First International Workshop on Developments in Computational Models

DCM 2005

Lisbon, Portugal

10 July, 2005

Preliminary Proceedings

Edited by Maribel Fernández and Ian Mackie

DCM 2005 is a satellite workshop of ICALP 2005.

Contents

Preface

Several new models of computation have emerged in the last few years, and many developments of traditional computational models have been proposed with the aim of taking into account the new demands of computer systems users and the new capabilities of computation engines. A new computational model, or a new feature in a traditional one, usually is reflected in a new family of programming languages, and new paradigms of software development.

The aim of this first DCM workshop is to bring together researchers who are currently developing new computational models or new features for traditional computational models, in order to foster their interaction, to provide a forum for presenting new ideas and work in progress, and to enable newcomers to learn about current activities in this area.

Topics of interest include all abstract models of computation and their applications to the development of programming languages and systems. This includes:

- Functional calculi: lambda-calculus, rho-calculus, term and graph rewriting;
- Object calculi;
- Interaction-based systems: interaction nets, games;
- Concurrent models: process calculi, action graphs;
- Calculi expressing locality, mobility, and active data;
- Quantum computational models;
- Biological or chemical models of computation;

The *First International Workshop on Developments in Computational Models* was held in Lisbon, Portugal on the 10th July 2005, as a satellite event of ICALP 2005. This volume contains the preliminary versions of the papers presented at DCM 2005. The final version will be published in Electronic Notes in Theoretical Computer Science. The Programme Committee selected 13 papers for presentation at the workshop. In addition the programme included an invited tutorial by Raja Nagarajan.

The Programme and Organising Committee consisted of:

- Vincent Danos, University of Paris 7 (France)
- Mariangiola Dezani, University of Torino (Italy)
- Maribel Fernández, King's College London (UK), co-chair
- Claude Kirchner, LORIA and INRIA (France)
- Cosimo Laneve, University of Bologna (Italy)

We would like to thank all those who contributed to DCM 2005. We are grateful to the Programme Committee for their careful and efficient work in reviewing and selecting the submitted papers, as well as to: José B. Almeida, Christopher Anderson, Martin Berger, Bob Coecke, Alessandra Di Pierro, Ross Duncan, Luigi Liquori, Rajagopal Nagarajan, Luca Paolini, Marco Pedicini, Simon Perdrix, Jorge S. Pinto, Nikolaos Siafakas, François-Régis Sinot, Gareth Stoyle and Daniele Varacca for their work as referees. DCM 2005 was sponsored by the Department of Computer Science, King's College London.

Maribel Fernández and Ian Mackie

London, 27 June, 2005.

A Generalized Higher-Order Chemical Computation Model

J.-P. Banâtre [a], P. Fradet [b] and Y. Radenac [a]

[a] *Irisa, Université de Rennes 1, Inria, Campus de Beaulieu, 35042 Rennes Cedex, France*
{jbanatre,yradenac}@irisa.fr

[b] *INRIA Rhône-Alpes, 655 avenue de l'Europe, 38330 Montbonnot, France*
Pascal.Fradet@inria.fr

Abstract

Gamma is a programming model where computation is seen as chemical reactions between data represented as molecules floating in a chemical solution. Formally, this model is represented by the rewriting of a multiset where rewrite rules model the chemical reactions. Recently, we have proposed the γ-calculus, a higher-order extension, where the rewrite rules are first-class citizen. The work presented in this paper increases further the expressivity of the chemical model with generalized multisets: multiplicities of elements may be infinite and/or negative. Applications of these new notions are illustrated by some programming examples.

1 Introduction

The Gamma formalism was proposed in [4] to capture the intuition of computation as the global evolution of a collection of atomic values interacting freely. Gamma can be introduced intuitively through the chemical reaction metaphor. The unique data structure in Gamma is the multiset which can be seen as a chemical solution. A simple program is made of a *reaction condition* and an *action*. Execution proceeds by replacing elements satisfying the reaction condition by the elements specified by the action. The result of a Gamma program is obtained when a stable state is reached that is to say when no more reactions can take place.

For example, the computation of the maximum element of a non empty set can be described by the reaction rule **replace** x, y **by** x **if** $x \geq y$ meaning that any couple of elements x and y of the multiset is replaced by x if the condition is fulfilled. This process goes on till a stable state is reached, that is to say, when only the maximum element remains. Note that, in this definition, nothing is said about the order of evaluation of the comparisons. If several disjoint pairs of elements satisfy the condition, the reactions can be performed in parallel.

Gamma can be formalized as a multiset rewriting language. The literature about Gamma as summarized in [1] is based on finite multisets (often called bags). However, one may think of extensions to this basic concepts by generalizing the multiplicity of elements in multisets to infinity (*infinite multisets*) and even multisets with elements possessing a negative multiplicity (*hybrid multisets*).

In this paper, we investigate these unconventional multiset structures (infinite and hybrid multisets) and show how they can be interpreted in a chemical programming framework. Section 2 presents the basic framework and introduces a higher-order chemical model [3]. Section 3 presents HOCl, a language based on the previous model which integrates infinite multisets and negative multiplicity. It sketches the semantics and implementation of the language using characteristic functions. We conclude in Section 4 with a short review of related work.

2 A higher-order chemical model

In this section, we introduce a higher-order chemical model called the γ-calculus [3]. In this chemical model of computation, every element (including reaction rules) is considered as a molecule. A program is a solution of

$$
\begin{array}{rcll}
M & ::= & x & \textit{; variable} \\
 & | & \gamma(P)\lfloor C \rfloor.M & \textit{; } \gamma\textit{-abstraction (reaction rule)} \\
 & | & M_1, M_2 & \textit{; multiset (AC)} \\
 & | & \langle M \rangle & \textit{; solution} \\
 & & & \\
P & ::= & x & \textit{; matches any molecule} \\
 & | & P_1, P_2 & \textit{; matches a compound molecule} \\
 & | & \langle P \rangle & \textit{; matches an inert solution}
\end{array}
$$

Grammar 1: Syntax of molecules.

molecules. A molecule can be (cf. Grammar 1) (1) a variable x that can represent any molecule, (2) a γ-abstraction $\gamma(P)\lfloor C \rfloor.M$ where P is the pattern which determines the format (or type) of the expected molecule, C is the reaction condition and M the result of the reaction, (3) a compound molecule (M_1, M_2) built with the associative and commutative constructor ",", or (4) a solution denoted by $\langle M \rangle$ which isolates a molecule M from the others.

Molecules can be freely reorganized using the associativity and commutativity of the multiset constructor ",":

$$(M_1, M_2), M_3 \equiv M_1, (M_2, M_3) \qquad M_1, M_2 \equiv M_2, M_1$$

These rules can be seen as a formalization of the Brownian motion of chemical solutions.

Another distinctive feature of chemical models is the reaction concept. In our model, it is represented by a rewrite rule:

$$(\gamma(P)\lfloor C\rfloor.M),\ N \rightarrow \phi M \qquad \text{if } P \text{ match } N = \phi \text{ and } \phi C$$

If a γ-abstraction "meets" a closed molecule N that matches the pattern P (modulo a substitution ϕ) and satisfies the reaction condition C (a boolean expression E), then they may react. The γ-abstraction $\gamma(P)\lfloor C\rfloor.M$ and the molecule N are replaced by the molecule ϕM (i.e. the body of the abstraction after substitution).

An execution consists in such rewritings ("chemical" reactions) until the solution inert (no further rewriting is possible). There are two structural rules:

$$\textit{locality}\ \frac{M_1 \rightarrow M_2}{M, M_1 \rightarrow M, M_2} \qquad \textit{solution}\ \frac{M_1 \rightarrow M_2}{\langle M_1\rangle \rightarrow \langle M_2\rangle}$$

The *locality* rule states that if a molecule M_1 can react then it can do so whatever its context M. The *solution* rule states that reactions can occur within nested solutions.

This model of computation is intrinsically parallel and nondeterministic. As long as reactions involve different molecules, they can take place at the same time in a solution. Furthermore, if a molecule contains several elements, it is not know a priori how they will combine because of the Brownian motion. For example, consider the solution $\langle(\gamma(x, y).x), \textbf{true}, \textbf{false}\rangle$, it may reduce to two distinct stable terms ($\langle\textbf{true}\rangle$ or $\langle\textbf{false}\rangle$) depending on the application of AC rules and whether the x will match **true** or **false**.

Note that abstractions $(\gamma(P)\lfloor C\rfloor.M)$ disappear in reactions: they are said to be *one-shot*. It is easy (using recursion) to define n-shot abstractions (denoted like in Gamma by $\textbf{replace}\,P\,\textbf{by}\,M\,\textbf{if}\,C$) which do not disappear in reactions. For instance, the following program:

$$\langle 2, 10, 5, 8, 11, 8, \textbf{replace}\,x, y\,\textbf{by}\,x\,\textbf{if}\,x \geq y\rangle$$

computes the maximum of some integers. The abstraction does not disappear and reacts as long as there are at least two integers in the solution. The resulting inert solution is $\langle 11, \textbf{replace}\,x, y\,\textbf{by}\,x\,\textbf{if}\,x \geq y\rangle$.

A solution can be matched only if it is inert (i.e. no more reaction can occur in it). This is an important property that permits the ordering of rewritings. For example, Program 1 below uses this restriction to sequence the different steps of the computation. Program 1 computes the largest prime number lower than a given integer. First, only reactions inside the sub-solution may

```
largestPrime10 =
    let $sieve$ = replace $x, y$ by $x$ if $x \text{ div } y$ in
    let $max$ = replace $x, y$ by $x$ if $x \geq y$ in
    $\langle\langle 2, 3, 4, \ldots, 10, sieve\rangle, \gamma\langle sieve, x\rangle.x, max\rangle$
```

Program 1: A program that computes the largest prime number lower than 10.

occur: the *sieve* abstraction computes all prime numbers lower than 10 by removing integers (y) for which a divider (x) is found. When the sub-solution becomes inert, the abstraction $\gamma\langle sieve, x\rangle.x, max$ matches it and extracts all the prime numbers (i.e. suppresses the reaction *sieve*) and computes their maximum using the reaction the *max*.

3 HOCl: multiplets, infinite multiplets and hybrid multisets

HOCl (*Higher Order Chemical Language*) is a programming language based on the previous model extended with infinite and hybrid multisets. Grammar 2 gives its syntax.

3.1 Multiplets

A *multiplet* is a multiset of identical elements. We introduce an exponential notation to denote and manipulate them:

$$a^n \equiv a^{(n-1)}, a \quad \text{and} \quad a^1 \equiv a \qquad \text{for any } n > 1$$

We consider only multiplets of a basic element (constants, abstractions and inert solutions) and not of a multiset.

