort of leftist trees can be defined as a subsort of binary trees by means of appropriate membership assertions. In order to compare quickly the ranks of two nodes, we need to save in each node its rank (in the same way that we saved the depth in each node in AVL trees). Since we are importing binary trees, which are parameterized with respect to the theory TRIV, we first define in the module TREE-NODE the construction of pairs formed by an element and a natural number, and then instantiate the module BIN-TREE of binary trees with the *parameterized view* Node(T). The view is parameterized because it still keeps as a parameter, as expected, the sort of the elements with a total order on top of which we are building the leftist trees.

The most important operation on this data structure is **merge** because both **insert** and **deleteMin** are easily defined from it. The operation **merge** is specified by structural induction on its arguments, with the help in the recursive case (when both arguments are non-empty) of an auxiliary operation **make** that takes

care of putting the tree with bigger rank at its root to the left of the tree being built.

```

(fmod TREE-NODE(T :: TOSET<=) is
  protecting NAT .
  sort Node(T) .
  op n : Nat T@Elt -> Node(T) [ctor] .
endfm)

(view Node(T :: TOSET<=) from TRIV to TREE-NODE(T) is
  sort Elt to Node(T) .
endv)

(fmod LEFTIST-TREES(T :: TOSET<=) is
  protecting BIN-TREE(Node(T)) .
  sorts NeLTree(T) LTree(T) .
  subsorts NeLTree(T) < LTree(T) < BinTree(Node(T)) .
  subsorts NeLTree(T) < NeBinTree(Node(T)) .

  op rank : BinTree(Node(T)) -> Nat .
  op rankL : LTree(T) -> Nat .
  op insert : T@Elt LTree(T) -> NeLTree(T) .
  op deleteMin : NeLTree(T) -> LTree(T) .
  op findMin : NeLTree(T) -> T@Elt .
  op make : T@Elt LTree(T) LTree(T) -> LTree(T) .
  op merge : LTree(T) LTree(T) -> LTree(T) .
  vars NeTL NeTR : NeLTree(T) .          vars M N N1 N2 : Nat .
  vars T TL TR TL1 TR1 TL2 TR2 : LTree(T) .  vars X X1 X2 : T@Elt .

  mb empty : LTree(T) .
  mb empty [n(1, X)] empty : NeLTree(T) .
  cmb NeTL [n(1, X)] empty : NeLTree(T) if X < findMin(NeTL) .
  cmb NeTL [n(N, X)] NeTR : NeLTree(T) if rank(NeTL) >= rank(NeTR) /\
    X < findMin(NeTL) /\ X < findMin(NeTR) /\ N = 1 + rank(NeTR) .
  eq rank(empty) = 0 .
  eq rank(TL [n(N, X)] TR) = 1 + rank(TR) .
  eq rankL(empty) = 0 .
  eq rankL(TL [n(N,X)] TR) = N .
  eq merge(empty, T) = T .
  eq merge(T, empty) = T .
  eq merge(TL1 [n(N1, X1)] TR1, TL2 [n(N2, X2)] TR2) =
    if X1 < X2 then make(X1,TL1,merge(TR1, TL2 [n(N2,X2)] TR2))
    else make(X2,TL2,merge(TL1 [n(N1,X1)] TR1,TR2)) fi .
  eq make(X, TL, TR) = if rankL(TL) >= rankL(TR)
    then TL [n(rankL(TR) + 1,X)] TR
    else TR [n(rankL(TL) + 1,X)] TL fi .
  eq insert(X,T) = merge(empty [n(1,X)] empty, T) .
  eq findMin(TL [n(N,X)] TR) = X .
  eq deleteMin(TL [n(N,X)] TR) = merge(TL,TR) .
endfm)

```

4 Proving Properties

Mechanical reasoning about specifications in Maude is supported by the experimental ITP tool [3], a rewriting-based theorem prover that can be used to prove inductive properties of equational specifications. It is written in Maude, and it is itself an executable specification. A key feature of the ITP is its reflective design, that allows the definition of customized *rewriting strategies* different from Maude's default one; currently, this capability is being used to extend the ITP with decision procedures for arithmetic, lists, and combinations of theories.

The ITP is still very much work in progress: it can work with any module present in Maude's database but not with those introduced in Full Maude; in particular, at present it offers no support for parameterized specifications. Therefore, in the examples that follow we illustrate its use with the concrete modules `LIST` and `ORD-LIST` of *lists of natural numbers*, obtained from the specifications in Sections 3.3 and 3.4 by removing the parameter `X` and renaming the sort `X@Elt` as `Nat`.

4.1 Concatenation is associative

The most basic property the `LIST` specification should satisfy is that concatenation of lists is associative. After loading the ITP with the instruction `in xitp-tool` and initializing its database with `loop init-itp .`, the property can be proved with the following commands enclosed in parentheses:

```
(goal list-assoc : LIST |- A{L1:List ; L2:List ; L3:List}
  (((L1:List ++ L2:List) ++ L3:List) = (L1:List ++ (L2:List ++ L3:List))) .)
(ind on L1:List .)
  (auto* .) --- base case []
  (auto* .) --- inductive step E : L
```

The first two lines introduce the desired goal: `list-assoc` is its name, `LIST` the module in which the goal is to be proved, and the symbol `A` (representing $\forall$) precedes a list of universally quantified variables. Notice that variables are always annotated with their types. Then we try to prove the property by structural induction on the first variable and the ITP generates a corresponding subgoal for each operator of range `List` that has been declared with the attribute `ctor`; in this case one for the empty list `[]` and another one for the constructor `_:_` (with its corresponding inductive hypothesis). Finally, both cases can be automatically proved with the command `auto*` that first transforms all variables into fresh constants and then rewrites the terms in both sides of the equation as much as possible.

4.2 Reverse is self-inverse

As another example we show that the result of applying the operation `rev` twice in a row to a list leaves it unchanged. Again, we try to prove the property by induction but this time the second `auto*` does not succeed and leaves us with the following subgoal to prove:

```
|- rev(rev(V0#1*List)++ V0#0*Elem : []) = V0#0*Elem : V0#1*List
```

This suggests proving the auxiliary lemma `conc-rev` below, which in turn requires a lemma to show that `[]` is the identity for concatenation and another one for associativity as above. Lemmas are introduced like goals but using the keyword `lem` and without the module's name; all three are straightforwardly proved by structural induction, and then the pending subgoal can be discharged with a final `auto*`.

```
(goal list-rev : LIST |- A{L:List}{(rev(rev(L:List))) = (L:List)) .)
  (ind on L:List .)
    (auto* .) --- []
    (auto* .) --- E : L
  (lem conc-id : A{L:List} ((L:List ++ []) = (L:List)) .)
    (ind on L:List .)
      (auto* .)
      (auto* .)
  (lem conc-assoc : A{L1:List ; L2:List ; L3:List}
    (((L1:List ++ L2:List) ++ L3:List) = (L1:List ++ (L2:List ++ L3:List))) .)
    (ind on L1:List .)
      (auto* .)
      (auto* .)
  (lem conc-rev : A{L1:List ; L2:List}
    ((rev(L1:List ++ L2:List)) = (rev(L2:List) ++ rev(L1:List))) .)
    (ind on L1:List .)
      (auto* .)
      (auto* .)
  (auto* .)
```

4.3 Ordered lists

The final example we consider is showing that the typing assigned to the operation `insert-list` in `ORD-LIST` is indeed correct; that is, that inserting a new element to an ordered list returns an ordered list. Note that in doing so we use an equationally defined predicate `sorted` instead of a membership assertion `_: OrdList`; again, this is due to the ITP's current restrictions.

The proof proceeds as usual by structural induction and, as in the previous example, we cannot discharge the second case, corresponding to a list of the form `E : L`, with a simple `auto*`. Note that this time, before proving the auxiliary lemma it is also necessary to distinguish the case in which the value `N` being inserted is less than or equal to `E`, from that in which it is not. This is done in the ITP with the `split` command: `N*` and `V0#0*Nat` are the constants to which `auto*` has transformed the original variables `N` and `E`. The rest of the proof follows the same pattern as the previous ones.

```
(goal SORTED : ORD-LIST |- A{L:List ; N:Nat}
  (((sorted(L:List)) = (true)) =>
    ((sorted(insert-list(L:List, N:Nat))) = (true))) .)
```

```

(ind on L:List .)
  (auto* .)
  (auto* .)
  (split on (N*Nat <= V0#0*Nat) .)
  (auto* .)
  (auto* .)
(lem SORTED2 : A{N:Nat ; M:Nat ; L:List}
  (((sorted(N:Nat : L:List)) = (true) & (N:Nat <= M:Nat) = (true)) =>
    ((sorted(N:Nat : insert-list(L:List, M:Nat))) = (true))) .)
(ind on L:List .)
  (auto* .)
  (auto* .)
  (split on (M*Nat <= V1#0*Nat) .)
  (auto* .)
  (auto* .)
(auto* .)

```

5 Ongoing and future work

As mentioned in the introduction, we consider this work as a first step to develop a set of basic data structures that could eventually lead to a shared library; now, other users' feedback will be necessary to improve this contribution. For such a library, one would expect an implementation as efficient as possible; however, there is a clear tension between this goal and that of using the specifications to illustrate programming in Maude.

There are a number of techniques to improve the efficiency in Maude, like the systematic use of *unconditional* equations with the help of the `if_then_else-fi` operator and the `owise` attribute. We have rewritten several of our examples using these techniques and obtaining a considerable gain in efficiency, as expected. They have the drawback of making formal reasoning about the specifications much more difficult or even impossible with the current version of the ITP. There is also the tradeoff between efficiency and very precise typing by means of memberships; these require typechecking during execution. If one is willing to work with less refined types, as is the case with other functional languages, one can forget about most of the memberships that appear in our specifications, thus obtaining another considerable speedup. Finally, one must distinguish also between the efficiency in the Full Maude prototype that works at the metalevel and the one in the Core Maude system implemented in C++; the transfer of parameterization techniques to the core level will help in this respect. In particular, according to the limited set of tests that we have run, Core Maude's performance in executing non-parametric specifications is close to that of GHCi for Haskell 98 [8], whereas Full Maude can be as much as three times slower.

Concerning the proof of properties, in collaboration with Manuel Clavel we have managed to prove more complex ones, including the correctness of the sorting operations `mergesort` and `quicksort`. However, we are still finding our way with proofs of basic properties about search trees. In the future, we would

like to consider more complex relationships such as those between 2-3-4 and red-black trees.

Acknowledgments. We warmly thank Ricardo Peña for our discussions on functional implementations of data structures, Manuel Clavel for his work on the ITP and his help in understanding its idiosyncrasies, José Meseguer for his encouragement, and the referees for their detailed and helpful comments.

References

1. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
2. F. M. Carrano and J. J. Prichard. *Data Abstraction and Problem Solving with C++. Third Edition*. Addison-Wesley, 2002.
3. M. Clavel. The ITP Tool. <http://geminis.sip.ucm.es/~clavel/itp>, 2004.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Que-sada. *A Maude Tutorial*. Computer Science Laboratory, SRI International, 2000. <http://maude.cs.uiuc.edu/papers>.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 2003, Proceedings*, LNCS 2706, pages 76–87. Springer, 2003.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.1)*, March 2004. <http://maude.cs.uiuc.edu/manual>.
7. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, 1999.
8. The GHC team. *The Glasgow Haskell Compiler*, March 2004. <http://www.haskell.org/ghc>.
9. E. Horowitz and S. Sahni. *Fundamentals of Data Structures in Pascal. Fourth Edition*. Computer Science Press, 1994.
10. N. Martí-Oliet, M. Palomino, and A. Verdejo. Data structures in Maude. <http://maude.sip.ucm.es/datastructures>, 2004.
11. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, LNCS 1376, pages 18–61. Springer, 1998.
12. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
13. M. Palomino, N. Martí-Oliet, and A. Verdejo. Playing with Maude. In *RULE'04 Fifth International Workshop on Rule-Based Programming. Proceedings*, Electronic Notes in Theoretical Computer Science, pages 4–19. Elsevier, 2004.
14. R. Peña. *Diseño de Programas. Formalismo y Abstracción. Segunda Edición*. Prentice Hall, 1998.
15. F. Rabhi and G. Lapalme. *Algorithms. A Functional Programming Approach*. Addison-Wesley, 1999.
16. M. A. Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley, 1998.

Towards (constructor) normal forms for Maude within Full Maude^{*}

Francisco Durán¹, Santiago Escobar², and Salvador Lucas²

¹ Universidad de Málaga, Campus de Teatinos, Málaga, Spain
duran@lcc.uma.es

² Universidad Politécnica de Valencia, Valencia, Spain
{sescobar,slucas}@dsic.upv.es

Abstract. Maude is able to deal with infinite data structures and avoid infinite computations by using *strategy annotations*, that is, positive indices that indicate which positions can be evaluated. However, they can eventually make the computation of the normal form(s) of some input expressions impossible. In [6,7], we used Full Maude to implement two new commands `norm` and `eval` which furnish Maude with the ability to compute (constructor) normal forms of initial expressions even when the use of strategy annotations together with the built-in computation strategy of Maude is not able to obtain them. These commands were integrated into Full Maude, making them available inside the programming environment like any other of its commands. Moreover, the type of annotations allowed was extended, giving to Maude the ability of dealing with on-demand strategy annotations, that is, negative indices that express *evaluation on-demand*, where the *demand* is a failed attempt to match an argument term with the left-hand side of a rewrite rule. In this paper, we recall these new commands and extensions.

1 Introduction

Handling infinite objects is a typical feature of lazy (functional) languages. Although reductions in Maude [4,5] are basically *innermost* (or eager), Maude is able to exhibit a similar behavior by using *strategy annotations* [17]. Maude strategy annotations are lists of non-negative integers associated to function symbols which specify the order in which the arguments are (eventually) evaluated in function calls. Specifically, when considering a function call $f(t_1, \dots, t_k)$, only the arguments whose indices are present as *positive* integers in the local strategy $(i_1 \dots i_n)$ for f are evaluated (following the specified order). Besides, if 0 is found, a reduction step on the whole term $f(t_1, \dots, t_k)$ is attempted. Every symbol without an explicit strategy annotation receives a standard local strategy $(1 \ 2 \ \dots \ k \ 0)$ by Maude, where k is the number of its arguments. Let us illustrate by an example how local strategies are used in Maude.

^{*} Work partially supported by CICYT under grants TIC2001-2705-C03-01 and TIC2001-2705-C03-02, by MCyT under grant HU 2003-0003, by Generalitat Valenciana under grants GRUPOS03/025 and GV04B-477, and by the EU-India Cross-Cultural Dissemination project ALA/95/23/2003/077-054.

Example 1. Consider the following Maude modules³ **LAZY-NAT**, which provides symbols **0** and **s** for specifying natural numbers in Peano syntax (sort **NatP**), **LAZY-LIST**, with a ‘polymorphic’ sort **List(X)** and symbols **nil** (the empty list) and **_. _** for the construction of polymorphic lists, and an instance of module **LAZY-LIST** for natural numbers, called **LIST-NAT**:

```
(fmod LAZY-NAT is
  sort NatP .
  op 0 : -> NatP .
  op s : NatP -> NatP [strat (0)] .
  op _+_ : NatP NatP -> NatP .
  vars M N : NatP .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
endfm)

(fth TRIV is
  sort Elt .
endfth)

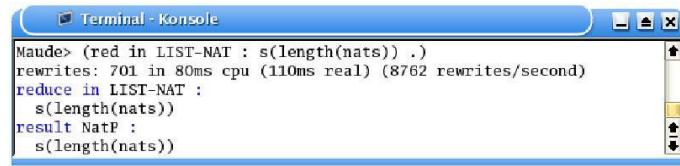
(fmod LAZY-LIST(X :: TRIV) is
  protecting LAZY-NAT .
  sort List(X) .
  subsort X@Elt < List(X) .
  op nil : -> List(X) .
  op _._ : X@Elt List(X) -> List(X) [strat (1 0)] .
  op length : List(X) -> NatP .
  var X : X@Elt .   var XS : List(X) .
  eq length(nil) = 0 .
  eq length(X . XS) = s(length(XS)) .
endfm)

(view NatP from TRIV to NAT is
  sort Elt to NatP .
endv)

(fmod LIST-NAT is
  protecting LAZY-NAT .
  protecting LAZY-LIST(NatP) .
  op incr : List(NatP) -> List(NatP) .
  op nats : -> List(NatP) .
  var X : NatP .   var XS : List(NatP) .
  eq incr(X . XS) = s(X) . incr(XS) .
  eq nats = 0 . incr(nats) .
endfm)
```

³ Actually, we will use the Full Maude extension instead of **Maude** in the examples. This is denoted by enclosing each module or command with parenthesis.

Strategy annotations can often improve the termination behavior of programs, i.e., they prune all infinite rewrite sequences starting from an expression at an argument position that is not present in the local strategy of the corresponding symbol. In the example above, the strategies (0) and (1 0) for symbols s and $_{-}._{-}$, respectively, guarantee that the resulting program is terminating⁴. For instance, the absence of 1 in the local strategy of s stops any infinite reduction in the term $s(\text{length}(\text{nats}))$, as shown by the standard reduction command `red` in Maude⁵

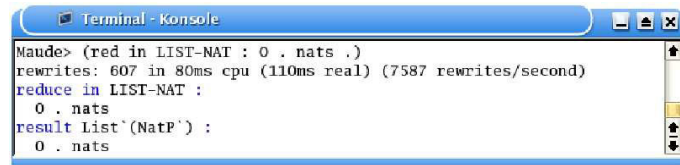


```

Maude> (red in LIST-NAT : s(length(nats)) .)
rewrites: 701 in 80ms cpu (110ms real) (8762 rewrites/second)
reduce in LIST-NAT :
  s(length(nats))
result NatP :
  s(length(nats))

```

and the absence of 2 in the local strategy of $_{-}._{-}$ stops any infinite reduction in the term `0 . nats`



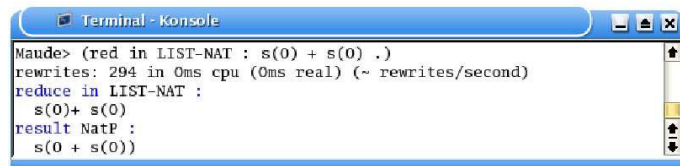
```

Maude> (red in LIST-NAT : 0 . nats .)
rewrites: 607 in 80ms cpu (110ms real) (7587 rewrites/second)
reduce in LIST-NAT :
  0 . nats
result List'(NatP') :
  0 . nats

```

Moreover, apart from improving termination, strategy annotations can also improve the efficiency of computations [10], e.g., by reducing the number of attempted matchings or avoiding useless or duplicated reductions.

Nevertheless, the absence of some indices in the local strategies can also jeopardize the ability of such strategies to compute normal forms. For instance, the evaluation of the expression $s(0) + s(0)$ w.r.t. Example 1 yields the following:



```

Maude> (red in LIST-NAT : s(0) + s(0) .)
rewrites: 294 in 0ms cpu (0ms real) (~ rewrites/second)
reduce in LIST-NAT :
  s(0)+ s(0)
result NatP :
  s(0 + s(0))

```

Due to the local strategy (0) for the symbol s , the contraction of the redex $0 + s(0)$ is not possible and the evaluation stops here with an expression that is not the intended value expected by the user.

The handicaps, regarding correctness and completeness of computations, of using (only) positive annotations have been pointed out in the literature, e.g. in [1,2,14,21,22], and a number of solutions have been proposed:

⁴ The termination of this specification can be formally proved by using the tool MU-TERM, see <http://www.dsic.upv.es/~slucas/csr/termination/muterm>.

⁵ The Maude 2.1 interpreter [5] is available at <http://maude.cs.uiuc.edu>.

1. Performing a *layered normalization*: when the evaluation stops due to the replacement restrictions introduced by the strategy annotations, it is resumed over concrete inner parts of the resulting expression until the normal form is reached (if any) [15];
2. transforming the program to obtain a different one which is able to obtain sufficiently interesting outputs (e.g., constructor terms) [2]; and
3. using strategy annotations with *negative* indices which allows for some extra evaluation *on-demand*, where the *demand* is a failed attempt to match an argument term with the left-hand side of a rewrite rule [21,22,1], like the (functional) lazy evaluation strategy.

In [6], we introduced two new **Maude** commands (**norm** and **eval**) to make techniques 1 and 2 available for the execution of **Maude** programs, and in [7], we extended the original **Maude** command **red** and the new command **norm** to make technique 3 also available into **Maude**. In this paper, we recall these new commands and extensions and refer the reader to [6,7] for details.

First, let us illustrate with the following example how negative indices can also improve the behavior of **Maude** programs.

Example 2. Continuing Example 1. The following **NATS-T0-BIN** module implements the binary encoding of natural numbers as lists of symbols 0 and 1 of sort **Binary** (starting from the least significative bit).

```
(fmod BINARY is
  sort Binary .
  ops 0 1 : -> Binary .
endfm)

(view Binary from TRIV to BINARY is
  sort Elt to Binary .
endv)

(fmod NATS-T0-BIN is
  protecting LAZY-NAT .
  protecting LAZY-LIST(Binary) .
  op natToBin : NatP -> List(Binary) .
  op natToBin2 : NatP NatP -> List(Binary) .
  vars M N : NatP .
  eq natToBin2(0, 0) = 0 .
  eq natToBin2(0, M) = 0 . natToBin(M) .
  eq natToBin2(s(0), 0) = 1 .
  eq natToBin2(s(0), M) = 1 . natToBin(M) .
  eq natToBin2(s(s(N)), M) = natToBin2(N, s(M)) .
  eq natToBin(N) = natToBin2(N, 0) .
endfm)
```

The evaluation of the expression **natToBin(s(0) + s(0))** should yield the binary representation of 2. However, we get:

```

Maude> (red natToBin(s(0) + s(0)) .)
rewrites: 320 in 0ms cpu (0ms real) (~ rewrites/second)
reduce in NATS-T0-BIN :
  natToBin(s(0)+ s(0))
result List'(Binary') :
  natToBin2(s(0 + s(0)),0)

```

The problem is that the current local strategy (0) for symbol s disallows the evaluation of subexpression $0 + s(0)$ in $\text{natToBin2}(s(0 + s(0)), 0)$, and thus disables the application of the last equation for natToBin2 . The use of the techniques 1 or 2 (through commands `norm` and `eval` introduced in [6]) do not solve this problem, since they just normalize non-reduced subexpressions but never replace the symbol at the root position:

```

Maude> (norm natToBin(s(0) + s(0)) .)
rewrites: 630 in 0ms cpu (0ms real) (~ rewrites/second)
normalize in NATS-T0-BIN :
  natToBin(s(0)+ s(0))
result List'(Binary') :
  natToBin2(s(s(0)),0)

```

In [21,22,1], *negative* indices are proposed to indicate those arguments that should be evaluated only “on-demand”, where the “demand” is a failed attempt to match an argument term with the left-hand side of a rewrite rule [22]. For instance, the evaluation of the subterm $0 + s(0)$ of the term $\text{natToBin2}(s(0 + s(0)), 0)$ in the previous example is *demand*ed by the last equation for symbol natToBin2 , i.e., by its left-hand side $\text{natToBin2}(s(s(N)), M)$. Specifically, the argument of the outermost occurrence of the symbol s in $\text{natToBin2}(s(0 + s(0)), 0)$ is rooted by a defined function symbol, $_{+}$, whereas the corresponding operator in the left-hand side is s . Thus, before being able to apply the rule, we have to further evaluate $0 + s(0)$ in order to eventually remove $_{+}$ by s at the root position.

Example 3. Continuing Example 2. Consider the specification resulting from replacing the declaration of the operator s by the following one with a new local strategy including a negative annotation that declares the first argument of s as evaluable only on-demand:

```
op s : NatP -> NatP [strat (-1 0)] .
```

The on-demand evaluation of $\text{natToBin}(s(0) + s(0))$ (through the extended command `red` introduced in [7]) obtains the appropriate head-normal form:

```

Maude> (red on-demand natToBin(s(0) + s(0)) .)
rewrites: 5703 in 10ms cpu (10ms real) (570300 rewrites/second)
reduce on-demand in NATS-T0-BIN :
  natToBin(s(0)+ s(0))
result List'(Binary') :
  0 . natToBin(s(0))

```

Specifically, from the expression `natToBin2(s(0 + s(0)), 0)` considered above, we have that the following evaluation step is demanded by the left-hand side `natToBin2(s(s(N)), M)`:

$$\text{natToBin2}(s(0 + s(0)), 0) \rightarrow \text{natToBin2}(s(s(0 + 0)), 0)$$

Then, the rule with left-hand side `natToBin2(s(s(N)), M)` can be applied, which implies that no evaluation is demanded:

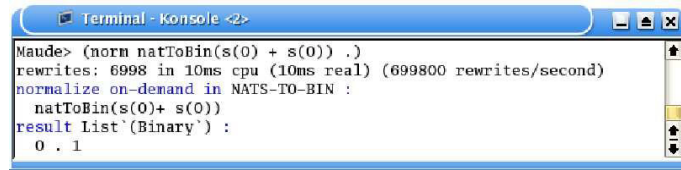
$$\text{natToBin2}(s(s(0 + 0)), 0) \rightarrow \text{natToBin2}(0 + 0, s(0))$$

Finally, the remaining reductions leading to the expression `0 . natToBin(s(0))` are not demanded and follow from the positive local indices given above. ■

From the previous examples, we can infer that, in some situations, the evaluation with (only) positive annotations either enters in an infinite derivation —e.g., for the term `length(nats)`, with the strategy `(1 0)` for symbol `s`— or does not provide the intended normal form —e.g., with the strategy `(0)` for symbol `s`—, whereas the strategy `(-1 0)` gives an appropriate local strategy for symbol `s`, since it makes its argument to be evaluated only “on demand”, and thus obtains the appropriate head-normal forms while still avoiding infinite derivations.

Nevertheless, our main motivation is to provide appropriate normal forms to programs with strategy annotations, and thus the redefinition of command `red` for on-demand evaluation is not enough, since it is not able to provide the normal form `0 . 1` for the program in Example 2 because the annotation 2 is missing in the strategy list for symbol `...` (see the output of the `red` command in Example 3). As it was explained above, this concrete problem is solved using either the layered normalization —technique 1— or a program transformation —technique 2—. And therefore, in [7], the output given by an on-demand evaluation with command `red` is used as the starting point of a layered evaluation coded into an appropriate redefinition of the command `norm` of [6] for dealing with negative indices.

Example 4. Consider the modules of Example 3. The layered normalization technique extended for on-demand annotations (through the extended command `norm` introduced in [7]) is able to provide the intended normal form associated to the expression `natToBin(s(0) + s(0))`.



```

Maude> (norm natToBin(s(0) + s(0)) .)
rewrites: 6998 in 10ms cpu (10ms real) (699800 rewrites/second)
normalize on-demand in NATS-TO-BIN :
  natToBin(s(0)+ s(0))
result List'(Binary') :
  0 . 1

```

It is worth to note that all these features and appropriate evaluations are achieved without entering in non-terminating evaluations. We refer the reader to [11] for a recent and detailed discussion about the use of on-demand strategy annotations in programming.

2 New commands for Maude within Full Maude

In the following, we provide some useful insights about how these features are implemented in Full Maude. The complete specifications can be found in

<http://www.dsic.upv.es/users/elp/toolsMaude>

2.1 Full Maude, reflection, and the META-LEVEL module

The reflective capabilities of Maude are the key for building these language extensions, which turn out to be very simple to use thanks to the infrastructure provided by Full Maude. Full Maude is an extension of Maude, written in Maude itself, that endows Maude with notation for object-oriented modules and with a powerful and extensible module algebra [4]. Its design, and the level of abstraction at which it is given, make of it an excellent metalevel tool to test and experiment with features and capabilities not present in (Core) Maude [8,9,4]. In fact, reflection, together with the good properties of the underlying *rewriting logic* [18] as a logical framework [20,19], makes quite easy the development of formal tools and execution environments in Maude for any logic $\mathcal{L}$ of interest, including rewriting logic itself (see e.g. [9,3]).

Maude's design and implementation systematically exploit the reflective capabilities of rewriting logic, providing key features of the universal theory in its built-in META-LEVEL module. In particular, META-LEVEL has sorts **Term** and **Module**, so that the representations of a term t and of a module $\mathcal{R}$ are, respectively, a term $\bar{t}$ of sort **Term** and a term $\bar{\mathcal{R}}$ of sort **Module**.

The basic cases in the representation of terms are obtained by subsorts **Constant** and **Variable** of the sort **Qid** of quoted identifiers. Constants are quoted identifiers that contain the name of the constant and its type separated by a dot, e.g., `'0.NatP`. Similarly, variables contain their name and type separated by a colon, e.g., `'N:NatP`. Then, a term is constructed in the usual way, by applying an operator symbol to a list of terms.

```
subsorts Constant Variable < Qid Term .
subsort Term < TermList .
op _,_ : TermList TermList -> TermList [assoc] .
op _[] : Qid TermList -> Term .
```

For example, the term `natToBin2(s(s(0)), 0)` of sort `List(Binary)` in the module `NATS-T0-BIN` is metarepresented as

```
'natToBin2['s['s['0.NatP]], '0.NatP]
```

The META-LEVEL module also includes declarations for metarepresenting modules. For example, a functional module can be represented as a term of sort **FModule** using the following operator.

```
op fmod_is_sorts_.....endfm : Qid ImportList SortSet
  SubsortDeclSet OpDeclSet MembAxSet EquationSet -> FModule .
```

Similar declarations allow us to represent the different types of declarations we can find in a module.

The module **META-LEVEL** also provides key metalevel functions for rewriting and evaluating terms at the metalevel, namely, **metaApply**, **metaRewrite**, **metaReduce**, etc., and also generic parsing and pretty printing functions **metaParse** and **metaPrettyPrint** [5,4]. For example, the function **metaReduce** takes as arguments the representation of a module $\mathcal{R}$ and the representation of a term t in that module:

```
op metaReduce : Module Term -> [ResultPair] .
op {_,_} : Term Type -> ResultPair .
```

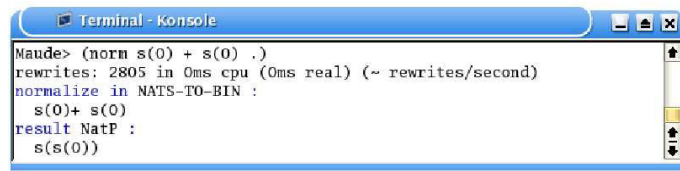
metaReduce returns the representation of the fully reduced form of the term t using the equations in $\mathcal{R}$, together with its corresponding sort or kind.

All these functionalities are very useful for metaprogramming. In particular, we have used the extensibility and flexibility of Full Maude to permit the definition of new commands such as **norm** and **eval** but also to adapt the usual Maude evaluation command **red** and the new command **norm** to the on-demand evaluation.

2.2 Extending Full Maude to handle the command **norm**

The *normalization via μ -normalization* procedure was introduced in [15,16] as a simple mechanism to furnish the *context-sensitive rewriting (CSR)* [13] with the ability to compute normal forms of initial expressions even though *CSR* itself is not able to compute them. The idea is very simple: if we are able to ensure that the normal forms computed by *CSR* are always head-normal forms, then it is safe to get into the maximal “*non-replacing*” subterms of the normal form s of a term t computed by *CSR* to (recursively) continue the computation. The technique works for left-linear, confluent TRSs $\mathcal{R}$ that use a *canonical* replacement map; see [15]. The procedure is sound in the context of Maude due to an appropriate formal connection between *CSR* and the evaluation mechanism of Maude [6].

This normalization via μ -normalization procedure has been implemented in [6] as a new command **norm** that uses the outcome of **red** to perform this layered evaluation of the initial expressions. The new command **norm** permits to obtain the intended value of many expressions, e.g. for the term $s(0) + s(0)$ given above:



```
Maude> (norm s(0) + s(0) .)
rewrites: 2805 in 0ms cpu (0ms real) (~ rewrites/second)
normalize in NATS-TO-BIN :
s(0)+ s(0)
result NatP :
s(s(0))
```

As for other commands in Maude, we may define the actions to take when the command is used by defining its corresponding meta-function. For instance,

a **red** command is executed by appropriately calling the **metaReduce** function. In the case of **norm**, we define an operation **metaNorm**. Basically, **metaNorm** calls the auxiliary function **metaNormRed**, which first reduces the term using **metaReduce**, and then proceeds recursively on each of the arguments of the resulting term. That is, for each argument of the symbol at the root of the resulting term, if the argument has evaluation restrictions, then it calls **metaNormRed**, otherwise it calls **metaNormNoRed**. **metaNormNoRed** proceeds as **metaNormRed**, but without reducing the term before going on the arguments, since it is not necessary.

2.3 Extending Full Maude to handle the command eval

When considering how the evaluation command **norm** works for a concrete input term t , we understand that it is interesting to isolate the replacement restrictions needed to achieve the head-normal form of t (produced by the command **red**) from the restrictions needed to get its normal form (produced by the command **norm**). In the case of constructor normal forms, we can rather consider a constructor (prefix) context $C[\]$ of the normal form of t such that $C[\] \in \mathcal{T}(\mathcal{B} \cup \{2\}, \mathcal{X})$ for some $\mathcal{B} \subseteq \mathcal{C}$. The intuitive idea is that the set of constructor symbols $\mathcal{B}$ characterizes those constructor symbols which are (or could eventually be) present in the normal form of t . Then, reductions *below* the outermost defined symbols should be performed only up to (constructor) head-evaluation (i.e., a term rooted by a symbol in $\mathcal{B}$ computed by command **red**), thus providing an incremental computation of the normal form of t . In [2], a program transformation aimed at achieving this goal was given. The key idea for the transformation is to duplicate every symbol in $\mathcal{B}_\tau$, for each sort τ in the specification that is involved in $\mathcal{B}$, to a fresh constructor symbol in the new set $\mathcal{B}'_\tau$, which does not have any restriction on its arguments. More specifically, given a sort τ , the set $\mathcal{C}_\tau^* \subseteq \mathcal{C}$ is the set of constructor symbols that can be found in constructor terms of sort τ . For instance, $\mathcal{C}_{\text{NatP}}^* = \{0, s_-\}$ and $\mathcal{C}_{\text{List}(\text{NatP})}^* = \{\text{nil}, -., 0, s_-\}$. Hence, the set $\mathcal{C}_\tau^*$ will tell us which constructor symbols must be renamed.

The renaming of the constructor symbols $c \in \mathcal{C}_\tau^*$ into new constructor symbols c' is performed by the following new rules, which are added to the original program during the transformation:

$$\text{quote}_{\text{sort}(c)}(c(x_1, \dots, x_k)) \rightarrow c'(\text{quote}_{\text{sort}(x_1)}(x_1), \dots, \text{quote}_{\text{sort}(x_k)}(x_k))$$

In practice, we use the overloading facilities of Maude and introduce a single (overloaded) symbol **quote**.

The evaluation of a term t would proceed by reducing **quote**(t) into a term with a constructor prefix context in $\mathcal{C}'$, which do not have any evaluation restriction. Then, we perform a postprocessing that recovers the original names of the constructor symbols after each argument of a symbol $c' \in \mathcal{C}'$ has been evaluated:

$$\text{unquote}(c'(x_1, \dots, x_k)) \rightarrow c(\text{unquote}(x_1), \dots, \text{unquote}(x_k))$$

Again, the symbol **unquote** is conveniently overloaded.

In [6], we implement a new command `eval` that uses this transformation to obtain the constructor normal form (if any) associated to a given input expression t . In contrast to the command `norm`, we first transform the module and, then, we simply reduce the expression `unquote(quote( $t$ ))` within the new module (using the `metaReduce` function). The new command `eval` permits to obtain the intended value of the term `s(0) + s(0)` given above:

```

Maude> (eval in LAZY-NAT : s(0) + s(0) .)
rewrites: 5114 in 40ms cpu (40ms real) (127850 rewrites/second)
transforming module LAZY-NAT for symbol _+_
transformed module QLAZY-NAT is complete for symbol _+_
reduce in QLAZY-NAT :
  unquote(quote(s(0)+ s(0)))
result NatP :
  s(s(0))

```

2.4 Extending Full Maude to handle on-demand strategy annotations

We have followed the computational model defined in [1] for dealing with negative annotations, where a local strategy for a k -ary symbol $f \in \mathcal{F}$ is a sequence of integers in $\{-k, \dots, -1, 0, 1, \dots, k\}$. In this computational model, each symbol in the term t to be evaluated is conveniently annotated with its local strategy, yielding a new class of annotated terms. The evaluation takes the annotated term and the strategy glued to its top symbol, and then proceeds as follows [1]: if a positive argument index is provided, then the evaluation jumps to the subterm at such argument position; if a negative argument index is provided, then the index is consumed but nothing is done; if a zero is found, then we try to find a rule to be applied on such a term. If no rule can be applied at the root position when a zero is found, then we proceed to look for its (demanded) evaluations, that is, we try to reduce one of the subterms at positions under a (consumed or present) negative index. All consumed indices (positive and negative) are glued also to each symbol in the term using an extra strategy list, so that demanded positions can appropriately be searched. See [1] for a formal description.

In [7], we provide the reduction of terms taking into account on-demand annotations as a redefinition of the usual evaluation command `red` of Maude (which considers only positive annotations). As for other commands in Full Maude, we provide a new metalevel operation `metaReduceOnDemand` which extends the reflective and metalevel capabilities of Maude. The redefined command `red` must then select between `metaReduce` and `metaReduceOnDemand` depending on whether negative annotations are present or not in the module associated to the expression to execute. This on-demand extension of the usual command `red` permits to produce the appropriate head-normal form of the term `natToBin(s(0) + s(0))`, as it was shown in Example 3.

It is worthy to note that although eager (or innermost) evaluation strategies correspond to local strategies of the shape $(1\ 2\ \dots\ k\ 0)$ (see [17] for the

exact relationship) lazy (or outermost) evaluation strategies do not directly correspond to local strategies of the shape $(-1 \ -2 \ \dots \ -k \ 0)$ [12]. Indeed, we plan to investigate this as future work.

2.5 Extending Full Maude with on-demand strategy annotations to layered normalization

The redefinition of command `norm` to deal with on-demand strategy annotations is almost identical to the implementation of the command `norm` given in [6] and reviewed in Section 2.2. The idea is that we keep the metalevel function `metaNorm` and define a new metalevel function `metaNormOnDemand` which calls `metaReduceOnDemand` instead of `metaReduce` when negative annotations are present in the module associated to the expression to execute. The redefinition of the command `norm` permits to obtain the intended value of the term `natToBin(s(0) + s(0))`, as it was shown in Example 4.

3 Conclusions

In this paper, we have presented two new commands `norm` and `eval` which furnish **Maude** with the ability to compute (constructor) normal forms of expressions even when the built-in computation strategy is not able to obtain them. We have also presented the extension of the type of annotations allowed in **Maude**, that gives to **Maude** the ability of dealing with on-demand strategy annotations, that is, negative indices that express evaluation on-demand like functional lazy evaluation.

Furthermore, the reader may note that strategy annotations have been used for many years in programming languages such as **Maude** but the formal analysis of computations under such annotated strategies has been recently addressed.

Acknowledgements. Thanks to the anonymous referees for their helpful remarks.

References

1. M. Alpuente, S. Escobar, B. Gramlich, and S. Lucas. Improving on-demand strategy annotations. In *Proc. of the 9th Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'02)*, LNCS 2514:1–18. Springer-Verlag, 2002.
2. M. Alpuente, S. Escobar, and S. Lucas. Correct and complete (positive) strategy annotations for OBJ. In *Proc. of the 4th Int'l Workshop on Rewriting Logic and its Applications, WRLA 2002*, ENTCS 71. Elsevier, 2003.
3. M. Clavel, F. Durán, S. Eker, J. Meseguer, and M. Stehr. Maude as a formal meta-tool. In *Proc. of the FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Volume II*, LNCS 1709:1684–1703. Springer-Verlag, 1999.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

5. M. Clavel, F. Durán, S. Eker, P. Lincoln, J. Meseguer N. Martí-Oliet, , and C. Talcott. The Maude 2.0 System. In *Rewriting Techniques and Applications, 12th Int'l Conf., RTA 2003*, LNCS 2706:76–87. Springer-Verlag, 2003.
6. F. Durán, S. Escobar, and S. Lucas. New evaluation commands for Maude within Full Maude. In *5th Int'l Workshop on Rewriting Logic and its Applications, WRLA'04*, ENTCS to appear. Elsevier, 2004.
7. F. Durán, S. Escobar, and S. Lucas. On-demand evaluation for Maude. In *Proc. of the 5th Int'l Workshop on Rule-Based Programming, RULE'04*, ENTCS to appear. Elsevier, 2004.
8. F. Durán and J. Meseguer. An extensible module algebra for Maude. In *Proc. of the 2nd Int'l Workshop on Rewriting Logic and its Applications, WRLA'98*, ENTCS 15:185–206. Elsevier, 1998.
9. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, Spain, June 1999.
10. S. Eker. Term rewriting with operator evaluation strategies. In *Proc. of the 2nd Int'l Workshop on Rewriting Logic and its Applications, WRLA 98*, ENTCS 15. Elsevier, 2000.
11. S. Escobar. *Strategies and Analysis Techniques for Functional Program Optimization*. PhD thesis, Universidad Politécnica de Valencia, Spain, October 2003.
12. O. Fissore, I. Gnaedig, and H. Kirchner. Outermost ground termination. In *Proc. of the 4th Int'l Workshop on Rewriting Logic and its Applications, WRLA 2002*, ENTCS 71. Elsevier, 2003.
13. S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):1–61, 1998.
14. S. Lucas. Termination of on-demand rewriting and termination of obj programs. In *Proc. of the 3rd Int'l ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP'01*, pages 82–93. ACM Press, 2001.
15. S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):294–343, 2002.
16. S. Lucas. Termination of (Canonical) Context-Sensitive Rewriting. In *Proc. of the 13th Int'l Conf. on Rewriting Techniques and Applications, RTA'02*, LNCS 2378:296–310. Springer-Verlag, 2002.
17. S. Lucas. Semantics of programs with strategy annotations. Technical Report DSIC-II/08/03, DSIC, Universidad Politécnica de Valencia, 2003.
18. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
19. J. Meseguer. Research directions in rewriting logic. In *Computational Logic, Proceedings of the NATO Advanced Study Institute on Computational Logic held in Marktoberdorf, Germany, July 29 – August 6, 1997*, volume 165 of *NATO ASI Series F: Computer and Systems Sciences*, pages 347–398. Springer-Verlag, 1998.
20. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In *Handbook of Philosophical Logic*, volume 9, pages 1–88. Kluwer Academic Publishers, 2002.
21. M. Nakamura and K. Ogata. The evaluation strategy for head normal form with and without on-demand flags. In *Proc. of the 3rd Int'l Workshop on Rewriting Logic and its Applications, WRLA 2000*, ENTCS 36. Elsevier, 2001.
22. K. Ogata and K. Futatsugi. Operational semantics of rewriting with the on-demand evaluation strategy. In *Proc. of 2000 Int'l Symposium on Applied Computing, SAC'00*, pages 756–763. ACM Press, 2000.

Invariant-Based Control of the Execution of Maude Specifications: The LTL Case

Francisco Durán and Manuel Roldán

LCC, Universidad de Málaga, Málaga, Spain.
`{duran,mrc}@lcc.uma.es`

Abstract. This work presents a general mechanism for executing specifications that comply with given invariants, which may be expressed in different formalisms. We exploit Maude’s reflective capabilities and its good properties as a semantic framework to provide a generic strategy that allows us to execute (possibly nonterminating and nonconfluent) Maude specifications taking into account invariants expressed in some logic. That is, the strategy is parameterized by the invariants and by the logic in which such invariants are expressed. Although other logics may be used, we focus in this work on the case of (Finite Future Time) Linear Temporal Logic.

1 Introduction

To deal with nonterminating and nonconfluent systems, we need good ways of controlling the rewriting inference process. In this line, different languages offer different mechanisms, including approaches based on metaprogramming, like Maude [3], or on strategy languages, like ELAN [1]. However, although the separation of logic and control greatly simplifies such a task, these mechanisms are sometimes hard to use, specially for beginners, and usually compromise fundamental properties like extensibility, reusability, and maintainability. Things may get really bad when trying to specify complex strategies. A declarative approach may help on this, if we are able to specify *what* must happen, instead of *how* it must be done.

Although not for controlling the execution process, formalisms like Z and UML allow the definition of invariants or constraints as part of the system specifications. In fact, given the advantages of specification executability, a great effort is being made in this direction in the Z and UML communities. Thus, we can find tools like Possum [9] which can do a reasonable simulation of Z specifications, and UML specifications can be made executable by specifying OCL constraints on them [14], with tools like USE [13] giving support to it.

In the case of rewriting logic based languages like Maude or ELAN the situation is different. The specifications in these languages are executable, being the possible transitions in the system expressed as rewrite rules. However, in nonterminating and nonconfluent systems the rewriting process may go in many undesired directions, and we need good ways to control it. Maude’s reflective capabilities allow the specification of *internal strategies* to guide the execution [2].

Although very expressive, the price for using such facilities is very high, even when considering very simple strategies.

Instead of considering invariants as properties that can be proved (or inferred) from the specifications they are defined on, we are interested in *constraining invariants*, that is, invariants that restrain the possible configurations and behavior of the system. Such invariants must be taken into account when executing or simulating the system being specified. Monitoring techniques use an external monitor to analyze whether the actual execution trace of the system satisfies a given property, analyzing the states obtained during the system execution for detecting nonvalid states, stopping the system if necessary, or trying to recover from it when possible. But, when executing specifications with constraining invariants, we do not want to stop the execution of the program when we reach a state in which the invariant is not satisfied, as we can do when monitoring. On the contrary, what we want is to *drive* the system execution using such invariants, hence preventing the system from getting into any invalid state.

The execution or simulation of specifications with constraining invariants is typically based on integrating somehow the invariants into the system code. However, such an integration is clearly unsatisfactory: the invariants get lost amidst the code, and become difficult to locate, trace, and maintain. Moreover, the programs and the invariants to be satisfied on them are usually expressed in different formalisms, and *live at different levels of abstraction*: invariants are defined *on* programs. Therefore, it is interesting to have some way of expressing them separately, thus avoiding the mixing of invariants and code. Moreover, we would like to be able to express the invariants in different logics, choosing the most appropriate one for each particular situation.

Maude does not provide direct support for expressing execution invariants. However, it does provide reflective capabilities and support to control the execution process, being also an excellent tool in which to create executable environments for various logics and models of computation [4]. Thus, it turns out to be a very good candidate for giving support to different types of invariants, which may be expressed in different formalisms.

In [5], we proposed the use of generic invariant-driven strategies to control the execution of systems by guaranteeing that the given invariants are always satisfied. Our strategies are generic in the sense that they are parameterized by: (1) the system whose execution they control, (2) the logic in which the invariants are expressed, and (3) the invariants themselves. The good properties of Maude as a logical and semantic framework [8], in which many different logics and formalisms can be expressed and executed allows us to say that other logics and formalisms may be used as parameters of our strategies. We considered the case of propositional logic in [5]. In this paper, we go a step forward and show how the logic in which such properties are expressed may be changed. In particular, we show how temporal logics, and in particular (finite future time) Linear time Temporal Logic, abbreviated LTL, can be used to express such invariants.

Future time Linear Temporal Logic was introduced by Pnueli [12] for stating properties about reactive and concurrent systems. LTL provides temporal operators that refer to the future/remaining part of an execution trace relative to a current point of reference. The standard models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests. Using temporal logic for proving programs correct or in testing them with respect to requirements specified as LTL formulae is an idea of broad practical and theoretical interest. As said above, we explore in this paper the possibility of controlling the execution of systems by stating LTL properties that must be satisfied along the execution. In particular a finite trace variant of LTL is considered, proposed by Havelund and Roşu for monitoring Java programs [7], which at the end of a trace remains stationary in the last state.

The paper is structured as follows. Section 2 serves as a brief introduction to rewriting logic and Maude. Section 3 introduces the definition of strategies in Maude, and serves as a basis for the introduction of invariant-guided strategies in Section 4. Section 5 describes as an example the case of invariants expressed using LTL calculus. Finally, Section 6 draws some conclusions.

2 Rewriting Logic and Maude

Maude [3] is a high-level language and a high-performance interpreter and compiler in the OBJ [6] algebraic specification family that supports membership equational logic [11] and rewriting logic [10] specification and programming of systems.

The underlying equational logic chosen for Maude is membership equational logic [11]. Membership equational logic is a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : S$, stating that a term t has sort S . Such a logic extends order-sorted equational logic, and supports sorts, subsort relations, subsort polymorphic overloading of operators, and the definition of partial functions with equationally defined domains.

Rewriting logic is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. In rewriting logic, the state space of a distributed system is specified as an algebraic data type in terms of an equational specification (Σ, E) , where Σ is a signature of sorts (types) and operations, and E is a set of (conditional) equational axioms. The dynamics of a system in rewriting logic is then specified by rewrite *rules* of the form $t \rightarrow t'$, where t and t' are Σ -terms. These rules describe the local, concurrent transitions possible in the system, i.e. when a part of the system state fits the pattern t , then it can change to a new local state fitting pattern t' . Rules may be conditional, in which case the guards act as blocking pre-conditions, in the sense that a conditional rule can only be fired if the condition is satisfied.

In Maude, a system module defines a rewrite theory and it represents a system specification by defining the rules whose evaluation will produce the progressive transformation of the system state. Object-oriented systems are specified by

object-oriented modules in which classes and subclasses are declared. A class is declared with the syntax `class C |  $a_1:S_1, \dots, a_n:S_n$` , where C is the name of the class, a_i are attribute identifiers, and S_i are the sorts of the corresponding attributes. Objects of a class C are then record-like structures of the form `< O : C |  $a_1:v_1, \dots, a_n:v_n$  >`, where O is the name of the object, and v_i are the current values of its attributes. Objects can interact in a number of different ways, including message passing. Messages are declared in Maude in `msg` clauses, in which the syntax and arguments of the messages are defined.

The general form of a rewrite rule is

```

cr1 [r] :
  < O1 : C1 | atts1 > ... < On : Cn | attsn >
  M1 ... Mm
=> < Oi1 : C'i1 | atts'i1 > ... < Oik : C'ik | atts'ik >
    < Q1 : C''1 | atts''1 > ... < Qp : C''p | atts''p >
    M'1 ... M'q
if Cond .