The exponential notation is also used for pattern-matching. Abstractions may select a fixed number of identical elements. For example, the abstraction matching three 1 is denoted by:

$$\gamma(x^3)\lfloor x = 1\rfloor.M \quad \equiv \quad \gamma(x, y, z)\lfloor x = y \wedge y = z \wedge x = 1\rfloor.M$$

Consider that an integer x is represented by a multiplet of x 1's, the integer division of x by y can be done by grouping y occurrences of 1's and replacing

	Program	
S ::=	M	*; a molecule*
$\mid$	$\emptyset$	*; nothing*
	Molecules	
M ::=	A	*; atom (basic molecule)*
$\mid$	M_1, M_2	*; compound molecule*
$\mid$	x	*; variable* ($x \in V$)
$\mid$	A^i	*; multiplet*
	Atoms	
A ::=	$x : U$	*; variable* ($x \in V$)
$\mid$	E	*; expression*
$\mid$	$\gamma(P)\lfloor E \rfloor.S$	*; one-shot reaction rule*
$\mid$	$\langle M \rangle$	*; solution*
$\mid$	$(name)A$	*; tagged molecule (*name *is a string)*
	Expressions	
E ::=	$\ldots$	*; usual integer and boolean expressions*
	Patterns	
P ::=	$x : T$	*; matches any molecule of type* T
$\mid$	ω	*; matches any molecule or nothing*
$\mid$	P_1, P_2	*; matches a compound molecule*
$\mid$	$(name)P$	*; matches a molecule tagged with name*
$\mid$	$\langle P \rangle$	*; matches an inert solution*
$\mid$	$(P_1 \mid P_2)$	*; matches a pair*
$\mid$	P^i	*; matches a multiplet*

Grammar 2: Syntax of programs.

them by a $\widehat{1}$ (a tagged integer denoted by a "hat" here) for the "quotient":

$$\gamma(\overline{y}).\, \mathbf{replace}\, 1^z\, \mathbf{by}\, \widehat{1}\, \mathbf{if}\ z = y$$

When the solution becomes inert, the multiplicity of $\widehat{1}$ is the quotient, and the multiplicity of the unmarked 1 is the remainder. For example, to divide 5 by 2 we write $\langle 1^5, \overline{2}, \mathit{cluster}\rangle$ which reduces to $\langle 1, \widehat{1}^2, \mathbf{replace}\, 1^z\, \mathbf{by}\, \widehat{1}\, \mathbf{if}\ z = 2\rangle$.

```
intdiv = $\gamma(x, y)$.
    $\mathbf{let}\ \mathit{cluster} = \gamma(\overline{y}).\, \mathbf{replace}\, 1^z\, \mathbf{by}\, \widehat{1}\, \mathbf{if}\ z = y\ \mathbf{in}$
    $\mathbf{let}\ \mathit{sumRemainder} = \mathbf{replace}\ x, y\ \mathbf{by}\ x + y\ \mathbf{in}$
    $\mathbf{let}\ \mathit{sumQuotient} = \mathbf{replace}\ \widehat{x}, \widehat{y}\ \mathbf{by}\ \widehat{x + y}\ \mathbf{in}$
    $\langle\langle 1^x, \overline{y}, \mathit{cluster}, \mathit{sumQuotient}\rangle, \gamma\langle x\rangle.x, \mathit{sumRemainder}\rangle$
```

Program 2: Integer division.

The size of a multiplet may be determined only at runtime. For example,

```
power =
    $\gamma(x, y).\, \mathbf{if}\ y = 0\ \mathbf{then}\ 1$
        $\mathbf{else\ if}\ y > 0\ \mathbf{then}\ x^y, (\gamma(u, v).u * v)^{y-1}$
        $\mathbf{else} (\frac{1}{x})^{-y}, (\gamma(u, v).u * v)^{-y-1}$
```

Program 3: Compute the power function.

the power function (cf. Program 3) computes $\mathit{exp}(x, y)$ $(x, y \in \mathbb{Z})$ by generating two multiplets of variable size: y copies of x and $y-1$ products, if $y > 0$. Here is an example of a reduction to compute $\mathit{exp}(5, 7)$:

$$\langle 5, 7, (\mathit{power})\rangle \rightarrow \langle 5^7, (\gamma(u, v).u * v)^6\rangle \rightarrow$$
$$\langle 25, 5^5, (\gamma(u, v).u * v)^5\rangle \rightarrow \ldots \rightarrow \langle 78125\rangle$$

Since we only consider multiplets of values, multiplets of active solutions are interpreted as the multiplet of the inert form obtained after reduction. For example let $\mathit{choose} = \gamma(x : \mathrm{Int}, y : \mathrm{Int}).x$ which takes two integers and returns one of them, then the solution $\langle 1, 2, \mathit{choose}\rangle^{42}$ denotes either the multiplet $\langle 1\rangle^{42}$ either $\langle 2\rangle^{42}$ but not $\langle 1\rangle^{20}$, $\langle 2\rangle^{22}$. The expression must be evaluated to an inert form before the multiplet can be produced. Similarly, in the abstraction $\gamma(x^3)\lfloor C\rfloor.M$, the variable x can only match an inert value.

In the Jackpot! program (Program 4), three solutions representing the three wheels of the machine evolve independently. When the three solutions become inert, the *win* abstraction checks if it is a multiplet of size 3, i.e. if jackpot =

$\quad\textbf{let}\, \mathit{choose} = \textbf{replace}\, x, y\, \textbf{by}\, x\, \textbf{in}$

$\quad\textbf{let}\, \mathit{wheel} = \langle \text{cherry}, \text{lemon}, \text{bell}, \text{bar}, \text{plum}, \text{orange}, \text{melon}, \text{seven}, \mathit{choose}\rangle\, \textbf{in}$

$\quad\textbf{let}\, \mathit{win} = \gamma(x^3).\,\text{“wonJackpot”}\, \textbf{in}$

$\quad\langle \mathit{wheel}, \mathit{wheel}, \mathit{wheel}, \mathit{win}\rangle$

Program 4: Jackpot!

the three wheels are identical. The solution $\langle \mathit{wheel}^3, \mathit{win}\rangle$ would be a wrong specification since in that case the draw is performed before the multiplet is produced. The machine would always produce a jackpot.

3.2 Infinite multiplets

An obvious generalization of multiplets is to allow infinite multiplets. An infinite multiplet is denoted by M^∞. It represents an infinity of copies of M, but it is also a molecule that can be produced or removed like any other molecule.

An application of infinite multiplets is to allow elements to take part in any number of reactions in parallel. For example, when a n-shot abstraction reacts, it is put back into the solution after the reaction where it may react again and so on. A parallel interpretation of a n-shot abstraction could be made using an infinite multiplet of the corresponding one-shot abstraction:

$$\textbf{replace}\, P\, \textbf{by}\, M\, \textbf{if}\, C \;\simeq\; (\gamma(P)\lfloor C\rfloor.M)^\infty$$

Instead of having one molecule taking part to one reaction at a time (sequential process), we consider having an infinity of abstractions that can react at the same time.

Another example is the quicksort program where all integers must be compared to a predefined pivot. In the following solution all integers lower or equal to the pivot (represented here by an integer tagged by the string *pivot*) are removed:

$$\langle (\mathit{pivot})5, 8, 3, 6, 4, 5, 3, \textbf{replace}(\mathit{pivot})x, y\, \textbf{by}(\mathit{pivot})x\, \textbf{if}\, x \geq y\rangle$$

Since the n-shot abstraction and the pivot are unique, only one reaction can occur at each reduction step. Considering the n-shot abstraction and the pivot as infinite multiplets, several comparisons can occur at the same time:

$$\langle ((\mathit{pivot})5)^\infty, 8, 3, 6, 4, 5, 3, ((\mathit{cmp})\gamma((\mathit{pivot})x, y)\lfloor x \geq y\rfloor.\emptyset)^\infty\rangle$$

By extending patterns, infinite multiplets can be manipulated as a single molecule. For example, when the previous computation is finished, the infinite multiplets can be removed in one reaction:

$$\gamma\langle((pivot)x)^\infty, ((cmp)y)^\infty, z\rangle.\langle z\rangle$$

Since we consider only multiplet of basic elements, the expression $(1, 2)^\infty$ is illegal. This restriction is motivated by representation issues (see Section 3.4).

3.3 Negative multiplicities

Hybrid multisets [6,9] are a generalization of multisets. In a hybrid multiset, the multiplicity of an element can be negative. A molecule a^{-1} can be viewed as an "antimatter": some positive and negative multiplet cannot cohabit in the same solution, they merge into one multiplet whose exponent is the sum of the multiplicities.

For example, assume that a rational number $\frac{p}{q}$ is represented by a molecule which contains the prime factorization of p and q but with negative multiplicities for the latter. Then, $\frac{20}{9}$ is represented by the molecule $2^2, 5, 3^{-2}$. By simply putting molecules representing rational numbers together in the same solution, we compute their product. For example, the product $\frac{20}{9} * \frac{15}{8}$ is represented by the following reaction $\langle 2^2, 5, 3^{-2}\rangle, \langle 3, 5, 2^{-3}\rangle, \gamma(\langle f\rangle, \langle g\rangle).\langle f, g\rangle \rightarrow \langle 5^2, 3^{-1}, 2^{-1}\rangle$.

Infinite negative multiplets can be seen as black holes. It can be used to remove all elements (present or to come) of a multiset. Let pi a reaction computing the product of a multiset of integers. Then, the integer 1, being the neutral element of the product, can be deleted prior to performing pi. The pi operator may be encoded by:

$$\text{pi} = \gamma\langle x\rangle.\langle 1^{-\infty}, x, (\textbf{replace}\, x, y\, \textbf{by}\, x * y)\rangle$$

Before considering any product, all 1's are annihilated, for example:

$$\langle 2, 9, 1, 5, 6, 1, 1, 2\rangle, \text{pi} \rightarrow \langle 1^{-\infty}, 2, 9, 5, 6, 2, (\textbf{replace}\, x, y\, \textbf{by}\, x * y)\rangle \rightarrow \ldots$$

After stabilization, $1^{-\infty}$ must be replaced by 1 (in case that the solution contained only 1's) and then the reaction can be removed.

Other examples that come to mind include a garbage collector that destroys useless molecules by generating their negative counterpart, or negative molecules used as anti-virus: a part of a system identifies a virus and generates an infinity of anti-virus (negative counterparts) that will spread in the whole system to clean it.

3.4 Representation with characteristic functions

Infinite multiplets cannot be represented by enumerating all their atoms. Representing them by generators (e.g. $M^\infty \equiv \textbf{replace}\, \emptyset\, \textbf{by}\, M$) makes it difficult to ensure that $M^\infty, M^\infty \equiv M^\infty$. Instead, we use the standard mathematical representation of a multiset, that is a function associating to each element of the multiset its number of occurrences (multiplicity): A molecule M is represented by a *characteristic function* $\{\!|M|\!\} \;:\; \mathbf{Atoms} \to \mathbb{Z} \cup \{+\infty, -\infty\}$ such that for all $x \in \mathbf{Atoms}$, $\{\!|M|\!\}(x)$ represents the number of occurrences of x in M. The set of values **Atoms** is defined as:

$$\mathbf{Atoms} ::= e \;\mid\; \gamma(P)\lfloor B\rfloor.M \;\mid\; \langle M\rangle \;\mid\; (\mathit{ident})A$$

Atoms are either an expression e in normal form (an integer, a boolean or a pair), a γ-abstraction, a solution (represented by its characteristic function) or a named molecule.

The representation of multisets in terms of functions depends on a notion of equality on **Atoms**. Since our multisets may contain programs, a true semantic equality cannot be implemented. A simple choice would be to use equality on basic values (integers, booleans and names), each composite value (γ-abstractions, sub-solutions, etc.) being only equal to themselves (structural equality). For example, the solution $\langle 1^{\infty}, 4, 5^{-2}, (\gamma(x,y).x), \langle 3, \mathbf{true}\rangle\rangle$ is represented by the function that for any $x \in \mathbf{Atoms}$:

$$f(x) = \begin{cases} \infty & \text{if } x = 1 \\ 1 & \text{if } x = 4 \\ -2 & \text{if } x = 5 \\ 1 & \text{if } x = \gamma(x,y).x \\ 1 & \text{if } \{\!|x|\!\} = g \\ 0 & \text{otherwise} \end{cases} \qquad \text{with } g(x) = \begin{cases} 1 \text{ if } x = 3 \\ 1 \text{ if } x = \mathbf{true} \\ 0 \text{ otherwise} \end{cases}$$

With this representation, infinite and/or negative multiplets can be represented and manipulated efficiently.

4 Conclusion

In this short paper, we have tried to convey the main ideas and applications of infinite and hybrid multisets. The interested reader will be able to find more details in [2].

As far as the higher-order generalization is concerned, a higher-order extension of Gamma has already been proposed in [8]. However, reactions were not first-class citizens since they were kept separate from multisets of data. The hmm-calculus [7] (for higher-order multiset machines) is an extension of Gamma where reactions are one-shot and first-class citizens. Although, the hmm-calculus is an interesting attempt to generalize the Gamma model of computation, it has not been thought as a programming language as HOCl.

Other models, inspired by Gamma, are worth to be mentioned. The chemical abstract machine or Cham was proposed in [5] to describe the operational semantics of process calculi. P-systems [10] are computing devices inspired from biology. It consists in nested membranes in which molecules reacts. A set of partially ordered rewrite rules is associated to each membrane. These rules describe possible reactions and communications between membranes of molecules. All these models are first-order.

Currently, we are investigating a complete semantics of infinite and negative multiplets with characteristic functions.

References

[1] Banâtre, J.-P., P. Fradet and D. Le Métayer, *Gamma and the chemical reaction model: Fifteen years after*, in: *Multiset Processing*, LNCS **2235** (2001), pp. 17–44.

[2] Banâtre, J.-P., P. Fradet and Y. Radenac, *Chemical programming with infinite and hybrid multisets* (2005), (available on request).

[3] Banâtre, J.-P., P. Fradet and Y. Radenac, *Principles of chemical programming*, in: S. Abdennadher and C. Ringeissen, editors, *Proceedings of the 5th International Workshop on Rule-Based Programming (RULE 2004)*, ENTCS **124** (2005), pp. 133–147.

[4] Banâtre, J.-P. and D. Le Métayer, *Programming by multiset transformation*, Communications of the ACM (CACM) **36** (1993), pp. 98–111.

[5] Berry, G. and G. Boudol, *The chemical abstract machine*, Theoretical Computer Science **96** (1992), pp. 217–248.

[6] Blizard, W., *Negative membership*, Notre Dame Journal of Formal Logic **31** (1990), pp. 346–368.

[7] Cohen, D. and J. Muylaert-Filho, *Introducing a calculus for higher-order multiset programming*, in: *Coordination Languages and Models*, LNCS **1061**, 1996, pp. 124–141.

[8] Le Métayer, D., *Higher-order multiset programming*, in: A. M. S. (AMS), editor, *Proc. of the DIMACS workshop on specifications of parallel algorithms*, Dimacs Series in Discrete Mathematics **18**, 1994.

[9] Loeb, D., *Sets with a negative number of elements*, Advances in Mathematics **91** (1992), pp. 64–74.

[10] Păun, G., *Computing with membranes*, Journal of Computer and System Sciences **61** (2000), pp. 108–143.

An Universal Accepting Hybrid Network of Evolutionary Processors

Florin Manea [1]

Faculty of Mathematics and Computer Science, University of Bucharest
Str. Academiei 14, 70109, Bucharest, Romania

Carlos Martín-Vide [2]

Research Group on Mathematical Linguistics, Rovira i Virgili University
Pça Imperial Tàrraco 1, 43005 Tarragona, Spain

Victor Mitrana [2]

Faculty of Mathematics and Computer Science, University of Bucharest
Str. Academiei 14, 70109, Bucharest, Romania
and
Research Group on Mathematical Linguistics, Rovira i Virgili University
Pça Imperial Tàrraco 1, 43005 Tarragona, Spain

Abstract

We propose a construction of an accepting hybrid network of evolutionary processors (AHNEP) which behaves as a universal device in the class of all these devices. We first construct a Turing machine which can simulate any AHNEP and then an AHNEP which simulates the Turing machine. We think that this approach can be applied to other bio-inspired computing models which are computationally complete.

Key words: networks of evolutionary processors, Turing machine, universality

1 Introduction

The line of research discussed in this paper lies among a wide range of computational models rooted in molecular biology [13]. A few words about the

[1] Email: `flmanea@funinf.cs.unibuc.ro`
[2] Email: {`carlos.martin,vmi`}`@urv.net`

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

history of introducing the model discussed here. A rather well-known architecture for parallel and distributed symbolic processing, related to the Connection Machine [9] as well as the Logic Flow paradigm [7] consists of several processors, each of them being placed in a node of a virtual complete graph, which are able to handle data associated with the respective node. Each node processor acts on the local data in accordance with some predefined rules, and then local data becomes a mobile agent which can navigate in the network following a given protocol. Only that data which can pass a filtering process can be communicated to the other processors. This filtering process may require to satisfy some conditions imposed by the sending processor, by the receiving processor or by both of them. All the nodes send simultaneously their data and the receiving nodes handle also simultaneously all the arriving messages, according to some strategies, see, e.g., [8,9].

Starting from the premise that data can be given in the form of words, Csuhaj-Varjú & Salomaa introduced in [5] a concept called network of parallel language processors with the aim of investigating this concept in terms of formal grammars and languages. In [1], this concept was modified in a way inspired from cell biology. Each processor placed in the nodes of the network is a very simple processor, an evolutionary processor. This is not a real, existing object but a mathematical concept. By an evolutionary processor it is meant a processor which is able to perform very simple operations, namely formal language theoretic operations that mimic the point mutations in a DNA sequence (insertion, deletion or substitution of a pair of nucleotides). More generally, each node may be viewed as a cell having genetic information encoded in DNA sequences which may evolve by local evolutionary events, namely point mutations. Each node is specialized just for one of these evolutionary operations. Furthermore, the data in each node is organized in the form of multisets of words (each word appears in an arbitrarily large number of copies), and all copies are processed in parallel such that all the possible events that can take place do actually take place. From the biological point of view, it cannot be expected that the components of any biological organism evolve sequentially or the cell reproduction may be modeled within a sequential approach. Cell state changes are modeled by rewriting rules like in formal grammars. The parallel nature of the cell state changes is modeled by the parallel execution of the symbols rewriting according to the rules applied. Consequently, hybrid networks of evolutionary processors might be viewed as bio-inspired computing models. Obviously, the computational process described here is not exactly an evolutionary process in the Darwinian sense. But the rewriting operations we have considered might be interpreted as mutations and the filtering process might be viewed as a selection process. Recombination is missing but it was asserted that evolutionary and functional relationships between genes can be captured by taking only local mutations into consideration [15]. Furthermore, we are not concerned here with a possible biological implementation, though a matter of great importance.

Our mechanisms introduced in [1] are further considered in a series of subsequent works [2,12] as language generating devices and their computational power in this respect is investigated. On the other hand, this model is considered as an accepting device in [11], where a new characterization of **NP** is obtained, as well as a problem solver in [12,10], where a few NP-complete problems are solved in linear time with polynomially bounded resources. The aforementioned models, besides the mathematical motivation, may also have a biological one. Cells always form tissues and organs interacting with each other either directly or via the common environment.

In this paper, we propose a construction of an accepting hybrid network of evolutionary processors which behaves as a universal device in the class of all these devices. We first construct a Turing machine which can simulate any AHNEP and then an AHNEP which simulates the Turing machine. The construction of the Turing machine is presented here while for the construction of the AHNEP the reader is referred to [11]. This result together with the fact that AHNEP is a deterministic and computationally complete device inspired from cell biology and amenable to be used as a problem solver (see [10], where a possible implementation of AHNEP using WWW is discussed) suggests the possibility to construct a sort of "tissue-like computer".

2 Basic definitions

We start by summarizing the notions used throughout the paper. An *alphabet* is a finite and nonempty set of symbols. The cardinality of a finite set A is written $card(A)$. Any sequence of symbols from an alphabet V is called *word (string)* over V. The set of all words over V is denoted by V^* and the empty word is denoted by ε. The length of a word x is denoted by $|x|$ while $alph(x)$ denotes the minimal alphabet W such that $x \in W^*$.

We say that a rule $a \to b$, with $a, b \in V \cup \{\varepsilon\}$ is a *substitution rule* if both a and b are not ε; it is a *deletion rule* if $a \neq \varepsilon$ and $b = \varepsilon$; it is an *insertion rule* if $a = \varepsilon$ and $b \neq \varepsilon$. The set of all substitution, deletion, and insertion rules over an alphabet V are denoted by Sub_V, Del_V, and Ins_V, respectively.

For two disjoint and nonempty subsets P and F of an alphabet V and a word w over V, we define the predicates:
- $\varphi^{(1)}(w; P, F) \equiv P \subseteq alph(w) \ \wedge \ F \cap alph(w) = \emptyset$
- $\varphi^{(2)}(w; P, F) \equiv alph(w) \subseteq P$
- $\varphi^{(3)}(w; P, F) \equiv P \subseteq alph(w) \ \wedge \ F \not\subseteq alph(w)$
- $\varphi^{(4)}(w; P, F) \equiv alph(w) \cap P \neq \emptyset \ \wedge \ F \cap alph(w) = \emptyset$.

An *evolutionary processor over* V is a tuple (M, PI, FI, PO, FO), where:
– Either M is a set of substitution, deletion or insertion rules over the alphabet V. Formally: $(M \subseteq Sub_V)$ or $(M \subseteq Del_V)$ or $(M \subseteq Ins_V)$. The set M represents the set of evolutionary rules of the processor. As one can see, a processor is "specialized" in one evolutionary operation, only.
– $PI, FI \subseteq V$ are the *input* permitting/forbidding contexts of the proces-

sor, while $PO, FO \subseteq V$ are the *output* permitting/forbidding contexts of the processor. Informally, the permitting input/output contexts are the set of symbols that should be present in a string, when it enters/leaves the processor, while the forbidding contexts are the set of symbols that should not be present in a string in order to enter/leave the processor.

We denote the set of evolutionary processors over V by EP_V.