```

where r is the rule label, $M_1 \dots M_m$ and $M'_1 \dots M'_q$ are messages, $O_1 \dots O_n$ and $Q_1 \dots Q_p$ are object identifiers, $C_1 \dots C_n$, $C'_{i_1} \dots C'_{i_k}$ and $C''_1 \dots C''_p$ are classes, $i_1 \dots i_k$ is a subset of $1 \dots n$, and *Cond* is a Boolean condition (the rule's *guard*). The result of applying such a rule is that: (a) messages $M_1 \dots M_m$ disappear, i.e., they are consumed; (b) the state, and possibly the classes of objects $O_{i_1} \dots O_{i_k}$ may change; (c) all the other objects O_j vanish; (d) new objects $Q_1 \dots Q_p$ are created; and (e) new messages $M'_1 \dots M'_q$ are created, i.e., they are sent. Rule labels and guards are optional.

For instance, the Maude module `DINING-PHILOSOPHERS` below specifies a variant of the well known problem of the hungry philosophers. The problem assumes five philosophers sitting around a table, on which five plates and five chopsticks are laid out. A philosopher can do two things, either think, or eat. When he thinks, a philosopher does not need the chopsticks; on the other hand, when thinking, he ends up being hungry. To eat, he needs the two chopsticks which are disposed on each side of his plate. Once he has finished eating, the philosopher releases the chopsticks and starts thinking, and then will be hungry again, etc. We assume a finite amount of spaghetti, so that the system becomes terminating.

Philosophers are modelled using a class with two attributes. The attribute `state` represents the state of the philosopher—which can be `thinking`, `hungry`, or `eating`—and the attribute `sticks` represents the number of chopsticks he holds. Moreover, a message `chopstick(N)` has been defined, indicating that the chopstick N is free. Philosophers and chopsticks are named with numbers from one to five, in such a way that the chopsticks besides the philosopher i are i and $i + 1$, or i and 1 if i is 5. Note the subsort declaration `Nat < Oid` making a natural number a valid object identifier. We also declare a class `Bowl`, which represents the bowl of spaghetti, with an attribute `amount` representing the amount of spaghetti left.

The system behavior is then defined by four rules, each one representing a local transition of the system. For example, the rule labelled as **grab** may be fired when a philosopher object I is hungry and it receives a message indicating that the chopstick J is free, being the chopstick J one of the chopsticks I can grab. As a result, the message is consumed, and the number of chopsticks grabbed by the philosopher is increased. The syntax for rules and conditional rules is, respectively, $\text{rl } [l] : t \Rightarrow t'$ and $\text{crl } [l] : t \Rightarrow t' \text{ if } c$, with l a rule label, t and t' terms, and c a rule condition.

Note that in Maude, those attributes of an object that are not relevant for an axiom do not need to be mentioned. Thus, attributes not appearing in the right-hand side of a rule will maintain their previous values unmodified.

```
(omod DINING-PHILOSOPHERS is
  protecting NAT .
  subsort Nat < Oid . *** Natural numbers are valid object identifiers
  sort Status .
  ops thinking hungry eating : -> Status [ctor] .
  op _can'use_ : Nat Nat -> Bool .

  class Philosopher | state : Status, sticks : Nat .
  class Bowl | amount : Nat .
  msg chopstick : Nat -> Msg .

  vars I J K B : Nat .
  eq I can use J = (I == J) or (s(I) == J) or (I == 5 and J == 1) .

  rl [hungry] :
    < I : Philosopher | state : thinking >
    => < I : Philosopher | state : hungry > .
  crl [grab] :
    < I : Philosopher | state : hungry, sticks : K > chopstick(J)
    => < I : Philosopher | sticks : K + 1 >
    if I can use J .
  crl [eat] :
    < I : Philosopher | state : hungry, sticks : 2 >
    < B : Bowl | amount : J >
    => < I : Philosopher | state : eating >
    < B : Bowl | amount : J + - 1 >
    if J > 0 .
  rl [full] :
    < I : Philosopher | state : eating >
    => < I : Philosopher | state : thinking, sticks : 0 >
    chopstick(I) chopstick((I rem 5) + 1) .
endom)
```

3 Execution strategies in Maude

System modules and object-oriented modules in Maude do not need to be Church-Rosser and terminating, therefore the system state may evolve in different direc-

tions depending on the order and the form in which we apply the rules describing such a system. Thanks to the reflective capabilities that Maude provides, we can define our own strategies, which in fact are defined using statements in a normal module.

Maude provides key metalevel functionality for metaprogramming and for writing execution strategies. In general, strategies are defined in extensions of the predefined module `META-LEVEL` by using predefined functions in it, like `metaReduce`, `metaApply`, `metaXapply`, etc. as building blocks. `META-LEVEL` also provides sorts `Term` and `Module`, so that the representations of a term T and of a module M are, respectively, a term $\overline{T}$ of sort `Term` and a term $\overline{M}$ of sort `Module`. Constants (resp. variables) are metarepresented as quoted identifiers that contain the name of the constant (resp. variable) and its type separated by a dot (resp. colon), e.g., `'true.Bool` (resp. `'B:Bool`). Then, a term is constructed in the usual way, by applying an operator symbol to a comma-separated list of terms. For example, the term $S \models \text{True}$ of sort `Bool` in the module `LTL-SATISFACTION` below is metarepresented as the term `'_|=_'S:State, 'True.Formula` of sort `Term`.

Of particular interest for our purposes are the partial functions `metaReduce` and `metaXapply`.¹ `metaReduce` takes a module $\overline{M}$ and a term $\overline{T}$, and returns the metarepresentation of the normal form of T in M , that is, the result of reducing T as much as possible using the equations in M . `metaXapply` takes as arguments a module $\overline{M}$, a term $\overline{T}$, and a rule label L , and returns the metarepresentation of the term resulting from applying the rule with label L in M on the term T .

To illustrate the general approach, and as a first step towards our final goal, let us suppose that we are interested in a strategy that rewrites a given term by applying on it all the rules in a given module, in any order. The strategy should just try to apply the rules one by one on the current term until it gets rewritten. Once a rule can be applied on it, the term resulting from such an application becomes the current term, and we start again. If none of the rules can be applied on a term, then it is returned as the result of the rewriting process. Such a strategy can be specified as follows:

```

op rew : Module Term -> Term .
op rewAux : Module Term ContStruct -> Term .

eq rew(M, T) = rewAux(M, T, cont(M)) .
ceq rewAux(M, T, C) = T if final(C) .
ceq rewAux(M, T, C)
  = if T' :: Term
    then rewAux(M, T', reset(C))
    else rewAux(M, T, next(C))
  fi
if L := getLabel(next(C)) /\ T' := metaXapply(M, T, L) .

```

¹ We have simplified the form of these functions for presentation purposes, since we do not need here their complete functionality. See [3] for the actual descriptions.

The operation **rew** takes two arguments: the (metarepresentation of) the module describing the system whose execution we wish to control, and the (metarepresentation of) the term representing the initial state of the system. **rewAux** takes three arguments: the module describing the system, the term being rewritten, and a continuation structure with the labels of the rules in the module, which allows us to iterate on the labels in some order. The strategy gives as result a term which cannot be further rewritten.

In the equations defining **rew** we assume a function **cont** that takes a module and returns a continuation structure for it. We also assume the following functions on the sort **ContStruct** of continuation structures: **final**, which returns a Boolean value indicating whether there are more labels in the structure; **reset**, which initializes the structure; **next**, which returns the structure with the next label set to be the next one; and **getLabel**, which returns the next label in the sequence. Note that we do not assume a concrete structure; depending on the particular structure used, and on the definition of these operations the order in which the labels are considered may be different.

A rewriting step $T \xrightarrow{L} T'$ is accomplished only if the rule labelled L is applicable on the term T , being T' the term returned by **metaXapply**(M , T , L). If the rule can be applied, then T' is a term (of sort **Term**), the membership assertion is evaluated to true, and a new rewrite step is considered, taking into account all the candidate rules; otherwise T' is an error term (of kind **[Term]**) and the next candidate rule is tried.

4 Using invariants to guide the system execution

Basically, an invariant is a property that a specification or program always requires to be true. However, we can distinguish between *deducible* or *external* invariants, which can be proved (or inferred) from the specifications they are defined on, and *constraining* or *internal* invariants, which form part of the specifications themselves (hence restricting the system behavior).

4.1 State-dependent satisfaction strategy

An interesting technique for checking invariants at runtime is *monitoring*, where we analyze whether the actual execution trace of the system satisfies a given property. Monitoring techniques study the states obtained during the system execution for detecting erroneous states, stopping the system if necessary, or trying to recover from it when possible. Instead of using an external monitor to verify a given system specification against an invariant, we propose using invariants as part of our specifications, making it *internal*. We suggest exploiting the possibility of defining execution strategies to *drive* the system execution in such a way that we can guarantee that every obtained state complies with the invariant, thus avoiding the execution of actions conducting the system to states not satisfying the invariant.

Now we will extend the strategy in Section 3 for defining a new strategy which guarantees a given invariant. We just need to check that the invariant is satisfied by the initial state and by every candidate to new state in a rewriting step. To implement this new strategy we assume a satisfaction predicate $_|_$ such that, given a state of the system S and an invariant I , then $S \models I$ evaluates to **true** or **false** depending on whether the state S satisfies the invariant I or not. The new strategy requires two additional parameters, namely (the metarepresentation of) the invariant predicate, and (the metarepresentation of) the module defining the satisfaction relation in the logic used for expressing such an invariant:

```

op rewInv : Module Module Term Term ~> Term .
op rewInvAux : Module Module Term Term ContStruct -> Term .

ceq rewInv(M, M', T, I)
  = rewInvAux(M, M', T, I, cont(M))
  if metaReduce(M', '_|=[T, I]) = 'true.Bool .
ceq rewInvAux(M, M', T, I, C) = T if final(C) .
ceq rewInvAux(M, M', T, I, C)
  = if T' :: Term and-then metaReduce(M', '_|=[T', I]) == 'true.Bool
    then rewInvAux(M, M', T', I, reset(C))
    else rewInvAux(M, M', T, I, next(C))
  fi
  if L := getLabel(next(C)) /\ T' := metaXapply(M, T, L) .

```

Now, the auxiliary function **rewInvAux** is invoked if the initial state satisfies the invariant. Notice that the operator **rewInv** is declared using $\sim\>$, meaning that if not reduced, it will return an error term of kind **[Term]**.² Moreover, the strategy takes a rewriting step only if the term can be rewritten using a particular rule and it yields to a next state which satisfies the invariant. An invariant I is checked by evaluating the expression $T' \models I$, for a given candidate transition $T \xrightarrow{L} T'$. Note however that the rewriting process takes place at the metalevel, and we use **metaReduce** for evaluating the satisfaction of the property.

Notice also that the rules describing the system can be written independently from the invariants applied to them, and the module specifying the system is independent of the logic in which the invariants are expressed, thus providing the right kind of independence and modularity between the system actions and the system invariants. In fact, the strategy is parameterized by the system to be executed (M), the invariant to be preserved (I) and the module defining the satisfaction relation (M'). This allows using different logics to express the invariant without affecting the strategy or the system to execute.

4.2 Trace-dependent satisfaction strategy

The strategy **rewInv** given in Section 4.1, although valid for the case of logics as propositional logic, where the satisfaction of a property can be decided looking at

² A kind is semantically interpreted as the set containing all the well-formed expressions in the sorts determining it, and also error expressions.

a particular state, does not work for other logics, as for example temporal logics. The satisfaction of LTL formulae cannot be decided considering particular states, but complete traces. For example, consider the invariant restriction $\Box P$ (always holds P). This invariant requires that any future state maintain the property P , and obviously this cannot be guaranteed by just considering the actual state. Since the execution of our specifications may terminate, we need to formalize what it means for a finite execution to satisfy a formula. Our approach to deal with LTL is similar to the one proposed by Havelund and Roşu in [7], based on the progressive transformation of the invariant restrictions when the system state evolves, possibly obtaining a new invariant when the system state changes. Thus, in the case of logics like LTL, in addition to a satisfaction relation $\models$, we will provide a transformation function $_ \{ _ \}$.

Given a state S and an LTL formula F , $F\{S\}$ returns a new formula F' . For example, a formula $\Diamond F$ (eventually F) will be transformed into $F \text{ or } \Diamond F$, that is, it has to be satisfied in the next state or eventually in the future. Notice that if F is not satisfied in the next state, $\Diamond F$ remains as an obligation to be satisfied. In case we reach a final state without satisfying the pending obligations, the states which originate such obligations (tentative states) become nonvalid states. We provide a strategy which backtracks in these situations.

Next, we show the strategy **rewBackInv**, which modifies the strategy **rewInv** by adding a backtracking mechanism. To implement backtracking it is necessary to save the execution path from the initial state until the actual one. This path consists of a sequence of nodes, each one containing the information necessary for continuing the execution from this point. Each node is represented as a tuple $\langle _ ; _ ; _ \rangle$ containing the continuation structure in a execution step, the term representing the state of the system, and the invariant to be satisfied. A path is then given as a sequence of nodes. When a backward step is necessary, we get the previous node in the sequence and we modify the actual state and the invariant, using the information it contains.

This new strategy takes a module **TR** as an additional parameter. Such a module is assumed to define the transformation function $_ \{ _ \}$ with which to transform the actual invariant after each computation step. Initially, the execution path consists of a single node which contains the initial state, the invariant it satisfies, and the continuation structure in a reset state.

The nonfinal states are handled by the last equation, which is similar to the one used in the **rewInv** strategy, but now handling the execution path and the transformation of the invariant predicate for the new state. Note how the invariant to be satisfied in the new state (I') is obtained by applying the transformation function to the actual state (T) and invariant (I). The **owise** attribute guarantees that this equation is not applied to final states.

In final states the satisfaction of the invariant has to be considered differently. For example, the invariant $\Diamond P$ is satisfied in any nonfinal state because, even if it is not satisfied now it may be satisfied in the future (it remains as a condition for the future, firing a backtracking mechanism if it does not occurs). However

the satisfaction of $\langle \rangle P$ in a final state requires the satisfaction of P in such a state.

First two equations for `rewBackInvAux` model what to do in a final state. The invariant is transformed taking into account that the current state is final (obtaining I''). Note that the transformation function is invoked after labelling the term representing the final state to be identified by the `final` operator. Then, it is checked whether the final state satisfies the transformed invariant (we assume the trace remains stationary in the last state). If it does, then it is a valid final state. Otherwise, if possible, the strategy backtracks (second equation). Notice that if the final state is not valid and there is no possibility of recovering a previous node (first equation), then an error term is returned.

```

op rewBackInv : Module Module Module Term Term -> Term .
op rewBackInvAux : Module Module Module Path -> Term .

ceq rewBackInv(M, ST, TR, T, I)
  = rewBackInvAux(M, ST, TR, < reset(cont(M)) ; T ; I >)
  if metaReduce(ST, '_|=_[T, I]) = 'true.Bool .
ceq rewBackInvAux(M, ST, TR, < C ; T ; I >)
  = T
  if final(C) /\ I'' := metaReduce(TR, '_{'[I, 'final[T]])
    /\ metaReduce(ST, '_|=_[T, I'']) == 'true .
ceq rewBackInvAux(M, ST, TR, < C ; T ; I > < C' ; T' ; I' > P)
  = if metaReduce(ST, '_|=_[T, I'']) == 'true
    then T
    else rewBackInvAux(M, ST, TR, < C' ; T' ; I' > P)
  fi
  if final(C) /\ I'' := metaReduce(TR, '_{'[I, 'final[T]]) .
ceq rewBackInvAux(M, ST, TR, < C ; T ; I > P)
  = if T' :: Term and-then metaReduce(ST, '_|=_[T', I']) == 'true
    then rewBackInvAux(M, ST, TR, < reset(C) ; T' ; I' > < C ; T ; I > P)
    else rewBackInvAux(M, ST, TR, < next(C) ; T ; I > P)
  fi
  if L := getLabel(next(C)) /\ T' := metaXapply(M, T, L)
    /\ I' := metaReduce(TR, '_{'[I, T]) [owise] .

```

5 Defining logics for driving the system execution: LTL

The logic in which the invariants are expressed is independent of the system to be executed. This would allow us to use one logic or another to express our invariants depending on our needs. If we want to use a specific logic to express the invariant predicates, we need to define the syntax of such a logic, a satisfaction relation, and a transformation operator for it. In the following sections, we study how to define these elements for linear time logic.

5.1 The syntax

The module `LTL-CALCULUS` defines LTL logic by defining the sorts `Proposition`, of atomic propositions, and `Formula`. Then, it declares the propositional and temporal operators. The nonconstructor connectives `[]_` (always) and `<>_` (eventually) are defined in terms of the basic constructor connectives `O_` (next), `_U_` (until), and `_R_` (release) by the appropriate equations. Assuming formulas `X` and `Y`, the operators have the following interpretation. The formula `[] X` holds if `X` holds in all time points, while `<> X` holds if `X` holds in some future time point. The formula `X U Y` (`X` until `Y`) holds if `Y` holds in some future time point, and until then `X` holds (we consider strict until). Finally, `O X` holds if `X` holds in the next time point. `_R_` is the dual of `_U_` (`X R Y = not((not X) U (not Y))`).

```
fmod LTL-CALCULUS is
  sort Proposition Formula .
  subsort Proposition < Formula .

  *** propositional operators
  ops True False : -> Formula .
  op _and_ : Formula Formula -> Formula [assoc comm prec 55] .
  op _xor_ : Formula Formula -> Formula [assoc comm prec 57] .
  op not_ : Formula -> Formula [prec 53] .
  op _implies_ : Formula Formula -> Formula [prec 61 gather (e E)] .
  vars A B C : Formula .
  eq True and A = A .
  eq False and A = False .
  eq A and A = A .
  eq False xor A = A .
  eq A xor A = False .
  eq A and (B xor C) = A and B xor A and C .
  eq not A = A xor True .
  eq A implies B = not(A xor A and B) .

  *** primitive LTL operators
  op O_ : Formula -> Formula [ctor prec 53 format (r o d)] .
  op _U_ : Formula Formula -> Formula [ctor prec 63 format (d r o d)] .
  op _R_ : Formula Formula -> Formula [ctor prec 63 format (d r o d)] .

  *** defined LTL operators
  op <>_ : Formula -> Formula [ctor prec 53 format (r o d)] .
  op []_ : Formula -> Formula [ctor prec 53 format (r d o d)] .
  eq <> A = True U A .
  eq [] A = False R A .
endfm
```

The module introduces the sort `Formula` of well-formed propositional formulae, with two designated formulae, namely `True` and `False`, with the obvious meaning. The sort `Proposition`, corresponding to the set of atomic propositions, is declared as subsort of `Formula`. `Proposition` is by the moment left

unspecified; we shall see below how such atomic propositions are defined for a given system module.

5.2 The satisfaction relation

Since the execution of our specifications may terminate, we need to formalize what it means for a finite execution to satisfy a formula. This is why we consider a finite LTL. Notice that finite LTL can behave quite differently from standard LTL. For example, the formula $\langle \rangle ([\] F \text{ or } [\] \text{ not } F)$ is not valid in LTL, but valid in finite trace LTL (the last state in the trace either satisfies F or it does not). Note that there are formulae which are satisfiable in LTL and not satisfiable in finite LTL, such as the negation of the above.

To handle the finite variant of LTL, we have to consider the final state as a special situation. Therefore, the satisfaction of each temporal logic connective is transformed in a different way depending on whether the considered state is final or not. Note that we define **State*** as a supersort of **State** and final states are labelled using the **final** operator. This allows us to distinguish final and nonfinal states when necessary.

```
(fmod LTL-SATISFACTION is
  protecting LTL-CALCULUS .
  sorts State State* .
  subsort State < State* .

  op final : State -> State* .
  op _|=_ : State* Formula ~> Bool .

  var S : State .
  var S* : State* .
  vars F F' : Formula .

  eq S* |= (F and F') = (S* |= F) and (S* |= F') .
  eq S* |= (F xor F') = (S* |= F) xor (S* |= F') .
  eq S* |= not F = not (S* |= F) .
  eq S* |= True = true .
  eq S* |= False = false .
  eq S |= 0 F = true .
  eq S* |= 0 F = S* |= F [owise] .
  eq S |= F U F' = (S |= F) or (S |= F') .
  eq S* |= F U F' = S* |= F' [owise] .
  eq S |= F R F' = S |= F' .
  eq S* |= F R F' = (S* |= F) and (S* |= F') [owise] .
endfm)
```

As said above, the satisfaction relation $_|=_$ is a Boolean predicate such that, given a state (the sort **State** will be defined for each particular problem) and a formula, evaluates to **true** or **false** depending on whether the given state satisfies such a formula or not. Notice that $_|=_$ takes a formula as second argument and returns a Boolean value, being **Bool** a predefined sort in Maude.

5.3 The transformation operator

The module `LTL-TRANSFORMER` defines the transformation operator `_{}_`, which takes a formula and a state and yields another formula. Note that as we did for the satisfaction relation, we have specified a different transformation for each temporal connective depending on whether the considered state is final or not.

```
(fmod LTL-TRANSFORMER is
  including LTL-CALCULUS .

  op _'{}_ : Formula State -> Formula .

  var S : State .
  var S* : State* .
  var P : Proposition .
  vars F F' : Formula .

  eq True { S* } = True .
  eq (not F) { S* } = not (F { S* }) .
  eq (F or F') { S* } = (F { S* }) or (F' { S* }) .
  eq (F and F') { S* } = (F { S* }) and (F' { S* }) .
  eq P { S* } = if S* |= P then True else False fi .
  eq (O F) { S } = F .
  eq (O F) { S* } = F { S* } [owise] .
  eq (F U F') { S } = F' { S } or (F { S } and (F U F')) .
  eq (F U F') { S* } = F' { S* } [owise] .
  eq (F R F') { S } = F' { S } and (F { S } or (F R F')) .
  eq (F R F') { S* } = F' { S* } and F { S* } [owise] .
endfm)
```

5.4 The definition of atomic propositions

If we want to use LTL to define invariant predicates for a given problem, we need to define the atomic propositions to be used to express the proposed invariant predicate, and the satisfaction relation on them. The `PHILOSOPHERS-LTL-PREDS` module defines the sorts left unspecified before—`State` and `Proposition`—. We define two propositions stating, respectively, if a given philosopher is **eating** or **hungry**, and a third one `chopstick(P, N)` which holds if the philosopher *P* has *N* chopsticks. Furthermore, for this example, a `State` is a `Configuration`. Our intention is to use these atomic propositions to guarantee that the philosophers will not get into deadlock (if each philosopher has a chopstick none of them will be able to eat), and some kind of fairness (if a philosopher gets hungry he/she will eventually eat).

```
(omod PHILOSOPHERS-LTL-PREDS is
  protecting DINING-PHILOSOPHERS .
  including LTL-SATISFACTION .

  subsort Configuration < State .
```

```

op eating : Oid -> Proposition .
op hungry : Oid -> Proposition .
op chopstick : Oid Nat -> Proposition .

vars I N M : Nat .
var C : Configuration .
var S : Status .
eq < I : Philosopher | state : S > C |= eating(I) = S == eating .
eq < I : Philosopher | state : S > C |= hungry(I) = S == hungry .
eq < I : Philosopher | sticks : N > C |= chopstick(I, M) = N == M .
endom)

```

Once we have defined the atomic propositions, they may be used to define the intended invariant for guiding the execution. The deadlock-free and fairness constraints may be expressed as follows:

```

deadlock-free = [] not(chopstick(1, 1) and chopstick(2, 1)
                        and chopstick(3, 1) and chopstick(4, 1)
                        and chopstick(5, 1))

fairness = [] (hungry(1) -> <> eating(1))
              and [] (hungry(2) implies <> eating(2))
              and [] (hungry(3) implies <> eating(3))
              and [] (hungry(4) implies <> eating(4))
              and [] (hungry(5) implies <> eating(5))

```

Finally, we can execute our example using as parameters the invariant, the module describing the logic we have used, and the module describing how to transform the invariant after each execution step. Let us denote $\bar{t}$ and $\bar{M}$ the metarepresentations of a term t and a module M .

```

red rewBackInv(DINING-PHILOSOPHERS,
               PHILOSOPHERS-LTL-PREDS,
               LTL-TRANSFORMER,
               initial-state,
               deadlock-free and fairness) .

```

With this command, we execute the system by allowing the nondeterministic application of the rules in the module, but with the guarantee that the invariant is satisfied by all the (non-tentative) states in the trace.

6 Concluding Remarks

We have proposed generic invariant-driven strategies, which control the execution of systems by guaranteeing that the given invariants are satisfied. Our strategies are generic in the sense that they are parameterized by the system whose execution they control, by the logic in which the invariants are expressed, and by the invariants themselves. This parameterization, together with the level

of modularization of the approach, allows improving quality factors such as extensibility, understandability, usability, or maintainability. We have illustrated its use with invariants expressed in (finite future time) linear time temporal logic. However, the good properties of Maude as a logical and semantic framework [8], in which many different logics and formalisms can be expressed and executed, allow us to use other logics as parameters of our strategies.

References

1. P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. ELAN v 3.3 user manual, Third edition. Technical report, INRIA Lorraine & LORIA, Nancy, France, Dec. 1998.
2. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, (285):187–243, 2002.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 manual. Available in <http://maude.cs.uiuc.edu/>, June 2003.
4. M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing, J. Woodcock, and J. Davies, eds., *FM'99 - Formal Methods (Vol. II)*, vol. 1709 of *LNCS*, pages 1684–1704. Springer, 1999.
5. F. Durán, M. Roldán and A. Vallecillo. Invariant-driven strategies for maude. In S. Antoy and Y. Toyama, eds., *4th Intl. Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004. Also available at <http://lcc.uma.es/~duran>.
6. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, eds., *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000.
7. K. Havelund and G. Roşu. Rewriting-based techniques for runtime verification. To appear in *Journal of Automated Software Engineering*. Also available at <http://fsl.cs.uiuc.edu/~grosu>.
8. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenther, eds., *Handbook of Philosophical Logic*, vol. 9, pages 1–87. Kluwer Academic Publishers, second edition, 2002.
9. T. McComb and G. Smith. Animation of Object-Z specifications using a Z animator. In *First Intl. Conference on Software Engineering and Formal Methods (SEFM'03)*, 2003.
10. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
11. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, ed., *Recent Trends in Algebraic Development Techniques*, vol. 1376 of *LNCS*, pages 18–61. Springer, 1998.
12. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
13. M. Richters and M. Gogolla. OCL: Syntax, semantics and tools. In T. Clark and J. Warmer, eds., *Advances in Object Modelling with the OCL*, vol. 2263 of *LNCS*, pages 43–69. Springer, 2001.
14. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.

A First-Order Functional Language for Reasoning about Heap Consumption*

Ricardo Peña and Clara Segura

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain

e-mail: {csegura,ricardo}@sip.ucm.es

Abstract. Most functional languages abstract the programmer from the details of the memory management done by his/her program at run time. This model is acceptable in most situations, being its main advantage that programmers are not bored, and programs are not obscured, with low level details about memory management. But in some other contexts like real-time programs or safety critical applications, time delays due to garbage collection or abortion due to memory exhaustion are not acceptable. On the other hand, many imperative languages offer low level mechanisms to allocate and free heap memory which give the programmer a complete control over memory usage, but without any safety requirement. We have chosen a semi-explicit approach to memory control by defining a functional language in which the programmer cooperates with the memory management system by providing some information about the intended use of data structures. The final aim of the system is to reason about the memory consumed by a program.

1 Introduction

Most functional languages abstract the programmer from the details of the memory management done by his/her program at run time. The runtime support system usually allocates fresh heap memory while program expressions are being evaluated. This underlying allocation mechanism continues to work as long as there is enough free memory available. Should the memory be exhausted, the garbage collector will come into the scene and decide which part of the heap is still being used and which one is dead. This normally implies the suspension of program execution for some time, after which, hopefully, some free memory has been recovered and the program can proceed with the previous execution-allocation scheme. Sometimes, not enough free memory has been recovered and the program simply aborts.

This model is acceptable in most situations, being its main advantage that programmers are not bored, and programs are not obscured, with low level details about memory management. But, in some other contexts, this scheme may not be acceptable for one or both of the following reasons:

* Work partially supported by the Spanish project TIN2004-07943-C04.

1. The time delay introduced by garbage collection prevents the program from providing an answer in a required reaction time. This is the case of real-time or highly interactive programs.
2. Memory exhaustion abortion may provoke unacceptable personal or economic damage to users. This is the case of safety-critical applications.

On the other hand, many imperative languages offer low level mechanisms to allocate and free heap memory, which the programmer may use in order to dynamically create and destroy pointer based data structures. These mechanisms give the programmer a complete control over memory usage but are very error prone. Well known problems that may arise when using a programmer controlled memory management are dangling references, undesired sharing between data structures with complex side effects as a consequence, and polluting memory with garbage.

Curiously, neither the completely implicit approach of functional languages, nor the completely explicit one of many imperative languages provide a clear logic the programmer may rely on to reason about the memory usage of programs. In particular, it is difficult to anticipate, or to prove, an upper bound on the amount of memory that guarantees non abortion due to memory exhaustion.

In this paper we have chosen a semi-explicit approach to memory control by defining a functional language in which the programmer cooperates with the memory management system by providing some information about the intended use of data structures. For instance, he/she may indicate that some particular data structure will not be needed in the future and that, as a consequence, it may be safely destroyed by the runtime system and its memory recovered. The language also allows to control the degree of sharing between different data structures. But, more important than the concrete facilities provided by the language, is the definition of a type system which guarantees that all this can be done in a safe way. In particular, it guarantees that a destroyed object is never referenced again and that garbage is always recovered. An ill-constructed program will be rejected by the type system. For lack of space we do not present here the typing rules but we will show the types of the functions in all the examples in order to indicate that all of them typecheck. The interested reader can find the rules explained in [5].

The benefits of the proposed approach are two:

- A garbage collector is not needed. Allocation and destruction of data structures are done as execution proceeds. Moreover, the cost of destruction can be anticipated and incorporated to the cost analysis of the algorithm, an issue that is usually not possible when garbage collectors are present.
- It is possible to reason about the memory consumed by the program. As we will see below in the paper, it is even possible to prove that the algorithm runs in constant heap space, independently of input size.

For the moment, these proofs are manually conducted but, in the longer term, we foresee that a more sophisticated type system than the one previously mentioned

would be able to compute, for many algorithms, an upper bound on the memory needed by the program.

The structure of the paper is as follows: In Section 2, a survey of the main principles and terminology underlying the language design is done. In Section 3, the syntax of the language and an informal explanation of its semantics are given. Some programming examples are given to illustrate its expressiveness. Section 4 is devoted to a more complete case study, *mergesort*, where its (constant) memory consumption is proved. Section 5 surveys the work that still remains to be done and Section 6 compares our work with three other approaches in the area.

2 Language Design Principles

One possible way of presenting the language is just “as a fact”: This is the syntax, this is the semantics and these are the programs you can construct with it. Then, the reader will judge its advantages and limitations. We will follow this approach in the following sections but, before embarking on the details, we present in this section the rationale which has led us to the current design. It has the form of a set of definitions and a set of axioms.

Definition 1. A **region** is a contiguous memory area in the heap where data structures can be constructed, read, or destroyed. It has a size known at run time and it is allocated and liberated as a whole, in constant time.

Definition 2. A **cell** is a small memory space, big enough to hold a data constructor. In implementation terms, a cell contains the mark (or code pointer) of the constructor, and a representation of the free variables to which the constructor is applied. These may consist, either of basic values, or of pointers to non-basic values.

Definition 3. A **data structure**, in what follows *DS*, is the set of cells obtained by starting at one cell considered as the root, and taking the transitive closure of the relation $C_1 \rightarrow C_2$, where C_1 and C_2 are cells of the same type T , and in C_1 there is a pointer to C_2 .

That means that, for instance in a list of type $[[a]]$, we are considering as a DS all the cells belonging to the *outermost* list, but not those belonging to the individual innermost lists. Each one of the latter constitute a separate DS.

Axiom 1 A DS completely resides in one region.

The rationale behind this decision is that, when a region dies, all the data structures in the region also die. Should we allow to split a DS into several regions, then we would obtain partially destroyed data structures. This axiom imposes an obligation on data constructors: Its recursive positions should only admit parameters belonging to the same region, and the resulting cell must also be constructed in that region.

Axiom 2 *Two DSs can be one of them part of the other, or they can share a third DS.*

By applying Axiom 1, all the three must reside in a single region. Of course, we want to have in our language as much sharing as possible, because this is a key for getting efficiency. Consider the difference in cost between inserting in a binary search tree, by sharing in the output the non-modified parts of the input, with creating a complete binary tree from scratch in the output. In the limit, conventional functional programming can be recovered in our system by having a single region in the whole program, and a potential sharing between all DSs.

Axiom 3 *The basic values —integers, booleans, etc.— do not allocate cells in regions. They live inside the cells of DSs, or in the stack.*

Axiom 4 *A function of n parameters can access, at most, to $n + 2$ regions.*

- *It can access the DSs of its n parameters, each one residing in a possibly different region.*
- *Every function returning a DS as a result, must construct it in its **output region**. There is at most one output region per function. Delivering this region identifier is the responsibility of the caller. The function receives the output region as an additional parameter.*
- *Every function has an optional **working region** where it may create intermediate DSs. The working region has the same lifetime as the function: It is allocated on each invocation and freed at function termination.*

The rationale behind this design is that, if we force functions to leave their result in an output region belonging to the caller, then the intermediate results computed by the function in order to produce its result may safely be deleted when the function terminates. In this way, heap working regions are stacked as nested functions are called, and it is easy to reason about the maximum heap needed by the program in order to run it safely.

Axiom 5 *If a parameter of a function is a DS, it can be destroyed by the function. We allow functions to behave in this way if the function programmer chooses to do so. We will say that the parameter is **condemned** because this capability depends on the function definition, not on its use. That means that every use of certain functions will destroy certain parameters. Destruction means that the caller can safely assume that all the cells occupied by the condemned DS are recovered and reused by the runtime system.*

Axiom 6 *The capabilities a function has on its accessible DSs and regions are:*

- *A function may only read a DS which is a read-only parameter.*
- *A function may read (before destroying it), and must destroy, a DS which is a condemned parameter.*
- *A function may construct, read, or destruct DSs, in either its output or its working region.*

$prog$	$\rightarrow dec_1; \dots; dec_n; expr$	
dec	$\rightarrow f \overline{x_i^n} r = expr$	{single-recursive, polymorphic function}
	$ f \overline{x_i^n} = expr$	
$expr$	$\rightarrow a$	{atom}
	$ (f \overline{a_i^n})@r$	{function application}
	$ (f \overline{a_i^n})$	
	$ (C \overline{a_i^n})@r$	{constructor application}
	$ \text{let } x_1 = expr_1 \text{ in } expr$	{non-recursive, monomorphic}
	$ \text{case } x \text{ of } \overline{alt_i^n}$	{read-only case}
	$ \text{case! } x \text{ of } \overline{alt_i^n}$	{destructive case}
a	$\rightarrow c$	{literal}
	$ x$	{variable}
	$ x@r$	{copy}
	$ x!$	{reuse}
alt	$\rightarrow C \overline{x_i^n} \rightarrow expr$	

Fig. 1. First-order functional language *Safe*

Having destroyable parameters is an abstract way of telling the system that a DS will no longer be used. Condemned parameters are reflected in the function type so that users know this feature just by looking at the type. The destroyed DSs must reside on a region on which the caller has destruction capabilities. A last aspect is guaranteeing that a destroyed DS is never used again. Using a linear type system, we have been able to typecheck these properties [5].

3 The Language itself

The syntax of *Safe*, the language whose design we have described in the previous section, is shown in Figure 1. This is a first-order eager functional language where sharing is enforced by using variables in function and constructor applications. A program *prog* is a sequence of single-recursive polymorphic function definitions followed by a main expression *expr* whose value is the result of the program. Each function definition may only call to previously defined functions and the main expression may call to any of them. Function definitions returning a DS will have an additional parameter *r*, which is the output region where the resulting DS is to be constructed. In the right hand side expression only *r* and its own working region, called *self*, may be used.

Polymorphic algebraic data types definitions are also allowed although, in order to simplify things, we will assume they are defined separately through **data** declarations. In all the examples given in this paper we will use lists [*a*] and binary trees *Tree a*, with constructors [] and : for lists, and *Empty* and *Node* for trees.

The program expressions include the usual: Variables, function and constructor applications, and also *let* and *case* expressions; but there are some additional expressions that we explain carefully.

$$\begin{aligned}
& revF :: \forall a, \rho_1, \rho_2. [a]@ \rho_1 \rightarrow \rho_2 \rightarrow [a]@ \rho_2 \\
& revF \ xs \ r = (revauxF \ xs \ []@r)@r \\
& revauxF :: \forall a, \rho_1, \rho_2. [a]@ \rho_1 \rightarrow [a]@ \rho_2 \rightarrow \rho_2 \rightarrow [a]@ \rho_2 \\
& revauxF \ xs \ ys \ r = \mathbf{case} \ x \mathbf{of} \\
& \quad [] \rightarrow ys \\
& \quad x : xx \rightarrow (revauxF \ xx \ (x : ys)@r)@r \\
\\
& revD :: \forall a, \rho_1, \rho_2. [a]!@ \rho_1 \rightarrow \rho_2 \rightarrow [a]@ \rho_2 \\
& revD \ x \ r = (revauxD \ x \ []@r)@r \\
& revauxD :: \forall a, \rho_1, \rho_2. [a]!@ \rho_1 \rightarrow [a]@ \rho_2 \rightarrow \rho_2 \rightarrow [a]@ \rho_2 \\
& revauxD \ x \ ys \ r = \mathbf{case!} \ x \mathbf{of} \\
& \quad [] \rightarrow ys \\
& \quad x : xx \rightarrow (revauxD \ xx \ (x : ys)@r)@r
\end{aligned}$$
Fig. 2. Functional and destructive list inversion

If x is a DS, only when x does not live in r the expression $x@r$ represents a copy of the DS accessed from x . Otherwise they share their contents. For example, if x is a list living in $r' \neq r$, $x@r$ is a new list with the same structure as x sharing its elements with x . This expression is useful when we want to work on a structure without modifying it (as an example, see function *msortN* at the end of the following section).

The expression $x!$ means that we want to reuse the DS to which x points but not through the name x . This is useful when we do not want to destroy completely a condemned parameter but we wish to reuse part of it, see Figure 3. We could make a copy allowing x to be destroyed but this would be less efficient.

In function application we have a special syntax $@r$ for including the additional region parameter. We use the same syntax to express that a constructor application is allocated in region r .

The (monomorphic) **let** expression allows to define an intermediate result. There are two variants of the **case** expression. The usual variant just consults the variable in the discriminant. The **case!** variant indicates that the variable x in the discriminant is condemned. In such case the outer constructor of x is disposed after the pattern matching, so x is not accesible any more. The recursive substructures may in the following be explicitly destroyed via another **case!** or reused via $!$. A condemned variable may be read but once we have destroyed it or reused its content in another structure we should not try to access it again. This is what the type system in [5] guarantees.

We show now with several examples how to use the language facilities. In these examples we show also the types that the functions have in the type system defined in [5]. Briefly ρ is a region type, $T \bar{s}_i^k@ \rho$ is the type of a DS residing in a region of type ρ , and $T \bar{s}_i^k!@ \rho$ is the type of a condemned argument.

The first example is the function that reverses a list. We show two versions of such function in Figure 2. In both of them we use the usual auxiliary function with an accumulator parameter.

```

insertD :: ∀a, ρ. a → Tree a!@ρ → ρ → Tree a@ρ
insertD x t r = case! t of
  Empty → (Node Empty@r x Empty@r)@r
  Node i y d →
    let c = compare x y
    in case c of
      LT → (Node (insertD x i)@r y d!)@r
      EQ → (Node i! y d!)@r
      GT → (Node i! y (insertD x d)@r)@r

```

Fig. 3. Destructive insertion with reuse

The first version, *revF*, is the functional version where the original list survives. Notice that the only difference with respect to the function that a functional programmer would write is the output region parameter *r*. The accumulator parameter is constructed in the output region as at the end it contains the result of the function. The second version, *revD*, is a destructive version where the original list is lost. Notice that the only difference with the previous one is that we use **case!** over the original list. The recursive application of the function destroys it completely. Those who call *revD* should know that the argument is lost in the inversion process, and should not try to use it any more.

The next example illustrates the reuse of a condemned structure. It is the function that inserts an element in a binary search tree. Here we only show in Figure 3 the destructive version where the original tree is partially destroyed. Everything but the path from the root to the inserted element is reused to build the new tree but these parts may not be accessed any more from the original tree. Notice that when the inserted element is already in the tree (*EQ* branch) we have to rebuild the tree we have just destroyed. The functional version is obtained by removing the **!** annotations and returning *t* in the *EQ* branch, as it has not been destroyed.

4 A Mergesort in Constant Heap Space

In this section we show a more involved example, a *mergesort* version that does not need additional heap space. First, we define the usual auxiliary functions to split the input list and merge two ordered lists in a single ordered list. In Figure 4 we show a destructive version of the splitting function. As in the previous examples, there are small differences with a purely functional version. In the base case ($n = 0$) we reuse the list in the output; in the recursive case we use a **case!** over the argument list. We also have to add **@r** where necessary.

In Figures 4 and 6 we use pair patterns in *let* bindings as an abbreviation of longer (typecheckable) expressions.

In Figure 5 we show the destructive version of the merging function. In the recursive calls to *mergeD* one of the parameters is one of the original lists. But we may not reference the original list as it has been condemned by a **case!**, so

$$\begin{aligned}
\text{splitD} &:: \forall a, \rho. \text{Int} \rightarrow [a]!@ \rho \rightarrow \rho \rightarrow ([a]@ \rho, [a]@ \rho)@ \rho \\
\text{splitD } n \text{ } xs \text{ } r &= \text{case } n \text{ of} \\
&\quad 0 \rightarrow ([]@r, xs!)@r \\
&\quad \text{succ}(n') \rightarrow \text{case! } xs \text{ of} \\
&\quad \quad [] \rightarrow ([]@r, []@r)@r \\
&\quad \quad y : ys \rightarrow \text{let } (ys_1, ys_2) = (\text{splitD } n' \text{ } ys)@r \\
&\quad \quad \quad \text{in } ((y : ys_1)@r, ys_2)@r
\end{aligned}$$

Fig. 4. Destructive partition of a list

$$\begin{aligned}
\text{mergeD} &:: \forall a, \rho. [a]!@ \rho \rightarrow [a]!@ \rho \rightarrow \rho \rightarrow [a]@ \rho \\
\text{mergeD } xs \text{ } ys \text{ } r &= \text{case! } xs \text{ of} \\
&\quad [] \rightarrow ys! \\
&\quad x : xx \rightarrow \text{case! } ys \text{ of} \\
&\quad \quad [] \rightarrow (x : xx!)@r \\
&\quad \quad y : yy \rightarrow \text{let } c = x \leq y \\
&\quad \quad \quad \text{in case } c \text{ of} \\
&\quad \quad \quad \quad T \rightarrow \text{let } ys' = (y : yy!)@r \text{ in} \\
&\quad \quad \quad \quad \quad \text{let } rs = (\text{mergeD } xx \text{ } ys')@r \text{ in} \\
&\quad \quad \quad \quad \quad \quad (x : rs)@r \\
&\quad \quad \quad \quad F \rightarrow \text{let } xs' = (x : xx!)@r \text{ in} \\
&\quad \quad \quad \quad \quad \text{let } rs = (\text{mergeD } xs' \text{ } yy)@r \text{ in} \\
&\quad \quad \quad \quad \quad \quad (y : rs)@r
\end{aligned}$$

Fig. 5. Destructive ordered merge of two lists

we need to rebuild the original list by reusing its components. This is the only detail we have to care about.

Finally, in Figure 6 we show the destructive mergesort *msortD*, that uses the previously defined functions. Both the input list *xs* and the intermediate results *xs*₁, *xs*₂, *xs*'₁ and *xs*'₂ are either destroyed or reused into the result. This will allow us to conclude that this function consumes a constant additional heap space. We can reason by induction on the length of the argument list:

- In the two base cases we reuse the argument list. This means that no additional heap space is needed.
- In the recursive case, we need to prove first that functions *splitD* and *mergeD* also consume constant heap space. This is easy as well, but we do not show the reasoning here.

We first call function *splitD* over the argument list *xs*. This function destroys *xs* and builds a pair of lists *xs*₁ and *xs*₂ such that $\text{length}(xs_1) + \text{length}(xs_2) = \text{length}(xs)$ (this can be easily proved by induction). So the only additional heap space needed is that occupied by a pair constructor. This is immediately disposed in order to extract its components.

$$\begin{aligned}
&msortD :: \forall a, \rho. [a]! @ \rho \rightarrow \rho \rightarrow [a] @ \rho \\
&msortD \ xs \ r = \mathbf{let} \ n = \mathit{length} \ xs \ \mathbf{in} \\
&\quad \mathbf{case} \ n \ \mathbf{of} \\
&\quad \quad 0 \rightarrow xs! \\
&\quad \quad 1 \rightarrow xs! \\
&\quad \quad n' \rightarrow \mathbf{let} \ n_2 = n \ 'div' \ 2 \ \mathbf{in} \\
&\quad \quad \quad \mathbf{let} \ (xs_1, xs_2) = (\mathit{splitD} \ n_2 \ xs) @ r \ \mathbf{in} \\
&\quad \quad \quad \mathbf{let} \ xs'_1 = (msortD \ xs_1) @ r \ \mathbf{in} \\
&\quad \quad \quad \mathbf{let} \ xs'_2 = (msortD \ xs_2) @ r \ \mathbf{in} \\
&\quad \quad \quad (\mathit{mergeD} \ xs'_1 \ xs'_2) @ r
\end{aligned}$$

Fig. 6. Destructive mergesort

Then two recursive calls are done which, by induction hypothesis, consume constant additional heap space. The resulting lists, xs'_1 and xs'_2 , are of the same length as the original ones, which can also be proved by induction. Finally we apply the function mergeD over xs'_1 and xs'_2 . This function destroys its arguments lists and produce a single list whose length is $\mathit{length}(xs'_1) + \mathit{length}(xs'_2)$, so no additional heap space is consumed.

For the moment our reasoning about heap consumption is done by induction but our final aim is to develop a type based analysis that automatically checks, or better infers, such consumption.

We can also write a version preserving the original argument:

$$msortN \ xs \ r = \mathbf{let} \ xs' = xs @ r \ \mathbf{in} \ (\mathit{msortD} \ xs') @ r$$

5 Towards a Type System for Full Space Analysis

In this paper we present a first-order functional language with facilities to control memory management. In [5] we have presented a type system to do this in a safe way. Here we have described the intended language semantics informally, but it is our aim to define formally an operational semantics, and prove that the type system is correct with respect to it. We would also like to write correct (and if possible complete) type checking and type inference algorithms for it.

The final aim of this system is to reason about heap consumption. We have done this here for an example by induction, but our intention is to define a type based analysis using sized types [3] that not only guarantees safe programming but also allows to automatically reason about memory consumption. A key aspect of this reasoning is that, as we have previously said, the implementation of the system must guarantee that a condemned structure is completely disposed. A type checking and/or type inference algorithm would be the core of the system. This could be used in first-order eager languages like Hume [4], which is intended to program systems with limited resources and safety critical applications. However, an extension to higher-order would be the following step in order to incorporate the system to a full functional language.

6 Related Work

Several approaches have been taken to memory management, some of which have inspired our work. Tofte and Talpin [6] introduced in KIT-ML the use of nested regions. Like ours, they are memory areas where data structures can be constructed, and they are allocated and deallocated as a whole. However, in our system regions are associated to functions, which have input, output and working regions with several use restrictions. Another difference is that our regions have a bounded size, that we intend to calculate with a sized type system. Additionally, although the working region is deallocated as a whole, we have an additional mechanism that allows us to destroy selectively DSs in the working or in the input regions.

Hughes and Pareto [2] incorporate in Embedded-ML also the concept of region to their sized types system so that heap and stack consumption can be checked. In this case, the size of regions is bounded. Our main differences to them are again the region-function association and the explicit dispose of structures. We think that their sized types system is a good starting point for us.

More recently, Hofmann and Jost [1] have developed in the framework of proof carrying code a type system to infer heap consumption. Theirs is also a first-order eager functional language with constructs similar to our **case!**. They associate to each function the space required by its execution and the remaining space after it. They also use a linear type system to safely use destructive facilities. Unlike us they do not use the concept of nested regions where DSs are allocated, so that sharing is not controlled in the same way.

References

1. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 185–197. ACM Press, 2003.
2. R. J. M. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space; Towards Embedded ML Programming. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, ICFP'99*, ACM Sigplan Notices, pages 70–81, Paris, France, September 1999. ACM Press.
3. R. J. M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT*, pages 410–423, 1996.
4. G. Michaelson. The Dynamic Properties of Hume: A Functionally-Based Concurrent Language with Bounded Time and Space Behaviour. In *Selected Papers of the 12th International Workshop on Implementation of Functional Languages, IFL'00*, volume 2011 of *LNCS*, pages 122–139. Springer-Verlag, 2001.
5. R. Peña and C. Segura. A First-Order Functional Language for Reasoning about Heap Consumption. In *Proceedings of the 16th International Workshop on Implementation of Functional Languages, IFL'04*, pages 64–80. 2004.
6. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

Tagged systems: a framework for the specification of history-dependent properties

Fernando Rosa Velardo, David de Frutos-Escrig, and Clara Segura Díaz

Dpto. de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
{fernandorosa, defrutos, csegura}@sip.ucm.es

Abstract. We propose a general method for the treatment of history-dependent runtime errors. When one has to control this kind of errors, a tagged version of the language is usually defined, in which tags capture only the necessary information of the history of processes. We will characterize such tagged languages as being quotients of the reachability tree defined by the computations of the original language. From this fact we can conclude that the property characterized by each tagged language is indeed a property of the original one. In this way, we can work in a common framework, instead of defining an ad hoc semantics for each property. In particular, we could still use the analysis machinery existing in the calculus in order to prove that or other related properties. We have applied this methodology to the study of resource access control in a distributed π -calculus, called $D\pi$. In particular, we have proved that the tagged version of $D\pi$ is indeed a *tagging* according to our definition.

1 Introduction

Several frameworks for the definition of security properties can be found in the literature [7, 9, 3, 4, 1]. In order to specify a property sometimes it is enough to detect some erroneous states of the analyzed systems. Some of those erroneous states can be statically identified, regardless the way in which the error-prone state has been reached. An example of such a security property is *confidentiality*, in which the erroneous states are those where a secret piece of data is being exhibited.¹

On the other hand, some security properties cannot be proved to hold by mere inspection of the current state. This is the case of history-dependent properties. An example of scenario in which this happens is when in order to access a resource, a process needs to have been given prior permission to use it [7].