An *accepting hybrid network of evolutionary processors* (AHNEP for short) is a 7-tuple $\Gamma = (V, U, G, N, \alpha, \beta, x_I, x_O)$, where:
• V and U are the input and network alphabets, respectively, $V \subseteq U$.
• $G = (X_G, E_G)$ is an undirected graph, called the *underlying graph* of the network. In this paper, we consider *complete* AHNEPs, i.e. AHNEPs having a complete underlying graph denoted by K_n, where n is the number of vertices.
• $N : X_G \longrightarrow EP_U$ is a mapping which associates with each node $x \in X_G$ the evolutionary processor $N(x) = (M_x, PI_x, FI_x, PO_x, FO_x)$.
• $\alpha : X_G \longrightarrow \{*, l, r\}$; $\alpha(x)$ gives the action mode of the rules of node x on the words existing in that node. Informally, this indicates if the evolutionary rules of the processor are to be applied at the leftmost end of the string, for $\alpha = l$, at the rightmost end of the string, for $\alpha = r$, or at any of its position, for $\alpha = *$.
• $\beta : X_G \longrightarrow \{(1), (2), (3), (4)\}$ defines the type of the *input/output filters* of a node. More precisely, for every node, $x \in X_G$, the following filters are defined:

$$\text{input filter: } \rho_x(\cdot) = \varphi^{\beta(x)}(\cdot; PI_x, FI_x),$$
$$\text{output filter: } \tau_x(\cdot) = \varphi^{\beta(x)}(\cdot; PO_x, FO_x).$$

That is, $\rho_x(w)$ (resp. τ_x) indicates whether or not the word w can pass the input (resp. output) filter of x. More generally, $\rho_x(L)$ (resp. $\tau_x(L)$) is the set of words of L that can pass the input (resp. output) filter of x.
• x_I and $x_O \in X_G$ is the *input node*, and the *output node*, respectively, of the AHNEP.

A *configuration* of an AHNEP Γ as above is a mapping $C : X_G \longrightarrow 2^{V^*}$ which associates a set of words with every node of the graph. A configuration may be understood as the sets of words which are present in any node at a given moment. A configuration can change either by an *evolutionary step* or by a *communication step*. When changing by an evolutionary step, each component $C(x)$ of the configuration C is changed in accordance with the set of evolutionary rules M_x associated with the node x and the way of applying these rules $\alpha(x)$. Formally, we say that the configuration C' is obtained in *one evolutionary step* from the configuration C, written as $C \Longrightarrow C'$, iff

$$C'(x) = M_x^{\alpha(x)}(C(x)) \text{ for all } x \in X_G.$$

When changing by a communication step, each node processor $x \in X_G$ sends one copy of each word it has, which is able to pass the output filter of x, to all the node processors connected to x and receives all the words sent by any node processor connected with x providing that they can pass its input filter. Formally, we say that the configuration C' is obtained in *one communication* *step* from configuration C, written as $C \vdash C'$, iff

$$C'(x) = (C(x) - \tau_x(C(x))) \ \cup \bigcup_{\{x,y\} \in E_G} (\tau_y(C(y)) \cap \rho_x(C(y))) \text{ for all } x \in X_G.$$

Note that words which leave a node are eliminated from that node. If they cannot pass the input filter of any node, they are lost.

Let Γ be an AHNEP, the computation of Γ on the input word $w \in V^*$ is a sequence of configurations $C_0^{(w)}, C_1^{(w)}, C_2^{(w)}, \ldots$, where $C_0^{(w)}$ is the initial configuration of Γ defined by $C_0^{(w)}(x_I) = w$ and $C_0^{(w)}(x) = \emptyset$ for all $x \in X_G$, $x \neq x_I$, $C_{2i}^{(w)} \Longrightarrow C_{2i+1}^{(w)}$ and $C_{2i+1}^{(w)} \vdash C_{2i+2}^{(w)}$, for all $i \geq 0$. By the previous definitions, each configuration $C_i^{(w)}$ is uniquely determined by the configuration $C_{i-1}^{(w)}$. Otherwise stated, each computation in an AHNEP is deterministic. A computation as above immediately halts if one of the following two conditions holds:

(i) There exists a configuration in which the set of words existing in the output node x_O is non-empty. In this case, the computation is said to be an *accepting computation.*

(ii) There exist two consecutive identical configurations.

In the aforementioned cases the computation is said to be finite. The language accepted by Γ is

$$L(\Gamma) = \{w \in V^* \mid \text{ the computation of } \Gamma \text{ on } w \text{ is an accepting one}\}.$$

3 Encoding complete AHNEPs

In this section we describe a way of encoding an arbitrary AHNEP using the fixed alphabet

$$A = \{\$, \#, r, l, *, (1), (2), (3), (4), 0, 1, 2, \bullet, \rightarrow\}.$$

Let $\Gamma = (V, U, K_n, N, \alpha, \beta, x_I, x_O)$ be an AHNEP, where
– $V = \{a_1, a_2, \ldots, a_m\}$,
– $U = \{a_1, a_2, \ldots, a_p\}$, $p \geq m$,
– the nodes of K_n are $x_1, x_2, \ldots, x_n$, with $x_1 = x_I$ and $x_2 = x_O$.

We encode every symbol a_i of U, and denote this encoding by $< a_i >$, in the following way:

$$< a_i >= \begin{cases} 10^i, 1 \leq i \leq m \\ 20^i, m+1 \leq i \leq p \end{cases}$$

Given w, a word over U as above, we define its encoding $< w >$ in the following way:

$< \varepsilon >= 1, < b_1 b_2 \ldots b_k >=< b_1 >< b_2 > \ldots < b_k >, k \geq 1, b_i \in U, 1 \leq i \leq k.$

Let $L \subseteq U^*$ be a finite language, $L = \{w_1, \ldots, w_k\}$. We encode this language by the word $< L >= \bullet < w_1 > \bullet < w_2 > \bullet \ldots \bullet < w_k > \bullet$. The empty language is encoded by $< \emptyset >= \bullet$.

As a direct consequence of the above, we describe how evolutionary rules are encoded:

– Substitution: $a \to b, a, b \in V$ is encoded as $< a > \to < b >$;
– Insertion: $\varepsilon \to a, a \in V$ is encoded as $1 \to < a >$;
– Deletion: $a \to \varepsilon$ is encoded as $< a > \to 1$.
We denote the encoding of the evolutionary rule r by $< r >$. A set of evolutionary rules: $R = \{r_1, \ldots, r_m\}$ is encoded:

$$< R > = \bullet < r_1 > \bullet < r_2 > \bullet \ldots \bullet < r_m > \bullet$$

For each node x we set
$< N(x) > = \# < M_x > \# < PI_x > \# < FI_x > \# < PO_x > \# < FO_x > \#$,
and $< x > = \#\alpha(x) < N(x) > \beta(x)\#$.

We now describe the way Γ is encoded. This is:
$< \Gamma > = \$ < K_n > \$ < x_1 > \$ < x_2 > \$ \ldots \$ < x_n >$, where $< K_n > = 2^n$.

4 Construction of a universal AHNEP

In this section we will prove that there exists an AHNEP Γ_U, such that if the input word of Γ_U is $< \Gamma >< w >$, for some AHNEP Γ and w the followings hold:
• Γ_U halts on the input $< \Gamma >< w >$ if and only if Γ halts on the input w.
• $< \Gamma >< w >$ is accepted by Γ_U if and only if w is accepted by Γ.

The first step of this construction is to define a Turing Machine that behaves as described in the next theorem.

Theorem 4.1 *There exists a Turing Machine T_U, with the input alphabet A, satisfying the following conditions on any input $< \Gamma >< w >$, where Γ is an arbitrary AHNEP and w is a word over the input alphabet of Γ:*
• *T_U halts on the input $< \Gamma >< w >$ if and only if Γ halts on the input w.*
• *$< \Gamma >< w >$ is accepted by T_U if and only if w is accepted by Γ.*

Proof: We describe the way T_U is obtained. Let T'_U be a 4-tapes Turing Machine, with the tapes labeled W, X, Y, Z. The algorithm that this machine implements is the following:
Initialization:
On tape W it is found the input word: $< \Gamma >< w >$. We assume that $\Gamma = (V, U, K_n, N, \alpha, \beta, x_I, x_O)$, the nodes of K_n being $x_1, x_2, \ldots, x_n$, with $x_1 = x_I$ and $x_2 = x_O$. We copy on tape X the encoding of the graph K_n. This means that on tape X we will have n occurrences of the symbol 2. This can be done by copying the part of $< \Gamma >$ between the first and the second occurrence of \$. Each symbol 2 on this tape will be used to keep track of the node that is processed at a given moment. On tape Y we construct the initial configuration of Γ. This is carried out in the following way:

– first we write $n + 1$ symbols \$ on tape Y, if on tape X are n symbols 1,

– then, between the first two occurrences of \$ on tape Y we copy $< w >$ from the input tape and place it between two bullet symbols,

– finally, put a bullet symbol between all the other occurrences of the symbol \$.

To summarize, the content of the tape Y will be:

$$\$\bullet < w > \bullet\$ \bullet \$ \bullet \ldots \bullet \$ \bullet \$.$$

Our strategy is the following:

1. The encoding of the configuration of the i-th node will be memorized between the i-th and $(i+1)$-th occurrences of symbol \$ on tape Y.

2. Tapes Y and Z, the latter containing initially $n+1$ symbols \$, will be used in the simulation of both evolutionary and communication steps.

3. The Evolution and Communication phases run alternatively one after another both being preceded by the Acceptance phase.

Evolution:
We assume that on tape Y we have the encoding of a configuration of Γ (assumption that holds after the Initialization phase). First, we transform tape Z into $\$ \bullet \$ \bullet \$ \bullet \ldots \bullet \$ \bullet \$$ and mark the leftmost unmarked symbol 2 on tape X. Assume that, after marking such a symbol, there are exactly k marked symbols on the tape X. We place the head of tape W on the $(k+1)$-th symbol \$ on this tape, and the heads of tapes Y and Z on the k-th symbol \$. It is not hard to see that the head of tape W is placed at the beginning of the encoding of the node x_k and the head of tape Y is placed at the beginning of the encoding of the configuration $C(x_k)$.

We now store in the current state $\alpha(x_k)$ and read from tape W the encoding of the first evolutionary rule of x_k. Let us assume that this encoding is the word $s_l \to s_r, s_l$; if $s_l \neq 1$, then we consider one by one all words in the encoding of $C(x_k)$ and for each of them we proceed as follows:

(I) If $\alpha(x_k) = *$, then look for the first occurrence of s_l.

(II) If no such occurrence is found insert the word followed by a bullet symbol on tape Z.

(III) If an occurrence of s_l was found, then proceed with one of the following tasks:
   (i) replace s_l by s_r, provided that $s_r \neq 1$,
   (ii) delete s_l, provided that $s_r = 1$ and the obtained word is not empty,
   (iii) replace s_l by 1, otherwise.

(IV) Insert the word obtained at (III) followed by a bullet symbol on tape Z. Look for the next occurrence of s_l in the original word and perform (III) until all occurrences of s_l were found.

(V) If $\alpha(x_k) = l$, then check whether the word starts with s_l. If this is not the case, perform (II), else perform (III) and insert the obtained word followed by a bullet symbol on tape Z.

(VI) If $\alpha(x_k) = r$, then check whether the word ends with s_l. If this is not the case, perform (II), else perform (III) and insert the obtained word followed by a bullet symbol on tape Z.