In the former case, all that is needed to ensure that no error can occur is to identify those erroneous states, for example by means of a suitable type system, in such a way that they will not be typeable. Then, if the original system is typeable and we have subject reduction for the typed semantics, we can guarantee that no error can ever occur.

¹ Clearly by such a simple property we could not cover implicit flows of information.

However, in the latter case, this is not possible, as errors are not a property of a single state, but of the whole computation. The most natural way of performing the necessary trace analysis is by somehow remembering along the computation the information that is needed for the characterization of the considered kind of error, tagging the processes with this information. In the existing papers on the subject, such as [7], the corresponding enriched semantics is defined in an ad hoc way, not relating it in principle with the original semantics, thus making necessary specific non-reusable proofs for the corresponding relation between the original and the enriched semantics.

We propose in this paper a systematic methodology for the treatment of these errors, that begins by defining a tagged language and a unary relation on tagged processes. Then we should prove that the defined tagged language is indeed a *tagging* of the original one, so that we could again use a type-based analysis to guarantee that no erroneous tagged state is ever reached along the computation.

The remainder of the paper is organized as follows: in Section 2 we define tagged transition systems; in Section 3 we define quotient transition systems and prove that they are essentially tagged transition systems. Section 4 shows simple ways of deriving tagged languages; in Section 5 we present an example of the use of this methodology: we show that tagged $D\pi$ [7, 8, 6] is a tagging according to our definition and, therefore, a quotient; finally, Section 6 presents our conclusions and proposes some future work.

2 Tagged Transition Systems

In this section we will formally define what a tagged language will be for us. We start by defining the tagged syntax generated by a given many-sorted signature [2, 5] with respect to a symbol of the signature and the considered set of labels.

Definition 1 *Given a many-sorted signature $(\mathcal{S}, \Sigma)$, let E be the $\mathcal{S}$ -sorted term algebra of Σ , f a symbol of Σ and $\mathcal{L}$ a set of labels $(\Gamma, \Delta, \dots \in \mathcal{L})$. We will call $(f, \mathcal{L})$ -tagged syntax of E , and we will denote it by $\bar{E}_f^{\mathcal{L}}$ (or sometimes just by $\bar{E}$), to the $\mathcal{S}$ -sorted term algebra of $\Sigma_f^{\mathcal{L}} = (\Sigma - \{f\}) \cup \{f_\Gamma \mid \Gamma \in \mathcal{L}\}$, where $\text{arity}(f_\Gamma) = \text{arity}(f)$ for every $\Gamma \in \mathcal{L}$.*

In terms of abstract syntax trees, tagged terms are usual terms where some nodes, either leaves in case the distinguished symbol f is a constant operator, or internal nodes otherwise, are annotated with labels in $\mathcal{L}$. A more general version of tagged syntax, $\bar{E}_{f_1, f_2, \dots}^{\mathcal{L}_1, \mathcal{L}_2, \dots}$, would consider a collection of symbols $f_1, f_2, \dots$ in Σ , and a (possibly) different set of labels for each of them.

Example 1. Let us consider $(\mathcal{S}, \Sigma)$, where $\Sigma = \{\text{Zero}, \text{Succ}, +, \times\}$, $\mathcal{S} = \{\text{Nat}\}$ with $\text{arity}(\text{Zero}) = \text{Nat}$, $\text{arity}(\text{Succ}) = \text{Nat} \rightarrow \text{Nat}$ and $\text{arity}(+) = \text{arity}(\times) = \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$; and the set of labels $\mathcal{L} = \{\square, \circ\}$. The $\mathcal{S}$ -sorted term algebra of Σ defines a set of arithmetic expressions. One tagged syntax is that generated by $\Sigma_{+, \times}^{\mathcal{L}, \mathcal{L}} = \{\text{Zero}, \text{Succ}, \oplus, \otimes, \boxplus, \boxtimes\}$, which defines ordinary expressions in which the operation symbols are substituted by “encircled” or “framed”

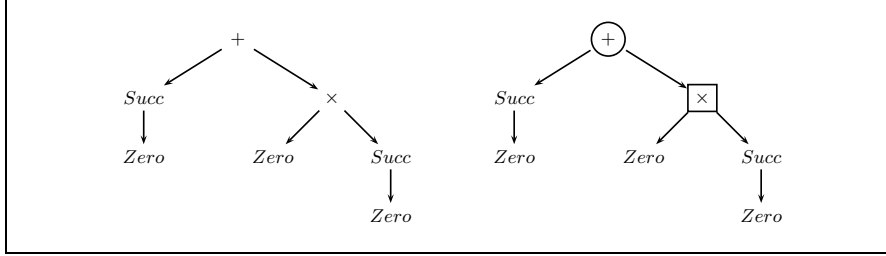


Fig. 1. Abstract syntax trees for $1 + (0 \times 1)$ and $1 \oplus (0 \otimes 1)$

symbols. Figure 1 shows the abstract syntax trees of expressions $1 + (0 \times 1)$ and $1 \oplus (0 \otimes 1)$.

We will use the usual definition of labelled transition system:

Definition 2 A labelled transition system S is a tuple $S = (E, V, \rightarrow, e_0)$, where E is the set of states, V is the set of transition labels, $e_0 \in E$ is the initial state and $\rightarrow \subset E \times V \times E$ is the transition relation. If $(e_1, a, e_2) \in \rightarrow$ we will write $e_1 \xrightarrow{a} e_2$.

We will write $e \Rightarrow e'$ if there exist $a_1, \dots, a_n \in V$ and $e_1, \dots, e_n \in E$ such that $e \xrightarrow{a_1} e_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} e_n = e'$. We will call the latter a computation sequence and denote $\mathbb{C}$ the set of computation sequences and $Reach(S) = \{e \in E \mid e_0 \Rightarrow e\}$ the set of reachable states.

Definition 3 Let $S = (E, V, \rightarrow, e_0)$ and $T = (F, V, \mapsto, f_0)$ be two labelled transition systems. We will say that S and T are isomorphic if there exists a mapping $h : Reach(S) \rightarrow Reach(T)$ such that:

1. h is a bijection
2. $h(e_0) = f_0$
3. $e \xrightarrow{a} e' \Leftrightarrow h(e) \xrightarrow{a} h(e')$ for all $e \in Reach(S)$

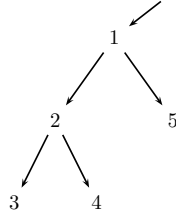
Now we will define when a transition system over the tagged syntax is a *tagging* of the original transition system. We will denote by $O(e)$ the term obtained from a tagged term e once we have removed all the tags in it.

Definition 4 Let E be the $\mathcal{S}$ -sorted term algebra of Σ and $\bar{E}$ a $(f, \mathcal{L})$ -tagged syntax of E . Also let $S = (E, V, \rightarrow, e_0)$ and $T = (\bar{E}, V, \mapsto, \bar{e}_0)$ be two transition systems with $O(\bar{e}_0) = e_0$. We will say that T is a tagging of S if the following conditions hold:

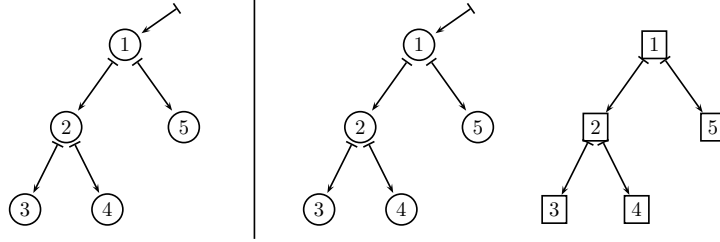
1. If $\bar{e} \mapsto \bar{e}'$ then $O(\bar{e}) \xrightarrow{a} O(\bar{e}')$.
2. If $e \xrightarrow{a} e'$ then for all $\bar{e}$ reachable from $\bar{e}_0$ such that $O(\bar{e}) = e$ there exists a unique $\bar{e}'$ with $O(\bar{e}') = e'$ such that $\bar{e} \xrightarrow{a} \bar{e}'$.

The first condition is a correctness property for tagged systems: they do not introduce new transitions, that is, all their transitions come from untagged ones. The second is a completeness condition: we have a unique tagged version for each original transition. Besides, the uniqueness states that we cannot separately remember different things about the same past.

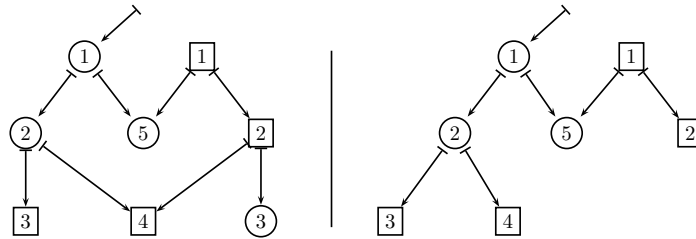
Example 2. Let us consider $\mathcal{S} = \{Nat\}$, $\Sigma = \{1, 2, 3, 4, 5\}$ with $arity(i) = Nat$ and the set of tags $\mathcal{L} = \{\circ, \square\}$. The terms obtained when we tag every symbol with tags in $\mathcal{L}$, are just the elements in $\{1, \dots, 5\}$ enclosed by $\circ$ or $\square$. Now let us consider the transition system defined by the following graph:



In this simple example, in which the graph defining the transition system is in fact a tree, a tagged system will consist of one or several copies of the original system, with at most one for each different way of annotating the initial symbol, like in the following two examples:



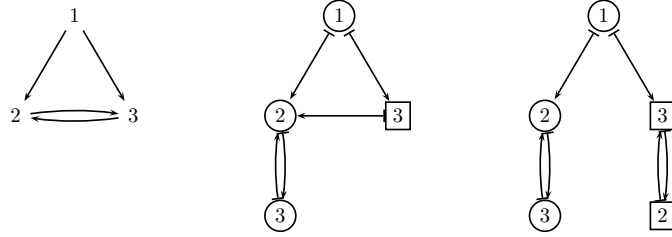
However, these copies could partially overlap for non-reachable terms, getting for instance:



Notice that in the definition of tagged systems we allow the presence of some *garbage nodes* (under certain conditions), that will not be reachable from the

initial state. We do this in order to get a more flexible definition of tagged systems, so that we do not need an exact knowledge of the set of reachable terms to define a tagged system. Of course, in practice, we will try to define them with the least possible amount of garbage, in order to have systems as simple as possible.²

Note that in this case all the tagged systems are indeed isomorphic: if we remove every non reachable term they become equal, up to the names of states. This is so in this example because the transition graph is a tree and, therefore, each node determines a unique path from the initial node (the transition graph and the reachability tree are isomorphic). As a different example, illustrating non isomorphic systems, let us consider the transition system defined by the graph on the left of the next figure, that is not a tree. The corresponding tagged systems on the right can now discriminate between different ways of reaching state 3, although the one in the middle *forgets* this difference once we move into state 2.



It is the existence of several reachable nodes with different labels what makes labels useful: just by checking the label in the reached state we have a (partial) knowledge of the way we reached that node.

3 Alternative characterization of tagged systems

In this section we will characterize the tagged systems defined in the previous section as quotients of their reachability tree. To be more precise, tagged systems are isomorphic (in the sense we have defined) to the computation tree, once we have identified some of its nodes. We start by defining the computation trees:

Definition 5 Given $S = (E, V, \rightarrow, e_0)$ a labelled transition system and $\mathcal{C}$ the set of computation sequences of S , we define the corresponding computation tree $S^t = (\mathcal{C}, V, \mapsto, e_0)$, as the transition system obtained taking $\mapsto \subset \mathcal{C} \times V \times \mathcal{C}$, defined by the rule

$$\frac{e \xrightarrow{a} e' \quad \text{last}(L) = e}{L \mapsto (L \xrightarrow{a} e')}$$

² Garbage nodes are useless, but at the same time they are absolutely innocuous, out of the increasing of the size of the system.

where “last” is the function that gives us the last state of each computation sequence.

The computation tree is simply the transition system defined by the reachability tree of the original system. It merely accumulates in its states the sequence of states that have been visited along the computation and the transitions between them. Thus, it contains all the information about the history of the process. We will sometimes call it the *totally tagged system*.

However, in general we will only be interested in a particular aspect of this information. We will use equivalence relations among computations to identify the required information.

Definition 6 We will say that an equivalence relation $\sim \subset \mathcal{C} \times \mathcal{C}$ is suitable if it satisfies the following conditions:

1. $L_1, L_2 \in \mathcal{C}$ and $L_1 \sim L_2 \Rightarrow (L_1 \xrightarrow{a} e) \sim (L_2 \xrightarrow{a} e)$
2. $L_1 \sim L_2 \Rightarrow \text{last}(L_1) = \text{last}(L_2)$

For example, the relations $\sim_\perp$ such that $L_1 \sim_\perp L_2 \Leftrightarrow \text{last}(L_1) = \text{last}(L_2)$ and $\sim_\top$ such that $L_1 \sim_\top L_2 \Leftrightarrow L_1 = L_2$ are trivial examples of suitable equivalences.

By means of suitable equivalences we will characterize the information about the past we are interested in. For instance, $\sim_\perp$ is the relation that forgets everything about the past, while $\sim_\top$ is the one remembering absolutely everything. Now we define the quotient transition system generated by such an equivalence relation:

Definition 7 Let $S = (E, V, \rightarrow, e_0)$ be a labelled transition system and $\sim \subset \mathcal{C} \times \mathcal{C}$ a suitable equivalence relation. We define $S^\sim = (\mathcal{C}/\sim, V, \mapsto_\sim, [e_0])$, and call it a quotient transition system of S with respect to $\sim$, as the transition system defined by the rule

$$\frac{L \xrightarrow{a} L'}{[L] \mapsto_\sim [L']}$$

where $L, L' \in \mathcal{C}$.

The first condition in Definition 6 is introduced so that the quotient transition system is uniformly defined, in the sense that its transitions do not depend on the choice of the representatives of the equivalence classes. The second states that the relation remembers at least the final state reached by the computation.

For the equivalences $\sim_\perp$ and $\sim_\top$ defined above, it turns out that $S^{\sim_\perp}$ and $S^{\sim_\top}$ are isomorphic to S and S^t , respectively.

Quotient transition systems are a conservative extension of ordinary transition systems in the following sense:

Proposition 1 If $[L_1], [L_2] \in \mathcal{C}/\sim$ and $[L_1] \xrightarrow{a} [L_2]$ then $\text{last}(L_1) \xrightarrow{a} \text{last}(L_2)$.

Now let us take a closer view at the structure of the set of quotient transition systems.

Definition 8 We denote by $\mathcal{S}$ the set of quotient transition systems, that is, $\mathcal{S} = \{S^\sim \mid \sim \text{ suitable}\}$, and define the operators $\sqcup$ and $\sqcap$ by $S^{\sim_1} \sqcup S^{\sim_2} \triangleq S^{\sim_1 \cap \sim_2}$ and $S^{\sim_1} \sqcap S^{\sim_2} \triangleq S^{\sim_1 \cup^* \sim_2}$, where $R_1 \cup^* R_2$ is the transitive closure of $R_1 \cup R_2$.

Then we have the following

Proposition 2 $(\mathcal{S}, \sqcap, \sqcup)$ is a complete lattice that has S and S^t as bottom and top elements, respectively.

Now let us state the main theorem of this section:

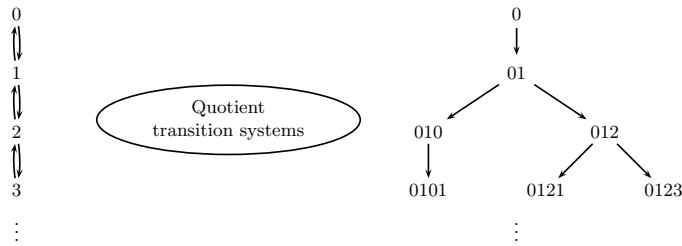
Proposition 3 Let $S = (E, V, \rightarrow, e_0)$ be a labelled transition system and let $T = (\bar{E}, V, \mapsto, \bar{e}_0)$ be a tagging of S . Then T is isomorphic to a quotient transition system of S .

The reciprocal is also true:

Proposition 4 Let $S = (E, V, \rightarrow, e_0)$ be a labelled transition system and $\sim$ a suitable equivalence relation. Then $S^\sim$ is isomorphic to a tagging of S .

In our Example 2 the lattice generated by the transition system is degenerate, since the bottom and the top elements are isomorphic. Therefore, all the tagged systems are also isomorphic. Let us now study a more interesting example:

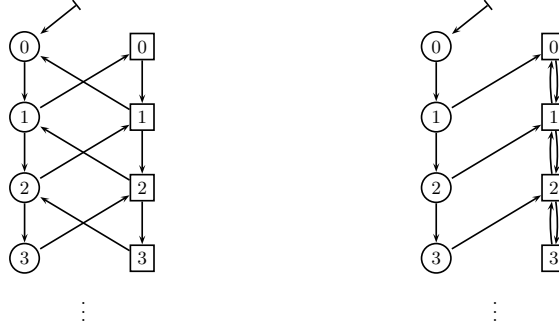
Example 3. Let us consider the language $D\pi^3$ [7] and the process $\ell\llbracket*(go\ \kappa.go\ \ell)\rrbracket$ which spawns trivial agents that go to location κ and back to ℓ , where they die. We will use the set of labels $\mathcal{L} = \{\square, \circ\}$ to label the transition system that the term generates. For each natural number n let us denote just by n the term $\kappa\llbracket go\ \ell \rrbracket \mid .^n \mid \kappa\llbracket go\ \ell \rrbracket \mid \ell\llbracket*(go\ \kappa.go\ \ell)\rrbracket$, be the term representing the state in which exactly n agents are at κ . Then we can present the transition graph (left) and the corresponding reachability tree (right) in the following way:



Then the following two transition graphs define a couple of quotient transition systems. The one on the left determines the parity of the number of agents that

³ That language will be formally presented in Section 5, although we consider that this particular example can be understood here without going into those formal details.

have died, while the one on the right only determines whether any agent has ever died. Therefore, in this second case state $\textcircled{n}$ corresponds to computation $01 \dots n$, while state $\boxed{n}$ identifies every computation sequence ending in state n , but $01 \dots n$. In terms of the used suitable relation, $\sim$ identifies every computation of the form $01 \dots n$ (the right-branch of the reachability tree) only with itself, while for the rest of the nodes, it relates every computation ending in the same state.



4 Deriving Tagged Languages

Now we address the problem of deciding whether an operational semantics, defined by a reduction relation over a tagged syntax, is in fact a tagging (and therefore a quotient) of the original reduction semantics, which was our initial goal.

Tagged semantics will be parameterized over the set of possible initial tagging functions $tag : E \rightarrow \bar{E}$, that will depend on the specific application we are interested in. In particular, tagging functions will satisfy $O \circ tag = Id_E$, but not the other way around. Typically, this function will capture on tags the static information needed to formalize the property in which we are interested.

In general, a semantics over the tagged syntax is a tagging of another operational semantics with respect to an initial tagging function tag if for every P the transition system that generates $tag(P)$ is a tagging of the transition system generated by P :

Definition 9 *We will say that a reduction relation $\bar{\mathcal{R}} = (\bar{E}, V, \mapsto)$ is a tagging of $\mathcal{R} = (E, V, \rightarrow)$ with respect to the initial tagging function tag if for every $e \in E$ the labelled transition system $(\bar{E}, V, \mapsto, tag(e))$ is a tagging of $(E, V, \rightarrow, e)$.*

We will just consider a particular case, that will be enough for our current purposes: that in which the reduction relation is structurally defined. Let us suppose that $\rightarrow$ is defined by means of a set of axioms $A_i : P_i \xrightarrow{a_i} Q_i$ for $i = 1, 2, \dots$ and a set of structural rules R_i of the form

$$\frac{P \xrightarrow{a} Q}{C_i(P) \xrightarrow{a} C_i(Q)}$$

with contexts $C_i = f^i(t_1, \dots, \square, \dots, t_{n_i})$, $i = 1, 2, \dots$, where the f^i 's used in the contexts are not distinguished symbols. Then, these structural rules can be directly considered as rules for the tagged syntax. In this setting, if we replace every axiom A_i by the set of corresponding tagged versions $\overline{A}_i$ for every way of tagging the P_i 's, we obtain a tagging of $\mathcal{R}$:

Proposition 5 *Let us consider $\mathcal{R}$ defined by means of the axioms A_i as above and a set of axioms $\overline{A}_i : \overline{P}_i \xrightarrow{a_i} \overline{Q}_i$, with $i = 1, 2, \dots$, such that $O(\overline{P}_i) = P_i$ and $O(\overline{Q}_i) = Q_i$; assume that each $\overline{A}_i$ is defined for every possible tagging⁴ of P_i . Let $\overline{\mathcal{R}}$ be the reduction relation defined by the set of axioms $\overline{A}_1, \overline{A}_2, \dots$ and the set of compositional rules $R_1, R_2, \dots$. Then $\overline{\mathcal{R}}$ is a tagging of $\mathcal{R}$.*

This is the simplest way of deriving a tagged semantics from a given semantics. The same can be done if the latter is defined by means of a structural congruence, when the distinguished symbols do not appear either in its axioms or in the rules. In this case these axioms and rules can also be applied to the tagged relation, thus defining a structural equivalence between tagged terms. Then, if we add to the set of previous reduction rules the corresponding congruence rule we also obtain a tagging of the original semantics. This is exactly the case we will study in the following section.

5 Tagged $D\pi$

In this section we will apply the proposed methodology to the language $D\pi$ [7], which is a distributed version of the π -calculus. In fact, for simplicity, we will consider the version presented in [6].

The syntax of the language is defined in Figure 2. The dots stand for the usual primitives of the π -calculus [10]: termination, composition, replication, input, output and conditional. A new construct $k[P]$, meaning that P is located at k , and a primitive *go* for movement are introduced. Also, created names can be channels, but also localities. In Figure 3 you will find the semantics of the language.

Tagged $D\pi$ is a version of $D\pi$ in which the construct $\llbracket \cdot \rrbracket$ is substituted by $\llbracket \cdot \rrbracket_\Gamma$, where Γ ranges over typing environments, that is, partial functions from locality identifiers to $\mathcal{K}$, the set of *locality types*, whose formal definition is here irrelevant [7]. Intuitively, they assign permissions (to use channels) to agents at each locality. For instance, if an agent is tagged with Γ and $\Gamma(k) = K$ then it has permissions at k specified by K . The semantics of Tagged $D\pi$ is defined by the axioms and rules in Figure 4. There, $\Gamma, \{_k e : E\}$ denotes the extension of Γ at k with the new name e having type E .

Then the set of tags $\mathcal{L}$ is the set of typing environments and the distinguished symbol is $\llbracket \cdot \rrbracket$. By mere inspection of the rules defining the tagged language it

⁴ To be exact, we will have an axiom $\overline{A}_{i, \overline{P}_i} : \overline{P}_i \xrightarrow{a_i} \overline{Q}_i$ for each $\overline{P}_i$ such that $O(\overline{P}_i) = P_i$, but usually all of them will have a homogenous structure, and this is why we propose above that more concise notation.

Names		Values	
$e ::= k$	locality	$u, v ::= \text{bv}$	base values
$ a$	channel	$ e$	name
		$ x$	variable
Patterns		$ u@v$	localized name
$X, Y ::= x$	variable	$ \tilde{u}$	tuple
$ X@z$	localized pattern		
$ \tilde{X}$	tuple		
Processes		Systems	
$P, Q ::= \dots$		$M, N ::= 0$	empty
$ go\ u.P$	movement	$ M N$	composition
$ (\nu e : E)P$	restriction	$ (\nu_k e : E)N$	restriction
		$ k[P]$	agent

Fig. 2. Syntax of $D\pi$

$M (\nu_k e : E)N \equiv (\nu_k e : E)(M N)$ if $e \notin fn(M)$	
(R-GO)	$\ell[go\ k.P] \rightarrow k[P]$
(R-COMM)	$k[a!\langle v \rangle P] k[a?(X : T).Q] \rightarrow k[P] k[Q\{X := v\}]$
(R-EQ ₁)	$k[\text{if } u = u \text{ then } P \text{ else } Q] \rightarrow k[P]$
(R-EQ ₂)	$k[\text{if } u = v \text{ then } P \text{ else } Q] \rightarrow k[Q]$ if $u \neq v$
(S-COPY)	$k[*P] \rightarrow k[P] k[*P]$
(S-SPLIT)	$k[P Q] \rightarrow k[P] k[Q]$
(S-NEW)	$k[(\nu e : E)P] \rightarrow (\nu_k e : E)k[P]$ if $e \neq k$
$(R-STR) \quad \frac{N \rightarrow N'}{(\nu e)N \rightarrow (\nu e)N'} \quad \frac{N \rightarrow N'}{M N \rightarrow M N'} \quad \frac{N \equiv M \quad M \rightarrow M' \quad M' \equiv N'}{N \rightarrow N'}$	

Fig. 3. Semantics of $D\pi$

$M (\nu_k e : E)N \equiv (\nu_k e : E)(M N)$ if $e \notin fn(M)$	
(R-GO)	$\ell[go\ k.P]_r \rightarrow k[P]_r$
(R-COMM)	$k[a!\langle v \rangle P]_r k[a?(X : T).Q]_\Delta \rightarrow k[P]_r k[Q\{X := v\}]_{\Delta \cap \{k v : T\}}$
(R-EQ ₁)	$k[\text{if } u = u \text{ then } P \text{ else } Q]_r \rightarrow k[P]_r$
(R-EQ ₂)	$k[\text{if } u = v \text{ then } P \text{ else } Q]_r \rightarrow k[Q]_r$ if $u \neq v$
(S-COPY)	$k[*P]_r \rightarrow k[P]_r k[*P]_r$
(S-SPLIT)	$k[P Q]_r \rightarrow k[P]_r k[Q]_r$
(S-NEW)	$k[(\nu e : E)P]_r \rightarrow (\nu_k e : E)k[P]_{r, \{k e : E\}}$ if $e \neq k$
$(R-STR) \quad \frac{N \rightarrow N'}{(\nu e)N \rightarrow (\nu e)N'} \quad \frac{N \rightarrow N'}{M N \rightarrow M N'} \quad \frac{N \equiv M \quad M \rightarrow M' \quad M' \equiv N'}{N \rightarrow N'}$	

Fig. 4. Tagged axioms of $D\pi$

can be seen that they satisfy the conditions imposed in the previous section. Indeed, we have a tagged version of each axiom, for every way of tagging the term on its left hand-side. The rest of the rules of the original language can also be considered as rules of the tagged one, since the symbol $\llbracket \cdot \rrbracket$ does not appear in their definition. Then we have the following

Theorem 6 *Tagged $D\pi$ is a tagging of $D\pi$.*

According to this result and our characterization of tagged systems it follows that the transition system generated by each term of *Tagged $D\pi$* is isomorphic to a quotient transition system. Therefore, every property defined over *Tagged $D\pi$* is also a property of $D\pi$. In particular, every subset of tagged terms (e.g., that of erroneous terms) defines a subset of computations (that of erroneous computations).

The result is true whatever the initial tagging function is. However, the choice is crucial to adequately formalize the runtime errors we have in mind: in this case the resource access errors. In [7] the chosen function *tag* is only defined for typed processes of the original calculus, so that if M is not typeable then $\text{tag}(M) = \emptyset$.

Another possibility is to tag every original process, simply by adding in the tags the created names, which are exactly those the agents have initially permission to use. We will do so with the help of the mappings $\text{tag}_\Gamma(M)$, where Γ is the initial knowledge of M . Let us consider the system defined by $(\nu a : A)(k\llbracket b!\langle a \rangle \rrbracket \mid k\llbracket b?(x : A).x!\langle c \rangle \rrbracket)$. Though the channel a is created with the two agents under its scope, only the agent on the left knows of its existence. In fact, the scope of the names can always be extended via extrusion. In particular, that system and $(\nu a : A)k\llbracket b!\langle a \rangle \rrbracket \mid k\llbracket b?(x : A).x!\langle c \rangle \rrbracket$ are equivalent. In order to deal with this fact, we will restrict ambient domains in $\text{tag}_\Gamma(M)$ to the set of free names in M , that is, we will have the invariant $\text{dom}(\Gamma) = \text{fn}(M)$, and we will denote by Γ_M the restriction $\Gamma \upharpoonright_{\text{fn}(M)}$. Now we can define the initial tagging function:

Definition 10 *Let Φ be the empty (partial) function in $\mathcal{L}$ (that with empty domain). Then we define the function $\text{tag} : \mathcal{M} \rightarrow \tilde{\mathcal{M}}$ by:*

- $\text{tag}(M) = \text{tag}_\Phi(M)$ if M is closed
- $\text{tag}_\Gamma(0) = 0$
- $\text{tag}_\Gamma(M \mid N) = \text{tag}_{\Gamma_M}(M) \mid \text{tag}_{\Gamma_N}(N)$
- $\text{tag}_\Gamma((\nu_\ell e : E)M) = (\nu_\ell e : E)\text{tag}_{\Gamma, \{\ell e : E\}}(M)$ if $e \in \text{fn}(M)$
- $\text{tag}_\Gamma((\nu_\ell e : E)M) = (\nu_\ell e : E)\text{tag}_\Gamma(M)$ if $e \notin \text{fn}(M)$
- $\text{tag}_\Gamma(\ell\llbracket P \rrbracket) = \ell\llbracket P \rrbracket_\Gamma$

In [7] an error relation on the corresponding tagged systems $M \rightarrow \text{err}$ is defined, meaning that a process can violate its permissions. To remove these errors, a type system $\vdash$ for the original language and another related one $\Vdash$ for the tagged systems were defined. We need the following lemma:

Lemma 7 *If $\Gamma \vdash M$ then $\Gamma \Vdash \text{tag}_\Gamma(M)$.*

A subject reduction theorem and a safety theorem for the tagged system were proved in [7]. Then we have our safety theorem:

Theorem 8 *If $\Gamma \vdash M$ and $\text{tag}_\Gamma(M) \mapsto^* M'$ then $M' \not\rightarrow err$.*

And the following

Corollary 9 *If $\Phi \vdash M$ and $\text{tag}(M) \mapsto^* M'$ then $M' \not\rightarrow err$.*

This result also appears in [7], but no interpretation of it is given in terms of the original semantics of the language. Instead, we can now interpret the fact that $M \not\rightarrow err$ as the impossibility to use a resource when we have not received the permission to do it along the current computation, since the tags remember exactly the set of accumulated permissions.

Example 4. As a very simple example, let us consider

$$M = (\nu_\ell : \text{loc}\{a : \text{res}\langle T \rangle\})(\nu_k b : \text{res}\langle T \rangle)(\nu_\ell c : T)\ell[b!\langle c \rangle]$$

We can tag M , getting

$$\text{tag}(M) = (\nu_\ell : \text{loc}\{a : \text{res}\langle T \rangle\})(\nu_k b : \text{res}\langle T \rangle)(\nu_\ell c : T)\ell[b!\langle c \rangle]_\Gamma$$

with

$$\Gamma = \{\ell : \text{loc}\{a : \text{res}\langle T \rangle, c : T\}, k : \text{loc}\{b : \text{res}\langle T \rangle\}\}$$

However, M is not typeable and, in fact, $\text{tag}(M) \rightarrow err$, since it attempts to use channel b at locality l , which is not allowed by Γ . Of course, since types cannot be complete, it could be the case that M is not typeable but $\text{tag}(M)$ will not produce any error.

6 Conclusion and Future Work

We think that the present work is a rather simple, but conceptually important justification of the use of tagged languages for the analysis of systems. The fact that these systems are quotients of the corresponding reachability trees tells us that the properties defined over them are, in fact, also properties of the original semantics, thus keeping us within the same original framework for every possible property of our interest. It may also be worthy a further study of the algebraic structure of the set of tagged languages and exploit it to study the combination of several properties. For example, if we have a language tagged with $\mathcal{L}$ and an injective mapping $\mathcal{L} \hookrightarrow \mathcal{L}'$ then we can consider the language as also being tagged with $\mathcal{L}'$. Therefore, if we have tagged a language in two different ways, say with $\mathcal{L}_1$ and $\mathcal{L}_2$, to treat two different properties, we can consider $\mathcal{L} = \mathcal{L}_1 \times \mathcal{L}_2$ and then both tagged languages would be two different taggings over the same tagged syntax. Then we could consider the language satisfying both properties simultaneously: if $\sim_1$ and $\sim_2$ were the equivalence

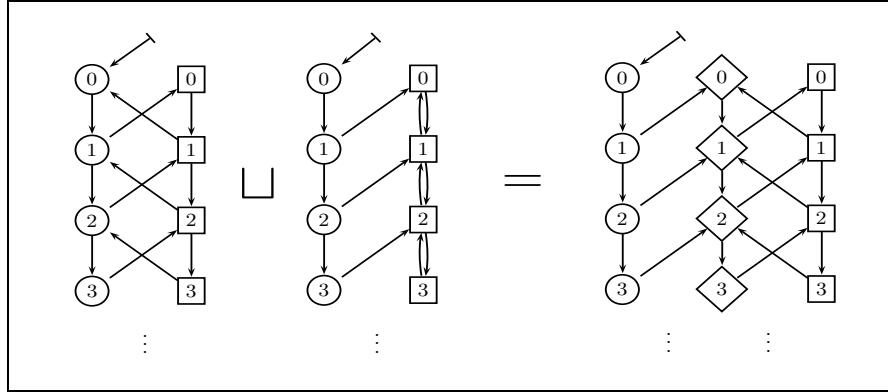


Fig. 5. Disjunction of tagged systems

relations defining each property, then the corresponding tagged language would be $S^{\sim 1} \sqcup S^{\sim 2}$.

Let us recall Example 3 in Section 3. If we want to consider the conjunction of the two properties there considered (whether any agent has died, and in such a case, whether this happened an even number of times) it suffices to consider $\mathcal{L} = \{\square, \circ\} \times \{\square, \circ\}$ and the corresponding joint transition system. If we take⁵ $\circ = (\circ, \circ)$, $\square = (\square, \square)$ and $\diamond = (\circ, \square)$ (we will not need the tag $(\square, \circ)$ since terms tagged with it will not be reachable) then we have the disjunction of the transition systems, as shown in Figure 5, where terms tagged with $\circ$ identify computations in which no agent has died, $\diamond$ identifies computations in which an odd number of agents have died and $\square$ identifies computations in which an even, but not null, number of agents have died. However, when we consider the meet of both transition systems (the disjunction of properties) we go all the way down to the bottom of the lattice.

So far, we have considered equivalences remembering at least the reached state, thus getting always some tagged systems. However, this constraint could be relaxed to obtain a superset of our class of quotient systems (see Figure 6). Then the bottom would represent the tagged system forgetting absolutely everything, even the reached state. Additionally, we would have transition systems lying in the diamond of the bottom of Figure 8, remembering only certain properties about the reached state, and transition systems combining both approaches, the ones in the diamonds at both sides of the figure, that remember some things about the past of the process but still only part of the information about the reached state.

We plan to study all these equivalences in a systematic way, thus trying to get their properties in a uniform way, to avoid the development of ad hoc proofs for each particular case in which we could be interested.

⁵ Obviously, we are overloading the notation here in order to preserve the symbols used in the figures.

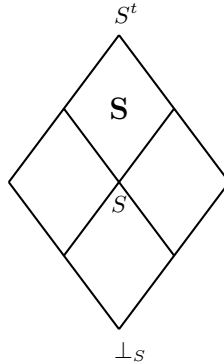


Fig. 6. Extended quotient systems lattice

References

- [1] M. Boreale and M. Buscemi. A framework for the analysis of security protocols. In *Concurrency Theory, 13th International Conference, CONCUR'02*, volume 2421 of *LNCS*, pages 483–498. Springer, 2002.
- [2] Hartmut Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I*. Springer-Verlag New York, Inc., 1985.
- [3] R. Foccardi and R. Gorrieri. Classification of security properties (Part I: Information flow). In *Foundations of Security Analysis and Design, FOSAD'00*, volume 2171 of *LNCS*, pages 331–396. Springer, 2001.
- [4] R. Foccardi, R. Gorrieri, and F. Martinelli. Classification of security properties (Part II: Network security). In *Foundations of Security Analysis and Design II, FOSAD'02*, volume 2946 of *LNCS*, pages 139–185. Springer, 2004.
- [5] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Wiley and Sons, 1990.
- [6] M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed systems. In *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS'03*, volume 2620 of *LNCS*, pages 282–298. Springer, 2003.
- [7] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *High-Level Concurrent Languages, HLCL'98*, volume 16 of *ENTCS*, pages 3–17. Elsevier, 1998.
- [8] M. Hennessy and J. Riely. Trust and partial typing in open systems of mobile agents. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages, POPL'99*, pages 93–104. ACM Press, 1999.
- [9] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(5):566–591, 2002.
- [10] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

Proving modal properties of rewrite theories using Maude's metalevel ^{*}

Isabel Pita and Miguel Palomino

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
{ipandreu,miguelpt}@sip.ucm.es

Abstract. Rewriting logic is a very expressive formalism for the specification of concurrent and distributed systems; more generally, it is a logic of change. In contrast, VLRL is a modal logic built on top of rewriting logic to reason precisely about that change. Here we present a technique to mechanically prove VLRL properties of rewrite theories using the reflective capability of rewriting logic through its Maude implementation.

1 Introduction

Rewriting logic [9] provides a formal framework for modelling concurrent systems in terms of states and state transitions, which is efficiently implemented in the Maude system [5]. It is a logic *of* change in which deduction directly corresponds to the change. In contrast, the Verification Logic for Rewriting Logic (VLRL) [7] is an action modal logic to talk *about* change in a more indirect and global manner, like other modal and temporal logics. VLRL was developed to prove abstract properties of systems specified in rewriting logic.

In VLRL, rewrite rules are captured as actions, transitions are represented by action modalities, and the structure of the state is represented by spatial modalities. In this way, action modalities allow the definition of properties of states after a particular sequence of rewrites, and the spatial modality allows the definition of properties of components of the state.

This kind of formulae can be proved at the object level with the help of Maude's LTL model checker if we are able to define a predicate *taken(a)*, holding in those states that arise from applying the action *a* to some other state, and a predicate *concurrent* stating that two actions can be executed concurrently. But in general, for a rewrite theory $\mathcal{R}$ it may be very difficult (or even impossible) to specify such predicates. In [10] we solved this problem by showing how to obtain a *semantically equivalent* theory $\mathcal{R}'$ in which those predicates can be straightforwardly defined.

In this paper we develop an alternative way of proving that a VLRL formula holds in a given state, that relies heavily on the use of *reflection* [6]. Reflection

^{*} Research supported by the Spanish Projects MELODIAS TIC2002-01167 and MIDAS TIC2003-01000.

allows a system to access its own metalevel and manipulate theories as ordinary objects, thus providing a powerful mechanism for controlling the rewriting process. Maude's metalevel provides convenient operations that permit us to control the execution of rewrite rules in the object system and we will use them to obtain the result from applying an action to a state as well as to look at the states' inner structure. This approach is simpler than the one explored before, as it requires neither a theory transformation nor translating the VLRL formulae to linear temporal logic and the use of Maude's model checker. For the sake of concreteness we illustrate our technique with a concrete example, whose complete code can be found at maude.sip.ucm.es/vlrl/metalevelprover/, but the method is general and can be easily adapted to any system.

2 Overview on rewriting logic

A distributed system is axiomatized in rewriting logic [9] by a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory describing its set of *states* as the algebraic data type $T_{\Sigma/E}$ associated to the initial algebra (Σ, E) . The system's *transitions* are axiomatized by the *conditional rewrite rules* R which are of the form $l : t \longrightarrow t' \text{ if } \textit{cond}$, with l a label, t and t' Σ -terms, possibly with variables, and $\textit{cond}$ a condition involving equations and rewrites. Under reasonable assumptions about E and R , rewrite theories are *executable*, and there are several rewriting logic language implementations, including ELAN [1], CafeOBJ [8], and Maude [4, 5]. In particular, Maude offers support for multiple sorts, subsort relations, operator overloading, and, unique among the other implementations, reflection.

We illustrate rewriting logic specifications by borrowing a distributed banking system example from the Maude manual [5]. Such a system is a “soup” of objects, which represent bank accounts, and messages, that represent actions to perform over those accounts.

```

mod BANK-ACCOUNT is
  protecting INT .
  including CONFIGURATION .
  op Account : -> Cid .
  op bal : _ : Int -> Attribute .
  ops credit debit : Oid Nat -> Msg .
  op from_to_transfer_ : Oid Oid Nat -> Msg .
  vars A B : Oid .
  vars M N N' : Nat .

  rl [credit] : < A : Account | bal : N > credit(A,M) =>
               < A : Account | bal : N + M > .
  crl [debit] : < A : Account | bal : N > debit(A,M) =>
               < A : Account | bal : N - M > if N >= M .
  crl [transfer] : (from A to B transfer M) < A : Account | bal : N >
                  < B : Account | bal : N' > =>
                  < A : Account | bal : N - M > < B : Account | bal : N' + M >

```

```

    if N >= M .
endm

```

Integers are imported in protected form in the second line, while the syntax for objects is imported from module **CONFIGURATION**. An object has the form $\langle A : C \mid \text{Atts} \rangle$, where A is the object's name, C its class, and Atts the list of the object's attributes. In the module **BANK-ACCOUNT** only one class, **Account**, was declared with an attribute **bal**. We also have three messages for credit, debit, and transfer of some money, each with an associated rewrite rule (the last two are conditional) that axiomatize the behaviour of the system when a message is received. Note that objects and messages are combined with an operator $_$ that is declared in **CONFIGURATION** to be associative, commutative, and with identity **none**, and that returns a term of sort **Configuration**. Finally, we can declare a new module that extends the previous one with a new class **Manager** whose objects will be in charge of creating new accounts.

```

mod BANK-MANAGER is
  inc BANK-ACCOUNT .
  op Manager : -> Cid .
  op new-account : Oid Oid Nat -> Msg [ctor] .
  vars O C : Oid .
  var N : Nat .
  rl [new] : < O : Manager | none > new-account(O, C, N) =>
    < O : Manager | none > < C : Account | bal : N > .
endm

```

3 The VLRL logic

VLRL permits the definition of observations over a system after choosing a distinguished sort *State*. In this way, the user defines the means by which he wants to talk about the behaviour of the system; see [7] for a complete presentation.

For example, in the bank account system we can choose **Configuration** as the distinguished sort. Then, we can observe if there is an account in the system with the help of an **account?** observation, and the current balance of an account with an observation **balance?**.

```

op account? : Oid -> VLRLBool .
op balance? : Oid -> VLRLInt .

```

This defines, for each object's name, two observations. Their actual meaning is given by means of interpretations I that take a term of sort *State* and an observation of sort s and return a term of sort s . For the “standard” interpretation one would expect, for example,

$$I(\langle A-002 : \text{Account} \mid \text{bal}: 300 \rangle, \text{balance?}(A-002)) = 300,$$

where **A-002** is an account name.

Actions We start by defining *preactions* α . These correspond to the quotient of the set of proof terms obtained through the following rules of deduction:

- *Identities*:¹ for each $[t]$, $\overline{[t] : [t] \rightarrow [t]}$,
- *Replacement*: for each rewrite rule $r : t(x_1, \dots, x_n) \longrightarrow t'(x_1, \dots, x_n)$ and terms $w_1, \dots, w_n$, $\overline{r(\overline{w}) : [t(\overline{w}/\overline{x})] \rightarrow [t'(\overline{w}/\overline{x})]}$, and
- Σ -*structure*: for each $f \in \Sigma$, $\frac{\alpha_1 : [t_1] \rightarrow [t'_1] \quad \dots \quad \alpha_n : [t_n] \rightarrow [t'_n]}{f(\alpha_1, \dots, \alpha_n) : [f(t_1, \dots, t_n)] \rightarrow [f(t'_1, \dots, t'_n)]}$,

modulo the following equations:

- *Identity transitions*: $f([t_1], \dots, [t_n]) = [f(t_1, \dots, t_n)]$,
- *Axioms in E* : $t(\overline{\alpha}) = t'(\overline{\alpha})$, for each equation $t = t'$ in E .

Actions are the preactions that rewrite terms of sort *State*.²

Intuitively, actions (or more generally, preactions) correspond to transitions where no sequential composition or nested applications of the replacement rule have taken place. It was proved in [9] that any transition in the initial model of a rewrite theory can be decomposed as an interleaving sequence of preactions.

The modal language The formulae in the modal language are given by

$$\varphi ::= \text{true} \mid t_1 = t_2 \mid \neg\varphi \mid \varphi \supset \varphi \mid \langle\alpha\rangle\varphi \mid \langle\langle\alpha\rangle\rangle\varphi \mid f_{\overline{d}}\langle\varphi_1, \dots, \varphi_n\rangle$$

where t_1 and t_2 are terms that may contain observations, α is an action, $f : s_1 \dots s_m \rightarrow \text{State} \in \Sigma$, $\overline{d}$ is a sequence of data terms corresponding to the arguments of f that are not of sort *State*, and the φ_i are in one-to-one correspondence with the arguments of f that are of sort *State*. We also have the dual action operators $[\cdot]$ and $[[\cdot]]$, and the dual spatial operator $f_{\overline{d}}[\varphi_1, \dots, \varphi_n]$.

Satisfaction of VLRL formulae at a given state $[t]$, observation interpretation I , and ground substitution σ is defined by structural induction in the usual way. An equation $t_1 = t_2$ holds in $[t]$ if the interpretations of t_1 and t_2 are the same; $\langle\alpha\rangle\varphi$, resp. $\langle\langle\alpha\rangle\rangle\varphi$, is true in $[t]$ if there exists a state $[t']$ that satisfies φ that can be reached from $[t]$ by applying action α at the top of $[t]$, resp. anywhere in $[t]$; and $f_{\overline{d}}\langle\varphi_1, \dots, \varphi_n\rangle$ is satisfied if there is a term of the form $f_{\overline{w}}(t_1, \dots, t_n)$ in the equivalence class $[t]$, where $\overline{w}$ is the value of $\overline{d}$ at state $[t]$ for the observation interpretation I and ground substitution σ , such that each $[t_i]$ satisfies φ_i . We refer to [7] for a formal definition.

4 VLRL in Maude

We now embark ourselves on specifying VLRL inside Maude, aiming at using it to automatically check whether a given VLRL formula holds in a given state.

¹ We use $[t]$ to denote both the equivalence class that represents the state and the identity transition for that state.

² The above definition of actions assumes that the rules in R are unconditional. The extension to conditional rules is straightforward (see [2]).

Syntax The first step in that direction consists in defining VLRL's syntax in a Maude module.

```
fmod VLRL-FORMULAE is
  sorts Action VLRLFormula .

  ops True False : -> VLRLFormula .
  op _->_ : VLRLFormula VLRLFormula -> VLRLFormula .
  op ~_ : VLRLFormula -> VLRLFormula .
  op <_>_ : Action VLRLFormula -> VLRLFormula .
  op <<_>>_ : Action VLRLFormula -> VLRLFormula .

  op [_]_ : Action VLRLFormula -> VLRLFormula .
  op [[_]_] : Action VLRLFormula -> VLRLFormula .
  op _/\_ : VLRLFormula VLRLFormula -> VLRLFormula .
  op _\/_ : VLRLFormula VLRLFormula -> VLRLFormula .

  vars X Y : VLRLFormula .
  var A : Action .

  eq [ A ] X = ~ ( < A > ~ X ) .
  eq [[ A ]] X = ~ ( << A >> ~ X ) .
  eq X \ / Y = ~ X -> Y .
  eq X /\ Y = ~ ( X -> ~ Y ) .
endfm
```

The module VLRL-FORMULAE above defines the syntax of the propositional and modal action formulae, which is system-independent. The concrete syntax for basic formulae $t_1 = t_2$, spatial formulae $f_{\vec{a}}\langle\varphi_1, \dots, \varphi_n\rangle$, and actions depends on the particular system at hand, and is defined in a supermodule. For the banking system we declare, in a module VLRL-SAT that imports BANK-MANAGER and VLRL-FORMULAE, the operators

```
op _=_ : VLRLInt VLRLInt -> VLRLFormula .
op _=_ : VLRLBool VLRLBool -> VLRLFormula .
```

for atomic formulae (one for each observation sort).

Then we define the spatial formulae. For each $f_{\vec{a}} : \text{State} \dots \text{State} \longrightarrow \text{State}$, we have to declare an operator $f_{\vec{a}}\langle \dots \rangle : \text{VLRLFormula} \dots \text{VLRLFormula} \rightarrow \text{VLRLFormula}$, and its dual $f_{\vec{a}}[\dots] : \text{VLRLFormula} \dots \text{VLRLFormula} \rightarrow \text{VLRLFormula}$. In our example, the only such operator is $_ _$ and we get:

```
op <_>_ : VLRLFormula VLRLFormula -> VLRLFormula .
op [_]_ : VLRLFormula VLRLFormula -> VLRLFormula .

vars X Y : VLRLFormula .
eq [ X Y ] = ~ < ( ~ X ) ( ~ Y ) > .
```

Finally, to capture preactions we declare a sort *PreAction-s* for each sort *s* in the original signature. Note that in the absence of conditional rules we only

need to declare preaction sorts for those sorts used in the definition of state constructors. Then the preaction corresponding to the sort selected as state of the system is made a subsort of **Action**.

```
sort PreActionConfiguration .
subsort PreActionConfiguration < Action .
```

For each sort s , actions arising from the identity rule are represented with an operator $[-] : s \longrightarrow \text{PreAction-}s$. To capture actions resulting from the application of the replacement rule we add, for each rewrite rule $l : t(\bar{x}) \longrightarrow t'(\bar{x})$, where we assume a fixed order in $\bar{x} = (x_1 : s_1, \dots, x_n : s_n)$, the operator $l : s_1 \dots s_n \longrightarrow \text{RPreAction-}s$. $\text{RPreAction-}s$ is a subsort of $\text{PreAction-}s$ used to identify precisely those preactions obtained through the replacement rule. Finally, actions obtained with the Σ -structure rule are represented by allowing the operators of the signature to apply to actions as well: for each operator $f : s_1 \dots s_n \longrightarrow s$, we declare $f : \text{PreAction-}s_1 \dots \text{PreAction-}s_n \longrightarrow \text{PreAction-}s$. In addition, in order to take into account the quotient described on page 4, all equations in the specification are duplicated so that now they also apply to preactions and the corresponding equations for the identity transitions are included. (Note that the new operator $--$ is declared with the same attributes as $--$ over **Configuration**.)

```
sort RPreActionConfiguration .
subsort RPreActionConfiguration < PreActionConfiguration .
op [-] : Configuration -> PreActionConfiguration .
op none : -> PreActionConfiguration .
op -- : PreActionConfiguration PreActionConfiguration ->
      PreActionConfiguration [assoc comm id: none] .
```

The **RPreActionConfiguration** operators depend on the rewriting rules of the particular system we are observing; in the banking system:

```
ops credit debit : Oid Int Int -> RPreActionConfiguration .
op transfer : Oid Oid Int Int Int -> RPreActionConfiguration .
op new : Oid Oid Int -> RPreActionConfiguration .
```

Formulae satisfaction To study satisfiability of VLRL formulae we first have to define a valid interpretation for the observations and then to extend it to arbitrary terms. For example, for the **balance?** observation and the “standard” interpretation we would declare:

```
op interp : Configuration VLRLInt -> Int .
op balance? : Oid -> VLRLInt .
op balance-aux : Configuration Oid -> Int .

var C : Configuration .      vars O O1 O2 : Oid .
vars N N1 N2 : Int .        var M : Msg .

eq balance-aux(none, 0) = 0 .
```

```

eq balance-aux(< 0 : Account | bal : N > C, 0) = N .
ceq balance-aux(< 01 : Account | bal : N > C, 02) = balance-aux(C, 02)
  if 01 /= 02 .
eq balance-aux(M C, 0) = balance-aux(C, 0) .

eq interp(C,balance?(0)) = balance-aux(C,0) .
eq interp(C,0) = 0 .
eq interp(C,s(N)) = s(interp(C,N)) .

```

Let us now make two assumptions. The first one is that we have at our disposal two operations

```

op nextState : State Action -> State .
op nextStateinContext : State Action -> StateList .

```

that given a state and an action return, respectively, the unique (if any) successor of the state that follows from applying that action *at the top* and the set of all successors that can be obtained from applying the action *in context*. The second assumption is that for each $f_{\vec{a}} : \text{State} \dots \text{State} \longrightarrow \text{State}$ we have an operation $\text{decomposable?} : \text{State VLRLFormula} \dots \text{VLRLFormula} \longrightarrow \text{Bool}$ such that $\text{decomposable?}(S, F_1, \dots, F_n)$ is true if S is of the form $f_{\vec{a}}(t_1, \dots, t_n)$, with each t_i satisfying F_i for $i = 1 \dots n$; for the bank system:

```

op decomposable? : State VLRLFormula VLRLFormula -> Bool .

```

Now, satisfiability of arbitrary VLRL formulae can be straightforwardly defined by structural induction. The distinguished sort, **Configuration**, is declared as a subsort of **State**, to which an **error** constant is also added to signal when an action cannot be applied to a given state.

```

subsort Configuration < State .
op error : -> State .
op _|=_ : State VLRLFormula -> Bool .

var S : State .          vars I1 I2 : VLRLInt .
vars B1 B2 : VLRLBool .  vars F1 F2 : VLRLFormula .
var A : Action .

ceq (S |= I1 = I2) = true if interp(S, I1) == interp(S, I2) .
ceq (S |= I1 = N2) = false if interp(S, I1) /= interp(S, I2) .
ceq (S |= B1 = B2) = true if interp(S, B1) == interp(S, B2) .
ceq (S |= B1 = B2) = false if interp(S, B1) /= interp(S, B2) .
eq (S |= True) = true .
eq (S |= False) = false .
eq (S |= (F1 -> F2)) = (S |= F2) or not (S |= F1) .
eq (S |= (~ F1)) = not (S |= F1) .

--- Actions
ceq (S |= < A > F1) = nextState(S,A) |= F1 if nextState(S,A) /= error .
eq (S |= < A > F1) = false [otherwise] .

```

```

ceq (S |= << A >> F1) = nextStateinContext(S,A) |= F1
    if nextStateinContext(S,A) /= nil .
eq (S |= << A >> F1) = false [owise] .
--- Spatial
ceq (S |= < F1 F2 >) = true if decomposable?(S, F1, F2) .
eq (S |= < F1 F2 >) = false [owise] .

```

In Maude, an *owise* equation is tried only if no other applies; its semantics can be defined purely in terms of equations [5, Chapter 4].

It only remains to specify the three operations that we took for granted, and it is precisely here where the main innovation of this paper lies. Note that it is in their nature that they should use as *input* the module that specifies the banking system itself, for instance to select a particular rule to apply to the state in the case of *nextState*. To define them, then, we require the use of Maude's metalevel, that we explain in the next section.

5 The module META-LEVEL

Maude's metalevel is specified in a predefined module called **META-LEVEL**. This module contains operators to (meta)represent terms and modules respectively as terms of sort **Term** and **Module**. A variable *X* and a constant *C* of sort *s* are represented by the quoted identifiers '*X:s*' and '*C.s*', respectively; the representation of a term *f(t₁, ..., t_n)* is '*f[$\bar{t}_1, \dots, \bar{t}_n$]*', where *t₁*, ..., *t_n* are the representations of *t₁* and *t_n*. Similar conventions apply to modules. In addition, Maude's metalevel supplies the user with efficient descent operations to reduce metalevel computations to object-level ones. We presently give a brief summary of the ones we need; for a complete and more accurate description we refer the reader to [5, Chapter 10].

- The operation **metaApply** takes five arguments: the metarepresentation of a module *M*, a term's metarepresentation, the name of a rule in *M*, a substitution's metarepresentation, and a natural number *n*. It tries to match the term with the lefthand side of the rule using the substitution, discards the first *n* matches, and applies the rule at the top of term with the *n* + 1 match. It returns a triple formed by the metarepresentation of the reduced term, its corresponding sort or kind, and the substitution used; we can obtain the first component of the triple with the operation **getTerm**.
- The operation **metaXapply** is analogous to **metaApply**, but the rule can be applied anywhere in the term.
- The operation **metaMatch**(*R*, *t*, *t'*, *Cond*, *n*) tries to match at the top the terms *t* and *t'* in the module *R* in such a way that the resulting substitution satisfies the condition *Cond*. The last argument is used to enumerate possible matches. It returns a term of sort **Substitution** if it succeeds, and **noMatch** otherwise. There is a corresponding generalization called **metaXmatch**.

- `upModule` takes as a first argument the name of a module already in Maude's database, and returns its representation as a term of sort `Module`. There is also a second argument that we can safely assume it is always `true`.
- The function `upTerm` takes a term and returns its metarepresentation; the operation `downTerm` works as its inverse.

6 The functions `nextState`, `nextStateinContext`, and `decomposable?`

The functionality required by `decomposable?` is supplied by `metaMatch` for free. Therefore, the semantics of spatial formulae can be defined by:

```
ceq (S |= < F1 F2 >) = true
  if metaMatch(upModule('VLRL-SAT+,true),
    '[_]'S1:State,'S2:State],
    upTerm(S),
    '[_]'S1:State,upTerm(F1)] = 'true.Bool /\
    '[_]'S2:State,upTerm(F2)] = 'true.Bool,
    0) /= noMatch .
ceq (S |= < F1 F2 >) = false [owise] .
```

We simply decompose the state into two substates, `S1` and `S2`, and require that each of them satisfies the corresponding property `F1` or `F2` by means of the metacondition

```
'[_]'S1:State,upTerm(F1)] = 'true.Bool /\
'[_]'S2:State,upTerm(F2)] = 'true.Bool
```

The specification of the functions `nextState` and `nextStateinContext` is however much more involved, although the idea is very simple: extract the labels from the actions and apply the corresponding rules with `metaApply` and `metaXapply`.

For that, we need two auxiliary operations over `RPreActionConfiguration`: `upAction`, to obtain the rule's label associated to the action, and `getSubs`, that constructs the substitution to be used when applying the rule.

```
op upAction : RPreActionConfiguration -> Qid .
op getSubs : RPreActionConfiguration -> Substitution .

vars O O1 O2 : Oid .
vars N M N1 N2 N3 : Int .

eq upAction(credit(O,N,M)) = 'credit .
eq upAction(debit(O,N,M)) = 'debit .
eq upAction(transfer(O1,O2,N1,N2,N3)) = 'transfer .
eq upAction(new(O1,O2,N)) = 'new .

eq getSubs(credit(O,N,M)) =
```

```

'A:Oid <- upTerm(0) ; 'N:Int <- upTerm(N) ; 'M:Int <- upTerm(M) .
eq getSubs(debit(0,N,M)) =
'A:Oid <- upTerm(0) ; 'N:Int <- upTerm(N) ; 'M:Int <- upTerm(M) .
eq getSubs(transfer(01,02,N1,N2,N3)) =
'A:Oid <- upTerm(01) ; 'B:Oid <- upTerm(02) ; 'M:Int <- upTerm(N1) ;
'N:Int <- upTerm(N2) ; 'N':Int <- upTerm(N3) .
eq getSubs(new(01,02,N)) =
'O:Oid <- upTerm(01) ; 'C:Oid <- upTerm(02) ; 'N:Int <- upTerm(N) .

```

Then, an identity action can be executed at the top of a state only if it corresponds to the whole state. The state does not change if an identity action is applied.

```

eq nextState(S, [ S ]) = S .
eq nextState(S, [ S1 ]) = error [otherwise] .

```

The state that results from applying a replacement action α to a state $[t]$ is obtained with the metalevel operation `metaApply` and the help of the auxiliary operations just defined.

```

ceq nextState(S, RA) = downTerm(getTerm(R:ResultTriple?),error)
  if R:ResultTriple? := metaApply(upModule('VLRL-SAT+',true),upTerm(S),
                                   upAction(RA),getSubs(RA),0) /\
  R:ResultTriple? /= failure .
eq nextState(S, RA) = error [otherwise] .

```

For actions $f(\alpha_1, \dots, \alpha_n)$ obtained with the Σ -structure rule, we define an operation `nextState-aux` that tries all possible decompositions $f(t_1, \dots, t_n)$ of the state by using the metalevel operation `metaMatch`, and chooses the one, if any, such that each action α_i can be applied to the state t_i . If there is no such a state decomposition the `error` state is generated. We first consider the operator `_;`; the functions `getFirstTerm` and `getSecondTerm` are used to extract the two terms in the substitution returned by `metaMatch`.

```

ceq nextState(S, A1 A2) = nextState-aux(S, A1 A2, 0)
  if A1 /= none /\ A2 /= none .

op nextState-aux : State Action Nat -> State .

ceq nextState-aux(S, A1 A2, i) =
  nextState(downTerm(getFirstTerm(R:Substitution?),error),A1)
  nextState(downTerm(getSecondTerm(R:Substitution?),error),A2)
  if A1 /= none /\ A2 /= none /\
  R:Substitution? := metaMatch(upModule('VLRL-SAT+',true),
                                '_[M1:State, M2:State],
                                upTerm(S), nil, i) /\
  R:Substitution? /= noMatch /\
  nextState(downTerm(getFirstTerm(R:Substitution?),error),A1)
  /= error /\

```

```

nextState(downTerm(getSecondTerm(R:Substitution?),error),A2)
      /= error .

ceq nextState-aux(S, A1 A2, i) = error
  if A1 /= none /\ A2 /= none /\
    metaMatch(upModule('VLRL-SAT+,true),
      '___['M1:State, 'M2:State],
      upTerm(S),
      nil,i) == noMatch .

ceq nextState-aux(S, A1 A2, i) = nextState-aux(S, A1 A2, s(i))
  if A1 /= none /\ A2 /= none /\
    R:Substitution? := metaMatch(upModule('VLRL-SAT+,true),
      '___['M1:State, 'M2:State],
      upTerm(S),
      nil,i) /\
    R:Substitution? /= noMatch /\
    (nextState(downTerm(getFirstTerm(R:Substitution?),error),A1)
      == error or
    nextState(downTerm(getSecondTerm(R:Substitution?),error),A2)
      == error) .

```

And finally, we also have the constant action `none`:

```

eq nextState(none,none) = none .
eq nextState(S,none) = error [owise] .

```

The idea behind the specification of `nextState` also applies to the operation `nextStateinContext`. Now, since the action can happen anywhere in the state and not only at the top, we use `metaXmatch` and `metaXapply` instead of `metaMatch` and `metaApply`. Because the same action can rewrite different parts of the state, the result of applying `nextStateinContext` won't be a single state in general, but a set of states (actually, a list in our implementation). Due to lack of space, we omit the specification, which can be obtained at <http://maude.sip.ucm.es/vlrl/metalevelprover>.

7 Some examples of properties

We illustrate some properties of the bank system for an initial bank configuration `S` with three accounts, `A-001`, `A-002` and `A-003`, three debit messages, a transfer order from `A-003` to `A-002`, and a new account message that we define in a module `VLRL-SAT+`. We want to decide, for a given formula `F`, whether `F` holds in `S` or not.

```

ops A-001 A-002 A-003 A-004 manager : -> Oid .
op bankConf : -> Configuration .
eq bankConf = < manager : Manager | none >
              < A-001 : Account | bal : 300 > debit(A-001, 150)

```

```

debit(A-001, 150) < A-002 : Account | bal : 250 >
debit(A-002, 400) < A-003 : Account | bal : 1250 >
(from A-003 to A-002 transfer 300)
new-account(manager, A-004, 1000).

```

Let us prove that a debit operation does not affect the existence of an account and that the balance of the account decreases in the debit amount. Notice that the action is done in context, since it only refers to the A-001 account and we have more terms in the state.

```

reduce in VLRL-SAT+ :
    bankConf |= << debit(A-001, 150, 300) >> (account?(A-001) = true /\
                                                balance?(A-001) = 150) .
result Bool: true

```

We can concurrently execute a debit operation on the A-001 account and create the new account A-004,

```

reduce in VLRL-SAT+ :
    bankConf |= << debit(A-001,150,300) new(manager,A-004,1000) >> True .
result Bool: true

```

but we cannot concurrently create a new account and make an operation on it:

```

reduce in VLRL-SAT+ :
    bankConf |= << debit(A-004,150,300) new(manager,A-004,1000) >> True .
result Bool: false

```

We cannot debit an account with more money than its current balance

```

reduce in VLRL-SAT+ : bankConf |= << debit(A-002, 400, 250) >> True .
result Bool: false

```

unless we first transfer some money to it:

```

reduce in VLRL-SAT+ :
    bankConf |= << transfer(A-003, A-002, 300, 1250, 250) >>
                << debit(A-002, 400, 250) >> True .
rewrites: 71 in 120ms cpu (120ms real) (591 rewrites/second)
result Bool: true

```

But note that both actions cannot be executed concurrently:

```

reduce in VLRL-SAT+ :
    bankConf |= << debit(A-002, 400, 250)
                transfer(A-003, A-002, 300, 1250, 250) >> True .
result Bool: false

```

We can also express concurrent actions by defining the part of the state in which they take place by means of the spatial operator. For example we can express a debit to the A-001 account and a transfer from account A-003 to the account A-002 by:

```

reduce in VLRL-SAT+ :
  bankConf |= < (< debit(A-001, 150, 300) > balance?(A-001) = 150)
              (<< transfer(A-003, A-002, 300, 1250, 250) >>
               balance?(A-002) = 550) > .
result Bool: true

```

The first action is done at the top while the second one is done in context. If we want to express all the actions at the top we can use the identity actions to fix the rest of the state. For example, in the following property we divide the state in three substates:

```

reduce in VLRL-SAT+ :
  bankConf |= < (< debit(A-001, 150, 300) > True)
              < (< new(manager, A-004, 1000) > True)
              (< [(((< A-003 : Account | bal : 1250 >
                  from A-003 to A-002 transfer 300)
                  debit(A-002, 400))
                < A-002 : Account | bal : 250 >)
                debit(A-001, 150)] > True) > > .
result Bool: true

```

8 Final remarks

The VLRL logic is a powerful logic for proving modal and temporal properties of systems specified in rewriting logic. It permits the specification of action and spatial properties that explore the structure of a system similar to the ones defined by the modal logic for mobile ambients of Cardelli and Gordon [3].

In this paper we have presented a method to check if a given VLRL formula holds in a given state that uses Maude's metalevel to decompose the state into its components and to execute the actions in a system. Our technique can be applied to arbitrary systems by slightly adapting to the particular operators at hand the description in this paper. Nevertheless, this description should already be widely applicable because distributed systems tend to be specified in rewriting logic with an associative and commutative operator as here; examples of a different kind can be found at maude.sip.ucm.es/vlrl/metalevelprover/.

Actually, and though not presented for the sake of generality, the definition of the operations `nextState` and `nextStateinContext` can be simplified for associative and commutative structure; in this case it is enough to distinguish the case in which the Σ -structure rule applies to an identity action from the case in which the rule applies to actions obtained by replacement. This definition is more efficient since it does not obtain all possible state decompositions, but it proceeds by matching some part of the state with some action and proving that the rest of the state fulfills the rest of the action.

Compared with the technique for proving VLRL formulae at the object level with the help of the LTL Maude model checker [10], this method does not require the transformation of the original theory but simply the definition of the

`nextState` and `nextStateinContext` operations for each function of the original signature. From the computational point of view, the use of the metalevel reduces drastically the number of rewrites needed to prove the formulae because the rewrite steps needed to simulate the execution of actions in the transformed theory are avoided. Nevertheless, in the examples we have run the time it takes to prove a formula is almost the same in both cases, due to the fact that computation at the metalevel is less efficient than computation at the object level, and the high performance of the LTL Maude model checker.

Acknowledgments. We thank Narciso Martí-Oliet for his helpful comments on a previous draft.

References

1. P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.
2. R. Bruni and J. Meseguer. Generalized rewrite theories. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Automata, Languages and Programming, ICALP 2003*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–256. Springer-Verlag, 2003.
3. L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *27th ACM Symposium on Principles of Programming Languages (POPL’00)*, pages 365–377, 2000.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.1). <http://maude.cs.uiuc.edu/manual/>, 2004.
6. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.
7. J. L. Fiadeiro, T. Maibaum, N. Martí-Oliet, J. Meseguer, and I. Pita. Towards a verification logic for rewriting logic. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques, WADT’99*, volume 1827 of *Lecture Notes in Computer Science*, pages 438–458. Springer-Verlag, 2000.
8. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
9. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
10. M. Palomino and I. Pita. Proving VLRL action properties with the Maude model checker. In N. Martí-Oliet, editor, *Fifth International Workshop on Rewriting Logic and its Applications, WRLA’04*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

Un Marco de Trabajo para la Construcción de Herramientas de Model Checking

Alejandro Carmona, María del Mar Gallardo, Jesús Martínez, Pedro Merino

Dpto. de Lenguajes y Ciencias de la Computación
University of Málaga, 29071 Málaga, Spain

Resumen Este artículo presenta trabajo en progreso para la obtención de un núcleo de código que permita la experimentación con estrategias para la detección automática de fallos en software para sistemas empujados y distribuidos. Se describe un marco de trabajo extensible que incorpora los algoritmos básicos para realizar model checking del código de sistemas concurrentes. El lenguaje interno empleado es suficientemente potente, y ampliable, como para permitir el análisis de diversos lenguajes de programación. El objetivo del trabajo es exportar nuestra experiencia en la combinación de model checking y abstracción del proyecto α SPIN al análisis de código fuente real.

1. Introducción

El empleo de herramientas automáticas o semiautomáticas para la detección de errores en el software es una necesidad paralela al desarrollo de la ingeniería del software. Uno de los enfoques más efectivos ha sido la aplicación de técnicas de verificación automática por exploración del espacio de estados, usualmente denominado “model checking” [4,5]. Esta técnica se ha aplicado normalmente al análisis de descripciones del software realizadas con técnicas formales, y su utilidad ya no se pone en duda (sobre todo a raíz del creciente número de errores software en proyectos de gran trascendencia). En este sentido, merece la pena destacar que ACM concedió en 2001 su premio al mejor sistema software al model checker SPIN [12]. Este galardón se ha concedido también a otros sistemas bien aceptados como UNIX, TeX, PostScript, SMALLTALK, TCP/IP, World-Wide-Web, TCL/TK o Java.

Sin embargo, son ya muchos los grupos de investigación que actualmente trabajan en su aplicación directa a código fuente usando lenguajes de programación habituales [8]. Un motivo para este cambio de orientación es la evolución de los algoritmos de model checking, y la disponibilidad de hardware más potente en los equipos de trabajo habituales de los desarrolladores. Ahora se pueden emplear estas herramientas automáticas sin incluir retrasos en el ciclo de producción de software. Otros motivos para el cambio de objetivo en model checking habría que buscarlos en las preferencias y en la formación de los desarrolladores de software que, en general, no aceptan el uso de notaciones adicionales para volver a describir la misma solución que ya tienen en su código.

En la línea de conseguir el model checking automático de lenguajes de programación hay diversos enfoques. Quizás el más natural consiste en traducir los programas a los formalismos que soportan los model checkers existentes, como SPIN o SMV [16], reutilizando la tecnología. Los proyectos Bandera [7] o FeaVer [15] siguen este enfoque. Otra alternativa es construir herramientas específicas para el código fuente, como hacen SLAM [1] o JPF [2]. Todas las herramientas citadas tienen su propia implementación de las tareas clásicas en model checking, como la generación del espacio de estados de forma eficiente.

Nuestra actividad en model checking, dentro del Grupo de Ingeniería del Software de la UMA, también está reorientándose en la línea de aplicarlos a lenguajes de programación. Para ello partimos de experiencias previas con model checkers académicos y comerciales. En particular, los hemos extendido con capacidades de abstracción automática, que es una de las principales características que debe abordar una herramienta para analizar código. En la herramienta α SPIN [10], extendemos SPIN con abstracción de datos para el lenguaje PROMELA. De esta forma conseguimos analizar especificaciones de sistemas complejos, sin que el usuario tenga que realizar la simplificación manualmente. En [11] se extiende STATEMATE para soportar abstracción de datos y de eventos. En los dos casos, se hace uso de transformaciones de código [9] para evitar la compleja tarea de modificar las implementaciones de las herramientas existentes. El enfoque para realizar la abstracción es similar a las facilidades de abstracción que implementa la herramienta Bandera, desarrollada en paralelo con nuestros trabajos. Sin embargo, hay notables diferencias en la forma de realizar la implementación.

Como fruto de estos trabajos previos, ahora se ha diseñado e implementado un marco de trabajo que incorpora los algoritmos necesarios y que es extensible para adaptarse a lenguajes de programación habituales y diferentes técnicas de optimización. El marco es suficientemente abierto como para poder capturar las particularidades de diferentes lenguajes de programación y para incluir nuevos algoritmos de análisis o de optimización (como la abstracción automática). De esta forma dispondremos del núcleo para la implementación de herramientas adecuadas a cada dominio de aplicación, y evitamos en gran medida las dificultades de extensión que tienen las herramientas clásicas (incluido SPIN). La implementación se ha realizado en Java, y la primera evaluación es prometedora en cuanto a consumo de recursos y tiempo de respuesta.

Resulta difícil comparar nuestra propuesta con enfoques anteriores para análisis de lenguajes de programación, ya que en casi todas las herramientas se prima la eficiencia para el lenguaje y el dominio de aplicación elegidos, y no la flexibilidad. Sin embargo, sí hay que destacar la existencia de otro proyecto que se desarrolla en paralelo en la Universidad de Kansas (donde se ha desarrollado Bandera). Se trata del marco Bogor [18], que está diseñado para hacer fácil la extensión de comprobadores de modelos con nuevos constructores para lenguajes de entrada y para optimizar los algoritmos de acuerdo con el dominio de aplicación. Un trabajo previsto es la comparación práctica de los dos marcos de trabajo para las mismas aplicaciones.

```

Proceso1::
m1: while true do
m2:  {seccion no critica 1}
m3:  c1 := 0;
m4:  wait until c2 = 1;
m5:  {seccion critica 1}
m6:  c1 := 1
m7: end;
Proceso2::
n1: while true do
n2:  {seccion no critica 2}
n3:  c2 := 0;
n4:  wait until c1 = 1;
n5:  {seccion critica 2}
n6:  c2 := 1
n7: end;

```

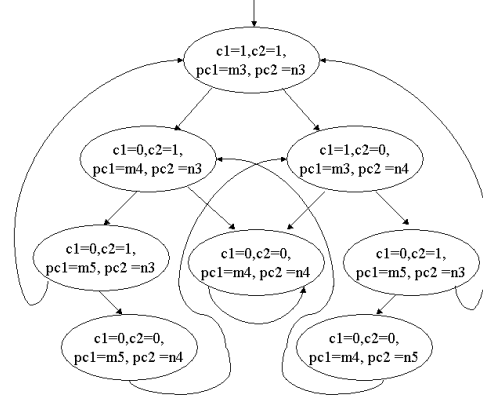


Figura 1. a) Exclusión mutua y b) Grafo de alcanzabilidad

El artículo continúa con una revisión de los fundamentos de model checking (Sección 2), que introduce los problemas a resolver. En la Sección 3 se describe el marco de trabajo, y en la Sección 4 se presenta la forma de ampliar sus funciones. La Sección 5 presenta las conclusiones y algunas líneas de trabajo futuro. Un Apéndice muestra una de las partes claves de la implementación: el mecanismo de exploración exhaustiva de estados.

2. Fundamentos de Model checking

Tres pasos son necesarios para analizar un sistema M mediante la técnica de *model checking*: (1) modelar el sistema utilizando algún *lenguaje de modelado*, (2) especificar las propiedades deseables f de M utilizando algún lenguaje de especificación, que suele ser alguna variante de la *lógica temporal*, (3) utilizar algún algoritmo de *model checking* para *comprobar* si el sistema diseñado es un *modelo* de la propiedad, lo que se denota con $M \models f$.

Los algoritmos de model checking se basan en la exploración exhaustiva del *espacio de estados* generado por el modelo M en busca de *contraejemplos* (trazas que violen la propiedad f). En esta sección se describen brevemente estos tres pasos.

2.1. Especificación del modelo

El model checking suele utilizarse para verificar sistemas concurrentes y distribuidos. Las estructuras de Kripke [5] son una abstracción de los lenguajes de modelado y permiten capturar de manera sencilla y elegante los aspectos temporales que más importan cuando se razona sobre este tipo de sistemas.

Sea AP un conjunto de *proposiciones atómicas*. Una *estructura de Kripke* M sobre AP es la tupla $M = (S, S_0, \rightarrow, L)$ donde S es un conjunto finito de estados, $S_0 \subseteq S$ es el conjunto de estados iniciales, $\rightarrow \subseteq S \times S$ es una relación de transición total ($\forall s \in S. \exists s' \in S$ tal que $(s, s') \in \rightarrow$) y $L : S \rightarrow 2^{AP}$ es una función que asocia a cada estado el conjunto de proposiciones atómicas ciertas en dicho estado.

Una *traza* de M a partir del estado $s \in S$ es una secuencia de estados $\pi = s_0 s_1 s_2 \dots$ tal que $s_0 = s$ y $\forall i \geq 0, (s_i, s_{i+1}) \in \rightarrow$. Se define $\mathcal{O}(M)$ como el conjunto de todas las trazas de M a partir de un estado de S_0 . Un estado s es *alcanzable* sii existe una traza a partir de un estado de S_0 que contiene a s .

Una estructura de Kripke $M = (S, S_0, \rightarrow, L)$ puede representarse de forma natural como un grafo dirigido $G(M) = (S', E, S_0)$, llamado *grafo de alcanzabilidad*, donde $S' \subseteq S$ es el subconjunto de los estados alcanzables, y $(s, s') \in E$ sii hay una transición $(s, s') \in \rightarrow$ en M . En el grafo, los estados iniciales de S_0 se representan con una flecha de entrada.

Las herramientas de model checking utilizan lenguajes de modelado muy cercanos a los lenguajes de programación. En la figura 1a se muestra como ejemplo el problema de la exclusión mutua descrito en un lenguaje imperativo. La estructura de Kripke asociada a este sistema puede obtenerse fácilmente considerando como conjunto de estados a $S = Bool \times Bool \times PC1 \times PC2$, donde las dos primeras componentes corresponden a los valores de las variables `c1` y `c2` y las dos últimas a los contadores de programa de los procesos.

La Figura 1b muestra el grafo de alcanzabilidad asociado. Es importante tener en cuenta que la eficacia de los algoritmos de model checking depende fuertemente de la representación interna que de este grafo haga la herramienta. El conjunto de estados considerado determina el tamaño del denominado *vector de estados*, que es la representación en memoria de cada estado. Es deseable que este tamaño sea el menor posible para evitar lo que se conoce como el *problema de la explosión de estados* que ocurre cuando el sistema a analizar es demasiado grande en comparación con los recursos disponibles.

2.2. Especificación de propiedades

Uno de los lenguajes de especificación más utilizados es la *lógica temporal* [17]. En la variante conocida como *lógica temporal lineal* (LTL), el tiempo se trata como si cada instante tuviera un único posible futuro, lo que la hace especialmente adecuada para describir el comportamiento de las trazas de ejecución definidas por una estructura de Kripke $M = (S, S_0, \rightarrow, L)$. En LTL, las fórmulas se construyen a partir del conjunto AP de proposiciones atómicas con los operadores Booleanos habituales y los operadores temporales $\bigcirc$ (*next*), $\Diamond$ (*eventually*), $\Box$ (*always*) y $\mathcal{U}$ (*until*).

Las fórmulas LTL se interpretan sobre trazas del tipo $\pi = s_0 s_1 s_2 \dots$. Así, dada una traza π y un instante $i \geq 0$, $\pi, i \models \varphi$ indica que la fórmula φ se satisface en el punto i de la computación π . Por ejemplo, si $\varphi \in AP$ entonces $\pi, i \models \varphi$ sii $\varphi \in L(s_i)$, $\pi, i \models \bigcirc \varphi$ sii $\pi, i+1 \models \varphi$ y $\pi, i \models \varphi_1 \mathcal{U} \varphi_2$ sii $\exists k \geq i. \pi, k \models \varphi_2$ y $\forall i \leq j < k. \pi, j \models \varphi_1$. El resto de los operadores temporales se define en

función de los anteriores. Por ejemplo, $\Diamond\varphi = \text{true } \mathcal{U} \varphi$, y $\Box\varphi = \neg\Diamond\neg\varphi$. Dada una fórmula φ , $\pi \models \varphi$ sii $\pi, 0 \models \varphi$, y $M \models \varphi$ sii $\forall \pi \in \mathcal{O}(M). \pi \models \varphi$.

2.3. Algoritmos de Model Checking

El model checking basado en autómatas es una de las técnicas más importantes de verificación automática y es el núcleo de las herramientas SPIN y SMV. Este método utiliza los *autómatas de Büchi* como lenguaje común al que se traducen tanto los modelos como las propiedades. Un autómata de Büchi es una tupla $A = (\Sigma, S, S_0, \rightarrow, L, F)$ donde Σ es el alfabeto finito, S es el conjunto finito de estados, $S_0 \subseteq S$ es el conjunto de estados iniciales, $\rightarrow \subseteq S \times S$ es la relación de transición, $L : S \rightarrow \Sigma$ es una función que etiqueta los estados y $F \subseteq S$ es el *conjunto de estados de aceptación*. Una *ejecución* de A sobre una palabra infinita $\omega = a_0a_1a_2\cdots$ es una secuencia $s_0s_1s_2\cdots$ tal que $\forall i. (s_i, s_{i+1}) \in \rightarrow$ y $L(s_i) = a_i$. Una ejecución $s_0s_1s_2\cdots$ es *de aceptación* sii pasa infinitas veces por algún estado de aceptación, es decir, sii $\exists s \in F$ tal que $\forall i \geq 0. \exists j \geq i$ y $s_j = s$. El autómata A acepta la palabra infinita ω sii existe una ejecución de aceptación de A sobre w . El lenguaje de las palabras infinitas aceptadas por el autómata A se denota con $\mathcal{L}(A)$. La relación de los autómatas de Büchi y la lógica temporal viene de la siguiente propiedad [5]: Dada una fórmula LTL φ y una estructura de Kripke $M = (S, S_0, \rightarrow, L)$ sobre AP , el autómata de Büchi $A_\varphi = (2^{AP}, S, S_0, \rightarrow, L, F)$ verifica que $\mathcal{L}(A_\varphi) = \{\omega \mid \exists \pi \in \mathcal{O}(M), \pi \text{ es una ejecución de aceptación de } A \text{ sobre } \omega \text{ y } \pi \models \varphi\}$. Por otro lado, una estructura de Kripke $M = (S, S_0, \rightarrow, L)$ sobre AP se convierte de manera sencilla en el autómata $A_M = (2^{AP}, S, S_0, \rightarrow, L, S)$ en el que todos los estados del modelo son de aceptación.

Esta relación permite reducir el problema de comprobar si $M \models \varphi$ a los tres pasos siguientes: (1) construir el autómata $A_{\neg\varphi}$, que corresponde a la negación de la fórmula φ , (2) construir el autómata intersección $A_{M, \neg\varphi}$ y (3) comprobar si el lenguaje $\mathcal{L}(A_{M, \neg\varphi})$ es vacío. Si lo es, entonces $M \models \varphi$, en otro caso, si existe $\omega \in \mathcal{L}(A_{M, \neg\varphi})$, entonces $M \not\models \varphi$ y, además, la traza π de $A_{M, \neg\varphi}$ sobre ω es un *contraejemplo* de la propiedad, es decir, una traza de M que no satisface φ . La ventaja de este método es que una vez que se ha construido $A_{\neg\varphi}$, existen algoritmos muy eficientes [6] para construir el autómata intersección $A_{M, \neg\varphi}$ y para comprobar si $\mathcal{L}(A_{M, \neg\varphi})$ es vacío. Muchas otras técnicas de compactación permiten representar sistemas con un gran número de estados [13,14].

3. El marco de trabajo

El objetivo principal de este marco es facilitar la incorporación de nuevas técnicas de simulación y análisis a los programadores. Para ello, se suministran mecanismos de extensión basados en plugins, con los que se incorporan nuevas capacidades o, en algún caso, reemplazan parte de la funcionalidad suministrada por defecto en el núcleo del marco.

El conjunto de funcionalidades básicas que aporta el núcleo del marco son:

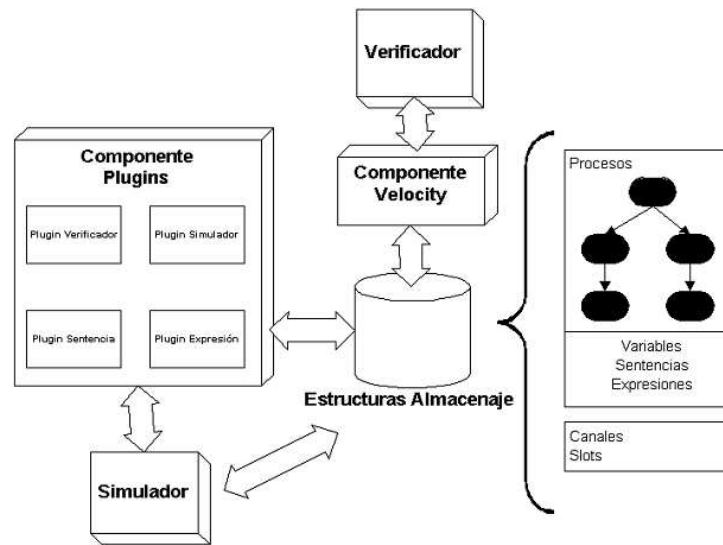


Figura 2. Arquitectura del marco de trabajo

1. Operaciones para manipular procesos (creación, destrucción)
2. Manipulación de sentencias (gotos, print, asignaciones, asertos,...)
3. Manipulación de expresiones (aritmético-lógicas y a nivel de bit)
4. Conjunto de simuladores con distintos modos de ejecución (aleatoria, guiada, primera disponible)
5. Verificador de propiedades de seguridad, basado en el algoritmo de búsqueda en profundidad DFS

Así, el marco será utilizado para crear un model checker específico (una aplicación Java), que ejecutará tareas de simulación o verificación cuyos componentes son suministrados de forma interna o externa a través de los mecanismos de extensión.

El marco de trabajo se encuentra estructurado en cinco componentes que interactúan entre sí. Estos componentes se muestran en la figura 2.

Componente Simulador: El simulador es la parte del marco de trabajo que, a partir de un modelo, se encarga de simular una traza de ejecución. El usuario puede escoger entre distintos tipos de simulación, tal y como se verá más adelante.

Componente Velocity: Velocity es un lenguaje de plantillas y un motor que genera código en tiempo de ejecución. En el marco, este componente genera las partes dinámicas de código Java pertenecientes a un verificador, y relacionadas con la definición de tipos de procesos o canales de comunicación.

Componente Verificador: El verificador es el encargado de realizar el análisis de alcanzabilidad y el chequeo de propiedades. De esta forma, manipula los vectores de estado, genera, almacena estados e intercambia información con los observadores de propiedades que se hayan suministrado (seguridad, viveza,...).

Componente Estructuras de Almacenaje: Los modelos de entrada del sistema, introducidos por el usuario, se dividen en procesos y se almacenan en una serie de estructuras de datos de forma sean accesibles por cualquier otro componente del marco. Los procesos están compuestos por secuencias de sentencias (o acciones), representados por máquinas de estado finitas. Las acciones se almacenan en las transiciones de estado. Junto con esta información, los procesos guardan también sus variables locales. Además de los procesos del sistema, también son accesibles los canales de comunicación y las variables globales al modelo.

Componente Plugins: Este componente del marco es el encargado de recoger las distintas funcionalidades implementadas por los usuarios e integrarlas en el sistema en tiempo de ejecución. Estas nuevas funcionalidades una vez cargadas, serán integradas con el marco de forma que el usuario pueda hacer uso de ellas tanto en sus modelos como en las simulaciones y verificaciones. Los distintos tipos de plugins admitidos por el marco son:

1. *Plugin verificador:* implementa un verificador que, en caso de que el usuario así lo requiera, pueda analizar una propiedad determinada sobre un modelo.
2. *Plugin simulador:* implementa un nuevo componente encargado de la simulación de un modelo, incluyendo el método de selección de una transición ejecutable de entre las disponibles en un momento de ejecución concreto.
3. *Plugin de definición de acciones:* Se pueden añadir tanto sentencias como expresiones (unarias, binarias o ternarias). Las acciones han de definir su comportamiento frente a un simulador y a un verificador en sus métodos correspondientes.

3.1. Representación interna del modelo

Los modelos de entrada son adaptados al marco de trabajo mediante un proceso de transformación que consiste en convertir toda la información del modelo a las estructuras de datos del núcleo del marco (autómatas). Este proceso analiza el modelo de entrada y lo divide en procesos. El conjunto de procesos del sistema contiene como mínimo uno de inicialización que debe existir en todos los modelos, y que se denomina INIT. Los procesos se estructuran almacenando todas sus instrucciones ejecutables. Para ello, cada proceso se representa con un autómata finito indeterminista que permite simular todas sus posibles ejecuciones. Las instrucciones se almacenan en las transiciones del autómata, y constan de guarda y sentencia. La guarda es la encargada de comprobar si la sentencia asociada es ejecutable. Si una transición no tiene guarda, se considera ejecutable. Cualquier expresión se estructura internamente en forma infija mediante árboles binarios y su evaluación se realiza mediante la evaluaciones recursivas de las ramas de dicho árbol. Las variables locales a un proceso, también se almacenan dentro de éste, así como las referencias a las variables globales usadas. Una vez dividido el modelo en procesos, se identifican los canales, que son los mecanismos de comunicación entre procesos y por lo tanto son globales al marco. Igualmente se identifican sus celdas (tipos de mensaje). Por último, se obtienen las variables globales del modelo y se inicializan.

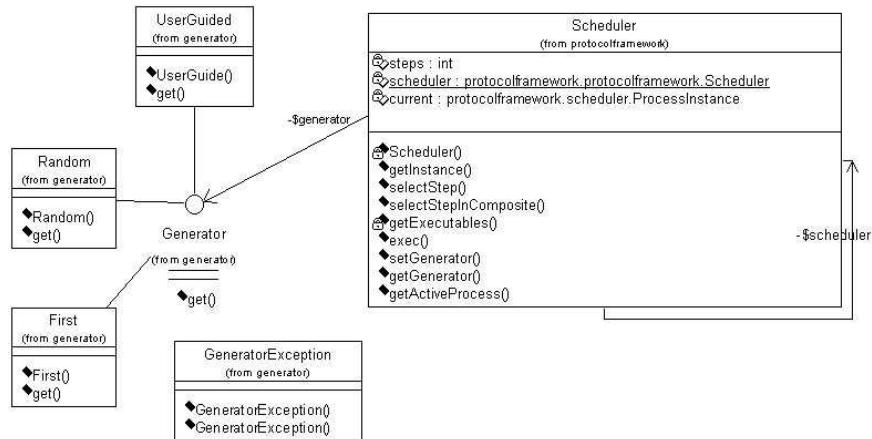


Figura 3. Estructura de clases del Simulador

3.2. Simulador

El simulador se encuentra estructurado de forma que pueda ser ejecutado por el usuario en cuanto el modelo de entrada está cargado completamente en las estructuras de árboles y los plugins se han inicializado e integrado en el marco. En principio, la clase principal que controla la simulación es la clase Scheduler (ver Figura 3) que es la encargada, por una parte, de la inicialización del Simulador escogido por el usuario y, por otra, de la interacción con los procesos del modelo para recabar información en cada paso de ejecución de la simulación. Esta información abarca desde recuperar las transiciones ejecutables de cada proceso hasta la ejecución de una de ellas.

La clase de simulación se crea a partir de una interfaz denominada Generator, que actúa como base para todos los simuladores. Esta interfaz únicamente obliga a implementar un método `get()` que, recibiendo como parámetro un conjunto de transiciones ejecutables, sea capaz de recuperar la que se va a ejecutar a continuación. La forma en que esta transición se selecciona depende del plugin de simulación escogido. En la versión inicial del marco se encuentran disponibles los siguientes plugins de simulador:

1. *Random*. Simulador que escoge aleatoriamente una transición de entre las ejecutables.
2. *First*. Plugin que escoge por defecto la primera transición ejecutable de la lista.
3. *UserGuided*. Este plugin muestra al usuario por pantalla las transiciones ejecutables, para que escoja la siguiente a ejecutar.

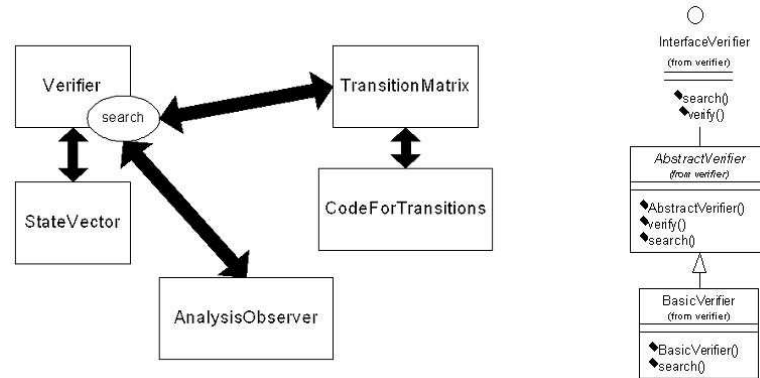


Figura 4. a) Estructura del Verificador y b) Plugin asociado

3.3. Verificador

El componente encargado de analizar propiedades sobre el modelo funciona internamente como sigue. El método `search()` del verificador (Verifier en la Figura 4a) va seleccionando las transiciones ejecutables en cada momento, haciendo uso de un vector de estados que tiene almacenado el contenido de canales y los valores de variables (globales o locales), así como el estado en el que se encuentra cada proceso. Las transiciones se encuentran en la clase `TransitionMatrix`, con información para la ejecución de acciones o para deshacer dichos cambios (backtracking). El código generado por cada transición se codifica en la clase `CodeForTransitions`. El fin del análisis (para cada paso de ejecución) se controla a través de la clase `AnalysisObserver`, que permite detectar errores, lanzando la excepción adecuada. Actualmente, el verificador básico incorporado en el marco proporciona análisis de seguridad (bloqueos, asertos y estados no alcanzados). Un nuevo plugin con un método `search()` y un `AnalysisObserver` modificados convenientemente, podrían analizar fórmulas LTL usando autómatas de Büchi (una de las posibles ampliaciones del marco).

4. Ampliando el marco

La introducción de nuevos plugins en el model checker se realiza mediante la redefinición de la clase `AdapterPlugin` (fig. 4b). A través de la misma se realizará el registro de las clases que implementen la nueva funcionalidad. Como ejemplo, la creación de un plugin de verificación se haría implementando el interfaz `InterfaceVerifier`, o una extensión de la clase `AbstractVerifier`. El programador ha de redefinir el método `search()`, responsable de realizar la generación de estados del modelo ayudado por las clases auxiliares mostradas en la fig. 4a). Los criterios de finalización de búsqueda los marca el objeto apropiado de la clase `AnalysisObserver`. Todas las clases que pertenecen a un nuevo plugin se empaquetan en un archivo de extensión `.jar`.

5. Conclusiones y trabajo futuro

A medida que aumenta la aplicación de técnicas de model checking en la verificación de software concurrente y distribuido, se hace patente la necesidad de crear herramientas orientadas tanto al lenguaje de programación analizado, como a las distintas estrategias de chequeo que se asuman óptimas en determinados escenarios. Este trabajo ha presentado un marco de trabajo orientado a objetos que permite la creación de model checkers a medida, usando mecanismos de extensión mediante plugins. El resultado permite la simulación y verificación de sistemas de forma independiente al lenguaje en el que fueron descritos. El beneficio del uso de plugins es bien conocido entre los programadores de entornos de desarrollo comerciales, o herramientas CASE: la herramienta resultante incorpora nueva funcionalidad a la existente hasta el momento. Nuestro marco es flexible para incorporar nuevas estrategias de simulación y verificación, aislando al programador de la complejidad de análisis del código fuente del núcleo.

El marco será utilizado próximamente para chequear propiedades de seguridad en software cliente/servidor desarrollado en C. Como futuros plugins, se plantean la incorporación de verificadores para propiedades de viveza, así como técnicas de optimización (orden parcial, supertraza y abstracción de datos).

Referencias

1. T. Ball and S. K. Rajamani, The SLAM project: debugging system software via static analysis. In *Proc. of POPL 2002*, pp. 1-3, (2002).
2. G. Brat, K. Havelund, S. Park and W. Visser, Java PathFinder - A second generation of a Java model checker. In *Proc. of the Post-CAV'00 Workshop on Advances in Verification*. (2000)
3. E.M. Clarke, O. Grumberg and D.E. Long. Model Checking and Abstraction, *ACM TOPLAS*, 16(5), (1994), 1512–1542.
4. E.M. Clarke, E. A. Emerson and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications, *ACM TOPLAS*, 8(2), 1986.
5. E. Clarke, O. Grumberg and D. Peled. *Model Checking* (MIT Press, 2000).
6. K. Etessami and G. Holzmann. Optimizing Buchi automata, in *Proc. of 11th Int. Conf. on Concurrency Theory*, pp. 153-167, (2000)
7. J. Hatcliff, M. Dwyer, C. Pasareanu and Robby. Foundations of the Bandera Abstraction Tools. *The Essence of Computation*, LNCS-2566, 172-203, (2003).
8. K. Havelund and W. Visser Program Model Checking as a new trend. *Software Tools for Technology Transfer (STTT)* 4:8-20, (2002).
9. M.M. Gallardo and P. Merino. A Framework for Automatic Construction of Abstract PROMELA Models. In *Theoretical and Practical Aspects of SPIN Model Checking*, LNCS-1680, pp. 184-199, (1999).
10. M.M. Gallardo, J. Martinez, P. Merino, E. Pimentel. aSPIN: A Tool for Abstract Model Checking. *International Journal on Software Tools for Technology Transfer*, 5 (2-3), pp. 165 - 184, 2004.
11. M.M. Gallardo, J. Martinez, P. Merino, E. Pimentel. Abstracting UML behavioral diagrams for verification. Chapter in In Hongji Yang (ed.), *Software Evolution with UML and XML*, IDEA Group Publishing, 2004, to appear.
12. G.J. Holzmann. The SPIN Model Checker. Addison-Wesley, Boston, USA, 2003.
13. G.J. Holzmann. An analysis of bitstate hashing. In *Formal Methods in Systems Design* (1998)
14. G.J. Holzmann and D. Peled. An Improvement in Formal Verification. *Proc. FORTE Conference*, Bern, Switzerland (1994)
15. G.J. Holzmann and M. H. Smith. A Practical Method for the Verification of Event Driven Systems. In *Proc. of ICSE99*, pp. 597-608, (1999).
16. K.L. McMillan. Symbolic Model Checking: An approach to the State Explosion Problem. Kluwer Academic, (1993).
17. Z. Manna and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, New York, NY, USA, 1992.
18. Robby, M. Dwyer, J. Hatckiff. Bogor: An Extensible and Highly-Modular Software Model Checking Framework. In *Proc. of ESEC/FSE'03* (2003).

Phrase Similarity through Approximate Tree Matching

F.J. Ribadas, M. A. Alonso, and D. Cabrero

Computer Science Dept., Univ. of Vigo	Computer Science Dept., Univ. of A Coruña
Campus As Lagoas, s/n.	Campus de Elviña, s/n.
32004 Orense, Spain	15071 A Coruña, Spain
<code>ribadas@uvigo.es</code>	<code>alonso@udc.es</code> , <code>cabrero@udc.es</code>

Abstract. We describe an algorithm to measure the similarity between phrases, integrating both the edit distance between trees and single-term similarity techniques, and allowing the pattern to be defined approximately, omitting some structural details. Such a technique is of interest in a variety of applications, such as information extraction/retrieval or question answering, where error-tolerant recognition allows incomplete sentences to be integrated in the computation process.

1 Introduction

Although similarity between documents is phrase-based, the approaches applied can be very different. While some authors do not take into account syntactic information, considering a vector space model, a latent semantic strategy, a semantically-based word-word proximity [6] or a combination of them; other authors incorporate syntactic [5] or structural information [4], which allows to introduce pattern-matching as operational mechanism to deal with. At this point, document similarity based on matching phrases seems to have a more significant effect on the quality and effectiveness of the resulting measures, due to its robustness in dealing with noisy terms [1]. So, finding phrase similarity models is a central question in dealing with document similarity measures.

In this context, the need for effective phrase similarity measures is growing as a result of the increasing amount of information available on-line. Most efforts have been targeted toward single-terms based techniques [2]. The work about the combination of parse structures and single-terms is limited, and nothing at all related to dealing with incomplete or vague structures. Our proposal seeks to provide these capabilities. In this sense, we look for integrate *variable length don't care* (VLDC) technology [9], initially developed to compare parse tree structures, in the definition of new phrase similarity proposals.