If $s_l = 1$, then we consider one by one all words in the encoding of $C(x_k)$ and for each of them we proceed as follows:

(I) If $\alpha(x_k) = *$, then look for the first occurrence of 1 or 2.

(II) Insert s_r before this occurrence and insert the obtained word followed by a bullet symbol on tape Z.

(III) Repeat (II) for all occurrences of 1 and 2 in the original word.

(IV) Append s_r at the end of the original word and insert the new word followed by a bullet symbol on tape Z.

(V) If $\alpha(x_k) = l$, then insert s_r before the first occurrence of 1 or 2 and insert the obtained word followed by a bullet symbol on tape Z.

(VI) If $\alpha(x_k) = r$, then append s_r at the end of the word and insert the new word followed by a bullet symbol on tape Z.

We repeat the above process for all evolutionary rules of x_k.

The we mark the $(k+1)$-th symbol 2 on tape X and move on the head of tape W on the $(k+2)$-th symbol \$ on this tape, and the heads of tapes Y and Z on the $(k+1)$-th symbol \$. For the new configuration of the Turing machine we proceed with the process described above.

When there are no more symbols to be marked on tape X, we unmark all the symbols and keep one copy only of the identical words existing on tape Z between any pair of symbols \$. In this moment, on the tape Z it is found the encoding of the configuration obtained in an evolutionary step of Γ from the configuration encoded on the tape Y.

We now move to the Communication phase.

Communication:
First, we transform tape Z into $\$ \bullet \$ \bullet \$ \bullet \ldots \bullet \$ \bullet \$$ and mark the leftmost unmarked symbol 2 on tape X. Assume that, after marking such a symbol, there are exactly k marked symbols on the tape X. We place the head of tape W on the $(k+1)$-th symbol \$ on this tape, and the heads of tapes Y and Z on the k-th symbol \$.

In the current state we memorize the way filters are used (which is encoded in the last symbol of $< x_k >$ before #) and read the sets defining the output filter of x_k. Since the filters defined by random-context conditions based on finite sets of symbols, it is easy to check whether or not a word from $C(x_k)$ on tape Z verifies the condition imposed by the output filter. All the words which cannot pass the output filters are marked on tape Z and inserted followed by a bullet symbol on tape Y. After this operation is carried out for all words of $C(x_k)$, we repeat the process marking a new symbol on tape X.

When there are no more symbols to be marked on the tape X we restore the original content of tape X. In this moment, on the tape Z are placed n words representing the encoding of configuration after the last evolutionary step some words in these sets being marked. The marked subwords in every $< C(x) >$ are exactly marked encoding of the words that could not leave the node x. Moreover, tape Y contains n words encoding the sets of words that could not leave the nodes.

Again, we mark the leftmost unmarked symbol 2 on tape X; assume that, after marking such a symbol, there are exactly k marked symbols on the tape X. We place the head of tape W on the $(k+1)$-th symbol \$ on this tape, the head of tape Y on the k-th symbol \$, and that of tape Z on the first symbol \$.

Now we read the sets defining the input filter of x_k. Then we check whether or not the unmarked words from $C(x_1)$, $C(x_2)$, $\ldots$, $C(x_n)$ satisfy the condition imposed by the input filter of x_k. All these words are inserted, followed by a bullet symbol, on tape Y. When the process is complete we go on and mark another symbol on tape X, and resume the process on tape Z. When no symbol on tape X can be marked anymore, restore the initial content of this tape, keep one copy only of the identical words existing on tape Y between any pair of symbols \$, and unmark all symbols on tape Z. In this moment, on the tape Y it is found the encoding of the configuration obtained in a communication step of Γ from the configuration encoded on the tape Z.

Acceptance:
If the configuration associated with x_O after an evolutionary or communication step is not empty, the computation stops and our machine accepts the input word. Otherwise, if, before an evolutionary or communication step, the words from tapes Y and Z are identical, the computation also stops, but the input word is not accepted.

Clearly, all the operations above can be actually implemented formally by a Turing Machine. We obtain that T'_U implements the desired behavior. From a classical result, it follows that there exist a 1-tape Turing Machine, T_U, with the same behavior as T'_U. This concludes the proof of the theorem. □

The final step of the construction of a universal AHNEP is based on the following theorem proved in [11]:

Theorem 4.2 [11] *For any Turing machine M recognizing a language L there exists an AHNEP Γ accepting the same language L.*

Moreover, from the proof of the above theorem it follows that the AHNEP Γ halts on exactly the same input words as M does. Consequently, we can construct an AHNEP Γ_U that implements the same behavior as T_U which is the universal AHNEP. Therefore, we have shown:

Theorem 4.3 *There exists an AHNEP Γ_U, with the input alphabet A, satisfying the following conditions on any input $<\Gamma><w>$, where Γ is an arbitrary AHNEP and w is a word over the input alphabet of Γ:*

- *Γ_U halts on the input $<\Gamma><w>$ if and only if Γ halts on the input w.*
- *$<\Gamma><w>$ is accepted by Γ_U if and only if w is accepted by Γ.*

References

[1] J. Castellanos, C. Martin-Vide, V. Mitrana, J. Sempere, Solving NP-complete problems with networks of evolutionary processors, *IWANN 2001* (J. Mira, A. Prieto, eds.), LNCS 2084 (2001), 621–628.

[2] J. Castellanos, C. Martin-Vide, V. Mitrana, J. Sempere, Networks of evolutionary processors, *Acta Informatica* 39(2003), 517-529.

[3] J. Castellanos, P. Leupold, V. Mitrana, Descriptional and computational complexity aspects of hybrid networks of evolutionary processors. *Theoret. Comput. Sci.* 330, 2(2005), 205-220.

[4] E. Csuhaj-Varjú, J. Dassow, J. Kelemen, G. Păun, *Grammar Systems.* Gordon and Breach, 1993.

[5] E. Csuhaj-Varjú, A. Salomaa, Networks of parallel language processors. *New Trends in Formal Languages* (G. Păun, A. Salomaa, eds.), LNCS 1218, Springer Verlag, 1997, 299 - 318.

[6] E. Csuhaj-Varjú, A. Salomaa, Networks of Watson-Crick D0L systems. Proc. *International Conference Words, Languages & Combinatorics III* (M. Ito, T. Imaoka, eds.), World Scientific, Singapore, 2003, 134 - 150.

[7] L. Errico, C. Jesshope, Towards a new architecture for symbolic processing. *Artificial Intelligence and Information-Control Systems of Robots '94* (I. Plander, ed.), World Scientific, Singapore, 1994, 31 - 40.

[8] S. E. Fahlman, G. E. Hinton, T. J. Seijnowski, Massively parallel architectures for AI: NETL, THISTLE and Boltzmann machines. Proc. *AAAI National Conf. on AI*, William Kaufman, Los Altos, 1983, 109 - 113.

[9] W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, 1985.

[10] F. Manea, C. Martin-Vide, V. Mitrana, Solving 3CNF-SAT and HPP in Linear Time Using WWW, *MCU 2004*, LNCS 3354 (2005), 269–280.

[11] M. Margenstern, V. Mitrana, M. Perez-Jimenez, Accepting hybrid networks of evolutionary processors, *Pre-proc. DNA 10*, 2004, 107–117.

[12] C. Martin-Vide, V. Mitrana, M. Perez-Jimenez, F. Sancho-Caparrini, Hybrid networks of evolutionary processors, *Proc. of GECCO 2003*, LNCS 2723 (2003), 401 - 412.

[13] G. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Berlin, 1998.

[14] G. Păun, *Membrane Computing. An Introduction.* Springer Verlag, Berlin, 2002.

[15] D. Sankoff, G. Leduc, N. Antoine, B. Paquin, B. F. Lang, R. Cedergren, Gene order comparisons for phylogenetic inference: Evolution of the mitochondrial genome. *Proc. Natl. Acad. Sci. USA*, 89(1992), 6575 - 6579.

Token-Passing Nets: Call-by-Need for Free

François-Régis Sinot [1,2]

LIX, École Polytechnique, 91128 Palaiseau, France

Abstract

Recently, encodings in interaction nets of the call-by-name and call-by-value strategies of the λ-calculus have been proposed. The purpose of these encodings was to bridge the gap between interaction nets and traditional abstract machines, which are both used to provide lower-level specifications of strategies of the λ-calculus, but in radically different ways. The strength of these encodings is their simplicity, which comes from the simple idea of introducing an explicit syntactic object to represent the flow of evaluation. In particular, no artifact to represent boxes is needed. However, these encodings purposefully follow as closely as possible the implemented strategies, call-by-name and call-by-value, hence do not benefit from the ability of interaction nets to easily represent sharing. The aim of this note is to show that sharing can indeed be achieved without adding any structure. We thus present the call-by-need strategy following the same philosophy, which is indeed not any more complicated than call-by-name. This continues the task of bridging the gap between interaction nets and abstract machines, thus pushing forward a more uniform framework for implementations of the λ-calculus.

1 Introduction

Interaction nets (INs) [4] are a graphical paradigm of computation that makes all the steps in a computation explicit and expressed uniformly. In particular, sharing is possible (as opposed to terms) and is dealt with explicitly (as opposed to termgraphs). Locality and strong confluence of reduction also make interaction nets well-suited as an intermediate formalism in the implementation of programming languages. However, despite their qualities and their popularity among theoreticians, it is sad to notice that they are less widely used by implementors of real-world programming languages. While it is difficult to say why, it is very probable that works such as those on optimal reduction have led some to think of interaction nets as a theoreticians-only

[1] Projet Logical, Pôle Commun de Recherche en Informatique du plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.
[2] Email: frs@lix.polytechnique.fr

tool. It might thus be worth it to bridge the gap between interaction nets and traditional tools such as abstract machines.

Interaction nets have been used for the implementation of optimal reduction [5,3,2] and for other efficient (non-optimal) implementations of the λ-calculus [7,8]. All of the above encodings of the λ-calculus have in common that a β-redex is always translated to an active pair (i.e. a redex in interaction nets), hence, paradoxically, while all reductions are equivalent, there is still the need for an external interpreter to find the redexes and manage them, which is typically implemented by maintaining a stack of redexes [9]. The fact that different β-reductions may be interleaved also has the nasty consequence that the encodings need to simulate *boxes* in a more or less complex and costly way.

On the contrary, in [10] as well as in this paper, the encodings stick as closely as possible to traditional strategies. In particular, they ensure that essentially only one reduction is possible at a time. This is simpler to implement and avoids the need for boxes. These encodings are based on the idea of a single evaluation token, which is a standard interaction agent, walking through the term as an evaluation function would do. They are thus very natural and easy to understand.

In [10] the encodings are in standard interaction nets, featuring a standard agent called the evaluation token. Reductions are essentially triggered by the evaluation token, whose unicity is guaranteed, hence reduction is essentially deterministic. On the other hand, some restrictions of interaction nets are tailored to ensure strong confluence, and thus are no longer necessary if all reductions are triggered by a unique token.

In this paper, we want to apply the same technology to the call-by-need strategy. This will force us to abandon the restriction to Lafont's interaction nets, as explained below, and adopt Alexiev's formalism of interactions nets with multiple principal ports [1], or simply nets. Still, since reduction is directed by a unique token, evaluation is fully deterministic.

Call-by-need was introduced by Wadsworth [11]. The idea is relatively intuitive: a subterm should be evaluated only if it is needed, and if so, it should be evaluated only once. And so is the original formulation, in terms of graph rewriting. There have been several attempts to formalise this idea in different ways, sometimes with some loss of intuitions. In contrast with these approaches, we present another graph-based formalisation, making explicit at the object-level the rewrite strategy used in [11]. Our presentation is thus more primitive than other works; we therefore prefer to refer to the intuitive definition of Wadsworth (in terms of graphs) rather than prove properties with respect to any of the latter formalisms, contrasting with the approach of [10]. The encoding will thus be direct. It can be seen as a graph-based abstract machine, which is still strikingly close to a term representation. The encoding of terms is almost the same as for call-by-name and reduction rules are not much more complicated, thus sharing is indeed obtained "for free".

The rest of this paper is structured as follows. In Section 2, we recall some background on net rewriting. We give the encoding of terms in Section 3, the evaluation rules in Section 4 and a few properties in Section 5. We conclude in Section 6.

2 Nets

Nets have been introduced in [1] under the name interaction nets with multiple principal ports. They are roughly interaction nets [4], but where the agents are allowed to have any number of principal ports, instead of just one.

A net is a graph (not necessarily connected), whose edges are binary (i.e. they are not hyperedges) and unlabelled and whose vertices are labelled, have a fixed arity and are called *agents*. The attachments points of agents are called *ports*. Each agent has a fixed number of *principal ports*, depicted by an arrow; the other ports are called *auxiliary*. The edges of the graph connect agents together at the ports such that there is only one edge at every port. The ports of an agent that are not connected to another agent are called free. Nets without agents (i.e. only with edges) are allowed and are called wirings; the extremes of wirings are also called free ports.

Two agents connected by principal ports on both sides form an *active pair* or *redex*. A net rewrite rule may replace an *active pair*, by an arbitrary net, provided that all free ports are preserved during reduction. This ensures that reduction is local, i.e. only the part of the net involved in the rewrite is modified.

We use the notation $\Longrightarrow$ for the one-step reduction relation and $\Longrightarrow^*$ for its transitive and reflexive closure. If a net does not contain any active pairs then we say that it is in normal form. Note that in general, no property of confluence can be expected from such a system.

If all agents in a net system have exactly one principal port and at most one rule can be applied to any active pair, then it is a system of interaction nets [4]. In this case, reduction is strongly confluent.

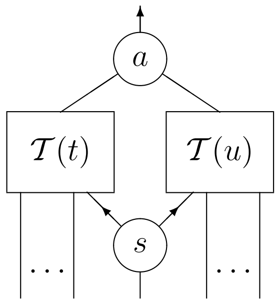
3 Encoding of Terms

The translation $\mathcal{T}(\cdot)$ of λ-terms into interaction nets is very natural. We basically represent terms by their syntax tree, where we group together several occurrences of the same variable by agents s (corresponding to sharing) and bind them to their corresponding λ node (this is sometimes referred to as a *backpointer*). The nodes for abstraction and application are agents λ and a with three ports; their principal port is directed towards the root of the term. Note that in traditional encodings, the application agent looks towards its left, so that interaction with an abstraction is always possible. Here, on the contrary, terms are translated to *packages* [6] and in particular there will be no spontaneous reduction, something will have to trigger them: the *evaluation* *token*.

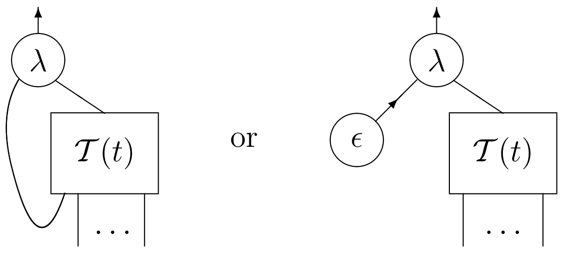
This is essentially the same encoding as in [10]. The only difference is that agents s representing sharing have two principal ports oriented upwards, so that they will not perform any copy before the evaluation token reaches them. In [10], these agents have one principal port oriented downwards, so that they can readily perform copying right after a β-reduction. These are in fact the agents c introduced below.

Variables. We consider only closed terms (open terms can be dealt with as in [10]), hence variables are not translated as such. They will simply be represented by edges between their binding λ and their grouped occurrence in the body of the abstraction, as explained below.

Application. The translation $\mathcal{T}(t\ u)$ of an application $t\ u$ is simply an interaction agent a whose principal port points at the root, and with $\mathcal{T}(t)$ and $\mathcal{T}(u)$ linked to its two auxiliary ports. If t and u share common free variables, then s agents (representing sharing) collect these together pairwise so that a single occurrence of each free variable occurs amongst the free edges (only one such copy is represented on the figure). Note that s agents have two principal ports oriented upwards, so that copy will not begin before an agent (the evaluation token) arrives from the top. These will be the only agents of the system with more than one principal port.



Abstraction. If $\lambda x.t$ is an abstraction, $\mathcal{T}(\lambda x.t)$ is obtained by introducing an agent λ, and simply linking its right auxiliary port to $\mathcal{T}(t)$ and its left one to the unique wire corresponding to x in $\mathcal{T}(t)$. If x does not appear in t, then the left port of the agent λ is linked to an agent ϵ.



To sum up, we represent λ-terms in a very natural way. In particular, there is no artifact to simulate boxes. Another point worth noticing is that, because of the explicit link between a variable and its binding λ, α-conversion comes from free, as it is often the case in graphical representations of the λ-calculus.

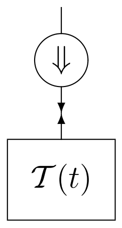
So far, we have only introduced agents λ and a strictly corresponding to the λ-calculus, as well as agents ϵ and s for the explicit resource managements necessary (and desirable: we do not want to hide such important things) in net rewriting. Also remark that the translation of a term has no active pair, hence is in normal form, whatever interaction rules are allowed. Moreover, it has exactly one principal port, at the root.

4 Evaluation by Interaction

The difference between call-by-name and call-by-need is only visible when sharing is involved. Consequently, the part of the encoding that does not deal with it is exactly the same as in [10] (Section 4.1). There is no reason either to change the way copying and erasing are done (Section 4.3); the difference is only on *when* copying should occur, i.e. when we should activate a sharing agent into a copying agent (Section 4.2).

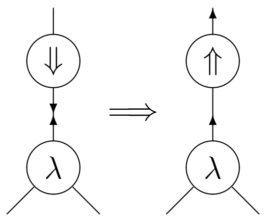
4.1 Linear part

We introduce two new unary agents $\Downarrow$ and $\Uparrow$. To start the evaluation, we simply build the following net, that we will denote $\Downarrow\mathcal{T}(t)$.

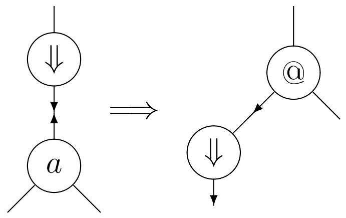


$\Uparrow\mathcal{T}(t)$ will be a net built in the same way, but with a $\Uparrow$ agent instead, with its principal port directed towards the root. In particular, $\Uparrow\mathcal{T}(t)$ is always a net in normal form. Relatively good intuition is carried by calling $\Downarrow$ "eval" and $\Uparrow$ "return".

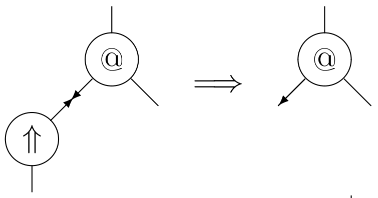
As for call-by-name, when we evaluate a term beginning by a λ, we should return that term:

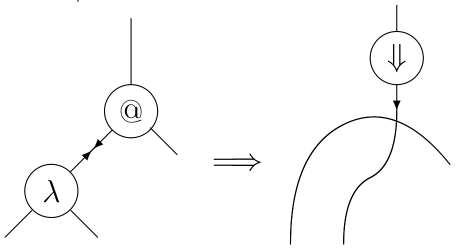


To evaluate a term whose head symbol is an application, we should first evaluate its left subterm. In other words, we should move the evaluation token to the left of the application. We also rename the agent a to @, which is still representing an application, but with its principal port no longer pointing to the root but to the left, so that interaction will be possible when the evaluation token returns.



Finally, when the agent ⇑ returns from a successful evaluation to a @, then we know for sure that there is a λ just below the ⇑, and a β-reduction should be performed. Due to the restriction to binary interaction, this takes two steps:

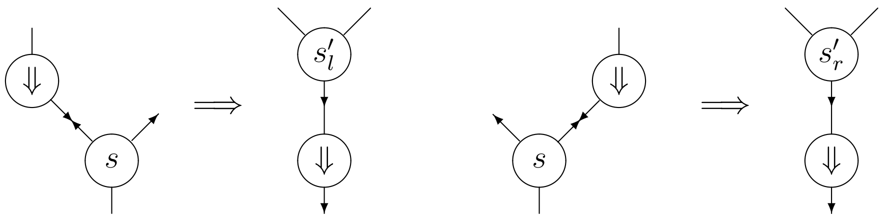




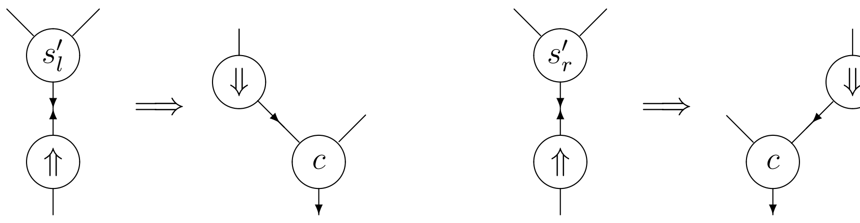
We link the variable port of the λ to the argument port of the @, which initiates the substitution; and we pursue evaluation on the body of the abstraction. This is the core of the interaction net machinery for linear λ-terms; it is the same as for call-by-name.

4.2 Sharing

Sharing is represented by agents s. When the evaluation token reaches an agent s that means that evaluation of the shared subterm is required. This is done very simply by moving the evaluation token down to the shared subterm. The agent s is then renamed to an agent s' looking down, so that interaction will be possible when the token returns. We also have to remember if the token comes from the left or from the right of the agent s, in order to resume evaluation from the same position. This is why we in fact introduce agents s'_l and s'_r.



When the token returns to an agent s', then we initiate the copying process with a c agent and resume evaluation from the original position (left or right, as remembered in the agent).

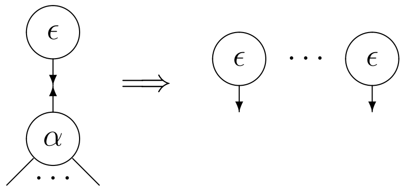


These two rules produce ⇓ agents, but they could equivalently produce ⇑ agents: we know that the agent under the initial ⇑ is a λ (possibly after some copying), so the c agent in the right-hand side of the rule will indeed copy that λ, and the ⇓ agent will be changed into a ⇑ agent. The interest of choosing to produce a ⇓ instead of a ⇑ is to force copying at that point instead of delaying it further.