2 The edit distance

Given P , a pattern tree, and D , a data tree, whose number of nodes is, respectively, $|P|$ and $|D|$, we define an *edit operation* as a pair $a \rightarrow b$, $a \in$

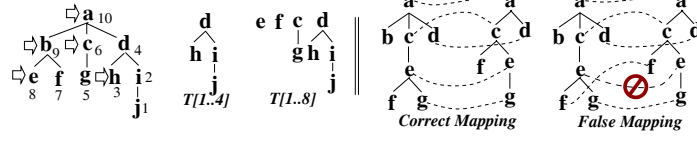


Fig. 1. Inverse postorder and examples of mappings

$\text{labels}(P) \cup \{\varepsilon\}$, $b \in \text{labels}(D) \cup \{\varepsilon\}$, $(a, b) \neq (\varepsilon, \varepsilon)$, where ε represents the empty string and $\text{labels}(T)$ the set of symbols labeling nodes in a tree T . We can delete a node ($a \rightarrow \varepsilon$), insert a node ($\varepsilon \rightarrow b$), and change a node ($a \rightarrow b$), as shown in Fig. 2. Each edit operation has a cost, $\gamma(a \rightarrow b)$, which satisfies the three properties of a metric. That is, it gives non negative values, $\gamma(a \rightarrow b) \geq 0$ and $\gamma(a \rightarrow a) = 0$, it is symmetrical, $\gamma(a \rightarrow b) = \gamma(b \rightarrow a)$ and it holds with the triangular inequality, $\gamma(a \rightarrow b) \leq \gamma(a \rightarrow c) + \gamma(c \rightarrow b)$. We extend γ to a sequence S of edit operations $s_1, s_2, \dots, s_n$ by letting $\gamma(S) = \sum_{i=1}^{|S|} (\gamma(s_i))$. The distance between P and D is defined by:

$$\delta(P, D) = \min\{\gamma(S) \mid S \text{ editing sequence taking } P \text{ to } D\}$$

Given an inverse postorder traversal, as is shown in the left-hand side of Fig. 1, to name each node i of a tree T by $t[i]$, a *mapping* from P to D is a triple (M, P, D) , where M is a set of integer pairs (i, j) satisfying, for each $1 \leq i_1 \leq |P|$, $1 \leq i_2 \leq |P|$ and $1 \leq j_1 \leq |D|$, $1 \leq j_2 \leq |D|$:

$$\begin{array}{ll} i_1 = i_2 & \text{if } j_1 = j_2 \\ p[i_1] \text{ is at the left of } p[i_2] & \text{if } d[j_1] \text{ is at the left of } d[j_2] \\ p[i_1] \text{ is an ancestor of } p[i_2] & \text{if } d[j_1] \text{ is an ancestor of } d[j_2] \end{array}$$

which corresponds to one-to-one assignation, sibling order preservation and ancestor order preservation; as is shown in the right-hand side of Fig. 1.

The cost, $\gamma(M)$, of a mapping (M, P, D) is computed from relabeling, deleting and inserting operations, as follows:

$$\gamma(M) = \sum_{(i,j) \in M} \gamma(p[i] \rightarrow d[j]) + \sum_{i \in \mathcal{D}} \gamma(p[i] \rightarrow \varepsilon) + \sum_{j \in \mathcal{I}} \gamma(\varepsilon \rightarrow d[j])$$

where $\mathcal{D}$ and $\mathcal{I}$ are, respectively, the nodes in P and D not affected by pairs in M . Tai [7] proves, given trees P and D , that $\delta(P, D) = \min\{\gamma(M) \mid M \text{ mapping from } P \text{ to } D\}$.

We have chosen to compute a modified edit distance, following Zhang *et al.* [11], where the data tree can be simplified by removing some irrelevant subtrees with no associated cost. We introduce $r(i)$ as the rightmost leaf descendant of the subtree rooted at $t[i]$ in a tree T , and $T[i..j]$ as the ordered subforest of T induced by the nodes numbered i to j , inclusive, as is shown in the left-hand side of Fig. 1. In particular, we have $T[r(i)..i]$ as the tree rooted at $t[i]$, that we denote, in short, as $T[i]$. We also introduce $r_keyroots(T)$ as the set of all nodes in T which have a right siblings plus the root, $root(T)$, of T , indicated by arrows in Fig. 1.

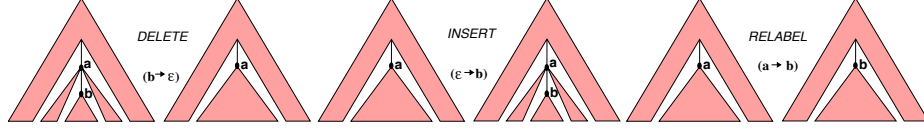


Fig. 2. Edit operations on trees

Cutting at node $t[j]$ means removing the subtree $T[j]$. A set $\mathcal{S}$ of nodes of $T[i]$ (resp. $T[i..j]$) is said to be a *set of consistent subtree cuts* in $T[i]$ (resp. $T[i..j]$) iff: (i) $t[k] \in \mathcal{S}$ implies that $t[k]$ is a node in $T[i]$ (resp. $t[k]$ is a node in $T[i..j]$), that is $i \leq k \leq j$, and (ii) $t[k], t[l] \in \mathcal{S}$ implies that neither is an ancestor of the other in $T[i]$ (resp. $T[i..j]$). We denote by $cut(T, \mathcal{S})$ (resp. $cut(T[i..j], \mathcal{S})$) the tree T (resp. subforest $T[i..j]$) with subtree removals at all nodes in $\mathcal{S}$, and we denote by $subtrees(T)$ (resp. $subtrees(T[i..j])$) the set of all possible sets of consistent subtree cuts in T (resp. $T[i..j]$). We illustrate these concepts in Fig. 3. Formally, given trees P and D , the *edit distance with cuts* is defined as

$$\delta_{\text{cut}}(P, D) = \min\{\delta(P, cut(D, S)) \mid S \in subtrees(D)\}$$

That is, δ_{cut} is the edit distance between the pattern and the data trees resulting from an optimal set of consistent cuts that yields a minimal distance. We now define the *forest edit distance with cuts* between a target subforest $P[s_1..s_2]$ and a data subforest $D[t_1..t_2]$, generalizing δ_{cut} in the form

$$\begin{aligned} fd(P[s_1..s_2], D[t_1..t_2]) &= \delta_{\text{cut}}(P[s_1..s_2], D[t_1..t_2]) = \\ &= \min\{\delta(P[s_1..s_2], cut(D[t_1..t_2], S)) \mid S \in subtrees(D[t_1..t_2])\} \end{aligned}$$

denoted $fd(s_1..s_2, t_1..t_2)$ when the context is clear. This computes the distance with cuts between nodes $p[s_2]$ and $d[t_2]$ in the context of their right siblings in the corresponding trees; while the tree distance $\delta_{\text{cut}}(P[s_2], D[t_2])$, denoted $td(s_2, t_2)$ in short, is computed only from their descendants. We compute the tree edit distance $td(\text{root}(P), \text{root}(D))$ in a bottom-up fashion, first determining mappings from all leaf r_keyroots, then for all r_keyroots at the next higher level, and so on to the root, applying, for each pair of r_keyroots, the formulae:

$$fd(\emptyset, \emptyset) = 0 \quad fd(\emptyset, r(j)..t) = 0 \quad fd(r(i)..s, \emptyset) = fd(r(i)..s-1, \emptyset) + \gamma(p[s] \rightarrow \varepsilon)$$

$$fd(r(i)..s, r(j)..t) = \min \begin{cases} fd(r(i)..s-1, r(j)..t) + \gamma(p[s] \rightarrow \varepsilon) \\ fd(r(i)..s, r(j)..t-1) + \gamma(\varepsilon \rightarrow d[t]) \\ fd(r(i)..s-1, r(j)..t-1) + \gamma(p[s] \rightarrow d[t]) \\ fd(r(i)..s, \emptyset) \end{cases}$$

if $r(s) = r(i)$ **and** $r(t) = r(j)$

$$fd(r(i)..s, r(j)..t) = \min \begin{cases} fd(r(i)..s-1, r(j)..t) + \gamma(p[s] \rightarrow \varepsilon) \\ fd(r(i)..s, r(j)..t-1) + \gamma(\varepsilon \rightarrow d[t]) \\ fd(r(i)..r(s)-1, r(j)..r(t)-1) + td(s, t) \\ fd(r(i)..s, r(j)..r(t)-1) \end{cases}$$

otherwise

where nodes $s \in P[i]$, $t \in D[j]$, and $i \in r_keyroots(P)$ and $j \in r_keyroot(D)$.

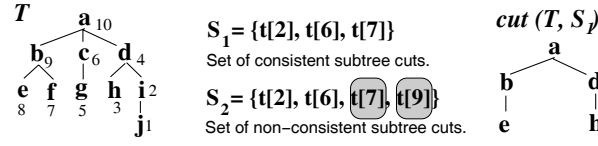


Fig. 3. An example of consistent subtree cuts on trees

We have three sets of formulae, the first one for initialization and the other two for distances between pairs of subtrees and for distances between pairs of actual subforests. In each step, every way to obtain the edit distance is computed, one for each edit operation and one for subtree cut, and the one with a minimum value is recorded. To compute $td(P, D)$ it is sufficient to take into account that

$$td(P, D) = fd(r(\text{root}(P)).. \text{root}(P), r(\text{root}(D)).. \text{root}(D))$$

Time complexity is, in the worst case,

$$\mathcal{O}(|P| \times |D| \times \min(\text{depth}(P), \text{leaves}(P)) \times \min(\text{depth}(D), \text{leaves}(D)))$$

where $|T|$, $\text{depth}(T)$ and $\text{leaves}(T)$ are the number of nodes, the depth and the number of leaves in a tree T ; respectively.

3 Approximate matching with VLDC symbols

We support the inclusion of *variable length don't care* (VLDC) symbols in the pattern tree, which allows us to omit some structural details in pattern specification and manage more general pattern trees. In this work we consider two different definitions of VLDC matching [11]:

- The VLDC substitutes part of a path from the root to a leaf of the data tree. We represent this, shown in Fig. 4, with a "||", and call it a *path*-VLDC.
- The VLDC matches part of such a path and all the subtrees emanating from the nodes of that path, except possibly at the lowest node of that path. At the lowest node, the VLDC symbol can substitute for a set of leftmost subtrees and a set of rightmost subtrees. We call this an *umbrella*-VLDC, and represent it with a "^", as shown in Fig. 4.

To formalize the use of pattern trees with VLDC symbols requires us to introduce the notion of *VLDC-substitution*. These VLDC-substitutions will assign to each VLDC node in the pattern tree P a set of nodes taken from the data tree D according to the definition given above for each kind of VLDC symbol. So, given a data tree D and a VLDC-substitution σ on P , and being $\sigma(P)$ the result of applying this substitution σ to the VLDC nodes in P , we define:

$$\delta_{VLDC}(P, D) = \min\{\delta_{\text{cut}}(\sigma(P), D) \mid \sigma \in \mathcal{S}\}$$

where $\mathcal{S}$ is the set of all possible VLDC-substitutions.

As a consequence, since the final distance is computed using the tree resulting from an optimal VLDC-substitution, no cost will be induced by these substitutions. To compute the edit distance when node $p[s]$ is either "|" or "^", only the formula for tree-to-tree distances must be adapted. We first assume that $p[s]$ is "|", when $r(i) = r(s)$ and $r(j) = r(t)$, we have that:

$$fd(r(i)..s, r(j)..t) = \min \begin{cases} fd(r(i)..s - 1, r(j)..t) + \gamma(p[s] \rightarrow \varepsilon) \\ fd(r(i)..s, r(j)..t - 1) + \gamma(\varepsilon \rightarrow d[t]) \\ fd(r(i)..s - 1, r(j)..t - 1) + \gamma(p[s] \rightarrow d[t]) \\ fd(\phi, r(j)..t - 1) + \min_{t_k} \{td(s, t_k) - td(\emptyset, t_k)\} \quad 1 \leq k \leq n_t, \\ fd(r(i)..s, \emptyset) \end{cases}$$

where nodes $d[t_1], d[t_2], \dots, d[t_{n_t}]$ are the n_t children of $d[t]$. When $p[s]$ is "^", the formula is the following:

$$fd(r(i)..s, r(j)..t) = \min \begin{cases} fd(r(i)..s - 1, r(j)..t) + \gamma(p[s] \rightarrow \varepsilon) \\ fd(r(i)..s, r(j)..t - 1) + \gamma(\varepsilon \rightarrow d[t]) \\ fd(r(i)..s - 1, r(j)..t - 1) + \gamma(p[s] \rightarrow d[t]) \\ \min_{t_k} \{td(s, t_k)\} \quad 1 \leq k \leq n_t, \\ \min_{t_k} \{sfd(r(i)..s - 1, r(j)..t_k)\} \quad \leq k \leq n_t \end{cases}$$

with nodes $d[t_k]$, $1 \leq k \leq n_t$ being the children of $d[t]$. As can be seen, in this formula we need to use an auxiliary distance to deal with $\wedge$ -VLDC. We define the *suffix forest distance* between F_P and F_D , forests in the pattern P and the data tree D respectively, denoted $sfd(F_P, F_D)$, as the distance between F_P and $\bar{F}_D$, where $\bar{F}_D$ is a subforest of F_D with some consecutive complete subtrees, all having the same parent, removed from the right. Formally, we have that

$$sfd(F_P, F_D) = \min_{\bar{F}_D} \{fd(F_P, \bar{F}_D)\}$$

and it can be proved [11] that sfd values can be computed as follows:

$$\begin{aligned} sfd(\emptyset, \emptyset) &= 0 \\ sfd(\emptyset, r(j)..t) &= \begin{cases} 0 & \text{if } r(t) = r(j) \text{ or } r(\text{parent}(t)) = r(j) \\ sfd(\emptyset, r(j)..t - 1) + \gamma(\varepsilon \rightarrow d[t]) & \text{otherwise} \end{cases} \\ sfd(r(i)..s, \emptyset) &= fd(r(i)..s, r(j)..t) \\ sfd(r(i)..s, r(j)..t) &= \min \begin{cases} fd(r(i)..s, \emptyset) \\ fd(r(i)..s, r(j)..t) \end{cases} \\ \text{if } r(s) &= r(i) \\ sfd(r(i)..s, r(j)..t) &= \min \begin{cases} sfd(r(i)..s - 1, r(j)..t) + \gamma(p[s] \rightarrow \varepsilon) \\ sfd(r(i)..s, r(j)..t - 1) + \gamma(\varepsilon \rightarrow d[t]) \\ sfd(r(i)..r(s) - 1, r(j)..r(t) - 1) + td(s, t) \end{cases} \\ \text{otherwise} & \end{aligned}$$

4 A semantical extension

We extend the VLDC matching algorithm in a similarity parse tree measure taking into account the semantic proximity between words. In essence, we propagate a similarity measure at word level through the nodes in accordance with the syntactic distances computed by tree matching.

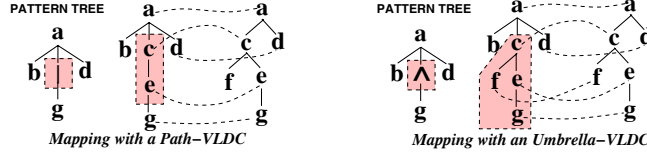


Fig. 4. Mapping with VLDC symbols

4.1 Semantic similarity at word level

We follow Lin's work [2], based on the use of the WORDNET [3] taxonomy, a computer dictionary that organizes its contents using semantic relationships between structures called *synsets*. A *synset* is a set of words sharing the same meaning. The semantic relationships are established between pairs of synsets, being the hyperonymy, i. e. the classical *is-a* relationship between a general concept and another more specific one, the most important one.

Lin assumes that the WORDNET taxonomy is a tree, and his approach is based on the information content of the synsets. Given two words, w_1 and w_2 , belonging to the synsets S_1 and S_2 , respectively, let $P(S_i)$ be the probability that a randomly selected word from WORDNET belongs to synset S_i or one of its more specific synsets. If S_0 is the most specific synset that subsumes both S_1 and S_2 according to the hyperonymy relation, the similarity of w_1 and w_2 is:

$$sim_{Lin}(w_1, w_2) = \frac{2 \times \log P(S_0)}{\log P(S_1) + \log P(S_2)}$$

The resulting measure gives values in $[0, 1]$, with values closest to 1 meaning a high semantical similarity, and values closest to 0, the greatest dissimilarity.

We use Lin's measure as a basis to compute the semantic cost, γ_{sem} , that we have to associate with the edit operations applied to the leaves of the trees we are comparing. Since we are dealing with a cost function, we must reverse the interpretation of the numerical values offered by this measure. Again, we denote the *insertion* of a word w_j as $\varepsilon \rightarrow w_j$, and the *deletion* of a word w_i as $w_i \rightarrow \varepsilon$.

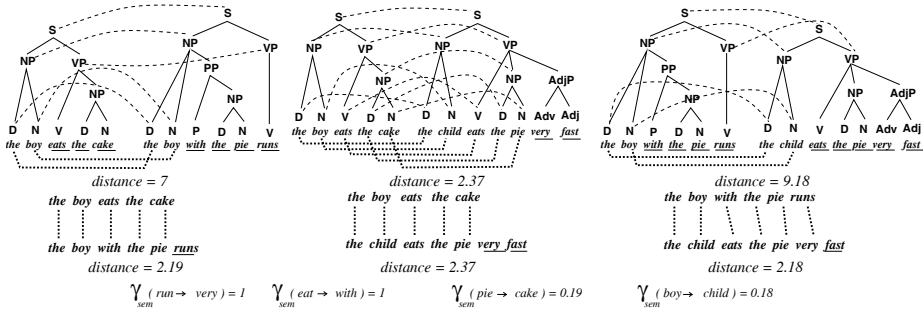


Fig. 5. String matching and tree matching mapping between words.

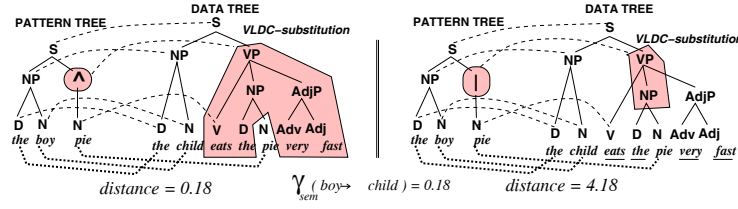


Fig. 6. Semantic distance with VLDC symbols

So, we compute $\gamma_{\text{sem}}(w_i \rightarrow w_j)$, as follows:

$$\gamma_{\text{sem}}(w_i \rightarrow w_j) = \begin{cases} 1 & \text{if } w_i = \varepsilon \text{ or } w_j = \varepsilon \\ 1 - \text{sim}_{\text{Lin}}(w_i, w_j) & \text{if } w_i \neq \varepsilon \text{ and } w_j \neq \varepsilon \end{cases}$$

When two words are close following Lin's measure, the cost will be near 0; and with non-similar words, the cost will be close to 1. This definition fulfills the properties of a metric.

4.2 Sentence similarity based on edit distance

Once we have established a way to measure the cost of the edit operations involving leaves, we extend it in order to compute a distance between pairs of whole sentences, taking into account both lexical and syntactic information.

As a first approach we could use a variation of the classical string matching algorithm by Wagner and Fischer [10] to locate the operations to be applied over the words in the two sentences. The algorithm will identify the sequence of modifications needed to obtain one of them from the other with a minimum cost. We will simply treat each word in the sentences as if it were a character, measuring the cost of the edit operations with the γ_{sem} function.

However, as shown in Fig. 5, the correspondence between words that the string matching approach yields can be rather odd. This method tries to get a minimum cost alignment between words, without taking into account whether that correspondence makes sense. So, in the first pair of sentences in Fig. 5, the algorithm aligns the verb "eat" with the preposition "with", which has no grammatical sense since their syntactic roles are different. This behaviour results in a deviation, masking the actual semantic proximity between the sentences.

This justifies the consideration of tree matching to identify the transformations from which semantic proximity should be computed. We use the edit distance algorithm proposed by Zhang *et al.* [11] and the distances obtained to select which words have to be modified in order to transform the sentences. The sets of partial distances between subforests computed by this algorithm are used to identify a minimal cost mapping, shown with dashed lines in Fig. 5. From this, we extract the edit operations applied over leaves, and from these, we then get the set of words whose semantic costs, computed by γ_{sem} , are accumulated.

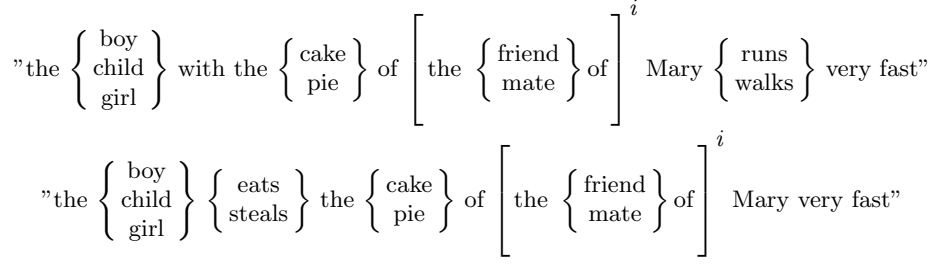


Fig. 7. Data sentences

In the left-most pair of sentences in Fig. 5, we can see how, using the tree matching approach, the words "eats" and "with" are no longer related, due to their different syntactic context.

5 Semantic similarity from word to sentence level

Having now outlined our proposal, we show how the propagation of semantic measures is effectively performed. Semantic costs and syntactic distances are computed in parallel, guided by the computation of partial subforest distances and following the same bottom-up strategy. We first introduce a cost function, γ , to be used within the tree matching algorithm. This takes into account the semantic similarity between words involved in tree edit operations:

$$\gamma(a \rightarrow b) = \begin{cases} 1 & \text{if } a = \varepsilon \text{ or } b = \varepsilon \text{ or } a \neq b \\ 0 & \text{if } a = b \text{ and } [a \notin \text{leaves}(P) \text{ or } b \notin \text{leaves}(D)] \\ \gamma_{\text{sem}}(\text{word}(a) \rightarrow \text{word}(b)) & \text{otherwise} \end{cases}$$

where we use the notation $\text{word}(t[i])$ to refer to the word associated with the $t[i]$ node in the tree T , when this node is a leaf. In this way, the two first cases of this function assign discrete costs to insert and delete operations, and the cost of relabeling is given by the γ_{sem} function applies on the corresponding words.

The *semantic forest distance*, fd_{sem} , is an extension of the concept at syntactic level [11]. Given trees, P and D , $fd_{\text{sem}}(P[s_1..s_2], D[t_1..t_2])$ is the cost of the set of edit operations performed over the individual words belonging to the subforests $P[s_1..s_2]$ and $D[t_1..t_2]$. The selection of the operations on the words is made according to the actual tree edit operation applied on the associated leaves, in order to obtain the optimal mapping that yields the $fd(P[s_1..s_2], D[t_1..t_2])$ value. In an analogous way, $td_{\text{sem}}(i, j)$ is equal to $fd_{\text{sem}}(r(i)..i, r(j)..j)$.

5.1 Computation of semantic distances without VLDC

What our approach does is to compute partial distances between the syntactic structure of the two parse trees, and use these fd values to decide which semantic values, fd_{sem} , should be propagated in each case. When no VLDC symbols are implicated this propagation is made according to the following set of formulae:

$$\begin{aligned}
(1) \quad & fd_{\text{sem}}(\emptyset, \emptyset) = 0 \\
& fd_{\text{sem}}(\emptyset, r(j)..t) = \begin{cases} fd_{\text{sem}}(\emptyset, r(j)..t-1) + \gamma_{\text{sem}}(\varepsilon \rightarrow \text{word}(d[t])) & \text{if } r(t) = t \\ fd_{\text{sem}}(\emptyset, r(j)..t-1) & \text{otherwise} \end{cases} \\
& fd_{\text{sem}}(r(i)..s, \emptyset) = \begin{cases} fd_{\text{sem}}(r(i)..s-1, \emptyset) + \gamma_{\text{sem}}(\text{word}(p[s]) \rightarrow \varepsilon) & \text{if } r(s) = s \\ fd_{\text{sem}}(r(i)..s-1, \emptyset) & \text{otherwise} \end{cases} \\
(2) \quad & fd_{\text{sem}}(r(i)..s, r(j)..t) = \min \left\{ \begin{aligned} & \begin{cases} fd_{\text{sem}}(r(i)..s-1, r(j)..t) + \gamma_{\text{sem}}(\text{word}(p[s]) \rightarrow \varepsilon) & \text{if } r(s) = s \\ fd_{\text{sem}}(r(i)..s-1, r(j)..t) & \text{otherwise} \end{cases} \\ & \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s-1, r(j)..t) + \gamma(p[s] \rightarrow \varepsilon) \\ & \begin{cases} fd_{\text{sem}}(r(i)..s, r(j)..t-1) + \gamma_{\text{sem}}(\varepsilon \rightarrow \text{word}(d[t])) & \text{if } r(t) = t \\ fd_{\text{sem}}(r(i)..s, r(j)..t-1) & \text{otherwise} \end{cases} \\ & \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s, r(j)..t-1) + \gamma(\varepsilon \rightarrow d[t]) \\ & \begin{cases} fd_{\text{sem}}(r(i)..s-1, r(j)..t-1) + \\ \gamma_{\text{sem}}(\text{word}(p[s]) \rightarrow \text{word}(d[t])) & \text{if } r(s) = s \text{ and } r(t) = t \\ fd_{\text{sem}}(r(i)..s-1, r(j)..t-1) & \text{otherwise} \end{cases} \\ & \text{if } fd(r(i)..s, r(j)..t) = \\ & \quad fd(r(i)..s-1, r(j)..t-1) + \gamma(p[s] \rightarrow d[t]) \\ & fd_{\text{sem}}(r(i)..s, \emptyset) + \sum_k \gamma_{\text{sem}}(\varepsilon \rightarrow \text{word}(d[k])), \quad d[k] \in \text{leaves}(D[t]) \\ & \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s-1, \emptyset) \\ & \text{if } r(s) = r(i) \text{ and } r(t) = r(j) \text{ and } p[s] \neq " \wedge " \text{ and } p[s] \neq " | " \end{aligned} \right\} \\
(3) \quad & fd_{\text{sem}}(r(i)..s, r(j)..t) = \min \left\{ \begin{aligned} & \begin{cases} fd_{\text{sem}}(r(i)..s-1, r(j)..t) + \gamma_{\text{sem}}(\text{word}(p[s]) \rightarrow \varepsilon) & \text{if } r(s) = s \\ fd_{\text{sem}}(r(i)..s-1, r(j)..t) & \text{otherwise} \end{cases} \\ & \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s-1, r(j)..t) + \gamma(p[s] \rightarrow \varepsilon) \\ & \begin{cases} fd_{\text{sem}}(r(i)..s, r(j)..t-1) + \gamma_{\text{sem}}(\varepsilon \rightarrow \text{word}(d[t])) & \text{if } r(t) = t \\ fd_{\text{sem}}(r(i)..s, r(j)..t-1) & \text{otherwise} \end{cases} \\ & \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s, r(j)..t-1) + \gamma(\varepsilon \rightarrow d[t]) \\ & fd_{\text{sem}}(r(i)..r(s)-1, r(j)..r(t)-1) + td_{\text{sem}}(s, t) \\ & \text{if } fd(r(i)..s, r(j)..t) = \\ & \quad fd(r(i)..r(s)-1, r(j)..r(t)-1) + td(s, t) \\ & fd_{\text{sem}}(r(i)..s, r(j)..r(t)-1) + \sum_k \gamma_{\text{sem}}(\varepsilon \rightarrow \text{word}(d[k])), \\ & \quad d[k] \in \text{leaves}(D[t]) \\ & \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s-1, r(t)-1) \end{aligned} \right\} \\
& \text{otherwise}
\end{aligned}$$

The computation of $fd_{\text{sem}}(r(i)..s, r(j)..t)$ is made in parallel with $fd(r(i)..s, r(j)..t)$. As in the original algorithm, we have three sets of formulae, the first one for initialization, numbered as (1), and the other two for distances between pairs of subtrees and between pairs of subforests, identified as (2) and (3). These formulae reflect how the edit operation from which the current $fd(r(i)..s, r(j)..t)$ was obtained is identified, and how the new fd_{sem} values are computed, propagating a previous semantic distance or adding a semantic cost to that value.

Zhang *et al.* [11] compute bottom-up partial distances, increasing the sizes of the subforests being considered. In each step, each way of obtaining the current distance is computed using partial distances obtained in previous steps, and the one giving the minimum value is recorded. In our proposal, for each syntactic distance we identify the previous partial ones used to compute it. This allows us to determine whether the last nodes at the considered subforests are involved in the optimal mapping and which edit operation has been applied to them.

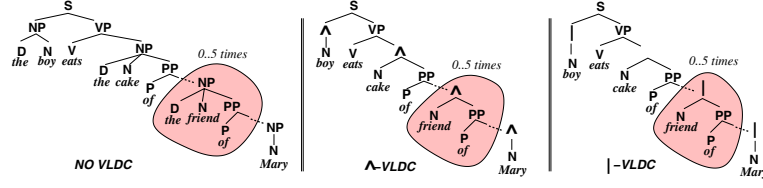


Fig. 8. Pattern trees

So, once the best edit operation has been identified, we compute the value $fd_{\text{sem}}(r(i)..s, r(j)..t)$ for the current pair of subforests. If no leaves were involved, we simply propagate the semantic distance associated with the previous fd value from which the current $fd(r(i)..s, r(j)..t)$ was obtained. Otherwise, when $r(s) = s$ or $r(t) = t$, we add to this fd_{sem} value recovered from the previous step, the semantic cost of the operation to be applied on those leaves.

In the case of subtree cuts, for both tree-to-tree and forest-to-forest distances, previous values are incremented with the cost of deleting every single word in the cut subtrees, as shown in the last cases of formulae (2) and (3). In this way, we avoid deviations in the final distance that subtree cuts could induce. So, although in the syntactic matching some parts of the data trees can be omitted and not be taken into account in the final distance, our proposal does not miss out these deleted words, which will be part of the final semantic cost. An additional refinement is required: when a syntactic distance can result from different edit operations, we must choose the option with the least semantic distance.

5.2 Computation of semantic distances with VLDC

When dealing with semantic distance propagation in presence of VLDC symbols we need to determine exactly the desired semantical behaviour for each kind of symbol. From the formulae in section 3, when cuts are allowed in the data tree, the two types of VLDC symbols can be interchanged without affecting to the mapping and the final syntactic distances. As Zhang *et al.* [11] prove, the subtrees included in the best VLDC-substitution for a $\wedge$ -VLDC will always be part of the optimal set of consistent subtree cuts if it were replaced by a $|$ -VLDC.

However, it is possible to difference the interpretation of both symbols when semantic distances are computed. So, for $|$ -VLDC symbols, we want the words in the cut leaves, shown underlined in the leftmost side of Fig.6, to be taken into account in the semantic distance. Whereas with the $\wedge$ -VLDC, leaves in subtrees not taken into account in the syntactic distance will not contribute to the final semantic distance with the deleting cost of words in their leaves. These subtrees, shaded in Fig. 6, are identified when sfd values are computed, and lead to the definition of a *semantic suffix forest distance*, sfd_{sem} , its semantic counterpart.

So, we shall start to define the formulae to compute tree-to-tree distances with $|$ -VLDC. Being $p[s] = "$ | ", $r(i) = r(s)$, $r(j) = r(t)$ and nodes $d[t_k]$, with $1 \leq k \leq n_t$, children of $d[t]$, we have that:

$$fd_{\text{sem}}(r(i)..s, r(j)..t) = \min \left\{ \begin{array}{l} \left\{ \begin{array}{l} fd_{\text{sem}}(r(i)..s-1, r(j)..t) + \gamma_{\text{sem}}(\text{word}(p[s]) \rightarrow \varepsilon) \text{ if } r(s) = s \\ fd_{\text{sem}}(r(i)..s-1, r(j)..t) \text{ otherwise} \end{array} \right\} \\ \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s-1, r(j)..t) + \gamma(p[s] \rightarrow \varepsilon) \\ \left\{ \begin{array}{l} fd_{\text{sem}}(r(i)..s, r(j)..t-1) + \gamma_{\text{sem}}(\varepsilon \rightarrow \text{word}(d[t])) \text{ if } r(t) = t \\ fd_{\text{sem}}(r(i)..s, r(j)..t-1) \text{ otherwise} \end{array} \right\} \\ \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s, r(j)..t-1) + \gamma(\varepsilon \rightarrow d[t]) \\ \left\{ \begin{array}{l} fd_{\text{sem}}(r(i)..s-1, r(j)..t-1) + \\ \gamma_{\text{sem}}(\text{word}(p[s]) \rightarrow \text{word}(d[t])) \text{ if } r(s) = s \text{ and } r(t) = t \\ fd_{\text{sem}}(r(i)..s-1, r(j)..t-1) \text{ otherwise} \end{array} \right\} \\ \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s-1, r(j)..t-1) + \gamma(p[s] \rightarrow d[t]) \\ fd_{\text{sem}}(\phi, r(j)..t-1) + td_{\text{sem}}(s, t_k) - td_{\text{sem}}(\emptyset, t_k) \\ \text{if } fd(r(i)..s, r(j)..t) = fd(\phi, r(j)..t-1) + td(s, t_k) - td(\emptyset, t_k) \\ \text{where } t_k = \text{argmin}_{1 \leq k \leq n_t} \{td(s, t_k) - td(\emptyset, t_k)\} \\ fd_{\text{sem}}(r(i)..s, \emptyset) + \sum_k \gamma_{\text{sem}}(\varepsilon \rightarrow \text{word}(d[k])), d[k] \in \text{leaves}(D[t]) \\ \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s-1, \emptyset) \end{array} \right.$$

To manage $\wedge$ -VLDC as we outlined above, being $d[t_k]$, $1 \leq k \leq n_t$ the children of $d[t]$, when $r(i) = r(s)$, $r(j) = r(t)$ and $p[s] = "$ $\wedge$ " we have that:

$$fd_{\text{sem}}(r(i)..s, r(j)..t) = \min \left\{ \begin{array}{l} \left\{ \begin{array}{l} fd_{\text{sem}}(r(i)..s-1, r(j)..t) + \gamma_{\text{sem}}(\text{word}(p[s]) \rightarrow \varepsilon) \text{ if } r(s) = s \\ fd_{\text{sem}}(r(i)..s-1, r(j)..t) \text{ otherwise} \end{array} \right\} \\ \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s-1, r(j)..t) + \gamma(p[s] \rightarrow \varepsilon) \\ \left\{ \begin{array}{l} fd_{\text{sem}}(r(i)..s, r(j)..t-1) + \gamma_{\text{sem}}(\varepsilon \rightarrow \text{word}(d[t])) \text{ if } r(t) = t \\ fd_{\text{sem}}(r(i)..s, r(j)..t-1) \text{ otherwise} \end{array} \right\} \\ \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s, r(j)..t-1) + \gamma(\varepsilon \rightarrow d[t]) \\ \left\{ \begin{array}{l} fd_{\text{sem}}(r(i)..s-1, r(j)..t-1) + \\ \gamma_{\text{sem}}(\text{word}(p[s]) \rightarrow \text{word}(d[t])) \text{ if } r(s) = s \text{ and } r(t) = t \\ fd_{\text{sem}}(r(i)..s-1, r(j)..t-1) \text{ otherwise} \end{array} \right\} \\ \text{if } fd(r(i)..s, r(j)..t) = fd(r(i)..s-1, r(j)..t-1) + \gamma(p[s] \rightarrow d[t]) \\ td_{\text{sem}}(s, t_k) \\ \text{if } fd(r(i)..s, r(j)..t) = td(s, t_k) \\ \text{where } t_k = \text{argmin}_{1 \leq k \leq n_t} \{td(s, t_k)\} \\ sf_{\text{sem}}(r(i)..s-1, r(j)..t_k) \\ \text{if } fd(r(i)..s, r(j)..t) = sf_{\text{sem}}(r(i)..s-1, r(j)..t_k) \\ \text{where } t_k = \text{argmin}_{1 \leq k \leq n_t} \{sf_{\text{sem}}(r(i)..s-1, r(j)..t_k)\} \end{array} \right.$$

To ensure the desired semantic behaviour for $\wedge$ -VLDC symbols, we introduce the *semantic suffix forest distance* and specify how these values should be computed. Being F_P and F_D , forests in the pattern P and the data tree D , the *semantic suffix forest distance*, $sf_{\text{sem}}(F_P, F_D)$, is the semantic distance between F_P and $\bar{F}_D$, where $\bar{F}_D$ is a subforest of F_D with some consecutive subtrees, all having the same parent, removed from the right. Formally:

$$sf_{\text{sem}}(F_P, F_D) = \min_{\bar{F}_D} \{fd_{\text{sem}}(F_P, \bar{F}_D)\}$$

As a result, we get the required behaviour for $\wedge$ -VLDC, since the words present in the removed subtrees will not take part in the final semantic distance. The computation procedure extends the formulae in section 3 to propagate semantic values in the following way:

$$\begin{aligned}
(1) \quad & sf_{d_{\text{sem}}}(\emptyset, \emptyset) = 0 \\
& sf_{d_{\text{sem}}}(\emptyset, r(j)..t) = \begin{cases} 0 & \text{if } r(t) = r(j) \text{ or } r(\text{parent}(t)) = r(j) \\ sf_{d_{\text{sem}}}(\emptyset, r(j)..t-1) + \gamma_{\text{sem}}(\varepsilon \rightarrow \text{word}(d[t])) & \text{if } r(t) = t \\ sf_{d_{\text{sem}}}(\emptyset, r(j)..t-1) & \text{otherwise} \end{cases} \quad \text{otherwise} \\
& sf_{d_{\text{sem}}}(r(i)..s, \emptyset) = fd_{\text{sem}}(r(i)..s, \emptyset) \\
(2) \quad & sf_{d_{\text{sem}}}(r(i)..s, r(j)..t) = \begin{cases} fd_{\text{sem}}(r(i)..s, \emptyset) & \text{if } fd(r(i)..s, \emptyset) < fd(r(i)..s, r(j)..t) \\ fd_{\text{sem}}(r(i)..s, r(j)..t) & \text{otherwise} \end{cases} \\
& \text{if } r(t) = r(j) \\
(3) \quad & sf_{d_{\text{sem}}}(r(i)..s, r(j)..t) = \min \begin{cases} \begin{cases} sf_{d_{\text{sem}}}(r(i)..s-1, r(j)..t) + \gamma_{\text{sem}}(\text{word}(p[s]) \rightarrow \varepsilon) & \text{if } r(s) = s \\ sf_{d_{\text{sem}}}(r(i)..s-1, r(j)..t) & \text{otherwise} \end{cases} \\ \begin{cases} sf_{d_{\text{sem}}}(r(i)..s, r(j)..t-1) + \gamma_{\text{sem}}(\varepsilon \rightarrow \text{word}(d[t])) & \text{if } r(t) = t \\ sf_{d_{\text{sem}}}(r(i)..s, r(j)..t-1) & \text{otherwise} \end{cases} \\ \begin{cases} sf_{d_{\text{sem}}}(r(i)..s, r(j)..t) = sf_{d_{\text{sem}}}(r(i)..s, r(j)..t-1) + \gamma(\varepsilon \rightarrow d[t]) \\ sf_{d_{\text{sem}}}(r(i)..r(s)-1, r(j)..r(t)-1) + td_{\text{sem}}(s, t) \\ \text{if } sf_{d_{\text{sem}}}(r(i)..s, r(j)..t) = \\ sf_{d_{\text{sem}}}(r(i)..r(s)-1, r(j)..r(t)-1) + td(s, t) \end{cases} \end{cases} \\
& \text{otherwise}
\end{aligned}$$

otherwise

By means of these formulae, we get two different kinds of semantic VLDC matching. As we shall see in next section, $|$ -VLDC offers a more restricted match, both at syntactic and semantic level, allowing to easily describe precise patterns. On the other hand, the $\wedge$ -VLDC is a more flexible and vague method to specify patterns, able to match a wider range of data parse trees.

6 A practical example

We now provide some preliminary results illustrating how our proposal can give a more precise notion of semantic similarity than the classic string matching one, and how the presence of VLDC symbols have an effect on the results. We consider the pico-grammar of English as running example:

- | | | |
|--------------------------------------|--------------------------------|-------------------------------------|
| (1) S $\rightarrow$ NP VP | (4) PP $\rightarrow$ Prep NP | (7) NP $\rightarrow$ Det Name |
| (2) VP $\rightarrow$ Verb NP AdjP | (5) AdjP $\rightarrow$ Adv Adj | (8) NP $\rightarrow$ Det Name PP |
| (3) VP $\rightarrow$ Verb NP PP AdjP | (6) NP $\rightarrow$ Name | (9) NP $\rightarrow$ Det Name PP PP |

We have created a bank of data sentences generated as is shown in Fig. 7. All possible combinations of the words "*boy, child, girl*", with "*eat, steal, run, walk*", with "*cake, pie*" and with "*friend, mate*" have been considered to compose those sentences, where i represents the number of repetitions of the substrings enclosed in braces, with values ranging from 0 to 7. We have also created a set of pattern sentences based on the following scheme, where i ranges from 0 to 5:

"the boy eats the cake of [the friend of] ^{i} Mary"

The presence of the rule " $\text{NP} \rightarrow \text{Det Name PP PP}$ " implies that the number of parse trees grows exponentially with i , and this number, C_i , is given by:

$$C_0 = C_1 = 1 \quad \text{and} \quad C_i = \binom{2i}{i} \frac{1}{i+1}, \quad \text{if } i > 1$$

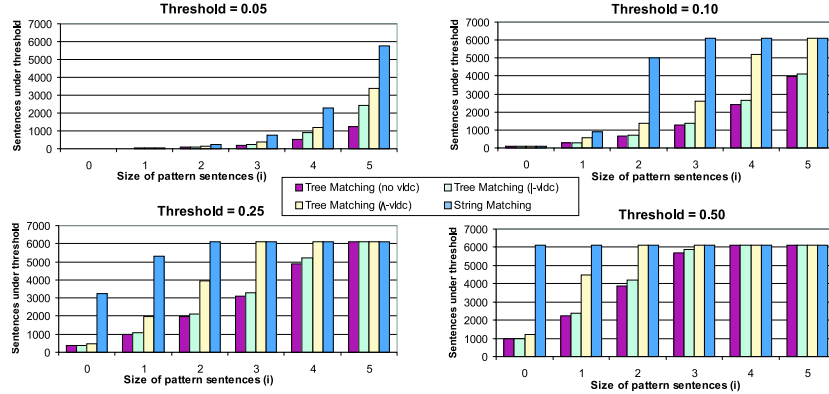


Fig. 9. Tree matching vs. string matching semantic measures

The frame described provides, in spite of its simplicity, a highly ambiguous environment to test our proposal with. To deal with it, we have generated the parse trees using ICE (*Incremental Compiler Environment*) [8], which represents them in a compact form as a shared forest. We have also incorporated the optimizations in the tree matching algorithm shown in [9], to take advantage of the structural sharing offered by ICE in order to avoid redundant computations.

In the case of the tree matching based approach three different sets of pattern trees were built from the deterministic parse of the pattern sentences with the structure shown in Fig. 8. The aim is to test how VLDC symbols work, and compare them with non-VLDC pattern trees. To test string matching, the pattern and data sentences were used directly, without parsing them. To compare our proposal with Lin's based method [2], we have normalized the resulting distances in $[0, 1]$, in both cases. In this context, to compare the different methods, we count the number of sentences whose similarity values with respect to a pattern sentence fall under a given threshold, assuming that more precise approaches should provide reduced sets of sentences, since they are based on stricter similarity criteria. We aim to determine whether the values obtained by the proposed methods are able to make the semantic proximity between sentences evident, and to identify non-similar ones without deviations.

Practical results are shown in Fig. 9, considering thresholds 0.05, 0.10, 0.25 and 0.50; and values of i for the pattern sentences from 0 to 5. Under the four thresholds our proposal seems to be more precise, showing a higher discrimination power and returning a smaller number of sentences. The string based approach tends to suffer deviations when the sentences compared share the same words with different syntactic roles, as is the case here. So, the number of sentences under thresholds is always greater for the string based one, showing that the tree based values provide a more accurate notion of semantic proximity.

With regard to the VLDC matching, it offers intermediate results, closer to non-VLDC tree matching than to the string based one. As expected, the $\mid$ -VLDC gives more accurate measures, quite similar to the ones obtained with non-VLDC pattern trees. The $\wedge$ -VLDC patterns are slightly less precise, but they do

not suffer the rapid degeneration shown by the string based measure when the thresholds become higher. In these cases, this method identifies all of the data sentences as being similar to the given pattern, without making any distinction between them. This indicates that the values offered by the string based approach are not a good criterion to determine if two sentences have similar meanings, since this metric assigns a high relevance to word-to-word correspondences, making other syntactic and semantic aspects irrelevant. Our technique has a better behavior, and it only loses efficiency under the highest thresholds.

7 Conclusions

We exploit the meaning of single-terms by integrating it into the edit distance construction. This allows us to deal with similarity problems where we can take advantage of the use of semantic information in pattern-matching processes. Preliminary results seem to support our approach as opposed to solutions based exclusively on syntactic structures or single terms.

Acknowledgments

This work was partially supported by the Spanish Government under project HF2002-0081, and the Autonomous Government of Galicia under projects PGIDIT03SIN30501PR, PGIDIT02SIN01E and PGIDIT02PXIB3051PR.

References

1. K. Hammouda, M. Kamel. Phrase-based document similarity based on an index graph model. *2002 IEEE Int. Conf. on Data Mining*, 203–210, Japan, 2002.
2. D. Lin. An information-theoretic definition of similarity. *Proc. of 15th Int. Conf. on Machine Learning*, 296–304, 1998.
3. G. Miller. WordNet: an online lexical database. *International Journal of Lexicography*, 3(4), 1990.
4. G. Navarro, R. Baeza-Yates. Proximal nodes: A model to query document databases by content and structure. *ACM Transactions on Information Systems*, 15(4):400–435, 1997.
5. Smeaton, A.F, O'Donell, R., Kelley, F.: Indexing Structures Derived from Syntax in TREC-3: System Description. *Proc. of 3rd Text REtrieval Conference*, 1994.
6. A.F. Smeaton, I. Quigley. Experiments on using semantic distances between words in image caption retrieval. *Proc. of the 19th Annual Int. ACM Conf. on Research and Development in Information Retrieval*, 174–180, 1996.
7. Kuo-Chung Tai. The Tree-to-Tree Correction Problem. *Journal of the ACM*, 26(3):422–433, 1979.
8. M. Vilares and B.A. Dion. Efficient incremental parsing for context-free languages. *Proc. of the 5th IEEE Int. Conf. on Computer Languages*, 241–252, France, 1994.
9. M. Vilares, F.J. Ribadas, and V.M. Darriba. Approximate VLDC pattern matching in shared-forest. *Lecture Notes in Artificial Intelligence*, 2004:483–494, 2001.
10. R.A. Wagner and M.J. Fischer. The string to string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.
11. K. Zhang, D. Shasha, and J.T.L. Wang. Approximate tree matching in the presence of variable length don't cares. *Journal of Algorithms*, 16(1):33–66, 1994.

Compiling Thread-Level Parallel Programs with a C-Compiler

Martti Forsell¹

¹VTT Electronics, Computing Platforms Group, Box 1100, FI-90571 Oulu, Finland

`Martti.Forsell@VTT.Fi`

`http://www.vtt.fi/ele/people/martti.forsell/work.html`

Abstract. Recent advances in multithreaded shared memory architectures have created a need for efficient and easy-to-use thread-level parallel (TLP) programming languages and tools supporting the synchronous shared memory model. In our previous work we have developed *e*, a fine-grained TLP programming language for multiprocessor architectures realizing such a model. The language uses a familiar c-like syntax and supports shared and private variables, arbitrary hierarchical groups of threads, and synchronous control structures. This allows a programmer to use various advanced TLP programming techniques like data parallelism, divide-and-conquer technique, different blocking techniques, and both synchronous and asynchronous programming style. In this paper we describe how an experimental compiler for *e* supporting virtual instruction-level parallelism exploitation can be set up by using a standard c-compiler. We will also shortly evaluate the compiler with real parallel programs on our scalable Eclipse network on chip architecture.

1. Introduction

Future computers can not be based on current architectures relying heavily on exploitation of *instruction-level parallelism* (ILP) in which multiple instructions are executed in multiple functional units in parallel because future silicon technologies will not be able to provide single clock cycle latency to long on-chip wires required to implement the internal communication network between the functional units of the processor [1]. Furthermore, large amounts of ILP is hard to extract [2], ILP lacks the possibility of exploiting control parallelism by definition, and the complexity of an efficient ILP machine increases quadratically in respect to the number of functional units due to need for forwarding [3]. *Thread-level parallelism* (TLP), i.e. executing a program consisting of parallel subtasks in multiple processors or thread units to solve a computational problem, avoids most of these problems by being much easier to extract, and inherently supporting control parallelism. Furthermore, some multithreaded architectures, e.g. superpipelined multithreading [4], are almost insensitive to long latencies since they allow a processor to execute other threads while a thread is accessing the memory via the interconnection network. Efficient exploitation of TLP requires determining how the functionality and data related to subtasks are distributed to the threads (or processors), how threads communicate with each other, and how synchronicity is maintained between the subtasks.

TLP architectures can be classified to message passing architectures and shared memory architectures according to the main method of communication: In *message passing architectures* (MPA) threads communicate by sending messages to each others and a programmer is responsible for explicitly defining communication, synchronizing subtasks, and describing data and program partitioning between threads making MPAs difficult to program. There exist MPAs, which use implicit messaging, but so far these systems relying on automatic parallelization can not compete with explicitly controlled systems in performance. In *shared memory architectures* (SMA) communication happens just by referring to the shared memory, data and program partitioning happens by allocating data structures from the shared memory making programming much easier. Unfortunately most SMAs consist

of multiple interconnected processor-cache pairs making cache coherency and synchronicity maintenance very expensive. Recent advances in TLP architectures [5, 6], however, suggest that it is possible to implement a cacheless and synchronous SMA realizing the *parallel random access machine* (PRAM) model [7] on a single chip computer or network on chip (NOC) [8]. This kind of machines are so far the best candidates for future generation general purpose parallel computers, because they have some very advanced properties: The single step accessible uniform shared memory, machine instruction-level lock-step synchronous execution, and an ability to exploit virtual ILP, a special kind of ILP-like parallelism occurring between multiple threads in forms of superpipelined and chained execution [9].

In our previous work we have developed an experimental programming language, called *e*, for fine-grained TLP programming on synchronous SMAs realizing the PRAM model [10]. It uses a familiar c-like syntax and provides support for shared and private variables, arbitrary hierarchical groups of threads, and synchronous control structures. This allows a programmer to use various advanced TLP programming techniques like data parallelism, divide-and-conquer techniques, different blocking techniques, and both synchronous and asynchronous programming style [11]. *E* can be used as an integral part of the application development flow in which computational problems are transformed to ILP and TLP optimized TLP binaries for synchronous SMAs [12] (see Figure 1).

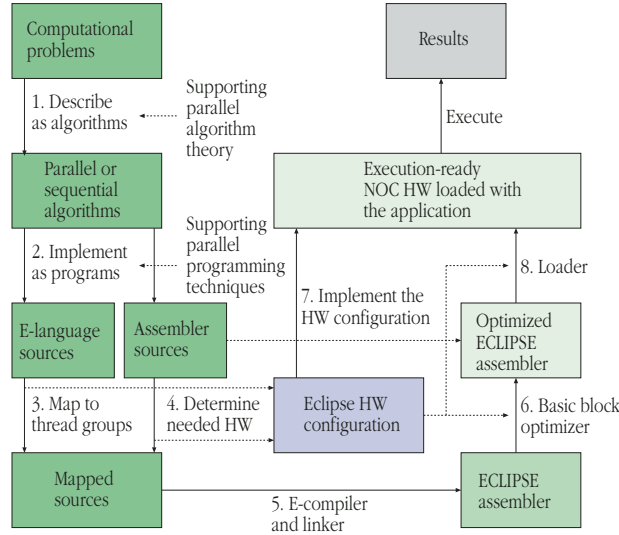


Fig. 1. The application development flow applied for the Eclipse architecture

In this paper we describe how *ec*, an experimental compiler for *e*, can be set up by using a standard c-compiler, set of add-on files, and some simple tools. We evaluate *ec* shortly by implementing a number of familiar TLP algorithms with *e*, compiling them with *ec* and executing them in our scalable high performance Eclipse NOC architecture [6] consisting of interthread superpipelined processors (MTAC) and interleaved memory modules connected together via a sparse mesh network.

The rest of the paper is organized so that in section 2 we take a brief look at related work in this area. In section 3 we give short overview of the *e*-language. In section 4 we describe *ec*, our implementation of the compiler for the *e*-language. A short evaluation including implementing a small number of real parallel algorithms with *e*-language, compiling them with our experimental *e*-compiler,

and executing them on the different configurations of the Eclipse architecture is described in section 5. Finally, in section 6 we give our conclusions.

2. Related Work

There exist compilers for a few TLP programming languages that have quite similar properties as the e-language. These languages are targeted for various PRAM models realized either as a simulator or as an experimental hardware. Among the best known are:

fcc. A compiler for Fork, a feature-rich parallel programming language supporting parallel recursive and synchronous MIMD programming using a c-like syntax [11]. Fork is targeted for arbitrary *concurrent read concurrent write* (CRCW) PRAM model with a fixed number of threads and hardware support for efficient multiprefix operations. (The arbitrary CRCW PRAM model allows multiple concurrent reads and writes per a shared memory location. If multiple values are tried to write simultaneously to a single memory location, an arbitrarily selected processor succeeds in writing.) The model is realized e.g. in the SB-PRAM parallel computer developed in the University of Saarbrücken [11]. The compiler is built on publicly available lcc compiler for c-language.

ll. A compiler for ll, a parallel language supporting parallel recursive and synchronous MIMD programming using Pascal-like syntax [13]. ll allows for a (virtual) processor to spawn new ones executing the loop body in parallel giving better support for the theoretical PRAM model with unbounded set of processors.

xcc. A compiler for a parallel language supporting expressing simplified TLP for a XMT machine providing realization of the CRCW PRAM model [14].

Some other PRAM languages, like pm2 [15] and Modula-2* [16], are more data parallelism oriented and provide no support for explicit group concept. None of these languages, however, can be compiled with a commercial quality c-compiler nor supports efficient virtual ILP exploitation [9].

3. E-language

The e-language [10] is an experimental TLP programming language created especially for synchronous SMA NOCs providing general purpose functionality, but it can be used also for other (multichip) synchronous SMAs like the IPSM framework [4]. The syntax of e-language is an extension of the syntax of the familiar c-language. E-language supports parallel recursive and synchronous MIMD programming for various PRAM models including the EREW PRAM model, the TLP programming model of Eclipse. (The EREW PRAM model allows only a single reference per a shared memory location.)

Variables in the e-language can be shared among a group of threads or they can be private to a thread. Shared variables can not be used as modal parameters because there is no mechanism to ensure that actual parameters of threads sharing a variable in a function really point to the same data in the memory. Passing a shared variable as the result value of a function may cause severe congestion if the machine supports the EREW model only. If an actual parameter is a shared variable, private copies of value or reference will be used in the function execution.

In order to support high-level TLP expressions with the *single program multiple data* (SPMD) paradigm, threads are automatically numbered from 0 to the number of threads -1 as new groups are created. Alternatively, a programmer can use static thread numbering.

E-language supports hierarchical groups of threads. In the beginning of a program there exists a single group containing all threads. A group can be divided into subgroups so that in each thread of the group is assigned into one of the subgroups. A subgroup may be split into further subgroups, but the existence of each level of subgroups ends as control returns back to the corresponding parent group. As a subgroup is created, dynamic thread identification variables are updated to reflect the new situation. As the subgroups join back to the parent group in the end of the statement the old values of these variables are restored.

Synchronous SMAs guarantee synchronous execution of instructions at machine instruction level. In e-language synchronicity and/or automatic subgroup creation through control structures having private enter/exit conditions can be maintained with special versions of control structures supporting automatic synchronization at the end of the structure, supporting automatic subgroup creation, or both automatic synchronization and subgroup creation. The basic control mode, called synchronous area, is the group-wise synchronous execution control structures or the SPMD convention in which one or more groups of threads execute the same portion of code synchronously. Asynchronous control structures, i.e. those with private enter/exit conditions but without synchronization, can be used only at the leaf level of group hierarchy. Entering to an asynchronous area happens by using an asynchronous control structure and returning back to the synchronous area happens by an explicit group-wide barrier synchronization assuming all threads of the group will reach the barrier.

4. E-compiler

We have implemented ec, an e-language compiler prototype for synchronous SMAs and configured it for the Eclipse architecture realizing the EREW PRAM model featuring T threads and providing a flexible synchronization hardware capable of multiple simultaneous synchronizations in constant time.

4.1 Structure

The compiler consists of a standard c-compiler (gcc 2.7.2 for the DLX processor) including the c-preprocessor (ccp), compiler itself (cc1), linker (ld), translator, virtual ILP optimizer, and loader (see Figure 2). The compiler excluding the machine dependent start-up code and run-time library is portable. A DLX version of gcc was used because it supports easy-to-prototype textual object file format, we had existing DLX to MTAC translator [17], and we considered reusing them easier than porting gcc directly to MTAC. Compiling happens in six parts:

- (1) *Apply the c-preprocessor to source files and e-specific header file (e.h).* During this part e-specific control structures are substituted with c-only structures taking care of subgroup creation and synchronization if necessary.
- (2) *Apply the c-compiler to the preprocessed files.* This part includes standard c-compilation and produces compiled files.
- (3) *Apply the linker to the compiled files and start-up code (ePrefix.s).* This part includes standard linking and produces a single executable file.

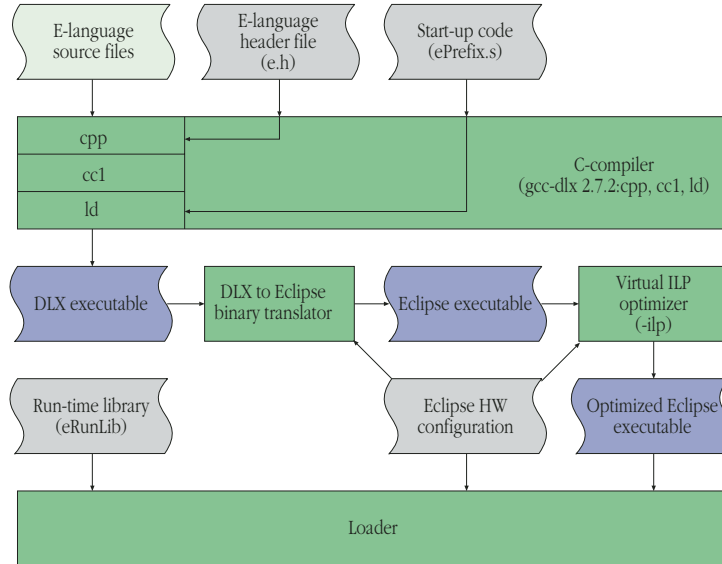


Fig. 2. The compiler structure

(4) *Apply the binary translator to the executable.* This part includes translating the (DLX) executable to the target (Eclipse) architecture. This part is needed only if there exists no good enough compiler for the target architecture.

(5) *Apply the virtual ILP optimizer to the translated executable.* This part includes optimizing the translated executable for the two-component computing model of the target machine.

(6) *Apply the loader to the optimized executable.* This part includes loading the optimized executable and the run time library (eRunLib) implementing various e-language primitives like group creation, maintenance and synchronization as well as some Eclipse-specific brute force primitives.

4.2 Shared and Private Memory Spaces and Referencing

The support for shared and private variables is implemented by dividing the memory space into a shared subspace and T private subspaces (see Figure 3). The binary translator and program loader take care of directing references to right addresses, assigning shared variable to the shared subspace, and private variables to corresponding private subspaces: Private locals and shared globals are referenced and allocated from private stacks and shared heap directly by instructions produced by the c-compiler. A private copy of each private global variable is allocated from every private heap simultaneously by the loader while the binary translator adds instructions adding the offset of a private space to each private global reference. Allocating space for shared locals happens collectively by macros and structures defined in the e-specific header file and given by a programmer. Referencing to shared locals happens via private copies of the shared stack pointer as specified in the e definition [10]. This kind of methodology simplifies pointer handling between memory spaces. The creation of shared subgroup subspaces is implemented by dividing the shared space of the parent group to two shared subspaces (see Figure 3). Allocation of space for subgroup-shared locals is implemented by decreasing the private copies of subgroup stack pointer by the allocation size of subgroup locals.

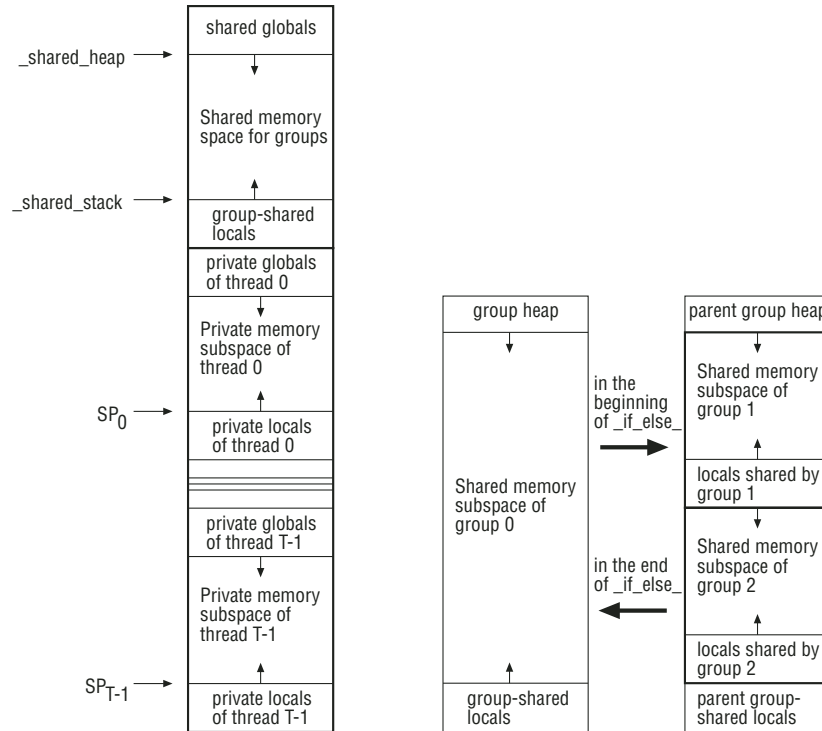


Fig. 3. The memory space implementation in the e-compiler (left) and the subspace creation in the case of `_if_else_` statement (right)

4.3 Control

The e-language supports a versatile set of control structures providing automatic subgroup creation, synchronization, or both. The standard c control structures are implemented directly by the instructions produced by the c-compiler and may enter threads to an asynchronous area. The non-c control structures are implemented as standard c preprocessor macros taking care of control, thread variable updating, and synchronization. Algorithm 1 shows how a two-branch control structure `_if_else_` (c,s,t) featuring automatic subgroup creation and post synchronization is implemented: First, the state of the current subgroup is saved to hidden local variables. Second, the threads are assigned to two execution flows according to the condition **c** of the structure. Third, the creation of shared subspaces for subgroups is implemented by dividing the shared subspace of the parent group to two shared subspaces. The allocation of space for subgroup-shared locals is implemented by decreasing the private copies of subgroup stack pointer by the allocation size of subgroup locals. The renumbering of threads and updating group specific variables is done with two run-time library primitives `_update_sub_1` and `_update_sub_2` creating two independent subgroups. Fourth, the substatements **s** and **t** are executed. Finally, the threads are joined back to a single instruction flow, the old thread state is restored, and threads are synchronized with a group-wide barrier synchronization primitive `presynchronize` of the run-time library.

```

#define _if_else_( c , s , t ) { \
    int _old_thread_id = _thread_id; \
    int _old_number_of_threads = _number_of_threads; \
    int _old_group_id = _group_id; \
    int _old_sub_id = _sub_id; \
    int _old_number_of_subs = _number_of_subs; \
    int _old_shared_stack; \
    int _old_shared_heap; \
    if (c) \
    { \
        _old_shared_stack = _shared_stack; \
        _shared_stack = ( _shared_heap + _shared_stack ) >> 1; \
        _shared_stack &= 0xfffffff; \
        _shared_stack -= 4; \
        _update_sub_1; \
        s \
    } \
    else \
    { \
        _old_shared_heap = _shared_heap; \
        _shared_heap = ( _shared_heap + _shared_stack ) >> 1; \
        _shared_heap &= 0xfffffff; \
        _update_sub_2; \
        t \
    } \
    _thread_id = _old_thread_id; \
    _number_of_threads = _old_number_of_threads; \
    _group_id = _old_group_id; \
    _sub_id = _old_sub_id; \
    _number_of_subs = _old_number_of_subs; \
    _shared_stack = _old_shared_stack; \
    _shared_heap = _old_shared_heap; \
    presynchronize; \
}

```

Alg. 1. The implementation of the `_if_else_` statement

4.4 Optimization

Available optimizations include those supported by the c-compiler (in the case of gcc e.g. -O2) optimizing TLP execution with standard tools and our implementation of the ILP optimization, denoted here with -ilp. The standard optimizations apply e.g. common subexpression elimination, more efficient register allocation to the instruction stream. Optimizations do not affect TLP execution since they have effect only on a single instruction stream used by all of the threads equally. The virtual ILP optimization schedules instructions of basic blocks for the two-component computing model (super-pipelined and chained VLIW execution) of the Eclipse architecture. Figure 4 shows how a simple basic block copying a block of words from a location to another in memory proceeds through the binary translation and virtual ILP optimization and provides 300% speedup factor on an Eclipse featuring 12-threaded 5 functional unit processors. Note that the virtual ILP optimization is able to speed up execution even if the code is strictly sequential, because execution of subinstructions happens sequentially. The optimization algorithm is described and evaluated in [9].

In order to make the overhead of synchronous execution on the EREW PRAM model reasonable we have replaced the original Eclipse memory module architecture [18] with active memory modules featuring a simple ALU and limited number of fast memory locations per module. By this method we are able implement architecture dependent limited and partial support for the arbitrary multiprefix CRCW PRAM model and the execution time of key primitives of group creation and synchronization,

i.e. thread renumbering and arbitrary simultaneous barrier synchronizations, drops from $O(\log T)$ to $O(1)$ in a T threaded Eclipse. We use this technique also to optimize some basic primitives, like spread a data word from a thread to all threads of the group and associative multioperations, in constant time. They are available in e for Eclipse as e-specific structures and implemented in the run-time library as Eclipse-specific assembler primitives. Algorithm 2 shows the implementation of the `spread_` primitive in Eclipse assembler for 5 functional unit machines.

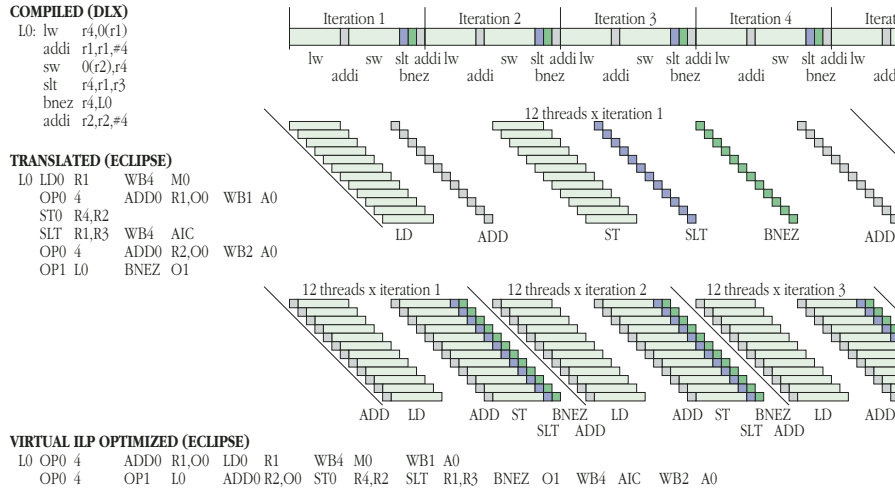


Fig. 4. Effect of binary translation and virtual ILP optimization on a block of code performing a memory block move in a 12-threaded Eclipse

```

ADD0 R29,O0 OP0 -8 ST0 R2,A0 ; Save registers
ADD0 R29,O0 OP0 -4 ST0 R3,A0
ADD0 O0,R32 OP0 __sub_id LD0 A0 WB1 M0 ; R1 <-- __sub_id
ADD0 O0,R32 OP0 __group_id LD0 A0 WB2 M0 ; R2 <-- __group_id
ADD0 O0,R32 OP0 __absolute_number_of_threads LD0 A0 WB3 M0 \
; R3 <-- __absolute_number_of_threads
SHL0 R3,O0 OP0 2 WB3 A0 ; R3 <-- 4 * __absolute_number_of_threads
ADD0 O0,R32 OP0 __absolute_thread_id LD0 A0 SEQ R1,M0 BEQZ O1 \
OP1 L10 ; If sub id ≠ absolute thread id goto L10
ADD0 R2,R3 LD0 A0 WB2 M0 ; R2 <-- data
ADD0 R1,R3 ST0 R2,A0 JMP O0 OP0 L20 \
; Replicate data to subgroups

L10 NOP0
NOP0
L20 ADD0 R1,R3 LD0 A0 WB1 M0 ; Replicate data to subgroup members
ADD0 R29,O0 OP0 -8 LD0 A0 WB2 M0 ; Restore registers
ADD0 R29,O0 OP0 -4 LD0 A0 WB3 M0 JMP R31

```

Alg. 2. The implementation the `spread_` statement

4.5 Limitations

The current version of the e-compiler has a number of limitations: goto-statements, multipoint exits from a function, break and continue-statements over group or shared locals block boundaries are not supported and may cause arbitrary effects. In addition, one should use explicit barrier synchroniza-

tion with these structures if private conditions realized as c-style short circuit evaluations break the synchronicity of the thread group in synchronous areas. Certain advanced gcc optimizations like loop unrolling should be used with care because they may break dependencies between threads and cause errors in execution. Finally, the support for shared dynamic variables is not implemented yet.

5. Evaluation

In order to evaluate the e-compiler in real parallel programming we wrote e-language versions of six TLP programs representing widely used primitives of parallel computing [19] (see Table 1). The programs we compiled with the level 2 optimizations (-O2) and our ILP optimization (-ilp) on, and executed in three configurations of our scalable Eclipse NOC architecture with the IPSMSim simulator [17]. The parameters of the configurations are listed in Table 2.

For each benchmark and configuration pair we measured the execution time of the program excluding the initialization of data and the utilization of functional units. The results of our measurements are shown in Figure 5.

Table 1. The benchmark programs

block	A parallel program that moves a block of integers in the shared memory from a location to another
fft	A parallel program that calculates a 64-point complex fast fourier transform for fixed point numbers
fir	A parallel program that applies finite response filter to a table of given integers in the shared memory
max	A parallel program that finds the maximum of given table of random integers in the shared memory
prefix	A parallel program that calculates the prefix sums for given table of integers in the shared memory
rqsort	A parallel program that sorts given table of T random integers in the shared memory using the recursive randomized parallel quicksort algorithm, where T is the number of threads

Table 2. The configurations of machines used in evaluations (c=processor clock cycles)

	E4	E16	E64
Processors	4	16	64
Threads per processor	512	512	512
Functional units	5	5	5
Bank access time	12 c	12 c	12 c
Bank cycle time	15 c	15 c	15 c
Length of FIFOs	16	16	16

The rough scalability of the execution times can be seen clearly from the curves of the benchmarks. The slight variance in scalability is due to the randomized nature of communication and that the number of threads is fixed making larger configurations relatively weaker than smaller ones. The utilization of functional units does not achieve the level measured in our earlier tests [9] suggesting that the quality of the code produced by the gcc is not nearly as good as that of hand compiling. The reason for this behavior is that we used -mtraps flag in compilation substituting some I/O routines with operating system traps, the benchmarks were not I/O intensive, and processors used VLIW coding.

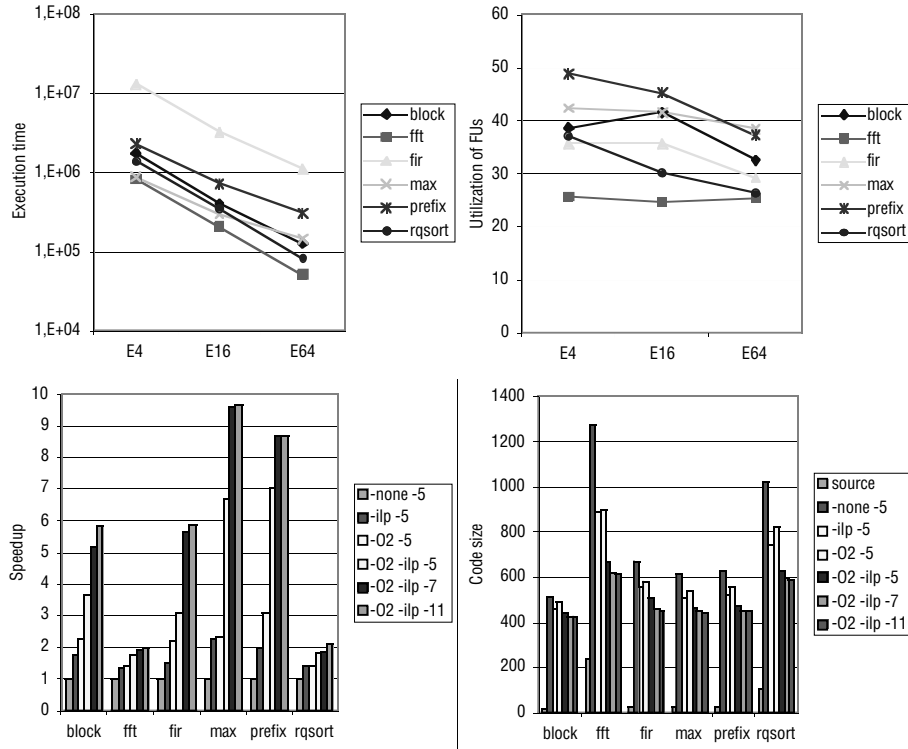


Fig. 5. The measured execution times in clock cycles, utilizations of functional units in E4, E16 and E64 (top). The measured speedups and sizes of source code (excluding start-up code, comments and run-time libraries in lines) and executables (in VLIW instructions) including the start-up code and libraries in E4 configuration for -O2, -ilp with 5, 7 and 11 functional units in respect to non-optimized code (bottom)

We evaluated also the effect of -O2 and -ilp optimizations for the E4 configuration as a speedup, the source code size and executable size in respect to the non-optimized execution. The results are shown in Figure 3. The sizes of executables remained modest although the start-up code and Eclipse-related run-time libraries were included. The optimizations speed up execution by the factors ranging from 2.0 to 9.6. The standard -O2 optimization seems to have stronger effect on the non-optimized code than the -ilp optimization. The reason for this may be that the -ilp optimization was limited by the low number of functional units used in the experiments and the fact that e-language specific routines taking care of subgroup creation and synchronization were hand-optimized for 5 functional unit configuration only and synchronization primitives took a lot of time reducing the percentage of actually optimized code remarkably. Our separate evaluations of -ilp optimization [9] suggest that speedup factors close to 500% by applying -ilp only, could be achieved, and the hardware-assisted optimization of synchronization and multioperation primitives give speedup factors higher than 750% and 1300%, respectively [18, 20].

In order to estimate the efficiency of our tools we determined the number of steps for a set of language primitives and compared them to those reported for fcc on a SB-PRAM [11] (see Table 3). Statements are assumed to be executed once and the execution time of substatements and possible waiting time are ignored. The fractional values should be interpreted so that they indicate the num-

ber of subinstructions needed to complete the primitive. Our compiler tool is currently in the early development stage leaving room for further development and optimization in especially for synchronous control structure implementations. With a good procedure inliner it could be possible to eliminate the subroutine-wise register allocation and call/return time components and achieve the base values dropping the performance of synchronous control structures to 40% to 112% of the performance of the Fork on SB-PRAM featuring much stronger computing model providing full support to CRCW and multiprefix operations. On the other hand, the variable referencing was at least 200% to 600% faster than that in Fork on SB-PRAM.

Table 3. Number of steps for e and Fork primitives compiled with ec and fcc, respectively (F =number of functional units, G =number of groups on path declaring->using group)

Primitive	Measured ec on a Eclipse with F functional units				fcc on SB-PRAM
	Base	Reg. alloc.	Call/ret.	Total	
Startup code				26	150
Barrier	16	7	9	32	18
private synch if-else	65	11	18	94	50
private synch while	83	15	27	125	33
private synch for	82	15	27	124	33
read private local variable				$2/F$	3
read private global variable				$1/F$	4
read shared local variable				$2/F$	$5 + 2G$
read shared global variable				$1/F$	6
write private local variable				$2/F$	2
write private global variable				$1/F$	3
write shared local variable				$2/F$	$4 + 2G$
write shared global variable				$1/F$	5
load a 32-bit constant				$1/F$	2

6. Conclusions

We have described how an experimental compiler for programming language e, aimed for fine-grained TLP programming on synchronous shared memory architectures realizing the PRAM programming model, can be set up by using a standard c-compiler. The example compiler is based on existing gnu c-compiler and implemented with macros, libraries and start-up code as well as some modifications to the loader and simulator/hardware. With the tools provided by the e-language a programmer is able to express the (parallel) functionality in a sophisticated way, make sure that the application is executed efficiently in the TLP hardware, and avoid pitfalls of current shared memory programming. According to our evaluation with the experimental e-compiler on the scalable Eclipse SMA, we get ease of use and scalable performance. In addition to standard c optimizations, the compiler is attached to a special virtual ILP optimizer scheduling the sequential stream of instructions for Eclipse's two-component computing model. The overall speedup factors to achieved with optimizations from a non-optimized version range from 2.0 to 9.6. We also compared the execution times of several e primitives compiled with e-compiler to an Eclipse to those achieved with Fork compiled with fcc to a SB-PRAM and found that the former provides much faster variable referencing, but somewhat slower synchronization and synchronous statement execution times than the latter.

Our future work includes refining some details of the language and the compiler. We plan also a thorough evaluation of the whole application development scheme being developed for synchronous SMAs using general purpose parallel applications as well as some digital signal processing domain applications [12]. Finally, we hope to be able to enhance our Eclipse SMA so that it would provide full support for stronger PRAM models like CRCW. The first steps towards this direction have already been taken [18]. From the e-language and e-compiler side there is no obstacle to use such a strong model even now.

References

1. Allan, A., et. al.: 2001 Technology Roadmap for Semiconductors, *Computer* 35, 1 (January 2002), 42-53.
2. Jouppi, N., Wall, D.: Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines, *Proceedings of the ASPLOS* 3, April 1989, 272-282.
3. Forsell, M.: Architectural differences of efficient sequential and parallel computers, *Journal of Systems Architecture* 47, 13 (July 2002), 1017-1041.
4. Forsell, M.: Implementation of Instruction-Level and Thread-Level Parallelism in Computers, *Dissertations* 2, Department of Computer Science, University of Joensuu, Finland, 1997.
5. Vishkin, U., Dascal, S., Berkovich, E., Nuzman, J.: Explicit Multithreading (XMT) Bridging Models for Instruction Parallelism, *UMIACS TR-98-05*, 1998.
6. Forsell, M.: A Scalable High-Performance Computing Solution for Network on Chips, *IEEE Micro* 22, 5 (September-October 2002), 46-55.
7. Fortune, S., Wyllie, J.: Parallelism in Random Access Machines, *Proceedings of 10th ACM STOC*, Association for Computing Machinery, New York, 1978, 114-118.
8. Jantsch, A., Tenhunen, H. (ed.): *Networks on Chip*, Kluwer Academic Publishers, Boston, 2003.
9. Forsell, M.: Using Parallel Slackness for Extracting ILP from Sequential Threads, *Proceedings of the SSRR-2003s*, International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet, July 28 - August 3, 2003, L'Aquila, Italy.
10. Forsell, M.: E—A Language for Thread-Level Parallel Programming on Synchronous Shared Memory NOCs, *WSEAS Transactions on Computers* 3, 3 (July 2004), 807-812.
11. Keller, J., Keßler, C., Träff, J.: *Practical PRAM Programming*, Wiley, New York, 2001.
12. Forsell, M.: Designing NOCs with a parallel extension of c, In the *Proceedings of the Forum on Specification and Design Languages*, September 13-17, 2004, Lille, France, 463-474.
13. Leon, C., Sande, F., Rodriguez, C., Garcia, F.: A PRAM oriented language, *Proceedings of EUROMICRO PDP'95 Workshop on Parallel and Distributed Processing*, January 1995, 182-191.
14. Naishlos, D., Nuzman, J., Tseng, C-W., Vishkin, U.: Towards a First Vertical Prototyping of an Extremely Fine-Grained Parallel Programming Approach, *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures*, July 2001, 93-102.
15. Juvaste, S.: The programming language pm2 for PRAM, *Technical Report B-1992-1*, Department of Computer Science, University of Joensuu, Finland, 1992.
16. Philippsen, M., Tichy, W.: Compiling for massively parallel machines, in *Code Generation: Concepts, Tools and Techniques* (editors R. Giegerich and S. Graham), Springer Workshops in Computing, 92-111, 1992.
17. Forsell, M.: Advanced Simulation Environment for Shared Memory Network-on-Chips, *Proceedings of the 20th IEEE NORCHIP Conference*, November 11-12, 2002, Copenhagen, Denmark, 31-36.
18. Forsell, M.: Efficient Barrier Synchronization Mechanism for Emulated Shared Memory NOCs, to appear in the *Proceedings of the International Symposium on System-on-Chip 2004*, November 16-18, 2004, Tampere, Finland.
19. Jaja, J.: *Introduction to Parallel Algorithms*, Addison-Wesley, Reading, 1992.
20. Forsell, M.: Realizing constant time parallel algorithms with active memory modules, to appear in the *International Journal of Electronics Business*, 2004.

Elimination of Local Variables from Definite Logic Programs ^{*}

Javier Álvarez and Paqui Lucio

Dpto. de LSI de la UPV\EHU. Paseo Manuel de Lardizabal, 1.
20080 - San Sebastian, Spain.
{jibalgij,jiplucap}@si.ehu.es

Abstract. In logic programming, a variable is said to be local if it occurs in a clause body but not in its head. It is well-known that local variables are the main cause of inefficiency (sometimes even incompleteness) in negative goal computation. The problem is that universal quantification is unavoidable for computing the negation of a clause body with some local variable and it is not easy to compute universal quantification in an efficient manner. In this paper, we introduce an effective method that takes any definite logic program and yields an equivalent definite logic program without local variables. Source and target programs are equivalent w.r.t. three-valued logical consequences of program completion. In further work, we plan to extend our results to normal logic programs.

Keywords: program transformation, local variables, logic programming.

1 Introduction

Local variables are very often used in logic programs to store intermediate results that are passed from one atom to another in a clause body. It is well-known that local variables cause several problems for solving negative goals, since they give raise to unavoidable universal quantification in the negation of a clause body. Depending on the LP or CLP approach, universal quantification affects simple goals or constrained goals. In the so-called *intensional negation* (cf. [2]) for the LP approach, universal quantification prevents to achieve a complete goal computation mechanism. Afterwards, constructive negation was introduced in [4, 5] and extended in [8, 15] to a complete and sound operational semantics for the whole class of normal logic programs in the CLP framework. Intensional negation was also extended to CLP in [3] where a complete operational semantics is provided. The computation mechanisms proposed in [3, 8, 15] deal with universally quantified (constrained) goals that, in general, are not easy to compute in an efficient manner. Besides, in [2] the so-called *negation technique* is introduced and local variable absence is claimed as a sufficient condition for the completeness of the technique. In the negation technique, the target program P' contains

^{*} This work has been partially supported by the Spanish Project TIC 2001-2476-C03.

a new predicate p' for each predicate p in the source program P , such that the computation of a negative literal $\neg p(\bar{t})$ in P is simulated by $p'(\bar{t})$ in P' .

In this paper, we present an effective transformation method for eliminating local variables from definite logic programs. The underlying aim is to improve the performance of a practical implementation of constructive negation (cf. [1]). This transformation provides a way for efficient computation of negative queries w.r.t. a definite logic program. Efficiency is achieved because: (1) the negative query is computed w.r.t. an equivalent definite logic program that does not contain local variables, hence universal quantification is avoided; and (2) the target program is built at compilation time. We would like to remark that the transformed program (without local variables) must be used only to compute negative literals, using the original for positive literals. Source and target programs are equivalent w.r.t. three-valued logical consequences of program completion. This is the standard Clark-Kunen semantics for normal (in particular, definite) logic programs. In further work, we plan to extend our results to normal logic programs.

We distinguish two classes of local variables: *auxiliary* and *isolated*. Auxiliary variables are the most common kind of local variables. They are local variables that appear in at least two different atoms, passing intermediate results. By the contrary, isolated variables occur exactly in one body atom, does not matter how many times. The problem of local variable elimination is reduced to auxiliary variable elimination since, for any isolated variable y , we can add an atom $p_new(y)$ (where p_new is a fresh predicate symbol) both to the body and to the program (as a fact). Then, y becomes an auxiliary variable. Of course, a real implementation uses this idea without introducing new predicates. Hence, along the rest of the paper we concentrate on auxiliary variable elimination.

This method is fold/unfold-based because the correctness of the transformations is given by an fold/unfold transformation sequence. Besides, it relies in a preliminary partition of the arguments in the atoms. This partition, called *mode specification*, associates a mode (input/output) to each argument position, that is inferred automatically. The inference depends on the auxiliary variables to be eliminated. The mode specification is only used during the auxiliary variable elimination and it has neither to do with restricting user-goals nor with the dataflow assumed by the programmer. Mode analysis and specification is used for several purposes such as compiler optimization, parallel goal-evaluation, etc. (for instance [7, 10]), which are far from the aim of this work.

Outline of the paper. In the next section we give some preliminary definitions. The third section introduces the notion of mode specification. In Section 4 we show how to eliminate all the auxiliary variables from a definite program in several phases. Finally, we give some conclusions and reflections about the presented, future and related work.

2 Preliminaries

Every program P is built from symbols of a *signature* $\Sigma \equiv \{FS_\Sigma, PS_\Sigma\}$ of function and predicate symbols respectively, and variables from X . Both function

and predicate symbols have associated a number $n \geq 0$, called its *arity*. A Σ -term is either a variable or a n -ary function symbol of FS_Σ applied to n Σ -terms. A bar is used to denote tuples, or finite sequences, of objects, like $\bar{x}$ as abbreviation of the n -tuple of variables $\langle x_1, \dots, x_n \rangle$. Concatenation of sequences is denoted by the infix operator $\cdot$ and $\langle \rangle$ stands for the empty sequence. We use the symbols $\setminus$ and $\cap$ as binary infix operators for difference and intersection over sequences, respectively, with the obvious meaning inherit from sets. From now on, t, r, s denote terms and x, y, z variables, possibly with bar and sub/super-scripts.

A *substitution* σ is a mapping from a finite set of variables, called its domain, into a set of terms. It is assumed that σ behaves as the identity for the variables outside its domain. The *most general unifier* of a set of terms $\{s_1, \dots, s_n\}$, denoted $mgu(\bar{s})$, is an idempotent substitution σ such that $\sigma(s_i) \equiv \sigma(s_j)$ for all $i, j \in 1..n$ and for any other substitution θ with the same property, $\theta \equiv \sigma' \cdot \sigma$ holds for some substitution σ' .

A Σ -atom $p(\bar{t})$ is a n -ary predicate symbol $p \in PS_\Sigma$ applied to a n -tuple of Σ -terms $\bar{t}$. We also use the two logical constants *True* and *False* as atoms. $Form(\Sigma)$ stands for the set of first-order Σ -formulas that can be built using predicate symbols from $PS_\Sigma \cup \{=\}$, connectives from $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ and quantifiers from $\{\forall, \exists\}$. The universal closure of a formula φ is denoted by $\varphi^\forall$. The *three-valued logical consequence* relation, between set of Σ -formulas and Σ -formulas, is denoted by the infix symbol $\models_3$.

A (definite) *clause* C is an expression of the form $a : - \bar{L}$ where a (*head*) is an atom and $\bar{L} \equiv a_1, \dots, a_m$ (*body*) is a conjunction of atoms a_i for $1 \leq i \leq m$. When the body is empty (or equivalent to *True*), the clause a is called a *fact*.

Let α be any syntactic object, we denote by $Var(\alpha)$ the sequence of all variables occurring in α . We call *auxiliary variables* to the variables that appear in at least two different atoms of a clause body but not appear in the clause head. We denote by $AuxVar(\alpha)$ the set of auxiliary variables of the object α .

A (definite) *program* P is defined by a collection of (definite) clauses. We use the term Σ -program whenever the signature is relevant. For a predicate p , we denote by $Def_P(p)$ the set of all clauses in P with head p . All definitions in this paper are given modulo reordering of the clause bodies and standardization apart is always assumed. Given $p, q \in PS_\Sigma$ and a Σ -program P , we say that p *directly depends* on q if q occurs in some clause in $Def_P(p)$. By reflexive transitive closure of this relation, we obtain the set $Dpd_P(p)$ such that, w.r.t. program P , p depends on all predicates in $Dpd_P(p)$ ($p \in Dpd_P(p)$ for every p). Besides, $MR_P(p) \equiv \bigcup \{q \mid q \in Dpd_P(p) \text{ and } p \in Dpd_P(q)\}$ is the set of all predicates that are mutually recursive with p and $MRDef_P(p) \equiv \bigcup \{Def_P(q) \mid q \in MR_P(p)\}$.

The standard declarative meaning of normal (in particular, definite) logic programs is the program completion (proposed by Clark in [6]) interpreted in three-valued logic (as proposed by [11]), that is also known as Clark-Kunen semantics. The *predicate completion formula* of a predicate p such that $Def_P(p) \equiv \{p(\bar{t}^i) : - \bar{q}^i(\bar{s}^i) \mid i \in 1..m\}$ is the sentence:

$$\forall \bar{x} (p(\bar{x}) \leftrightarrow \bigvee_{i=1}^m \exists \bar{y}^i (\bar{x} = \bar{t}^i \wedge \bar{q}^i(\bar{s}^i)))$$

where each $\bar{y}^i \equiv \text{Var}(\bar{t}^i) \cdot \text{Var}(\bar{s}^i)$. The *Clark's completion* of a program P , namely $\text{Comp}(P)$, consists of the universal closure of the set P^* of the predicate completion formulas for every $p \in PS_\Sigma$ together with the *free equality theory* $FET(\Sigma)$. Program transformation preserves some equivalence relation on programs. Different equivalence relations are induced by the different semantics (see [12] for a systematic comparison). Since we plan to extend our results to normal programs, we are interested in completion semantics. Shepherdson's operators T^P and F^P (for a program P) were introduced in [14] and provide a characterization of the three-valued logical consequences of program completion.

Definition 1. ([14]) Let $p \in PS_\Sigma$ defined by a set of clauses $\text{Def}_P(p)$ as above:

$$\begin{aligned} T_0^P(p(\bar{x})) &\equiv \text{False} & T_{n+1}^P(p(\bar{x})) &\equiv \bigvee_{i=1}^m \exists \bar{y}^i (\bar{x} = \bar{t}^i \wedge T_n^P(\bar{q}^i(\bar{s}^i))) \\ F_0^P(p(\bar{x})) &\equiv \text{False} & F_{n+1}^P(p(\bar{x})) &\equiv \bigwedge_{i=1}^m \neg \exists \bar{y}^i (\bar{x} = \bar{t}^i \wedge \neg F_n^P(\bar{q}^i(\bar{s}^i))) \end{aligned}$$

where $\bar{y}^i \equiv \text{Var}(\bar{t}^i) \cdot \text{Var}(\bar{s}^i)$. Besides, $T_n(\text{True}) \equiv \text{True}$ and $F_n(\text{True}) \equiv \text{False}$ for all n . The extension to connectives and quantifiers is the obvious one. ■

Notice that $T_k^P(p(\bar{x}))$ and $F_k^P(p(\bar{x}))$ are an equality formulas that represent the successes and failures of the atom $p(\bar{x})$ which can be derived from P in k steps and that facts produce the one level successes, since their clause body is *True*.

Theorem 2. For any normal (in particular, definite) program P and any sentence φ , $\text{Comp}(P) \models_3 \varphi \iff$ there exists $n \in \mathbb{N}$ such that $FET(\Sigma) \models_3 T_n^P(\varphi)$.

Proof. Lemma 4.1 in [14] proves the equivalence $\Phi_P^n \models_3 \varphi \iff FET(\Sigma) \models_3 T_n^P(\varphi)$ for all $n \in \mathbb{N}$, where Φ_P^n is the n -iteration of Fitting's immediate consequence operator Φ_P (cf. [9]). Besides, Kunen's Theorem 6.3 in [11] proves that $\text{Comp}(P) \models_3 \varphi \iff$ there exists $n \in \mathbb{N}$ such that $\Phi_P^n \models_3 \varphi$. ■

As a consequence, the three equivalence relations respectively induced on the set of programs by Fitting's and Shepherdson's operators and by logical consequence of program completion have the same strength. *TF*-equivalence is a useful representative of these three notions. Below, we give a more precise definition of the *TF*-equivalence ($\overset{TF}{\cong}$) relation on programs. A strictly stronger equivalence relation is given by logical equivalence of program completions. That means to require $FET(\Sigma) \models_3 \text{Comp}(P_1) \leftrightarrow \text{Comp}(P_2)$ for the equivalence of the programs P_1 and P_2 . The interested reader is referred to [12] for an example of *TF*-equivalent programs whose completions are not logically equivalent.

Definition 3. Let P_1 and P_2 be two Σ -programs:

- (i) $P_1 \preceq_{TF} P_2$ iff for all $p \in PS_\Sigma^1 \cap PS_\Sigma^2$ and for all $k \in \mathbb{N}$, there exist $k' \in \mathbb{N}$ such that $FET(\Sigma) \models_3 (T_k^{P_1}(p(\bar{x})) \rightarrow T_{k'}^{P_2}(p(\bar{x})) \wedge F_k^{P_1}(p(\bar{x})) \rightarrow F_{k'}^{P_2}(p(\bar{x})))^\forall$
- (ii) $P_1 \overset{TF}{\cong} P_2$ iff $P_1 \preceq_{TF} P_2$ and $P_2 \preceq_{TF} P_1$. ■

Intuitively, TF -equivalent programs have equivalent sets of answers, but not necessarily obtained at the same iteration step. We transform a Σ -program into a TF -equivalent Σ' -program without auxiliary variables, where $\Sigma' \supseteq \Sigma$.

3 Mode Specification

In a given atom $p(t_1, \dots, t_n)$, we distinguish between the argument position j for some $j \in \{1, \dots, n\}$ and the corresponding argument t_j . We associate a *mode*, *input* (**in**) or *output* (**out**), to each argument position. The resulting partition of the n argument positions is a *mode specification* for $p(\bar{t})$. In this section, we explain how to *automatically infer a mode specification*, starting with the two atoms that share the auxiliary variables to be eliminated (see Definition 19). Then, we extend the mode specification to the program clauses related by computational dependence with these atoms.

Definition 4. The *mode specification* of an atom a (with n -ary predicate p) is either $\perp$ or a n -tuple $\text{MS}(a) \equiv (\text{MS}_1(a), \dots, \text{MS}_n(a)) \subseteq \{\text{in}, \text{out}\}^n$. $\text{MS}(a)$ is *undefined* if it is $\perp$. Otherwise, it is *defined*. ■

Definition 5. Let $C \equiv B : -B_1, \dots, B_m$ be a program clause, its *mode specification* $\text{MS}(C)$ is a $m+1$ -tuple $(\text{MS}^0(B), \text{MS}^1(B_1), \dots, \text{MS}^m(B_m))$. If $\text{MS}^i(a)$ is defined for $0 \leq i \leq m$, then $\text{MS}(C)$ is *total*. Otherwise, $\text{MS}(C)$ is *partial*. ■

We would like to remark that mode specifications of atoms are about positions, but not about terms being their actual holders. Notice that the same term (in particular, variable) could occur in distinct atoms (or, even, in the same atom) in positions with different modes. Moreover, the mode specification does not restrict the goals and has nothing to do with the dataflow assumed by the programmer.

For the rest of the paper, we adopt the following notation that allows us to manage the terms that are actual arguments in input and output positions.

Notational Convention 6. Suppose some fixed mode specification $\text{MS}(p(\bar{t}))$. The atom $p(\bar{t})$ will be written as $p(\langle \bar{t}_I \cdot \bar{t}_O \rangle)$ whenever we want to denote that $\bar{t}_I$ and $\bar{t}_O$ are the order-preserving subsequences of $\bar{t}$ that respectively corresponds with the input and the output positions. ■

For example, $p(\langle g(a), x \rangle \cdot \langle f(a, y), y \rangle)$ denotes the atom $p(f(a, y), g(a), y, x)$ provided that its mode specification is $(\text{out}, \text{in}, \text{out}, \text{in})$.

Now, we show how to extend a mode specification $\text{MS}(p(\bar{t}))$ from the clause head to the clause body.

Definition 7. Let $C \equiv p(\bar{t}) : -B_1, \dots, B_m$ be a clause such that $\text{MS}(C)$ is partial and $\text{MS}^0(p(\bar{t}))$ is defined. $\text{MS}(C)$ is *extended to a total mode specification* by successively assigning a mode specification to the leftmost atom $B_i \equiv q(s_1, \dots, s_n)$ such that $\text{MS}^i(B_i)$ is undefined, accordingly with the following criteria and for each argument position $1 \leq j \leq n$:

- If $Var(s_j) \setminus AuxVar(s_j) \neq \langle \rangle$: $MS_j^i(B_i) := \text{in}$ whenever $Var(s_j) \subseteq Var(\bar{t}_I)$. Otherwise, $MS_j^i(B_i) := \text{out}$.
- Otherwise: $MS_j^i(B_i) := \text{out}$ whenever B_i is the leftmost atom where some $y \in AuxVar(s_j)$ occurs in and $MS_j^i(B_i) := \text{in}$ if it is not. ■

Example 8. Consider the following clause C :

$$w_balanced(tree(x_1, x_2, x_3), x_4) : - w_balanced(x_2, y_1), \\ w_balanced(x_3, y_2), equal(y_1, y_2), sum(y_1, y_2, y_3), sum(x_1, y_3, x_4)$$

such that $MS(C) \equiv ((\text{in}, \text{out}), (\text{in}, \text{out}), (\text{in}, \text{out}), \perp, \perp, \perp)$. We extend $MS(C)$ to a total mode specification as follows. $MS(equal(y_1, y_2)) \equiv (\text{in}, \text{in})$ since the auxiliary variables y_1 and y_2 also occur in the previous atom. $MS(sum(y_1, y_2, y_3)) \equiv (\text{in}, \text{in}, \text{out})$ because it is the leftmost occurrence of the auxiliary variable y_3 (the two first arguments, as the previous case). $MS(sum(x_1, y_3, x_4)) \equiv (\text{in}, \text{in}, \text{out})$ since it is not the leftmost occurrence of y_3 and the variables x_1 and x_4 occur in input and output arguments respectively in the clause head. ■

We extend a mode specification $MS(p(\bar{t}))$ to the set of clauses $MRDef_P(p)$.

Definition 9. The *extension of the mode specification* $MS(p(\bar{t}))$ to $MS(MRDef_P(p))$ is made as follows:

1. Assign $MS(p(\bar{t}))$ to every atom of predicate p in any clause $C \in MRDef_P(p)$.
2. For each clause $C \equiv B : - B_1, \dots, B_m \in MRDef_P(p)$ where $MS(C)$ is partial and $MS^0(B)$ is defined, extend $MS(C)$ to a total mode specification (see Definition 7). Besides, for $B_i \equiv q(\bar{s})$ such that $q \in Dpd_P(p)$, the obtained $MS^i(B_i)$ is assigned to every atom $q(\bar{r})$ in any clause in $MRDef_P(p)$. ■

Definition 10. A mode specification $MS(MRDef_P(p))$ is *coherent* iff it is total and for every predicate $q \in MR_P(p)$ the following two conditions hold:

1. The mode specification assigned to every atom $q(\bar{t})$ is equal.
2. For each atom $q(\bar{r})$ in a clause body, $Var(\bar{r}_I^j) \setminus AuxVar(\bar{r}_I^j) \subset Var(\bar{s}_I)$, being $\bar{s}_I$ the terms placed in input arguments in the clause head. ■

Lemma 11. Let P be a program and $MS(p(\bar{t}))$ a mode specification. There exists a program P' such that the extension $MS(MRDef_P(p))$ is coherent and $P \stackrel{TF}{\cong} P'$.