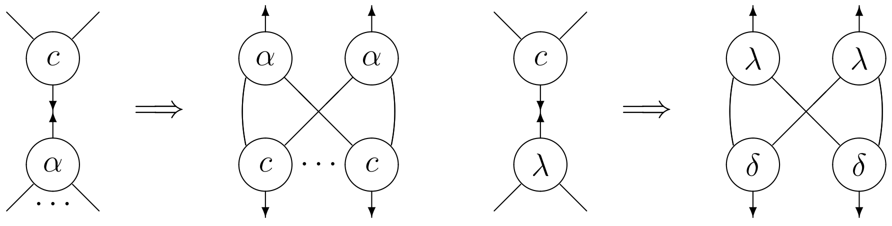
Remark 4.1 *Because of our encoding and because we follow a normal order strategy (we always go left in an application), it will often be the case that the token reaches a s on its left port. However, this is not always true, for instance in the term $(\lambda x.(\lambda y.\lambda z.z\,y)\,x\,x)\,(\lambda u.u)$. This is why we have to abandon the restriction to interaction nets.*

4.3 Copying, erasing

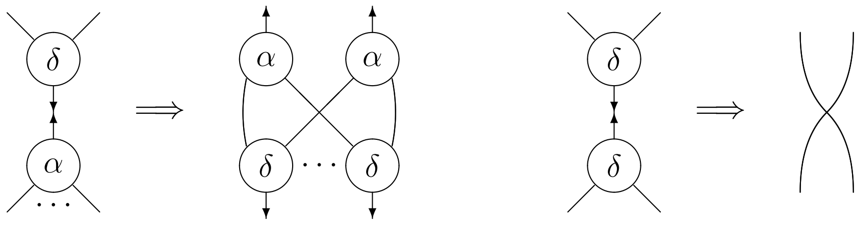
Copying and erasing are done in a classical way, by agents ϵ, c and δ. The auxiliary agent δ is introduced to duplicate abstractions, as explained below. The agent ϵ erases any agent and propagates according to the following schema (where α represents any agent except s):



In general, the agent c duplicates any agent it meets. To duplicate an abstraction, we need an auxiliary agent δ that will also duplicate any agent, but will stop the copy when it meets another δ agent. Note that an agent c will thus never interact with another agent c. Here, α represents any agent except λ and s.



The agent δ duplicates any agent, except itself. If it interacts with itself, it just annihilates. Here, α represents any agent except δ and s.



A c agent always tries to copy an evaluated subnet, so agents c and s never meet (because the s would have been activated first). However, we may have to copy an agent s inside an abstraction using a δ. We know that if the agent s was in fact a c, there would be no problem. Hence the simplest (although not the most efficient) option when an agent other than the evaluation token meets an agent s is to simply activate the agent s into an agent s'' that behaves similarly to agents c, except that each reduction restores it into an agent s (instead of s''). This brings us back to the simpler framework without agents s and helps to preserve strong confluence (see below). The details (and pictures) are omitted. The sharing obtained is exactly call-by-need in the usual sense (e.g. like in Haskell): no reduction is shared inside an abstraction. A finer tuning of this issue could lead to an implementation of the *fully lazy* strategy [11], but could also make the issue of matching δ's harder.

5 Properties

Definition 5.1 *A net N is said to be valid if there exists a λ-term t such that $\Downarrow\mathcal{T}(t) \Longrightarrow^* N$.*

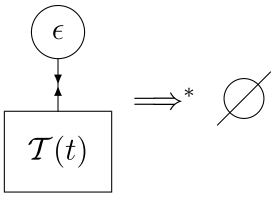
In a valid net, there may be several redexes involving agents c, δ or ϵ, however, we have the following result.

Proposition 5.2 *In a valid net there is exactly one occurrence of $\Downarrow$, $\Uparrow$ or of a $\lambda{-}@$ active pair.*

Proof. By induction, using the rules. □

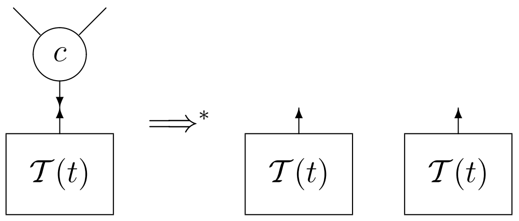
Classical results on packages [6] allow to state the two following properties:

Proposition 5.3 • *If t is a closed λ-term, then:*



(where the right-hand side of the rule denotes the empty net).

- *If t is a closed λ-term, then:*



Rules can be partitioned into *evaluation rules* involving a token $\Downarrow$ or $\Uparrow$, or an active pair $\lambda - @$, and *administrative rules* involving agents c, δ, ϵ or s''. Remark that it is indeed a partition.

Proposition 5.4 *Reduction is strongly confluent on valid nets, i.e. if M is a valid net such that $M \Longrightarrow P$ and $M \Longrightarrow Q$ (with $P \neq Q$), then there exists a net N such that $P \Longrightarrow N$ and $Q \Longrightarrow N$.*

Proof. In a valid net, there is at most one evaluation rule applicable (by Proposition 5.2), so at least one of the reduction is administrative. But notice that there is no overlap between evaluation and administrative rules, so in this case, the reductions are independent and the diverging pair can be joined by applying the other rule. The remaining case thus involves two administrative rules. Again, if they are applied at different places, the pair is easy to join. The only remaining cases involve an agent s and any agent among δ, ϵ, s'' on both its principal ports. In these cases, the agent s is simply transformed into an agent s'', and indeed $P = Q$. This completes the proof. □

Proposition 5.5 $\Downarrow\mathcal{T}(t) \Longrightarrow^* \Uparrow\mathcal{T}(v)$ *if and only if t reduces to v by the call-by-need strategy.*

Proof. There are two points:

- if the issue of sharing is ignored, the strategy followed is call-by-name (see [10]), hence by classical results, we do not evaluate a subterm unless it is needed;
- when a shared subterm is needed, the rules for sharing are clear: the subterm is evaluated first, and copied only afterwards (thanks to Proposition 5.3).□

6 Conclusion

We have presented a simple approach to express call-by-need in net rewriting. The approach is so simple that it is indeed a good alternative to working with terms. It is obtained from an encoding of call-by-name without adding any extra structure, and without much complication. The framework of interaction nets had to be abandoned, but no property is lost. This can be seen both as a simple formalisation of Wadsworth original formulation, and as the basis for a graph-based abstract machine.

References

[1] Alexiev, V., "Non-deterministic Interaction Nets," Ph.D. thesis, University of Alberta (1999).

[2] Asperti, A., C. Giovannetti and A. Naletto, *The Bologna optimal higher-order machine*, Journal of Functional Programming **6** (1996), pp. 763–810.

[3] Gonthier, G., M. Abadi and J.-J. Lévy, *The geometry of optimal lambda reduction*, in: *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)* (1992), pp. 15–26.

[4] Lafont, Y., *Interaction nets*, in: *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)* (1990), pp. 95–108.

[5] Lamping, J., *An algorithm for optimal lambda calculus reduction*, in: *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)* (1990), pp. 16–30.

[6] Lippi, S., "Théorie et pratique des réseaux d'interaction," Ph.D. thesis, Université de la Méditerranée (2002).

[7] Mackie, I., *YALE: Yet another lambda evaluator based on interaction nets*, in: *Proceedings of the 3rd International Conference on Functional Programming (ICFP'98)* (1998), pp. 117–128.

[8] Mackie, I., *Efficient λ-evaluation with interaction nets*, in: V. van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, Lecture Notes in Computer Science **3091** (2004), pp. 155–169.

[9] Pinto, J. S., *Sequential and concurrent abstract machines for interaction nets*, in: J. Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, Lecture Notes in Computer Science **1784** (2000), pp. 267–282.

[10] Sinot, F.-R., *Call-by-name and call-by-value as token-passing interaction nets*, in: *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA'05)*, Lecture Notes in Computer Science **3461** (2005), pp. 386–400.

[11] Wadsworth, C. P., "Semantics and Pragmatics of the Lambda-Calculus," Ph.D. thesis, Oxford University (1971).

Supporting Function Calls within PELCR

Antonio Cosentino [1]

Dipartimento di Informatica, Sistemi e Produzione, Università degli Studi di Roma "Tor Vergata", Viale del Politecnico 1, Rome, Italy

Marco Pedicini [2]

Istituto per le Applicazioni del Calcolo "M. Picone", CNR, Viale del Policlinico 137, Rome, Italy.

Francesco Quaglia [3]

Dipartimento di Informatica e Sistemistica, Università degli Studi di Roma "La Sapienza", Via Salaria 113, Rome, Italy.

Abstract

In [10,11], PELCR has been introduced as an implementation derived from Geometry of Interaction in order to perform virtual reduction on parallel/distributed computing systems.

In this paper we provide an extension of PELCR with computational effects based on directed virtual reduction [2], namely a restriction of virtual reduction [3], which is a particular way to compute the Geometry of Interaction [5] in analogy with Lamping's optimal reduction, [6]. Moreover, the proposed solution preserves scalability of the parallelism arising from local and asynchronous reduction as studied in [11].

Key words: Functional Programming, Optimal Reduction, Linear Logic, Geometry of Interaction, Virtual Reduction, Parallel Implementation.

[1] e-mail: `antocos@tiscali.it`

[2] e-mail: `marco@iac.cnr.it`

[3] e-mail: `quaglia@dis.uniroma1.it`

[4] Partially supported by CNR/CNRS cooperation project n. 16251 "Interaction and Complexity", 2004-2005.

1 Introduction

PELCR (Parallel Environment for optimal Lambda Calculus Reduction) is a software package supporting optimal lambda calculus reduction on parallel/distributed computing systems. It is devised as an interpreter for pure lambda calculus (complete) reduction, whose development relies on the Geometry of Interaction [5] and successive results in the field of functional programming and linear logic [3], which have shown that the reduction of lambda terms can be mapped onto a graph rewriting technique known as Directed Virtual Reduction (DVR), see [2]. Specifically, PELCR implements a particular strategy for DVR, referred to as Half Combustion (HC), see [10,11], which permits great exploitation of parallelism by allowing the composition between two edges coincident on a same node of the graph as soon as these edges become available to the processor hosting that node. A set of optimisations are also implemented within PELCR allowing a reduction of the communication overhead, and a fair policy for distributing dynamically originated load (i.e. new nodes and edges generated during the reduction) among processors.

Although pure lambda calculus has a variety of applications, many functional programming languages tend to deviate from it in order to become more attractive and effective for programmers, and to enrol the use of non-functional constructs. With respect to this point, let us cite the most diffused examples of functional languages, namely ocaml and haskell, both having additional base types and facilities for explicit interactions with the underlying operating system.

In this paper we show how it is possible to support similar extensions in PELCR, without preventing the possibility to exploit parallelism (and hence to achieve run-time effectiveness) arising from Geometry of Interaction. Our starting point is Mackie's work on the implementation of the Geometry of Interaction where the author extends Girard's algebra with extra generators for natural numbers and for the successor function. The new generators form an equational theory which defines a particular abstract data type.