Proof. The first condition is ensured by Definition 9. If some atom $q(\bar{s})$ violates the second one in a clause C , we define a new predicate p' such that $Def_P(q') \equiv Def_P(q)[q'/q]$ and substitute $q'(\bar{s})$ for $q(\bar{s})$ in the clause C . The predicate q' together with the mode specification $MS(q'(\bar{s}))$ obtained after the extension is associated to q . We proceed in this way until obtaining a coherent mode specification. This process clearly ends since the number of possible mode specifications for each predicate is finite, assuming that we do not define a new predicate if there already exists an associated one with the same mode specification. The original and final programs are trivially *TF*-equivalent. ■

4 The Elimination Method

In this section, we will show the method for transforming any definite program in order to eliminate the auxiliary variables and preserving TF -equivalence. In the next three subsections, we explain the details of the main three subtask: (a) to normalize a clause C w.r.t. the auxiliary variable occurrences; (b) to transform the definition of a predicate p into a tail recursive definition (w.r.t. a mode specification); (c) to eliminate the auxiliary variables that occur in a pair of atoms of a normalized clause C . In a forth subsection, we give an algorithm that, using this three subtasks, accomplishes the auxiliary variable elimination.

4.1 Auxiliary Variable Normalization

Definition 12. A clause $p(\bar{t}) : -p_1(\bar{s}^1), \dots, p_m(\bar{s}^m) \in P$ is *normalized w.r.t. the auxiliary variables* (or *normalized* for short) iff each auxiliary variable exactly occurs in $p_j(\bar{s}^j)$ and $p_{j+1}(\bar{s}^{j+1})$ for some $1 \leq j \leq m-1$. ■

That is, each auxiliary variable exactly occurs in two consecutive atoms and does not occur anymore in the clause.

Lemma 13. Any program P can be transformed into a TF -equivalent normalized program P' such that each clause C' in the program P' is normalized.

Proof. Let $C \equiv p(\bar{t}) : -p_1(\bar{s}^1), \dots, p_m(\bar{s}^m)$ be any clause in P such that C is not normalized. Let $p_j(\bar{s}^j)$ be the leftmost atom such that some auxiliary variable y occurs in and either: (1) y does not occur in the next atom $p_{j+1}(\bar{s}^{j+1})$, or (2) it also occurs in $p_k(\bar{s}^k)$ for some $j+2 \leq k \leq m$. We define a new clause C' by:

- Replacing in C the atom $p_{j+1}(\bar{s}^{j+1})$ with $p'_{j+1}(\bar{r})$ where: (a) p'_{j+1} is a new predicate, and (b) $\bar{r} \equiv \bar{s}^{j+1} \cdot y \cdot y'$, being y' a new auxiliary variable.
- In the resulting clause, replacing each occurrence of the variable y in the atoms $p_k(\bar{s}^k)$ for every $j+2 \leq k \leq m$ with the new variable y' .

and we add a new clause $p'_{j+1}(\bar{x} \cdot z \cdot z) : -p_{j+1}(\bar{x})$ where there is a distinct variable x_i in $\bar{x}$ for each term in $\bar{s}^{j+1}$, all different from z . The new clause C' is normalized w.r.t. the auxiliary variable y . If necessary, this process is repeated until obtaining a set of normalized clauses w.r.t. the auxiliary variables. Notice that the process clearly ends. The original and final programs are TF -equivalent by unfolding each introduced atom p'_{j+1} using the corresponding clause. ■

Example 14. Given the following program:

E14.1 : $preorder(nil, [])$
 E14.2 : $preorder(tree(x_1, x_2, x_3), x_4) : -preorder(x_2, y_1),$
 $preorder(x_3, y_2), append([x_1|y_1], y_2, x_4)$

We obtain a TF -equivalent normalized program by substituting the following two clauses for the clause E14.2:

E14.3 : $preorder(tree(x_1, x_2, x_3), x_4) : -preorder(x_2, y_1),$
 $preorder'(x_3, y_2, y_1, y_3), append([x_1|y_3], y_2, x_4)$
 E14.4 : $preorder'(x_1, x_2, x_3, x_3) : -preorder(x_1, x_2)$ ■

4.2 Tail Recursive Transformation

A Prolog-predicate is said to be tail recursive whenever the recursive call is made by the right-most literal and no more computation is made after. We relativize tail recursion to the specification mode, defining a slightly stronger notion.

Definition 15. The definition $\text{Def}_P(p)$ of a predicate p in a program P is *tail recursive w.r.t. a mode specification* $\text{MS}(p(\bar{t}))$ iff it exclusively consists of clauses of one of the following two forms (in notation of Convention 6):

- (1) $p(\bar{r}_1 \cdot \bar{r}_0) : - \bar{L}$
- (2) $p(\bar{s}_1 \cdot \bar{z}) : - \bar{M}, p(\bar{s}'_1 \cdot \bar{z})$

where $p \notin \text{Dpd}_P(q)$ (that is, $q \notin \text{MR}_P(p)$) for any atom q in $\bar{L}$ and $\bar{M}$, and $\bar{z}$ is a tuple of pairwise distinct variables. ■

We show how to transform a definition into a TF -equivalent tail recursive definition w.r.t. a mode specification. It is based on the well-known technique that uses a stack for storing the recursive calls and we use new constants with predicate names and clauses as super/sub-scripts to be stored in the stack.

Definition 16. The *tail recursive definition* $\text{TailRDef}_P(p)$ of a predicate p w.r.t. a mode specification $\text{MS}(\text{MRDef}_P(p))$ is defined by the following clauses:

- The following two initial clauses:

$$(1) \quad p(\bar{x} \cdot \bar{z}) : - p_tr(\llbracket \bar{x}, c_p \rrbracket \cdot \bar{z}) \qquad (2) \quad p_tr(\llbracket \bar{x} \rrbracket \cdot \bar{x})$$

where p_tr is a new predicate and the constant c_p corresponds to p .

- For each clause $h(\bar{s}_1 \cdot \bar{s}_0) : - q_1(\bar{r}_1^1 \cdot \bar{r}_0^1), \dots, q_n(\bar{r}_1^n \cdot \bar{r}_0^n) \in \text{MRDef}_P(p)$ such that $\{q_1, \dots, q_n\} \cap \text{MR}_P(p) = \emptyset$, a clause of the form (c_h corresponds to h):

$$(3) \quad p_tr(\llbracket \bar{s}_1, c_h | S \rrbracket \cdot \bar{z}) : - q_1(\bar{r}_1^1 \cdot \bar{r}_0^1), \dots, q_n(\bar{r}_1^n \cdot \bar{r}_0^n), p_tr(\llbracket \bar{s}_0 | S \rrbracket \cdot \bar{z})$$

- For each clause $C \equiv h(\bar{s}_1 \cdot \bar{s}_0) : - q_1(\bar{r}_1^1 \cdot \bar{r}_0^1), \dots, q_n(\bar{r}_1^n \cdot \bar{r}_0^n)$ such that $\{q_1, \dots, q_n\} \cap \text{MR}_P(p) \neq \emptyset$ for some $1 \leq j \leq n$, two clauses of the form:

$$(4) \quad p_tr(\llbracket \bar{s}_1, c_h | S \rrbracket \cdot \bar{z}) : - p_tr(\llbracket \bar{s}_1, \bar{w}^1, c_{q_1}^C, \dots, \bar{w}^n, c_{q_n}^C, \bar{w}, c_h^C | S \rrbracket \cdot \bar{z})$$

$$(5) \quad p_tr(\llbracket \bar{r}_0^n, \bar{w}, c_h^C | S \rrbracket \cdot \bar{z}) : - p_tr(\llbracket \bar{s}_0 | S \rrbracket \cdot \bar{z})$$

Besides, we add one of the following two clauses for each $1 \leq j \leq n$: the clause (6) with $\bar{w}^j \equiv \text{Var}(\bar{r}_1^j \cdot \bar{r}_0^j) \setminus \text{Var}(\bar{r}_0^{j-1})$ if $q_j \notin \text{MR}_P(p)$ or the clause (6') with $\bar{w}^j \equiv \text{Var}(\bar{r}_1^j) \setminus \text{Var}(\bar{r}_0^{j-1})$ otherwise¹:

$$(6) \quad p_tr(\llbracket \bar{r}_0^{j-1}, \bar{w}^j, c_{q_j}^C | S \rrbracket \cdot \bar{z}) : - q_j(\bar{r}_1^j \cdot \bar{r}_0^j), p_tr(\llbracket \bar{r}_0^j | S \rrbracket \cdot \bar{z})$$

$$(6') \quad p_tr(\llbracket \bar{r}_0^{j-1}, \bar{w}^j, c_{q_j}^C | S \rrbracket \cdot \bar{z}) : - p_tr(\llbracket \bar{r}_1^j, c_{q_j}^C | S \rrbracket \cdot \bar{z})$$

where $\bar{w} \equiv \text{Var}(\bar{s}_0) \setminus \text{Var}(\bar{r}_0^n)$ and the distinct constants correspond to the atoms in the clause C . ■

¹ By convention: $\bar{r}_0^{(-1)} \equiv \bar{s}_1$.

Theorem 17. *Let P be a normalized program and $\text{MS}(\text{MRDef}_P(p))$ be a coherent mode specification, then P and $P \setminus \text{Def}_P(p) \cup \text{TailRDef}_P(p)$ are TF -equivalent.*

Proof. We define a new program P' composed of the following clauses:

$$\begin{aligned}
C17.1 &: p_tr(\llbracket \bar{x} \rrbracket \cdot \bar{x}) \\
C17.2 &: p_tr(\llbracket x_1, x_2 | S \rrbracket \cdot \bar{z}) : - h'(\llbracket x_1, x_2 \rrbracket \cdot x_3), p_tr(\llbracket x_3 | S \rrbracket \cdot \bar{z}) \\
C17.3 &: p_tr(\llbracket x_1, x_2, x_3 | S \rrbracket \cdot \bar{z}) : - h'(\llbracket x_1, x_2, x_3 \rrbracket \cdot x_4), p_tr(\llbracket x_4 | S \rrbracket \cdot \bar{z}) \\
C17.4^* &: h(\llbracket \bar{x} \rrbracket \cdot \bar{z}) : - h'(\llbracket \bar{x}, c_h \rrbracket \cdot \bar{z}) \\
C17.5^* &: h'(\llbracket \bar{s}_1, c_h \rrbracket \cdot \bar{s}_0) : - q_1(\bar{r}_1^1 \cdot \bar{r}_0^1), \dots, q_n(\bar{r}_1^n \cdot \bar{r}_0^n) \\
C17.6^* &: h'(\llbracket \bar{s}_1, c_h \rrbracket \cdot \bar{z}) : - h'(\llbracket \bar{r}_1^1, \bar{w}^1, c_{q_1}^C \rrbracket \cdot \bar{r}_0^1), \dots, h'(\llbracket \bar{r}_1^n, \bar{w}^n, c_{q_n}^C \rrbracket \cdot \bar{r}_0^n), \\
&\quad h'(\llbracket \bar{s}_0, \bar{w}, c_h^C \rrbracket \cdot \bar{z}) \\
C17.7^* &: h'(\llbracket \bar{s}_0, \bar{w}, c_h^C \rrbracket \cdot \bar{s}_0) \\
C17.8^* &: h'(\llbracket \bar{r}_1^j, \bar{w}^j, c_{q_j}^C \rrbracket \cdot \bar{r}_0^j) : - q_j(\bar{r}_1^j \cdot \bar{r}_0^j)
\end{aligned}$$

where p_tr is a new predicate and the clauses marked with $*$ are schemes. More precisely, for every $h \in \text{MR}_P(p)$ we have: one clause $C17.4$; a clause $C17.5$ for each $C \equiv h(\bar{s}_i, \bar{s}_o) : - q_1(\bar{r}_i^1, \bar{r}_o^1), \dots, q_n(\bar{r}_i^n, \bar{r}_o^n) \in \text{Def}_P(h)$ such that $q_j \notin \text{MR}_P(p)$ for all $1 \leq j \leq n$; and a clause $C17.6$, a clause $C17.7$ and n clauses $C17.8$ for each $C \equiv h(\bar{s}_i, \bar{s}_o) : - q_1(\bar{r}_i^1, \bar{r}_o^1), \dots, q_n(\bar{r}_i^n, \bar{r}_o^n) \in \text{Def}_P(h)$ such that $q_j \in \text{MR}_P(p)$ for some $1 \leq j \leq n$. The sets variables and new constants are like in Definition 16. Since $\text{MS}(\text{MRDef}_P(p))$ is coherent, the distinct sets of variables $\bar{w}$ and $\bar{w}^j$ ensure that no new local variable is generated. P and P' are TF -equivalent since by unfolding $h'(\llbracket \bar{x}, c_h \rrbracket \cdot \bar{z})$ in each clause $C17.4$ and then unfolding each atom $h'(\llbracket \bar{r}_i^j, \bar{w}^j, c_{q_j}^C \rrbracket \cdot \bar{r}_o^j)$ in the resulting clauses, we obtain P . To see that P' and $P \setminus \text{Def}_P(p) \cup \text{TailRDef}_P(p)$ are TF -equivalent, it suffices to unfold $h'(\llbracket x_1, x_2 \rrbracket \cdot x_3)$ and $h'(\llbracket x_1, x_2, x_3 \rrbracket \cdot x_4)$ in the clauses $C17.2$ and $C17.3$, respectively. We then fold each tuple of atoms of predicates h' and p_tr using the clauses $C17.2$ and $C17.3$ in the obtained ones, generating each time a new atom of predicate p_tr . When no atom of predicate h' remains, we get $P \setminus \text{Def}_P(p) \cup \text{TailRDef}_P(p)$. ■

Example 18. Given the normalized program defined by the clauses $E14.1$, $E14.3$ and $E14.4$ in the Example 14 and the mode specification $\text{MS}(\text{preorder}(x', x'')) \equiv (\text{in}, \text{out})$, this mode specification is extended to $\text{MRDef}_P(\text{preorder})$ as follows:

$$\begin{aligned}
\text{MS}(E14.1) &\equiv ((\text{in}, \text{out})) \\
\text{MS}(E14.3) &\equiv ((\text{in}, \text{out}), (\text{in}, \text{out}), (\text{in}, \text{out}, \text{in}, \text{out}), (\text{in}, \text{in}, \text{out})) \\
\text{MS}(E14.4) &\equiv ((\text{in}, \text{out}, \text{in}, \text{out}), (\text{in}, \text{out}))
\end{aligned}$$

Then, $\text{TailRDef}_P(\text{preorder})$ w.r.t. $\text{MS}(\text{MRDef}_P(\text{preorder}))$ is defined by²:

$$\begin{aligned}
E18.1 &: \text{preorder}(x, z) : - \text{preorder_tr}([x, c_1], z) \\
E18.2 &: \text{preorder_tr}([x], x) \\
E18.3 &: \text{preorder_tr}([nil, c_1 | S], z) : - \text{preorder_tr}([] | S], z) \\
E18.4 &: \text{preorder_tr}([tree(x_1, x_2, x_3), c_1 | S], z) : - \\
&\quad \text{preorder_tr}([tree(x_1, x_2, x_3), [], c_2, [x_3], c_3, [x_1], c_4, [], c_5 | S], z) \\
E18.5 &: \text{preorder_tr}([tree(x_1, x_2, x_3), [], c_2 | S], z) : - \text{preorder_tr}([x_2, c_1 | S], z)
\end{aligned}$$

² For brevity, we use numbers, instead of predicates names, as constant sub-scripts.

$$\begin{aligned}
E18.6 : & \text{preorder_tr}([y_1, [x_3], c_3|S], z) :- \text{preorder_tr}([(x_3, y_1), c_6|S], z) \\
E18.7 : & \text{preorder_tr}([(y_2, y_3), [x_1], c_4|S], z) :- \text{append}([x_1|y_3], y_2, y), \\
& \quad \text{preorder_tr}([y|S], z) \\
E18.8 : & \text{preorder_tr}([x_4, [], c_5|S], z) :- \text{preorder_tr}([x_4|S], z) \\
E18.9 : & \text{preorder_tr}([(x_1, x_3), c_6|S], z) :- \\
& \quad \text{preorder_tr}([(x_1, x_3), [], c_7, [x_3], c_8|S], z) \\
E18.10 : & \text{preorder_tr}([(x_1, x_3), [], c_7|S], z) :- \text{preorder_tr}([x_1, c_1|S], z) \\
E18.11 : & \text{preorder_tr}([x_2, [x_3], c_8|S], z) :- \text{preorder_tr}([(x_2, x_3)|S], z) \quad \blacksquare
\end{aligned}$$

Notice that for every program P , after transforming the definition of a predicate p in this way, no other predicate is mutually recursive to p . That is, $\text{MR}_P(p) \equiv \{p\}$.

4.3 A Single Step of Auxiliary Variable Elimination

In this subsection, we introduce the method for eliminating the auxiliary variables from the leftmost pair of consecutive atoms that contains any. To do that, we first assign a mode specification to these two consecutive atoms as follows.

Definition 19. Let $C \equiv h(\bar{t}) :- \bar{N}, p_j(\bar{t}^j), p_{j+1}(\bar{t}^{j+1}), \bar{O}$ be a normalized clause such that $p_j(\bar{t}^j)$ and $p_{j+1}(\bar{t}^{j+1})$ is the leftmost pair of consecutive atoms that contain auxiliary variables. We define $\text{MS}(p_j(\bar{t}^j))$ and $\text{MS}(p_{j+1}(\bar{t}^{j+1}))$ as follows:

- For each $1 \leq k \leq m$ (being m the arity of p_j): $\text{MS}_k^j(p_j(\bar{t}^j)) := \text{out}$ whenever $\text{Var}(t_k^j) \cap \text{AuxVar}(\bar{t}^j \cdot \bar{t}^{j+1}) \neq \langle \rangle$. Otherwise, $\text{MS}_k^j(p_j(\bar{t}^j)) := \text{in}$.
- For each $1 \leq k \leq n$ (being n the arity of p_{j+1}): $\text{MS}_k^{j+1}(p_{j+1}(\bar{t}^{j+1})) := \text{in}$ if $\text{Var}(t_k^{j+1}) \cap \text{AuxVar}(\bar{t}^j \cdot \bar{t}^{j+1}) \neq \langle \rangle$. Otherwise, it is **out**. $\blacksquare$

That is, in the left-hand atom, we assign the mode **out** to the argument positions whose terms contain auxiliary variables and **in** to the remaining ones. We do the opposite in the right-hand atom. Now, we explain how to eliminate the auxiliary variables that these atoms share:

Definition 20. Let $C \equiv h(\bar{t}) :- \bar{N}, p_j(\bar{t}_1^j \cdot \bar{t}_0^j), p_{j+1}(\bar{t}_1^{j+1} \cdot \bar{t}_0^{j+1}), \bar{O}$ be a clause in a normalized program P such that Definition 19 holds, $p_j \notin \text{MR}_P(h)$ and $\text{Def}_P(p_j)$ is tail recursive w.r.t. $\text{MS}(p_j(\bar{t}_1^j \cdot \bar{t}_0^j))$. The set of clauses $\text{AVF}(C)$ is defined by:

- One clause:

$$h(\bar{t}) :- \bar{N}, p'(\bar{t}_1^j \cdot \bar{w}) \cdot \bar{t}_0^{j+1}, \bar{O}$$

where p' is new and $\bar{w} \equiv \text{Var}(\bar{t}_0^j \cdot \bar{t}_1^{j+1}) \setminus (\text{AuxVar}(\bar{t}_0^j \cdot \bar{t}_1^{j+1}) \cdot \text{Var}(\bar{t}_1^j \cdot \bar{t}_0^{j+1}))$.

- One clause:

$$p'(\bar{t}_1^j \cdot \bar{w}\sigma) \cdot \bar{z} :- \bar{L}, p_{j+1}(\bar{t}_1^{j+1}\sigma) \cdot \bar{z}$$

for each $p_j(\bar{r}_1 \cdot \bar{r}_0) :- \bar{L} \in \text{Def}_P(p_j)$, where $\sigma \equiv \text{mgu}(\bar{r}_0, \bar{t}_0^j)$ and $\bar{z}$ are fresh.

- One clause:

$$p'(\bar{s}_1 \cdot \bar{w}) \cdot \bar{z} :- \bar{M}, p'(\bar{s}_1' \cdot \bar{w}) \cdot \bar{z}$$

for each clause $p_j(\bar{s}_1 \cdot \bar{z}) :- \bar{M}, p_j(\bar{s}_1' \cdot \bar{z}) \in \text{Def}_P(p_j)$ where $\bar{z}$ are also fresh.

Notice that, since the tuples of terms $\bar{t}_0^j$ and $\bar{t}_1^{j+1}$ have disappeared, we have eliminated the auxiliary variables occurring in the leftmost pair of atoms. ■

Theorem 21. *Let $C \equiv h(\bar{t}) : - \bar{N}, p_j(\bar{t}_1^j \cdot \bar{t}_0^j), p_{j+1}(\bar{t}_1^{j+1} \cdot \bar{t}_0^{j+1}), \bar{O}$ be a clause in a normalized program P that satisfies the conditions of Definition 20. Then, the programs P and $P \setminus \{C\} \cup AVF(C)$ are TF-equivalent.*

Proof. Let P' be the program that results of adding to P the following clause:

$$C21.1 : p'((\bar{x} \cdot \bar{w}) \cdot \bar{z}) : - p_j(\bar{x} \cdot \bar{t}_0^j), p_{j+1}(\bar{t}_1^{j+1} \cdot \bar{z})$$

where p' is a new predicate. By Definition 20, the predicate p_j is defined in P' by clauses of the following two forms (the asterisk * shows that they are schemes):

$$\begin{aligned} C21.2* : p_j(\bar{r}_1 \cdot \bar{r}_0) &: - \bar{L} \\ C21.3* : p_j(\bar{s}_1 \cdot \bar{z}) &: - \bar{M}, p_j(\bar{s}_1' \cdot \bar{z}) \end{aligned}$$

It is easy to see that $P \stackrel{TF}{\cong} P'$, then we will show that $P' \stackrel{TF}{\cong} P \setminus \{C\} \cup AVF(C)$. By unfolding the atom $p_j(\bar{x} \cdot \bar{t}_0^j)$ in the clause $C21.1$ using the clauses of the form $C21.2$ and $C21.3$, we get clauses of the following two forms:

$$\begin{aligned} C21.4* : p'((\bar{r}_1 \cdot \bar{w}\sigma) \cdot \bar{z}) &: - \bar{L}, p_{j+1}(\bar{t}_1^{j+1} \sigma \cdot \bar{z}) \\ C21.5* : p'((\bar{s}_1 \cdot \bar{w}) \cdot \bar{z}) &: - \bar{M}, p_j(\bar{s}_1' \cdot \bar{t}_0^j), p_{j+1}(\bar{t}_1^{j+1} \cdot \bar{z}) \end{aligned}$$

where $\sigma \equiv mgu(\bar{r}_0, \bar{t}_0^j)$. We now fold each clause of the form $C21.5$ using the clause $C21.1$ and we obtain clauses of the form:

$$C21.6* : p'((\bar{s}_1 \cdot \bar{w}) \cdot \bar{z}) : - \bar{M}, p'((\bar{s}_1' \cdot \bar{w}) \cdot \bar{z})$$

By folding the clause C using the clause $C21.1$ again we obtain:

$$C21.7 : h(\bar{t}) : - \bar{N}, p'((\bar{t}_1^j \cdot \bar{w}) \cdot \bar{t}_0^{j+1}), \bar{O}$$

$AVF(C)$ is composed of $C21.7$ and the sets of clauses $C21.4$ and $C21.6$. ■

Example 22. Following the previous Example 18, it only remains to eliminate the auxiliary variable y in the clause $E18.7$:

$$E18.7 : preorder_tr([(y_2, y_3), [x_1], c_4|S], z) : - append([x_1|y_3], y_2, y), \\ preorder_tr([y|S], z)$$

By Definition 19, we have that $MS(append([x_1|y_3], y_2, y))$ is (in, in, out) and $MS(preorder_tr([y|S], z))$ is (out, in) . Let $append/3$ be defined by the clauses:

$$\begin{aligned} E22.1 : append([], x, x) \\ E22.2 : append([x_1|x_2], x_3, [x_1|x_4]) : - append(x_2, x_3, x_4) \end{aligned}$$

Extending $MS(append([x_1|y_3], y_2, y))$ to the clauses in $MRDef_P(append)$, we obtain $MS(E22.1) \equiv ((in, in, out))$ and $MS(E22.2) \equiv ((in, in, out), (in, in, out))$. Since $MRDef_P(append)$ is not tail recursive³ w.r.t. $MS(append([x_1|y_3], y_2, y))$, we obtain the set of clauses $TailRDef_P(append)$ w.r.t. $MS(append([x_1|y_3], y_2, y))$:

³ It is in the Prolog sense, but not in our sense which is relative to a mode specification.

$E22.3 : \text{append}(x_1, x_2, z) : - \text{append_tr}([(x_1, x_2), c_9], z)$
 $E22.4 : \text{append_tr}([x], x)$
 $E22.5 : \text{append_tr}([([]], x), c_9|S], z) : - \text{append_tr}([x|S], z)$
 $E22.6 : \text{append_tr}([(x_1|x_2], x_3), c_9|S], z) : -$
 $\quad \text{append_tr}([(x_1|x_2], x_3), [], c_{10}, [x_1], c_{11}|S], z)$
 $E22.7 : \text{append_tr}([(x_1|x_2], x_3), [], c_{10}|S], z) : - \text{append_tr}([(x_2, x_3), c_9|S], z)$
 $E22.8 : \text{append_tr}([x_4, [x_1], c_{11}|S], z) : - \text{append_tr}([x_1|x_4]|S], z)$

and substitute $\text{append_tr}([(x_1|y_3], y_2), c_9], [y])$ for $\text{append}([x_1|y_3], y_2, y)$ in the clause $E18.7$ (see next subsection), whose mode specification remains $\text{MS}(E18.7) \equiv ((\text{in}, \text{out}), (\text{out}, \text{in}))$. Then, $\text{AVF}(E18.7)$ is defined by the following set of clauses:

$E22.9 : \text{append_tr}'([x], [S], z) : - \text{preorder_tr}([x|S], z)$
 $E22.10 : \text{append_tr}'([([]], x), c_9|S], [S'], z) : - \text{append_tr}'([x|S], [S'], z)$
 $E22.11 : \text{append_tr}'([(x_1|x_2], x_3), c_9|S], [S'], z) : -$
 $\quad \text{append_tr}'([(x_1|x_2], x_3), [], c_{10}, [x_1], c_{11}|S], [S'], z)$
 $E22.12 : \text{append_tr}'([(x_1|x_2], x_3), [], c_{10}|S], [S'], z) : -$
 $\quad \text{append_tr}'([(x_2, x_3), c_9|S], [S'], z)$
 $E22.13 : \text{append_tr}'([x_4, [x_1], c_{11}|S], [S'], z) : - \text{append_tr}'([x_1|x_4]|S], [S'], z)$

4.4 An Auxiliary Variable Elimination Algorithm

In this subsection, we give an algorithm that use the previous transformations in order to eliminate all the auxiliary variables from a definite program. By technical convenience, we suppose that normalization has been already applied to every clause with auxiliary variables.

Roughly speaking, for each clause C (let h be the predicate of its head atom) we select the leftmost pair of atoms $(p_j(\bar{t}^j), p_{j+1}(\bar{t}^{j+1}))$ that contain auxiliary variables and we assign to them a mode specification (see Definition 19). If $p_j \notin \text{MR}_P(h)$, then we transform $\text{Def}_P(p_j)$ into a tail recursive definition w.r.t. $\text{MS}(p_j(\bar{t}^j))$ (if necessary) and substitute $\text{AVF}(C)$ (see Definition 20) for C . Otherwise, we transform $\text{Def}_P(h)$ into a tail recursive definition w.r.t. $\text{MS}(p_j(\bar{t}^j))$.

In the former, the auxiliary variables contained in the leftmost pair of atoms are directly eliminated. In the latter, since it is not possible to directly eliminate them, we first transform $\text{Def}_P(h)$ into $\text{TailRDef}_P(h)$ w.r.t. $\text{MS}(p_j(\bar{t}^j))$, that makes $\text{MR}_P(h) \equiv \{h\}$. This transformation itself often eliminates many of the auxiliary variables (see Examples 14 and 18 where two auxiliary variables are eliminated). For the remaining ones in $\text{TailRDef}_P(h)$, any clause in this set holds the first *if* condition in Figure 1. Then, auxiliary variables can be eliminated from these clauses. Besides, when a clause $D \notin \text{TailRDef}_P(h)$ is selected, $\text{TailRDef}_P(h)$ can never be affected by the auxiliary variable elimination. Besides, none transformational step introduce new auxiliary variables. Therefore, our procedure eventually finishes when no auxiliary variable remains.

```

Pre:  $\{P \equiv P_0 \text{ is a normalized definite program}\}$ 
Repeat
  Select any clause  $C \equiv h(\bar{t}) : -\bar{N}, p_j(\bar{t}^j), p_{j+1}(\bar{t}^{j+1}), \bar{O}$  such that  $p_j(\bar{t}^j)$  and
     $p_{j+1}(\bar{t}^{j+1})$  is the leftmost pair of atoms with auxiliary variables
  Assign  $MS(p_j(\bar{t}^j))$  and  $MS(p_{j+1}(\bar{t}^{j+1}))$ 
  If  $p_j \notin MR_P(h)$  Then
    If  $Def_P(p_j)$  is not tail recursive w.r.t.  $MS(p_j(\bar{t}^j))$  Then
      Transform  $Def_P(p_j)$  into  $TailRDef_P(p_j)$  w.r.t.  $MS(p_j(\bar{t}^j))$  using  $p_j\text{-}tr$ 
      Substitute  $p_j\text{-}tr([\bar{t}_1^j, c_{p_j}] \cdot \bar{t}_0^j)$  for  $p_j(t_1^j \cdot t_0^j)$  in  $C$ 
    End If
    Substitute  $AVF(C)$  for the clause  $C$ 
  Else Transform  $MRDef_P(h)$  into  $TailRDef_P(h)$  w.r.t.  $MS(p_j(\bar{t}^j))$ 
  End If
Until no auxiliary variable remains in  $P$ 
Post:  $\{P \stackrel{TF}{\cong} P_0 \text{ is an auxiliary variable free program}\}$ 

```

Fig. 1. Auxiliary Variable Elimination Algorithm

5 Conclusions

The presented method can be automatically applied to every definite logic program and eliminates all its local variables. Each definite clause (with local variables) is substituted by a set of definite clauses with new auxiliary predicates. As a result, the negative version of the target program does not require universal quantification. Hence, much efficiency is gained in negative goal computation. However, positive goals should be computed with respect to the source program since, in general, the new predicates that are introduced by the transformation jeopardize the efficiency of the positive goal computation. The aim of the present work is to prove the existence of a general algorithmic method for local variable elimination in definite logic programs. Much work and further improvements should be made on implementation and experimentation in order to obtain more efficient target programs.

A method for eliminating local (there called, unnecessary) variables from definite logic programs was introduced in [13]. Their main aim was to eliminate the redundant computations that are made by means of local variables, hence the target program yields more efficient SLD-computations. This motivation is essentially different from ours. They present different strategies for guiding the application of unfold/fold transformations in order to achieve local variable elimination. The strategies are, like our method, syntactically based. However, the strategies only warrant the complete elimination of all local variables for a syntactic subclass of definite logic programs.

We plan to extend our results to normal logic programs. By now, we think that full generality could not be achieved in this case. However, we believe that the method can be easily adapted for a wide subclass of normal logic programs. Future work also includes the extension to constraint logic programming.

Acknowledgments. We are grateful to the referees for their helpful comments.

References

1. J. Álvarez, P. Lucio, F. Orejas, E. Pasarella, and E. Pino. Constructive negation by bottom-up computation of literal answers. In *ACM Symposium on Applied Computing (SAC2004)*, 2004.
2. M. Barbuti, P. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *Journal of Logic Programming*, 8:201–228, 1990.
3. P. Bruscoli, F. Levi, G. Levi, and M. C. Meo. Compilative constructive negation in constraint logic programs. In S. Tison, editor, *Proc. of the Trees in Algebra and Programming 19th Int. Coll.(CAAP '94)*, volume 787 of *LNCS*, pages 52–67. Springer-Verlag, 1994.
4. D. Chan. Constructive negation based on the completed database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. of the 5th Int. Conf. and Symp. on Logic Progr.*, pages 111–125. MIT Press, 1988.
5. D. Chan. An extension of constructive negation and its application in coroutining. In E. Lusk and R. Overbeek, editors, *Proc. of the NACLP'89*, pages 477–493. MIT Press, 1989.
6. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322, New York, 1978. Plenum Press.
7. S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229, 1988.
8. W. Drabent. What is failure? an approach to constructive negation. *Acta Informatica*, 32:27–59, 1995.
9. M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.
10. A. King, K. Shen, and F. Benoy. Lower-bound time-complexity analysis of logic programs. In J. Maluszynski, editor, *International Symposium on Logic Programming*, pages 261 – 276. MIT Press, November 1997.
11. K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.
12. M. J. Maher. Equivalences of logic programs. In J. Minker, editor, *Foundations of deductive databases and logic programming*, pages 627–658, Los Altos, CA, 1988. Morgan Kaufmann.
13. M. Proietti and A. Pettorossi. Unfolding - definition - folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.
14. J. Shepherdson. Language and equality theory in logic programming. Technical Report No. PM-91-02, University of Bristol, 1991.
15. P. J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, 1995.

Unfolding-based Improvements on Fuzzy Logic Programs [★]

P. Julián, G. Moreno, and J. Penabad

Departments of Computer Science and Applied Mathematics. U. Castilla-La Mancha.
{Pascual.Julian},{Gines.Moreno},{Jaime.Penabad}@uclm.es

Abstract. Unfolding is a semantics-preserving program transformation technique that consists in the expansion of subexpressions of a program using their own definitions. In this paper we define two unfolding-based transformation rules that extend the classical definition of the unfolding rule (for pure logic programs) to a fuzzy logic setting. We use a fuzzy variant of Prolog where each program clause can be interpreted under a different (fuzzy) logic. We adapt the concept of a *computation rule*, a mapping that selects the subexpression of a goal involved in a computation step, and we prove the independence of the computation rule. We also define a basic transformation system and we demonstrate its strong correctness, that is, original and transformed programs compute the same fuzzy computed answers. Finally, we prove that our transformation rules always produce an improvement in the efficiency of the residual program, by reducing the length of successful Fuzzy SLD-derivations.

Keywords: Fuzzy Logic Programming, Program Transformation.

1 Introduction

Logic Programming [12] has been widely used for problem solving and knowledge representation in the past. Nevertheless, traditional logic programming languages do not incorporate techniques or constructs in order to treat explicitly uncertainty and approximated reasoning. *Fuzzy Logic* provides a mathematical background for modeling uncertainty and/or vagueness. Fuzzy logic relies on the concept of fuzzy set, the theory of fuzzy connectives (t-norms, t-conorms, etc.) and the extension of two-values classical predicate logic to a logic where formulas can be evaluated in the range of the $[0, 1]$ real interval (see [23] or [13] for a comprehensive introduction of this subject). Fuzzy sets [22] are objects introduced to deal with the fuzziness or vagueness we find in the real world when we try to describe phenomena that have not sharply defined boundaries. Given a set U , an ordinary subset A of U can be defined in terms of its *characteristic function* $\chi_A(x)$ (that returns 1 if $x \in A$ or 0 otherwise) which neatly specifies whether or not an element x is in A . On the other hand, a *fuzzy subset* A of U is a function $A : U \rightarrow [0, 1]$. The function A is called the *membership function*, and the value $A(x)$ represents the *degree of membership* (it is not meant to convey

[★] This work is partially supported by CICYT under grant TIC 2001-2705-C03-03.

the likelihood that x has some particular attribute such as “young” [13]) of x in the fuzzy set A . Different functions A can be considered for a fuzzy concept and, in general, they will present a soft shape instead of the characteristic function’s crisp slope of an ordinary set.

Fuzzy Logic Programming is an interesting and still growing research area that agglutinates the efforts to introduce Fuzzy Logic into Logic Programming. During the last decades, several fuzzy logic programming systems have been developed, where the classical inference mechanism of SLD-Resolution is replaced with a fuzzy variant which is able to handle partial truth and to reason with uncertainty. Most of these systems implement the fuzzy resolution principle introduced by Lee in [10], such as the Prolog-Elf system [4], Fril Prolog system [2] and the F-Prolog language [11].

On the other hand, there is also no agreement about which fuzzy logic must be used when fuzzifying Prolog. Most systems use min-max logic (for modeling the conjunction and disjunction operations) but other systems just use Lukasiewicz logic [7]. Other approaches are parametric with respect to the interpretation of the fuzzy connectives, letting them unspecified to obtain a more general framework [21]. Recently, it has been appeared in [20] a theoretical model for fuzzy logic programming which deals with many values implications. Finally, in [19] we find an extremely flexible scheme where, apart from introducing negation and dealing with interval-valued fuzzy sets [8], each clause on a given program may be interpreted with a different logic. In this paper, we follow this last extension in a very natural way.

The fold/unfold transformation approach (also known as “rules+strategies” approach [15]) was first introduced in [3] to optimize functional programs and then used for logic programs [18] and integrated functional-logic programs [1]. This approach is based on the construction, by means of a *strategy*, of a sequence of equivalent programs —called *transformation sequence* and usually denoted by $\mathcal{P}_0, \dots, \mathcal{P}_n$ — where each program $\mathcal{P}_i$ is obtained from the preceding ones $\mathcal{P}_0, \dots, \mathcal{P}_{i-1}$ by using an *elementary* transformation rule. The essential rules are *folding* and *unfolding*, i.e., contraction and expansion of subexpressions of a program using the definitions of this program (or of a preceding one). Other rules which have been considered are, e.g., definition introduction/elimination, and algebraic replacement.

Focusing in unfolding, the objective of this program transformation operation is to replace a program rule by the set of rules obtained after applying a *symbolic computation* step (in all its possible forms) on the body of the selected rule [15]. Depending on the concrete paradigm taken into account (functional [3], logic [18] or integrated functional-logic [1]) the considered computation step will be performed using —some variant of— its associated operational mechanism (rewriting, resolution or narrowing, respectively).

In this paper we study the extension of two unfolding-based transformations to a fuzzy context. In particular, besides defining a fuzzy variant of the unfolding transformation of [18], we also introduce a new transformation, called *T-Norm replacement*, operating on the fuzzy component of an expression, which has some

similarities with the algebraic replacement of [3]. As we will see, the adaptation of the unfolding rule from classical LP to fuzzy LP can not be made in a naive way, if we really want to define it in a natural and correct way. In particular we need to take into account the “functional” dimension associated to the fuzzy component and, what it is more important, it is crucial to introduce an intermediate language (not required neither in LP, nor in FP) for coding the programs involved during the transformation process.

Apart from [5] and as far as we now, this is the first approach described in the literature for introducing fuzzy transformation rules. The present work largely improves [5] in the following points: i) **Language**: We use a richer extension of the fuzzy logic language described in [21] which directly affects all the subsequent definitions, results and proofs. ii) **Transformations**: Now we present a clearer transformation scheme by providing two unfolding-based transformation rules, each one focused in a language component: whereas the fuzzy unfolding rule concentrates into the logic part, the T-Norm replacement rule handles the fuzzy component. Moreover, the effects obtained by the (fourth variant of the) T-Norm replacement operation can not be achieved by the single transformation rule introduced in [5]. iii) **Properties**: Besides correctness results, we proof the reduction in length of successful Fuzzy SLD-derivations (when they are computed in the transformed program).

The outline of this paper is as follows. In the next section, we summarize an extension of the fuzzy Prolog dialect described in [21]. In the new extension, that we call **f-Prolog**, each program clause can be interpreted under a different logic. Section 3 presents the operational semantics of our language. We also adapt the concept of a *computation rule* and we prove a result which is the fuzzy counterpart of the independence of the computation rule theorem demonstrated in [12]. In Section 4 we define two unfolding-based transformation rules for a labeled mark variant of **f-Prolog** and we study its correctness properties. Finally, in Section 5, we give our conclusions.

2 Fuzzy Prolog Programs

Among the variety of fuzzy logic programming languages in the literature, the one described in [21] is specially appropriated to define the concept of unfolding of fuzzy logic programs. In this section we present an extension of this language, that we call **f-Prolog** (*fuzzy Prolog*) and allow us to give a more flexible interpretation of the logical connectives in the style introduced by [19].

Let $\mathcal{L}$ be a first order language containing variables, function symbols, predicate symbols, constants, quantifiers $\forall$ and $\exists$, and connectives $\neg$, seq , et_1 , and et_2 (the intended meaning is that seq is an implication —the left-arrow version is written as qes —, et_1 is a conjunction evaluating *modus ponens* with seq , and et_2 is a conjunction typically occurring in the body of clauses). Although et_1 and et_2 are binary connectives, we usually generalize them as functions with an arbitrary number of arguments. That is we write, for instance, $\text{et}_2(x_1, \dots, x_n)$ instead of $\text{et}_2(x_1, \text{et}_2(x_2, \dots, \text{et}_2(x_{n-1}, x_n) \dots))$.

A (definite) *clause* is a formula $\forall(A \text{ qes et}_2(B_1, \dots, B_n))$, and a (definite) *goal* is a formula $\forall(\text{qes et}_2(B_1, \dots, B_n))$, where A and each B_i are atomic formulas. In general, we call them (seq, et₂)-*formulas* or simply *formulas* if the kind of connectives used in their writing is not important or can be inferred by the context. Also, we write $A \leftarrow B_1, \dots, B_n$ as syntactic sugar of $\forall(A \text{ qes et}_2(B_1, \dots, B_n))$, etc. As usually, A is said to be the head of the clause and $(B_1, \dots, B_n)$ the body. Clauses with an empty body are called *facts*, whereas clauses with a head and a body are called *rules*. A sort of degenerate clause is the *empty clause*, denoted by ‘ $\square$ ’, representing a contradictory formula.

The meaning function for the connective seq, denoted by $\llbracket \text{seq} \rrbracket$, is the residual implication defined as: $\llbracket \text{seq} \rrbracket(x, y) = \sup \{z \in [0, 1] : \llbracket \text{et}_1 \rrbracket(x, y) \leq z\}$. Therefore $\llbracket \text{seq} \rrbracket$ is a R-implication, since it is only defined beginning from a t-norm, but it is not necessarily a S-implication, neither a QM-implication [17]. Nevertheless, the meaning function for the connective seq we use in this paper is the one proposed by [21], where $\llbracket \text{seq} \rrbracket$ is linked to $\llbracket \text{et}_1 \rrbracket$ in such a way that modus ponens is a sound rule, what is essential in order to proof the correctness of the Fuzzy SLD-Resolution Calculus.

Note also that, in the discussion above, the meaning functions for connectives et₁ and et₂ were let unspecified as arbitrary t-norms ¹ $\llbracket \text{et}_i \rrbracket : [0, 1]^2 \rightarrow [0, 1]$, properly extended as many valued functions, and they are intended to be fixed in the range of the whole program. However, as it has been told in [19], it may be useful from a practical point of view to associate a concrete interpretation for each operator et₁ or et₂ in the context of a program clause instead of a fixed interpretation for the whole program environment. In particular, note that, the conjunction is defined by Zadeh via the *min* operator but it is widely accepted that no single operator for conjunction can model the wide variety of expressions that is necessary to formalize.

Example 1. Given a clause $p(x) \leftarrow q(x), r(x)$ if we interpret $p(x)$ as “ x is sportsman”, $q(x)$ as “ x is young” and $r(x)$ as “ x is healthy”, the predicates of the body are of positive influence (that is, they mutually reinforce themselves). In this case the conjunction et₂ of body is usually understood as *min*. Similarly, if $q(x)$, and $r(x)$ are interpreted as independent or noninteractive predicates. However, if $q(x)$ is interpreted as “ x is low” and $r(x)$ as “ x is tall”, that is, the predicates $q(x)$ and $r(x)$ has a contradictory meaning, the conjunction et₂ of body is advisable to be interpreted as *max*.

We can redefine the concept of fuzzy theory that appears in [21] in order to cope with this problem.

Definition 1. A fuzzy theory is a partial mapping T applying a triple $\langle \alpha, le_1, le_2 \rangle$ in $(0, 1] \times Sem \times Sem$, to each formula, where $(0, 1]$ is the domain of truth degrees and Sem is a set of semantics labels indicating the associated meaning for et₁ and et₂ respectively.

¹ They are commutative, associative, and monotone in both arguments and $\llbracket \text{et}_i \rrbracket(x, 1) = x$ (hence, they subsume classical conjunction $\{0, 1\}^2 \rightarrow \{0, 1\}$) [16]. Note also that, in general, meaning functions for et_i connectives are not distributive.

The real α is a *truth degree* expressing the confidence which the user of the system has in the truth of the clause $\mathcal{C}$. A truth degree $\alpha = 1$ means that the user believes the clause $\mathcal{C}$ is true; on the other hand, a truth degree less than 1 represents the degree of uncertainty or lost of confidence on the truth of a belief; a truth degree near 0 expresses the lack of confidence on the truth of a belief. We can use labels indicating the meaning assigned to the et_1 or et_2 operator in the clause $\mathcal{C}$. For instance, a label $le_i = \text{lukasiewicz}$ interprets an operator et_i as a Lukasiewicz t-norm, that is, $\llbracket \text{et}_i \rrbracket(x, y) = \max(0, x + y - 1)$. Other possible labels would be: **min**, if $\llbracket \text{et}_i \rrbracket(x, y) = \min(x, y)$; **product**, if $\llbracket \text{et}_i \rrbracket(x, y) = x \cdot y$; etc. A **void** value in *Sem* is employed to express that no meaning for et_1 or et_2 is selected. Note that, operationally, since **void** must be linked to a t-norm, $\text{void}(1, x) = \text{void}(x, 1) = x$.

Definition 2. A definite f-Prolog program, $\mathcal{P}$, is a fuzzy theory such that:

1. $\text{Dom}(\mathcal{P})$ is a set of (seq, et_2)-definite program clauses or facts,
2. for $\mathcal{C}_1, \mathcal{C}_2 \in \text{Dom}(\mathcal{P})$, we say $\mathcal{C}_1 \approx \mathcal{C}_2$ if and only if $\mathcal{C}_1$ is a variant of $\mathcal{C}_2$ and $\mathcal{P}(\mathcal{C}_1) = \mathcal{P}(\mathcal{C}_2)$, and
3. $\text{Dom}(\mathcal{P})/\approx$ is finite.

Roughly speaking, a program can be seen as a set of pairs $(\mathcal{C}; \langle r, l1, l2 \rangle)$, where $\mathcal{C}$ is a clause, r is the truth degree of the clause $\mathcal{C}$, and $l1, l2$ are the semantics labels associated with the operators et_1 and et_2 for this clause. However we prefer to write $\mathcal{C}$ with $\langle r, l1, l2 \rangle$, or more descriptively: $\mathcal{C}$ with $\alpha = r$ and $le_1 = l1$ and $le_2 = l2$. If clause $\mathcal{C}$ is a fact, le_1 and le_2 are **void** and we simply write: $\mathcal{C}$ with $\alpha = r$; omitting the values for le_1 and le_2 . Similarly, a goal $\mathcal{G}$ has only associated a semantic label for et_2 , but no initial truth degree or semantic label for et_1 , and we write: $\mathcal{G}$ with $le_2 = l2$.

3 Operational Semantics and Labeled Fuzzy Prolog

Given a goal $\mathcal{G}$ its truth degree, α , is obtained by evaluating a sequence of Fuzzy SLD-Resolution steps leading to an empty clause. In the sequel we formalize the concepts of Fuzzy SLD-Resolution, Fuzzy SLD-Derivation and Fuzzy computer answer, with some variations with regard to the definitions that appear in [21].

Let $\mathcal{P}$ be a program and $\mathcal{G}$ a goal. Since $\text{Dom}(\mathcal{P})/\approx$ is a classical definite program, the classical SLD-resolution should still work. Therefore, the main operational problem is to define the machinery for evaluating truth degrees. The truth degree of a expression is a semantic notion that must be evaluated using meaning functions. Considering a program rule $\mathcal{C} \equiv A \leftarrow B_1, \dots, B_m$, with $\alpha = q$, and a goal $\mathcal{G} \equiv \leftarrow A'$, where A' unifies with the head A of $\mathcal{C}$, it is possible a SLD-resolution step leading to the resolvent $\mathcal{G}' \equiv \leftarrow (B_1, \dots, B_m)$. If we want to evaluate the truth degree of $\mathcal{G}$, we have to compute the truth degrees $r_1, \dots, r_n$ of all subgoals $B_1, \dots, B_m$ before the truth degree q of the rule can be applied to obtain $\llbracket \text{et}_1 \rrbracket(q, \llbracket \text{et}_2 \rrbracket(r_1, \dots, r_n))$, the truth degree of the goal $\mathcal{G}$. We need a mechanism in order to remember that a program rule was

applied in former steps, since it is necessary to distinguish when to apply $\llbracket \text{et}_1 \rrbracket$ or $\llbracket \text{et}_2 \rrbracket$. In [21] a context grammar was introduced to solve this problem. This grammar contains left and right marks ($\boxed{L_q}$ and $\boxed{R_q}$) labeled by a real value, to remember the exact point where a rule with $\alpha = q$ was applied. Hence the previous resolution step can be annotated as: $\boxed{L_q} B_1, \dots, B_m \boxed{R_q}$. For our case, in order to manage the resolution process properly while extending the expressive power of our language, it is also necessary to expand the label mechanism to distinguish what is the meaning operator $\llbracket \text{et}_1 \rrbracket$ or $\llbracket \text{et}_2 \rrbracket$ that must be applied. Hence, we introduce the marks $\boxed{L_{\langle q, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle}}$ and $\boxed{R_{\langle q, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle}}$. We call **lf-Prolog** the extended language obtained by adding labeled marks and real numbers to the **f-Prolog** alphabet. A **lf-expression** is an atom, a sequence of real numbers, or a real number enclosed between labeled marks. The following definition makes use of **lf-Prolog** in the formalization of Fuzzy SLD-Resolution (we write $\bar{o}$ for the -possibly empty- sequence of syntactic objects $o_1, \dots, o_n$).

Definition 3. Let $\mathcal{G} \equiv \leftarrow Q$ with $le_2 = \llbracket \text{et}_2 \rrbracket$ be a **lf-Prolog** goal and let ϑ be a substitution, a **lf-Prolog** state is a pair $\langle Q; \vartheta \rangle$. Let $\mathcal{E}$ be the set of **lf-Prolog** states. Given a **f-Prolog** program $\mathcal{P}$, we define Fuzzy SLD-Resolution as a state transition system, whose transition relation $\rightarrow_{FR} \subseteq (\mathcal{E} \times \mathcal{E})$ is the smallest relation satisfying the following rules:

Rule 1. (Clause Resolution Rule)

$\langle \bar{X}, A_m, \bar{Y}; \vartheta \rangle \rightarrow_{FR} \langle \bar{X}, \boxed{L_{\langle q, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle}} B_1, \dots, B_l \boxed{R_{\langle q, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle}}, \bar{Y} \rangle \theta; \vartheta \theta$ if

1. A_m is the selected atom,
2. θ is an mgu of A_m and A ,
3. $C \equiv (A \leftarrow B_1, \dots, B_l$ with $\langle q, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle \rangle \in \mathcal{P}$ and $l \geq 1$.

Rule 2. (Fact Resolution Rule)

$\langle \bar{X}, A_m, \bar{Y}; \vartheta \rangle \rightarrow_{FR} \langle \bar{X}, r, \bar{Y}; \vartheta \theta \rangle$ if

1. A_m is the selected atom,
2. θ is an mgu of A_m and A , and
3. $C \equiv (A \leftarrow$ with $r) \in \mathcal{P}$.

Rule 3. ($\llbracket \text{et}_1 \rrbracket$ Resolution Rule)²

$\langle \bar{X}, \boxed{L_{\langle q, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle}} r \boxed{R_{\langle q, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle}}, \bar{Y}; \vartheta \rangle \rightarrow_{FR} \langle \bar{X}, \llbracket \text{et}_1 \rrbracket(q, r), \bar{Y}; \vartheta \rangle$ if

1. r is a real number.

Rule 4. ($\llbracket \text{et}_2 \rrbracket$ Resolution Rule)

$\langle \bar{X}, \boxed{L_{\langle q, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle}} \bar{r} \boxed{R_{\langle q, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle}}, \bar{Y}; \vartheta \rangle \rightarrow_{FR}$
 $\langle \bar{X}, \boxed{L_{\langle q, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle}} \llbracket \text{et}_2 \rrbracket(\bar{r}) \boxed{R_{\langle q, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle}}, \bar{Y}; \vartheta \rangle$ if

1. $\bar{r} \equiv r_1, \dots, r_n$, where $n > 1$, are real numbers.

² In [21], the $\llbracket \text{et}_1 \rrbracket$ resolution rule is expressed as a combination of our third and fourth rules but our reformulation is completely equivalent to the original when the labels le_1 and le_2 are fixed in the whole program.

All familiar logic programming concepts can be extended for the fuzzy case, assuming also that clauses involved in fuzzy SLD-computation steps are renamed before being used. In the following, symbols $\rightarrow_{FR1}$, $\rightarrow_{FR2}$, $\rightarrow_{FR3}$ and $\rightarrow_{FR4}$ may be used for explicitly referring to the application of each one of the four fuzzy resolution rules. When needed, the exact lf-expression and/or clause used in the corresponding step, will be also annotated as a super-index of the $\rightarrow_{FR}$ symbol. In order to extend the notion of computer answer in our fuzzy setting, in the following definition we use id to refer to the empty substitution, $\mathcal{V}ar(s)$ denotes the set of distinct variables occurring in the syntactic object s , and $\theta[\mathcal{V}ar(s)]$ corresponds to the substitution obtained from θ by restricting its domain, $Dom(\theta)$, to $\mathcal{V}ar(s)$.

Definition 4. Let $\mathcal{P}$ be a f-Prolog program and $\mathcal{G} \equiv \leftarrow \mathcal{Q}$ with $le_2 = \llbracket et_2 \rrbracket$ be a lf-Prolog goal. A pair $\langle r; \theta \rangle$ consisting of a real number r and a substitution θ is a fuzzy computed answer (f.c.a.) if there is a sequence $\mathcal{E}_0, \dots, \mathcal{E}_n$ (called f-derivation) such that:

1. $\mathcal{E}_0 = \langle \boxed{L_{\langle 1, void, \llbracket et_2 \rrbracket \rangle}}, \mathcal{Q}, \boxed{R_{\langle 1, void, \llbracket et_2 \rrbracket \rangle}}; id \rangle$,
2. for each $0 \leq i < n$, $\mathcal{G}_i \rightarrow_{FR} \mathcal{G}_{i+1}$ is a fuzzy SLD-resolution step,
3. $\mathcal{E}_n = \langle r; \theta' \rangle$ and $\theta = \theta'[\mathcal{V}ar(\mathcal{Q})]$.

We illustrate the last definition by means of the following example.

Example 2. Let $\mathcal{P}$ be the f-Prolog program,

$\mathcal{C}_1 :$	$p(X) \leftarrow q(X, Y), r(Y)$	with $\langle 0.8, \text{prod}, \text{min} \rangle$
$\mathcal{C}_2 :$	$q(a, Y) \leftarrow s(Y)$	with $\langle 0.7, \text{prod}, \text{void} \rangle$
$\mathcal{C}_3 :$	$q(Y, a) \leftarrow r(Y)$	with $\langle 0.8, \text{luka}, \text{void} \rangle$
$\mathcal{C}_4 :$	$r(Y) \leftarrow$	with 0.7
$\mathcal{C}_5 :$	$s(b) \leftarrow$	with 0.9

The following is a successful f-derivation for the program $\mathcal{P}$ and the goal “ $\leftarrow p(X), r(a)$ with min” with f.c.a. $\langle 0.504; \{X/a\} \rangle$:

$$\begin{array}{ll}
\langle \boxed{L_{\psi_0}} p(X), r(a) \boxed{R_{\psi_0}}; \sigma_0 \rangle & \rightarrow_{FR1}^{C_1} \\
\langle \boxed{L_{\psi_0}} \boxed{L_{\psi_1}} q(X_1, Y_1), r(Y_1) \boxed{R_{\psi_1}} r(a) \boxed{R_{\psi_0}}; \sigma_1 \rangle & \rightarrow_{FR1}^{C_2} \\
\langle \boxed{L_{\psi_0}} \boxed{L_{\psi_1}} \boxed{L_{\psi_2}} s(Y_2) \boxed{R_{\psi_2}} r(Y_2) \boxed{R_{\psi_1}} r(a) \boxed{R_{\psi_0}}; \sigma_2 \rangle & \rightarrow_{FR2}^{C_5} \\
\langle \boxed{L_{\psi_0}} \boxed{L_{\psi_1}} \boxed{L_{\psi_2}} 0.9 \boxed{R_{\psi_2}} r(b) \boxed{R_{\psi_1}} r(a) \boxed{R_{\psi_0}}; \sigma_3 \rangle & \rightarrow_{FR2}^{C_4} \\
\langle \boxed{L_{\psi_0}} \boxed{L_{\psi_1}} \boxed{L_{\psi_2}} 0.9 \boxed{R_{\psi_2}} 0.7 \boxed{R_{\psi_1}} r(a) \boxed{R_{\psi_0}}; \sigma_4 \rangle & \rightarrow_{FR2}^{C_4} \\
\langle \boxed{L_{\psi_0}} \boxed{L_{\psi_1}} \boxed{L_{\psi_2}} 0.9 \boxed{R_{\psi_2}} 0.7 \boxed{R_{\psi_1}} 0.7 \boxed{R_{\psi_0}}; \sigma_5 \rangle & \rightarrow_{FR3} \\
& // \text{since product}(0.9, 0.7) = 0.63 \\
\langle \boxed{L_{\psi_0}} \boxed{L_{\psi_1}} 0.63, 0.7 \boxed{R_{\psi_1}} 0.7 \boxed{R_{\psi_0}}; \sigma_5 \rangle & \rightarrow_{FR4} \\
& // \text{since min}(0.63, 0.7) = 0.63
\end{array}$$

$$\begin{array}{ll}
\langle \boxed{L_{\Psi_0}} \boxed{L_{\Psi_1}} 0.63 \boxed{R_{\Psi_1}} 0.7 \boxed{R_{\Psi_0}} ; \sigma_5 \rangle & \rightarrow_{FR3} \\
& // \text{since } \text{product}(0.63, 0.8) = 0.504 \\
\langle \boxed{L_{\Psi_0}} 0.504, 0.7 \boxed{R_{\Psi_0}} ; \sigma_5 \rangle & \rightarrow_{FR4} \\
& // \text{since } \min(0.504, 0.7) = 0.504 \\
\langle \boxed{L_{\Psi_0}} 0.504 \boxed{R_{\Psi_0}} ; \sigma_5 \rangle & \rightarrow_{FR3} \\
& // \text{ since } \text{void}(1, 0.504) = 0.504 \\
\langle 0.504 ; \sigma_5 \rangle &
\end{array}$$

where $\Psi_0 \equiv \langle 1, \text{void}, \min \rangle$, $\Psi_1 \equiv \langle 0.8, \text{prod}, \min \rangle$ and $\Psi_2 \equiv \langle 0.7, \text{prod}, \max \rangle$, are the triples associated with the original goal and clauses $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively. Also the substitutions $\sigma_0 = id$, $\sigma_1 = \{X/X_1\}$, $\sigma_2 = \{X/a, X_1/a, Y_1/Y_2\}$, $\sigma_3 = \{X/a, X_1/a, Y_1/b, Y_2/b\}$, $\sigma_4 = \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b\}$ and $\sigma_5 = \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b, Y_4/a\}$.

In [21], the authors established the correctness results for the Fuzzy SLD-Resolution operational mechanism (following a technique similar as the one proposed by Lloyd, in [12], for classical logic programming), but extending all results with the treatment of truth degrees. These correctness results can be easily adapted to our case.

As for the classical SLD-Resolution calculus, we assume the existence of a fixed selection function, also called *fuzzy computation rule*, deciding, for a given goal, which is the selected lf-expression to be exploited in the next fuzzy SLD-Resolution step. For instance, when building the f-derivation shown in Example 2, we have used a computation rule similar to the left to right selection rule of Prolog but delaying the application of the $\llbracket \text{et}_1 \rrbracket$ and $\llbracket \text{et}_2 \rrbracket$ resolution rules until all atoms have been resolved. Given a fuzzy computation rule $\mathcal{R}$, we say that a fuzzy SLD-derivation is *via* $\mathcal{R}$ if the selected lf-expression in every step is obtained by the application of the mapping $\mathcal{R}$ to the corresponding goal in that step. The following result extends to our fuzzy setting the independence of the computation rule proved in [12] for the pure logic programming case.

Theorem 1 (Independence of the Fuzzy Computation Rule).

Let $\mathcal{P}$ be a lf-Prolog program, and $\mathcal{G} \equiv \leftarrow \mathcal{Q}$ with $le_2 = \llbracket \text{et}_2 \rrbracket$ be a lf-Prolog goal. Then, for any pair of different fuzzy computation rules $\mathcal{R}$ and $\mathcal{R}'$, we have that:

$$\langle \boxed{L_{\Psi}} \mathcal{Q} \boxed{R_{\Psi}} ; id \rangle \rightarrow_{FR\mathcal{R}}^n \langle r; \theta \rangle \quad \text{iff} \quad \langle \boxed{L_{\Psi}} \mathcal{Q} \boxed{R_{\Psi}} ; id \rangle \rightarrow_{FR\mathcal{R}'}^n \langle r; \theta \rangle$$

where $\Psi \equiv \langle 1, \text{void}, \llbracket \text{et}_2 \rrbracket \rangle$ and n is the (same) number of fuzzy SLD-resolution steps in both derivations.

4 Unfolding-based Transformations for Fuzzy Programs

In essence, the unfolding transformation traditionally considered in pure logic programming consists in the replacement of a program clause C by the set of clauses obtained after applying a symbolic computation step in all its possible forms on the body of C [15]. In [5] we gave a first approach to the fuzzy extension

of unfolding, by considering the complete set of fuzzy SLD-resolution rules in Definition 3 (when performing symbolic computation steps) in order to generate all alternative clauses. However, a deeper look at Definition 3, reveals us that only rules 1 and 2 reproduce the essence of classical logic programming by exploiting atoms and generating unifiers and, in this sense, they are more appropriate to be used during the unfolding process to simulate the original definition. On the other hand, rules 3 and 4 neither reduce atoms nor produce unifiers, but simply perform numerical manipulations to produce truth degrees (what, in some way, reflects the fuzzy component of this enriched context). Therefore, rules 3 and 4 should be more appropriately used for defining other kind of transformations (as we will see in Definition 6).

In this paper we adopt this new point of view and in the next two sections, we define a set of program transformations based on (fuzzy variants of) the classical unfolding operation for pure logic programs defined in [18]. We also prove their strong correctness, i.e., they are sound and complete w.r.t. the semantics of fuzzy computed answers obtained by fuzzy SLD-resolution.

4.1 The Fuzzy Unfolding Transformation Rule

As we have seen in the previous sections, the differences between f-Prolog and lf-Prolog programs appear only at the syntactic level: whereas the body B of a (non unit) f-Prolog program clause (which in essence, is no more than a simple goal, that is, an atom or a conjunction of atoms) respects the grammar $B \rightarrow B, \dots, B \mid atom$, we need to enrich this set of grammar rules with $B \rightarrow \boxed{L_\psi} B \boxed{R_\psi} \mid number$, if we really want to cope with the possibility of including marks and real numbers in the body of lf-Prolog clauses (which intuitively have the same structure of any initial, intermediate or final goal appearing in fuzzy SLD-derivations). This implies that any f-Prolog program is also a lf-Prolog program, although the contrary is not always true (i.e., the set of f-Prolog programs is a proper subclass of the set of lf-Prolog programs). Apart from this simple fact (which, on the other hand, is mandatory to define the fuzzy SLD-resolution principle) both languages share the same operational semantics.

On the other hand, note that the application of rules 1 and 2 always generates clauses whose bodies include marks or numbers which implies that an unfolding transformation based on these rules is able to preserve the syntactic structure of lf-Prolog programs but, even in the case that original programs be also f-Prolog programs, the transformed ones will never belong to this subclass: the marks or real numbers incorporated into the transformed clauses by unfolding steps force the lost of the original f-Prolog syntax. In order to avoid this inconvenience, our transformation rules focus on the general framework of lf-Prolog programs instead of the more restricted subclass of f-Prolog programs. Note also that this fact does not suppose a problem in practice: the most important purpose of transformations rules, apart from preserving the program semantics, is to optimize code, independently of the object language. Classical fold/unfold based transformation systems optimize programs by returning code which uses the

same source language, but unfolding has also played important roles in the design of compilers (see [6]) which generate an object code written in a target language. In this sense, our transformation system can be seen as a mixed technique that optimizes f-Prolog programs and compiles it into lf-Prolog programs, with the advantage in our case that both programs are executable with exactly the same operational principle.

In the following, we consider a fixed transformation sequence $(\mathcal{P}_0, \dots, \mathcal{P}_k)$, $k \geq 0$, where we only require that the initial program $\mathcal{P}_0$ be an f-Prolog program since the remaining ones will necessarily adopt the lf-Prolog syntax.

Definition 5 (Fuzzy Unfolding).

Let $(\mathcal{P}_0, \dots, \mathcal{P}_k)$ be a transformation sequence of lf-Prolog programs starting from an f-Prolog program $\mathcal{P}_0$. Assume that $\mathcal{C} \equiv (H \leftarrow \overline{B} \text{ with } \Psi) \in \mathcal{P}_k$ is a (non unit) lf-Prolog program clause, such that $\Psi \equiv \langle p, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle$ and A is an atom included in the body $\overline{B}$. Then, the next program $\mathcal{P}_{k+1}$ in the transformation sequence can be obtained by fuzzy unfolding of clause $\mathcal{C}$ at atom A in program $\mathcal{P}_k$ as follows: $\mathcal{P}_{k+1} = (\mathcal{P}_k - \{\mathcal{C}\}) \cup \{H\sigma \leftarrow \overline{B}' \text{ with } \Psi \mid \langle \boxed{L_\Psi} \boxed{\overline{B}} \boxed{R_\Psi}; id \rangle \rightarrow_{FR^A} \langle \boxed{L_\Psi} \boxed{\overline{B}'} \boxed{R_\Psi}; \sigma \rangle\}$

There are some remarks to do regarding our definition. Similarly to the classical SLD-resolution based unfolding rule presented in [18], the substitutions computed by resolution steps during unfolding are incorporated to the transformed rules in a natural way, i.e., by applying them to the head of the clause. On the other hand, the propagation of truth degrees during the fuzzy unfolding process is done at two different levels: by directly assigning the complete tuple Ψ (containing the truth degree p and labels for $\llbracket \text{et}_1 \rrbracket$ and $\llbracket \text{et}_2 \rrbracket$) of the original clause to the transformed one, and by introducing marks and/or real numbers in its body. The following example illustrates these facts.

Example 3. Consider again the set of clauses of Example 2 as the initial program, $\mathcal{P}_0$, of a transformation sequence. It is easy to see that the unfolding of clause $\mathcal{C}_2$ w.r.t. $\mathcal{P}_0$ (exploiting the second rule of Definition 3) generates the new program $\mathcal{P}_1 = (\mathcal{P}_0 - \{\mathcal{C}_2\}) \cup \{\mathcal{C}_6\}$, where $\mathcal{C}_6$ is the new unfolded rule “ $q(a, b) \leftarrow 0.9$ with $\langle 0.7, \text{product}, \text{max} \rangle$ ”.

On the other hand, if we want to unfold now clause $\mathcal{C}_1$ in program $\mathcal{P}_1$, we must firstly generate the following one-step Fuzzy SLD-derivations³:

$$\begin{aligned} \langle \boxed{L_{\Psi_1}} \underline{q(X, Y)}, r(Y) \boxed{R_{\Psi_1}}; id \rangle &\rightarrow_{FR^1}^{C_6} \langle \boxed{L_{\Psi_1}} \boxed{L_{\Psi_6}} 0.9 \boxed{R_{\Psi_6}}, r(b) \boxed{R_{\Psi_1}}; \sigma_1 \rangle \\ \langle \boxed{L_{\Psi_1}} \underline{q(X, Y)}, r(Y) \boxed{R_{\Psi_1}}; id \rangle &\rightarrow_{FR^1}^{C_3} \langle \boxed{L_{\Psi_1}} \boxed{L_{\Psi_3}} r(Y_1) \boxed{R_{\Psi_3}}, r(a) \boxed{R_{\Psi_1}}; \sigma_2 \rangle \end{aligned}$$

where $\Psi_1 \equiv \langle 0.8, \text{prod}, \text{min} \rangle$, $\Psi_6 \equiv \langle 0.7, \text{prod}, \text{max} \rangle$ and $\Psi_3 \equiv \langle 0.8, \text{luka}, \text{min} \rangle$, are the triples corresponding to clauses $\mathcal{C}_1$, $\mathcal{C}_6$ and $\mathcal{C}_3$ respectively; also the unifiers $\sigma_1 = \{X/a, Y/b\}$ and $\sigma_2 = \{X/Y_1, Y/a\}$.

So, the unfolded program $\mathcal{P}_2 = (\mathcal{P}_1 - \{\mathcal{C}_1\}) \cup \mathcal{U}$ where $\mathcal{U}$ contains the new clauses:

$$\begin{aligned} \mathcal{C}_7 : p(a) &\leftarrow \boxed{L_{\langle 0.7, \text{prod}, \text{max} \rangle}} 0.9 \boxed{R_{\langle 0.7, \text{prod}, \text{max} \rangle}}, r(b) \quad \text{with } \langle 0.8, \text{prod}, \text{min} \rangle \\ \mathcal{C}_8 : p(Y_1) &\leftarrow \boxed{L_{\langle 0.8, \text{luka}, \text{min} \rangle}} r(Y_1) \boxed{R_{\langle 0.8, \text{luka}, \text{min} \rangle}}, r(a) \quad \text{with } \langle 0.8, \text{prod}, \text{min} \rangle \end{aligned}$$

³ Both steps are performed with Rule 1 of Definition 3 (the first one uses clause $\mathcal{C}_6$ and the second one clause $\mathcal{C}_3$) and exploit the same selected -underlined- atom $q(X, Y)$.

Finally, by performing a new resolution step with the second rule of Definition 3 on atom $r(b)$ in the body of clause $\mathcal{C}_7$, we obtain the new unfolded program $\mathcal{P}_3 = \{\mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5, \mathcal{C}_6, \mathcal{C}_8, \mathcal{C}_9\}$ (note that clauses $\mathcal{C}_1, \mathcal{C}_2$ and $\mathcal{C}_7$ have been removed after being unfolded) where $\mathcal{C}_9 \equiv p(a) \leftarrow \boxed{L_{\langle 0.7, \text{prod}, \text{max} \rangle}} 0.9 \boxed{R_{\langle 0.7, \text{prod}, \text{max} \rangle}}, 0.7$ with $\langle 0.8, \text{prod}, \text{min} \rangle$.

It is important to note now that the application of this last rule to the goal “ $\neg p(X), r(a)$ with **min**” simulates the effects of the first four resolution steps shown in the derivation of Example 2, which evidences the improvements achieved by unfolding on transformed programs.

4.2 The T-Norm Replacement Transformation Rule

Although we have seen in the previous section that the actions performed by unfolding on the body of transformed clauses drastically rebound in the computation/propagation of truth degrees when solving goals against transformed programs, the ‘compiled-in’ information collected on the body of unfolded rules admits significant numerical manipulations to eliminate or, at least to simplify, marks and real numbers. The following transformation performs this task in a very similar way to rules 3 and 4 of Definition 3.

Definition 6 (T-Norm Replacement).

Let $(\mathcal{P}_0, \dots, \mathcal{P}_k)$ be a transformation sequence of lf-Prolog programs starting from an f-Prolog program $\mathcal{P}_0$. Assume that $\mathcal{C} \equiv (H \leftarrow \overline{B} \text{ with } \Psi) \in \mathcal{P}_k$ is a (non unit) lf-Prolog program clause, such that $\Psi \equiv \langle p, \llbracket \text{et}_1 \rrbracket, \llbracket \text{et}_2 \rrbracket \rangle$. Then, the next program $\mathcal{P}_{k+1}$ in the transformation sequence can be obtained by T-Norm Replacement of clause $\mathcal{C}$ in program $\mathcal{P}_k$ as follows: $\mathcal{P}_{k+1} = (\mathcal{P}_k - \{\mathcal{C}\}) \cup \{H \leftarrow \overline{B}' \text{ with } \Psi'\}$, such that:

1. if $\overline{B} \equiv (\overline{X}, \boxed{L_{\langle p', \llbracket \text{et}_1 \rrbracket', \llbracket \text{et}_2 \rrbracket' \rangle}} r \boxed{R_{\langle p', \llbracket \text{et}_1 \rrbracket', \llbracket \text{et}_2 \rrbracket' \rangle}}, \overline{Y})$, where r is a real number, then $\overline{B}' \equiv (\overline{X}, \llbracket \text{et}_1 \rrbracket'(p', r), \overline{Y})$ and $\Psi' \equiv \Psi$.
2. if $\overline{B} \equiv (\overline{X}, \boxed{L_{\langle p', \llbracket \text{et}_1 \rrbracket', \llbracket \text{et}_2 \rrbracket' \rangle}} \bar{r} \boxed{R_{\langle p', \llbracket \text{et}_1 \rrbracket', \llbracket \text{et}_2 \rrbracket' \rangle}}, \overline{Y})$, where $\bar{r} \equiv r_1, \dots, r_n$ ($n > 1$) are real numbers, $\overline{B}' \equiv (\overline{X}, \boxed{L_{\langle p', \llbracket \text{et}_1 \rrbracket', \llbracket \text{et}_2 \rrbracket' \rangle}} \llbracket \text{et}_2 \rrbracket'(\bar{r}) \boxed{R_{\langle p', \llbracket \text{et}_1 \rrbracket', \llbracket \text{et}_2 \rrbracket' \rangle}}, \overline{Y})$ and $\Psi' \equiv \Psi$.
3. if $\overline{B} \equiv r$, where r is a real number, then $\overline{B}'$ is empty and $\Psi' \equiv \llbracket \text{et}_1 \rrbracket(p, r)$.
4. if $\overline{B} \equiv \bar{r}$, where $\bar{r} \equiv r_1, \dots, r_n$ ($n > 1$) are real numbers, then $\overline{B}' \equiv \llbracket \text{et}_2 \rrbracket(\bar{r})$ and $\Psi' \equiv \Psi$.

The previous definition remembers the so called “algebraic replacement” traditionally used in transformation systems for pure functional programs [3, 15], where a functional expression, in a program rule, is replaced by other equivalent expression w.r.t. a given algebraic property. For instance, the pure functional rule $\mathbf{f}(\mathbf{X}, \mathbf{Y}) \rightarrow \mathbf{X} + \mathbf{Y}$ can be transformed by algebraic replacement into $\mathbf{f}(\mathbf{X}, \mathbf{Y}) \rightarrow \mathbf{Y} + \mathbf{X}$, thanks to the commutativity of the sum. Observe also that similarly to many other pure functional transformation rules (including algebraic replacement and pure functional unfolding) our T-Norm replacement replaces an original clause by a single new one, which contrasts with the unfolding rule defined for pure logic and/or fuzzy logic programs. All these facts are not surprising, since the

fuzzy component of our logic language has a functional taste, in the sense that the numerical manipulations performed by rules 3 and 4 of Definition 3 recall functional evaluations. This property is also inherited by Definition 6 where we define some particular instances of algebraic replacement, but exclusively focused on t-norm operations. For that reason, in our fuzzy setting, we decided to use the name of “T-Norm replacement” instead of “algebraic replacement” to designate this kind of transformations.

Apart from algebraic replacement, the T-Norm replacement transformation has also some similarities with fuzzy unfolding. In fact, note that the following unfolding-like definition subsumes the first kind of T-Norm replacement: $\mathcal{P}_{k+1} = (\mathcal{P}_k - \{C\}) \cup \{H \leftarrow \overline{B'} \text{ with } \Psi \mid \langle \boxed{L_\Psi} \overline{B} \boxed{R_\Psi}; id \rangle \rightarrow_{FR3} \langle \boxed{L_\Psi} \overline{B'} \boxed{R_\Psi}; \sigma \rangle\}$, where a simple fuzzy SLD-resolution step of kind 3 have been applied for performing a T-Norm replacement of type $\llbracket et_1 \rrbracket$. Observe also that, by replacing $\rightarrow_{FR3}$ with $\rightarrow_{FR4}$ in the previous definition, we obtain an alternative, unfolding-like definition, of the second and fourth kinds of T-Norm replacement (depending on which is the $\llbracket et_2 \rrbracket$ -based lf-expression exploited: inside of $\overline{B}$ or directly the whole $\boxed{L_\Psi} \overline{B} \boxed{R_\Psi}$ expression). Anyway, the T-Norm replacement of kind 3 has never been considered in the literature, neither implicitly nor explicitly (although it was proposed as future work in [5]), and its application is able to transform a non unit program clause into a fact (i.e., a clause with empty body). Observe that no other transformation has this capability, which indirectly imply that, in the best case, although programs to be transformed by this transformation must necessarily belong to the **lf-Prolog** superclass, once transformed by T-Norm replacement of kind 3, they may recover the **f-Prolog** syntax.

The following example illustrates the application of transformations based on T-Norm replacement and some of their advantages.

Example 4. Let us continue with the transformation sequence started in example 3 by performing now some T-Norm replacements. So, the next program in the sequence is $\mathcal{P}_4 = (\mathcal{P}_3 - \{C_9\}) \cup \{C_{10}\}$, where $C_{10} \equiv p(a) \leftarrow 0.63, 0.7 \text{ with } \langle 0.8, \text{prod}, \text{min} \rangle$. It has been obtained by T-Norm replacement of kind 1 on clause C_9 (note that $\text{prod}(0.9, 0.7) = 0.63$). Moreover, since $\text{min}(0.63, 0.7) = 0.63$, a T-Norm replacement of kind 4 on this last clause generates the new one $C_{11} \equiv p(a) \leftarrow 0.63 \text{ with } \langle 0.8, \text{prod}, \text{min} \rangle$. Finally clause C_{11} becomes in the fact $C_{12} \equiv p(a) \leftarrow \text{with } 0.504$ (since $\text{prod}(0.63, 0.8) = 0.504$) after the last T-Norm replacement of kind 3. Hence, the final program is $\mathcal{P}_6 = \{C_3, C_4, C_5, C_6, C_8, C_{12}\}$ and now the derivation shown in example 2 can reduce its length in six steps thanks to the use of clause C_{12} , which evidences once again the improvements achieved not only by unfolding, but also by T-Norm replacement, on transformed programs.

To finish this section, we formalize the best properties one can expect in a transformation system like ours, namely: i) on the theoretical side, the exact and total correspondence between fuzzy computed answers for goals executed on the original and the transformed programs and, ii) on the practical side, the gains in efficiency when executing transformed programs by reducing the number of fuzzy SLD-resolution steps needed to solve a goal.

Theorem 2 (Strong Correctness of the Transformation System).

Let $(\mathcal{P}_0, \dots, \mathcal{P}_k)$ be a transformation sequence of lf-Prolog programs such that each program in the sequence, except the initial f-Prolog program $\mathcal{P}_0$, is obtained from the immediately preceding one by applying fuzzy unfolding or T-Norm replacement. Then, for any lf-Prolog goal $\mathcal{G} \equiv \leftarrow Q$ with $le_2 = \llbracket et_2 \rrbracket$, we have that:

$$\langle \boxed{L_\Psi} \ Q \ \boxed{R_\Psi} ; id \rangle \rightarrow_{FR}^n \langle r; \theta \rangle \text{ in } \mathcal{P}_0 \text{ iff } \langle \boxed{L_\Psi} \ Q \ \boxed{R_\Psi} ; id \rangle \rightarrow_{FR}^m \langle r; \theta \rangle \text{ in } \mathcal{P}_k$$

where $\Psi \equiv \langle 1, void, \llbracket et_2 \rrbracket \rangle$ and the number of fuzzy SLD-resolution steps in each derivation verify that $m \leq n$.

5 Conclusions

This work introduces two safe unfolding-based transformation rules for optimizing fuzzy logic programs. To the best of our knowledge, this is the first time the issue, of integrating transformation techniques in the context of fuzzy logic languages, is treated in the literature.

After an inspection of the main proposals for the inclusion of fuzzy logic into a logic programming setting, we have selected an extension of the language described in [21], that we call lf-Prolog, since we think it is the best suited to deal with the problems that may arise in the transformation process of logic programs. It is remarkable that lf-Prolog is provided with a labeled mark language. Inspired in [19], we have extended this language in order to be able to code different fuzzy logics inside the same program, which greatly enhances the expressive power of the former language. Also, as an auxiliary result, we have established the independence of the Fuzzy Computation Rule for lf-Prolog programs and goals (Theorem 1).

We have defined the fuzzy unfolding and T-Norm replacement of lf-Prolog programs (Definition 5 and 6) and we have demonstrated the (strong) correctness (Theorem 2) of the transformation system. Moreover, we have proved that transformation sequences can be guided in a blind way since any transformation step based on fuzzy unfolding and T-Norm replacement always produces an improvement on transformed programs. This contrasts with other transformation rules, like definition introduction or folding that may degrade the efficiency of programs, if appropriate “transformation strategies” are not used to drive the construction of the transformation sequence.

Finally, it is important to say that the results in this paper can be thought as a basis to optimize fuzzy prolog programs and they are the first step in the construction of a global fold/unfold framework (including more transformation rules and strategies) for optimizing this class of programs.

References

1. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Rules + Strategies for Transforming Lazy Functional Logic Programs. *Theoretical Computer Science*, 311:479–525, 2004.

2. J. F. Baldwin, T. P. Martin, and B. W. Pilsworth. *Fril- Fuzzy and Evidential Reasoning in Artificial Intelligence*. John Wiley & Sons, Inc., 1995.
3. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
4. M. Ishizuka and N. Kanai. Prolog-ELF Incorporating Fuzzy Logic. In *Proc. of IJCAI'85*, pages 701–703. Morgan Kaufmann, 1985.
5. P. Julian, G. Moreno, and J. Penabad. Unfolding Fuzzy Logic Programs. In *Proc. of ISDA'04*, pages 595–600. IEE Computer Society, 2004.
6. P. Julian and C. Villamizar. Analizing Definitional Trees: Looking for Determinism. In *Proc. of FLOPS'04*, pages 55–69. Springer-Verlag LNCS 2998, 2004.
7. Frank Klawonn and Rudolf Kruse. A Łukasiewicz logic based Prolog. *Mathware & Soft Computing*, 1(1):5–29, 1994.
8. G. J. Klir and B. Yuan. *Fuzzy Sets and Fuzzy Logic*. Prentice-Hall, 1995.
9. J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.
10. R.C.T. Lee. Fuzzy Logic and the Resolution Principle. *Journal of the ACM*, 19(1):119–129, 1972.
11. Deyi Li and Dongbo Liu. *A fuzzy Prolog database system*. John Wiley & Sons, Inc., 1990.
12. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
13. H.T. Nguyen and E.A. Walker. *A First Course in Fuzzy Logic*. Chapman & Hall, 2000.
14. C. Palamidessi. Algebraic Properties of Idempotent Substitutions. In *Proc. of ICALP'90*, pages 386–399. Springer LNCS 443, 1990.
15. A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
16. B. Schweizer and A. Sklar, Probabilistic Metric Spaces. North-Holland, 1983.
17. E. Trillas, C. del Campo and S. Cubillo, When QM-Operators Are Implication Functions and Conditional Fuzzy Relations. *International Journal of Intelligent Systems*, 15:647–655, John Wiley and Sons, Inc. 2000.
18. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In *Proc. of ICLP'84*, pages 127–139, 1984.
19. C. Vaucheret, S. Guadarrama, and S. Muñoz. Fuzzy prolog: A simple general implementation using *clp(r)*. In *Proc. of LPAR'02*, pages 450–463. LNAI 2514, Springer Verlag, 2002.
20. P. Vojtas. Fuzzy Logic Programming. *Fuzzy Sets and Systems*, 124(1):361–370, 2001.
21. Peter Vojtas and L. Paulík. Soundness and completeness of non-classical extended SLD-resolution. In *Proc. of ELP'96 Leipzig*, pages 289–301. LNCS 1050, Springer Verlag, 1996.
22. L. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.
23. L. Zadeh. Calculus of fuzzy restrictions. *Fuzzy Sets and Their Applications to Cognitive and Decision Processes*, pages 1–39, 1975.

Integrated Transformation Strategies for Integrated Declarative Programs ^{*}

Soledad González and Ginés Moreno

Department of Computer Science, EPSA, University of Castilla-La Mancha,
02071 Albacete, Spain. (sgonzalez,gmoreno)@info-ab.uclm.es

Abstract. This paper investigates the optimization by fold/unfold of declarative programs that integrate the best features from both (lazy) functional and logic programming. Transformation sequences are guided by a mixed strategy which, in three low-level transformation phases, successfully combines two well-known heuristics -composition and tupling-, thus avoiding the construction of intermediate data structures and redundant computations. In particular, whereas composition is able to produce a single function definition for some nested (composed) functions, the tupling method merges non-nested functions calls into a new function definition called *eureka*. We solve the non trivial problem of discovering the set of calls to be tupled in an incremental way, i.e. by chaining different *eureka* definitions where only non-nested calls sharing common variables are taken into account. Moreover, by appropriately combining both strategies, together with a simplification pre-process based on a kind of normalization, we automatically optimize a wide range of programs (with nested and/or non-nested function calls) at a very low cost.

Functional logic programming languages combine the operational methods and advantages of the most important declarative programming paradigms, namely functional and logic programming. The operational principle of such modern integrated declarative languages is usually based on *narrowing*. A *narrowing step* instantiates variables in an expression and applies a reduction step to a redex of the instantiated expression. *Needed narrowing* (symbolized by $\leadsto$) is the currently best lazy narrowing strategy defined for *inductively sequential* programs, a wide subclass of CB, orthogonal (confluent and non-terminating) programs.

On the other hand, the fold/unfold transformation approach was first introduced in [2] to optimize functional programs and then used for logic [4] and lazy functional logic programs [1]. It is based on the construction, by means of a *strategy* (composition, tupling), of a sequence of equivalent programs each obtained from the preceding ones by using an *elementary* transformation rule. Here, we use the following set of transformation rules formally described in [1]:

Definition Introduction: This operation adds to the last program in the sequence a new rule (called “eureka”) of the form $f(\overline{x_m}) \rightarrow r$, where r is any valid expression containing the set of variables $\{\overline{x_m}\}$ and f is a *new* symbol.

Unfolding: It basically consists in the replacement of a program rule $l \rightarrow r$ by the set of rules $\{\sigma(l) \rightarrow r' \mid r \leadsto_{\sigma} r'\}$ obtained after applying a needed

^{*} This paper abstracts the work presented in the *9th Ibero-American Conference on Artificial Intelligence, IBERAMIA'2004* (Springer, LNAI 3315, pp. 419-429), which has been partially supported by CICYT under grant TIC 2001-2705-C03-03.

narrowing step (in all its possible forms) on its rhs. When σ is empty, we call normalizing to the resulting (rewriting-based) unfolding transformation.

Folding: It replaces in the rhs of a program rule, an instance of the rhs of an eureka rule by the corresponding instance of its lhs.

Abstraction: In essence, to abstract a set of sub-expressions $e_1, \dots, e_n$ in the rhs of a program rule consists in replacing all its occurrences by a set of fresh variables $x_1, \dots, x_n$ and adding the local declarations $x_1 = e_1, \dots, x_n = e_n$.

Transformation strategies are used to drive the application of the previous rules in three phases. For the simpler case of composition, an eureka rule containing a nested expression $f(g(\square))$ in its rhs is initially created by the application of the definition introduction rule. Then, it is repeatedly unfolded until finding and folding (w.r.t. the original eureka) instances (regularities) of $f(g(\square))$ in its rhs. It suffices to improve most cases, such as function **sum-gen** (programs $\mathcal{R}_0 - \mathcal{R}'_0$) in table 4. Moreover, for the case of tupling, we find three important problems which can be solved in a similar way to [3], as shown in the following table.

Table 1. The Tupling Algorithm

INPUT:	Initial Program $\mathcal{R}$ and Program Rule $R = (l \rightarrow r) \in \mathcal{R}$
BODY:	*****DEFINITION INTRODUCTION PHASE***** 1. Let T be the tuple of patterns sharing variables in r 2. DEFINITION_INTRODUCTION: generate $R_{def} = (f_{new}(\bar{x}) \rightarrow T)$ ***** UNFOLDING PHASE *****
	3. Let $\mathcal{R}_{unf} = \{R_{def}\}$ be a program 4. Repeat $\mathcal{R}_{unf} = \text{NORMALIZE}(\text{UNFOLD}(\mathcal{R}_{unf}, \mathcal{R}), \mathcal{R})$ until every rule $R' \in \mathcal{R}_{unf}$ verifies $\text{TEST}(R', R_{def}) > 0$ ***** FOLDING PHASE *****
	5. Let $\mathcal{R}_{fold} = (\mathcal{R}_{unf} \cup R)$ be a program 6. For every rule $R' \in \mathcal{R}_{fold}$ verifying $\text{TEST}(R, R_{def}) = 2$ $\mathcal{R}_{fold} = (\mathcal{R}_{fold} - \{R'\}) \cup \{\text{FOLD}(\text{ABSTRACT}(R', R_{def}))\}$
OUTPUT:	Transformed Program $\mathcal{R}_{fold}$

The first problem is to discover the set of calls (arbitrarily distributed on the original rule) to be tupled when generating the initial eureka. As shown in table 1 (steps 1 and 2), we solve it by simply considering patterns sharing common variables. The same technique is used for stopping the unfolding loop (steps 3 and 4 in table 1, where unfolded rules are normalized once obtained). The key point is the use of function **TEST** described in table 2, which compares the tuples of patterns sharing variables of eureka and unfolded rules in order to decide when no more unfolding steps are needed (base case), a regularity has been found or a new tupling cycle must be (incrementally) re-initiated. The last problem consists in packing the appropriate set of calls to be folded during the folding phase (steps 5 and 6 of table 1) when function **TEST** returns 2. This is done by abstracting (note that this step is not needed when performing composition) once again patterns sharing variables. In contrast with prior, rather complicated, tupling approaches, we remark the benefits of using this easy (purely syntactic method) along the three transformation phases as well as the elegance of function **TEST**, as illustrated by programs $\mathcal{R}_0 - \mathcal{R}'_0$ (function **average**) and $\mathcal{R}_1 - \mathcal{R}'_1$ in table 4.

Table 2. Function TEST

INPUT:	Original eureka	$f_{new}(\bar{x}) \rightarrow T$	such that $T = \langle t_1, \dots, t_n \rangle$
	Unfolded Rule	$f_{new}(\bar{s}) \rightarrow T'$	such that $T' = \langle t'_1, \dots, t'_n \rangle$
BODY:	If $S = \langle s'_1, \dots, s'_m \rangle$ is the tuple of patterns sharing variables in T' , then look if one of the following Stopping Conditions holds		
	SC1:	$m < n$	$\Rightarrow$ return 1 <i>* base case definition *</i>
	SC2:	S is instance of T	$\Rightarrow$ return 2 <i>***** foldability *****</i>
	SC3:	$m > n$	$\Rightarrow$ return 3 <i>*** new tupling loop **</i>
	Otherwise:		$\Rightarrow$ return 0 <i>** continue unfolding *</i>

Finally, we propose three final improvements which can be naturally embedded in our tupling algorithm as shown in table 3. The global algorithm submits all rules of a given program to the transformation process (step 2 in table 3). Before being treated by tupling (step 2.3), each program rule R , is firstly simplified by normalization (generating rule R' in step 2.1) and composition (generating rule R'' in step 2.2), which suffices to optimize programs $\mathcal{R}_2 - \mathcal{R}'_2$ (which has been traditionally considered a difficult problem to be solved by tupling) and $\mathcal{R}_3 - \mathcal{R}'_3$ in table 4, respectively. Finally, our last improvement (step 2.4 in table 3) introduces our incremental notion of tupling by generating chained eureka definitions and optimizing complex cases such as programs $\mathcal{R}_4 - \mathcal{R}'_4$ in table 4.

To finish, it is important to remark that, similarly to most classical approaches, termination is not always ensured in our algorithm (mainly due to the normalizing/unfolding loops and the recursive tupling calls). Nevertheless, our strategies inherits strong correctness from the set of transformation rules of [1].

Table 3. Improved Algorithm: Normalization+Composition+Incremental Tupling

INPUT:	Original Program $\mathcal{R}$
BODY:	1. Initialize program $\mathcal{R}' = \mathcal{R}$ 2. For each rule $R \in \mathcal{R}'$ do 2.1. $\{R'\} = \text{NORMALIZE}(\{R\}, \mathcal{R}')$ 2.2. $\mathcal{R}' = \text{Composition_Algorithm}((\mathcal{R}' - \{R\}) \cup \{R'\}, R')$ 2.3. $\mathcal{R}' = \text{Tupling_Algorithm}(\mathcal{R}', R'')$ 2.4. For every rule $R^* \in \mathcal{R}'$ verifying $\text{TEST}(R^*, R_{def}) = 3$ $\mathcal{R}' = \text{Tupling_Algorithm}(\mathcal{R}', R^*)$
OUTPUT:	Transformed Program $\mathcal{R}'$

References

1. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Rules + Strategies for Transforming Lazy Functional Logic Programs. *Theoretical Computer Science*, 311:479–525, 2004.
2. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
3. G. Moreno. Automatic Optimization of Multi-Paradigm Declarative Programs. In *Proc. of the 8th Ibero-American Conference on Artificial Intelligence, IBERAMIA'2002*, pages 131–140. Springer LNAI 2527, 2002.
4. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In *Proc. of the 2nd Int'l Conf. on Logic Programming, Uppsala, Sweden*, pages 127–139, 1984.

Table 4. Examples of Program Transformation

$\mathcal{R}_0$:	***** Original program $\mathcal{R}_0$: some operations with lists of numbers ***** $(R_1): \text{len}([]) \rightarrow 0$ $(R_2): \text{len}([H T]) \rightarrow s(\text{len}(T))$ $(R_3): \text{sum}([]) \rightarrow 0$ $(R_4): \text{sum}([H T]) \rightarrow H + \text{sum}(T)$ $(R_5): \text{gen}(0) \rightarrow []$ $(R_6): \text{gen}(s(N)) \rightarrow [s(N) \text{gen}(N)]$ $(R_7): \text{sum_gen}(X) \rightarrow \text{sum}(\text{gen}(X))$ $(R_8): \text{average}(L) \rightarrow \text{sum}(L)/\text{len}(L)$
$\mathcal{R}'_0$:	***** After “Composition”, rule R_7 is replaced by the set of rules: $\text{sum_gen}(X) \rightarrow \text{new}_1(X)$, $\text{new}_1(0) \rightarrow 0$ and $\text{new}_1(s(X)) \rightarrow s(X) + \text{new}_1(X)$ ***** Moreover, after “Tupling”, rule R_8 is replaced by the set of rules: $\text{average}(L) \rightarrow U/V$ where $\langle U, V \rangle = \text{new}_2(L)$, $\text{new}_2([]) \rightarrow \langle 0, 0 \rangle$ and $\text{new}_2([H T]) \rightarrow \langle H + U, s(V) \rangle$ where $\langle U, V \rangle = \text{new}_2(T)$
GAIN	Function new_1 avoids the generation and subsequent traversal of an auxiliary list, whereas the two traversals of a list performed when computing its average are reduced to only one by eureka new_2.
$\mathcal{R}_1$:	***** Original program $\mathcal{R}_1$ (fibonacci numbers): $(R_1): \text{fib}(0) \rightarrow s(0)$ $(R_2): \text{fib}(s(0)) \rightarrow s(0)$ $(R_3): \text{fib}(s(s(X))) \rightarrow \text{fib}(s(X)) + \text{fib}(X)$
$\mathcal{R}'_1$:	***** After “Tupling”: $\text{fib}(s(s(N))) \rightarrow U + V$ where $\langle U, V \rangle = \text{new}(N)$ $\text{new}(0) \rightarrow \langle s(0), s(0) \rangle$ $\text{new}(s(N)) \rightarrow \langle U + V, U \rangle$ where $\langle U, V \rangle = \text{new}(N)$
GAIN	Tupling reduces to linear the exponential complexity of $\mathcal{R}_1$.
$\mathcal{R}_2$:	***** Original program $\mathcal{R}_2$: list of factorials of even numbers ***** $(R_1): \text{fact}(0) \rightarrow s(0)$ $(R_2): \text{fact}(s(N)) \rightarrow s(N) * \text{fact}(N)$ $(R_3): \text{fli}^{\text{even}}(0) \rightarrow []$ $(R_4): \text{fli}^{\text{even}}(s(0)) \rightarrow []$ $(R_5): \text{fli}^{\text{even}}(s(s(N))) \rightarrow [\text{fact}(s(s(N)))] \text{fli}^{\text{even}}(N)$
$\mathcal{R}'_2$:	***** After “Normalization+Tupling”, rule R_5 is replaced by: $\text{fli}^{\text{even}}(s(s(N))) \rightarrow [s(s(N)) * s(N) * U V]$ where $\langle U, V \rangle = \text{new}^{\text{even}}(N)$, $\text{new}^{\text{even}}(0) \rightarrow \langle s(0), [] \rangle$, $\text{new}^{\text{even}}(s(0)) \rightarrow \langle s(0) * s(0), [] \rangle$ and $\text{new}^{\text{even}}(s(s(N))) \rightarrow \langle s(s(N)) * s(N) * U, [s(s(N)) * s(N) * U V] \rangle$ where $\langle U, V \rangle = \text{new}^{\text{even}}(N)$
GAIN	The algorithmic complexity of $\mathcal{R}_2$ is reduced from $O(n^2)$ to $O(n)$.
$\mathcal{R}_3$:	** Original program $\mathcal{R}_3 = \mathcal{R}_0 \cup \{ (R_9): f(N) \rightarrow \text{sum}(\text{gen}(N))/\text{len}(\text{gen}(N)) \}$
$\mathcal{R}'_3$:	***** After “Composition+Tupling”, rule R_9 is replaced by the new rules: $f(N) \rightarrow U/V$ where $\langle U, V \rangle = \text{new}(N)$, $\text{new}(0) \rightarrow \langle 0, 0 \rangle$ and $\text{new}(s(N)) \rightarrow \langle s(N) + U, s(V) \rangle$ where $\langle U, V \rangle = \text{new}(N)$
GAIN	Neither (two) lists are created nor traversed (twice) in $\mathcal{R}'_3$.
$\mathcal{R}_4$:	***** Original program $\mathcal{R}_4$: the Hanoi’s Towers problem ***** $(R_1): \text{ap}([], L) \rightarrow L$, $(R_2): \text{ap}([H T], L) \rightarrow [H \text{ap}(T, L)]$, $(R_3): \text{h}(0, A, B, C) \rightarrow []$ $(R_4): \text{h}(s(N), A, B, C) \rightarrow \text{ap}(\text{h}(N, A, C, B), [\text{mov}(A, B) \text{h}(N, C, B, A)])$
$\mathcal{R}'_4$:	**** After “Tupling+Tupling” (incremental tupling), rule R_4 is replaced by: $\text{h}(s(N), A, B, C) \rightarrow \text{ap}(Z_1, \text{mov}(A, B) : Z_2)$ where $\langle Z_1, Z_2 \rangle = \text{new}_1(N, A, B, C)$ $\text{new}_1(0, A, B, C) \rightarrow \langle [], [] \rangle$ $\text{new}_1(s(N), A, B, C) \rightarrow \langle \text{ap}(Z_1, [\text{mov}(A, C) Z_2]), \text{ap}(Z_3, [\text{mov}(C, B) Z_1]) \rangle$ where $\langle Z_1, Z_2, Z_3 \rangle = \text{new}_2(N, A, B, C)$ $\text{new}_2(0, A, B, C) \rightarrow \langle [], [], [] \rangle$ $\text{new}_2(s(N), A, B, C) \rightarrow \langle \text{ap}(Z_1, [\text{mov}(A, B) Z_2]), \text{ap}(Z_3, [\text{mov}(B, C) Z_1]),$ $\text{ap}(Z_2, [\text{mov}(C, A) Z_3]) \rangle$ where $\langle Z_1, Z_2, Z_3 \rangle = \text{new}_2(N, A, C, B)$
GAIN	The two recursive calls performed in rule R_4 are avoided in $\mathcal{R}'_4$, where function definitions only admit a single recursive call.