By generalising Mackie's approach, we extend the applicability of PELCR as an environment for the execution of functional and imperative languages through automatic and adaptive distribution, where the functional parts take into account functional dependencies and where external functions, let say x*-functions* for short, make calls to imperative code implementing parts less prone to be specified in a functional language. Also, compared to Mackie's original proposal, which deals with functions with a single parameter, we address the case of functions with multiple parameters.

The rest of this paper is structured as follows. In Section 2, we provide the theoretical framework for the previously mentioned extension. In Section 3, we present the interpretation of extended lambda calculus in dynamic graphs with x-functions to be executed on top of directed virtual reduction. How to support the extension within PELCR is described in Section 4. In Section 5 we give an example of code using x-functions and report experimental data related to the course of execution time while varying the number of used processors.

2 Geometry of Interaction and Extended $\mathsf{L}^\star$

The extension we provide is obtained following the approach introduced by Mackie in [7] and then expanded by Pinto in [13]. This technique can be summarised with the addition of generators in $\mathsf{L}^\star$ with computational effects consisting of data to be manipulated and executables to be evaluated when interactions occur. This work explores this direction, and in fact we map those generators onto data structures and functions defined in external libraries implemented in C language.

Let us recall Pinto's example, we consider a new generator S representing the successor function $S : \mathbb{N} \to \mathbb{N}$ and a generator n for every integer $n \in \mathbb{N}$. Then a pair of specific interaction equations are given

$$\mathsf{S}^\star \mathsf{n} = (\mathsf{n}+1)\mathsf{S}^\star \tag{1}$$
$$\mathsf{S}^\star \mathsf{S} = 1 \tag{2}$$

and added to the algebra $\mathsf{L}^\star$. Note that, while the evaluation of Equation (1) is presented as a rewriting, it has attached a computational task to compute the result of $S(n)$. In fact, these rules are better understood in the following terms: we add to $\mathsf{L}^\star$ a generator S for the function and a generator N which stands for a ground type object, a natural number in this case. Any generator added to the algebra has a corresponding allocated memory space:

- to store its state n in the case of an object of type N, denoted by $\mathsf{N} : n$
- to store a program address p in the case of a function, denoted by $\mathsf{S} : p$.

So now Equations (1) and (2) can be rephrased as

$$(\mathsf{S} : p)^\star(\mathsf{N} : n) = (\mathsf{N} : p(n))(\mathsf{S} : p)^\star, \tag{3}$$
$$(\mathsf{S} : p)^\star(\mathsf{S} : p) = 1. \tag{4}$$

Note that $p(n)$ is obtained by calling the function with address p with argument stored in N, and by storing the result in the space allocated for N.

We have two kinds of problems with this approach. First, we need to consider how to extend such an approach to functions with more than one argument: $f : A_1 \times A_2 \times \ldots A_k \to B$. Second, we have to consider generators for partially evaluated functions since, as in currification, the generator $\mathsf{F} \in \mathsf{L}^\star$ associated with f, interacts with its arguments one by one.

In order to give the general form of equations (1), and (2), we introduce a new family of generators, let say x-generators, with identifier i and constant lift 1, denoted by x_i; from the algebraic point of view x_i behaves like exponential generators of lift 1, we then specify the computational task associated with any generator, i.e. its computational effect.

For any x_i we have a type $\tau(i)$ and a $state(i)$, the state stores information

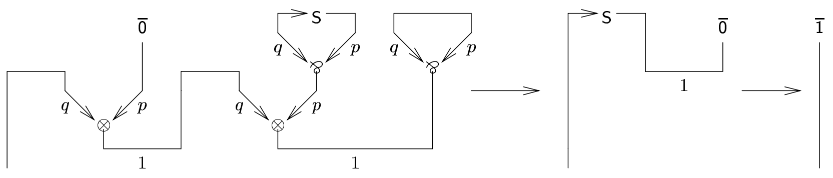


Fig. 1. Pinto's reduction of a term involving successor $((\lambda x.x)\lambda a.\mathtt{S}(a)\,\mathtt{0})$

on the type of the computational task, and on its evaluation status; essentially we have two classes of evaluation states:

- in case of *data*, denoted by $x_i : a$, x_i type is a ground type and its evaluation state is the stored value a;
- in case of *functions*, denoted by $x_i : (p, v)$, x_i type is a functional type and its evaluation state is given by a function pointer p and by an ordered list of values v of length strictly lesser than the arity of the function, note that the vector may possibly be the empty one.

For the sake of simplicity we suppose a unique ground type σ and the arrow type constructor, so the set S of types is $S := \sigma|\sigma \to S$. We denote $\sigma \to (\sigma \to \dots (\sigma \to \sigma) \dots)$ by $\sigma^n \to \sigma$.

Definition 2.1 For any x_i we have that $state(i) = \langle \tau, p, v\rangle$ where its type $\tau = \sigma^n \to \sigma \in S$, p is a reference to a function code, and $v = (a_1, a_2, \dots, a_m)$ with $0 \le m < n$. The case of data is treated as a particular case where $state(i) = \langle \sigma, \mathtt{null}, (a_1)\rangle$.

We now introduce the definition of the Geometry of Interaction algebra extended with x-generators. Interaction rules for x-generators are defined only for well typed data/function; all the other types of interaction are undefined. In Definition 2.2, and in particular in Equation (6) in Definition 2.3 below, we suppose that x_i has a functional type and x_j a ground data type, moreover we denote by s_i (respectively by s_j) its state $state(x_i) = \langle \sigma^n \to \sigma, p, (a_1, \dots, a_m)\rangle$ (resp. $state(x_j) = \langle \sigma, \mathtt{null}, b\rangle$):

Definition 2.2 For any pair of x-generators x_i and x_j the $\mathtt{eval}$ function acts on the respective states s_i and s_j as follows:

$$\mathtt{eval}(s_i, s_j) = \begin{cases} \langle \tau(i), p, ()\rangle & m = n-1, \\ \langle \tau(i), p, (a_1, \dots, a_m, b)\rangle & m < n-1, \end{cases}$$

and

$$\mathtt{eval}(s_j, s_i) = \langle \gamma, \mathtt{null}, p(a_1, \dots a_{n-1}, b)\rangle,$$

if $\tau(i) = \sigma_1, \dots, \sigma_n \to \gamma$.

The next definition extends the usual presentation of Girard's dynamic algebra with interactions corresponding to the evaluation of the computational effects associated with x-generators:

Definition 2.3 The extended monoid $\mathsf{L}^\star$ of the Geometry of Interaction is the free monoid with a morphism $!(.)$, an involution $(.)^\star$ and a zero, generated by p, q, a family $W = (w_i)_i$ of exponential generators, and a family $X = (x_i)_i$ of x-generators such that for any $u \in \mathsf{L}^\star$:

$$a^\star b = \delta_{ab} \qquad \text{for } a, b = p, q, w_i, \tag{5}$$

$$(x_i : s_i)^\star (x_j : s_j) = \begin{cases} (x_i : \mathtt{eval}(s_i, s_j))^\star & \text{if } i \neq j \text{ and } m = n-1, \\ (x_j : \mathtt{eval}(s_j, s_i))(x_i : \mathtt{eval}(s_i, s_j))^\star & \text{if } i \neq j \text{ and } m < n-1, \\ 1 & \text{if } i = j. \end{cases} \tag{6}$$

$$!(u)a = a!^{e(a)}(u), \qquad \text{where either } a = w_i \text{ either } a = x_i \tag{7}$$

where δ_{ab} is the Kronecker operator, $e(a)$ is an integer associated with a called the *lift* of a; note that $e(x_i) = 1$ for all i, i is called the *name* of w_i or x_i and we will often write $w_{i,e(i)}$ to explicitly denote the lift of the exponential generator.

Orienting the equations (5-7) from left to right, one gets a rewriting system which is terminating and confluent, provided that x-function calls eventually return. The non-zero normal forms, known as *stable forms*, are the terms $ab^\star$ where a and b are *positive* (i.e., written without *s).

Example 2.4 Let us consider the following interaction:

$$(x_1 : (\sigma^2 \to \sigma, \&\mathtt{ADD}(), ()))^\star (x_2 : (\sigma, \mathtt{null}, 1))(x_2 : (\sigma, \mathtt{null}, 3)),$$

it implicates the functional generator x_1 associated with function $ADD()$ of arity 2 from integers to integers, and reference &ADD(), and the generator x_2 of a ground type for integer; it is reduced in the following way:

$$\begin{aligned} &(x_1 : (\sigma^2 \to \sigma, \&\mathtt{ADD}(), ()))^\star (x_2 : (\sigma, \mathtt{null}, 1))(x_2 : (\sigma, \mathtt{null}, 3)) \to \\ &\quad \to (x_1 : (\sigma^2 \to \sigma, \&\mathtt{ADD}(), (1)))^\star (x_2 : (\sigma, \mathtt{null}, 3)) \to \\ &\quad\quad \to (x_2 : (\sigma, \mathtt{null}, 4))(x_1 : (\sigma^2 \to \sigma, \&\mathtt{ADD}(), ()))^\star. \end{aligned}$$

3 Encoding Extended Lambda Calculus in PELCR

In the previous section, we have introduced the extension of the dynamic algebra, and illustrated how to use it while considering the evaluation of arbitrary arity functions. In this section we sketch out how to fill the gap between the natural extension of term interpretation in the Geometry of Interaction by Mackie and Pinto, in Figure 1, and the analogous interpretation of an arity 2 function, see Figure 2.

We have to trade-off between apparently clashing requirements:

- to have a single execution path weighted with x-generators (see Figure 3.a);
- to map multiple arity functions to nodes with multiple links (see Figure 3.b);

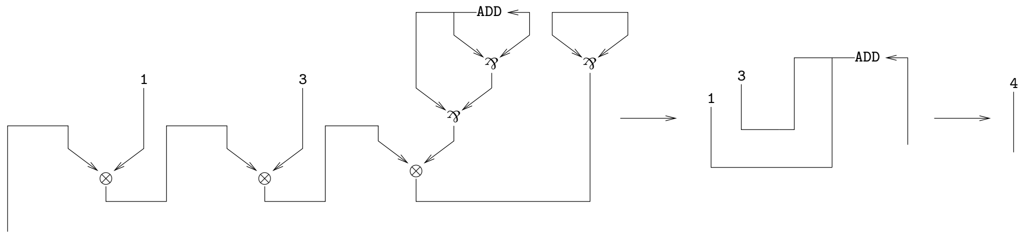


Fig. 2. Reduction of the term corresponding to $((\lambda x.x)\lambda a.\lambda b.\mathtt{ADD}(a,b)\,\mathtt{1}\,\mathtt{3})$

- to execute interactions by using the parallel directed virtual reduction provided by PELCR (see Figure 3.d).

In fact, we have included in PELCR features allowing us to obtain all the above mentioned requirements. The first step is obtained by using the internalisation in DVR of a synchronisation scheme which reduces x-function and arguments interaction to a linear path with the correct configuration, i.e., like in Example 2.4, $F^{\star}A_1\dots A_n$. This approach appeared in a simplified form as a construction to accommodate the conditional term of PCF, in Mackie's work on interaction nets, [8].

Once we have obtained a single execution path, it must be proved not to alter global properties of directed virtual reduction, namely splitness and square-freeness [3], which are the basic properties to prove confluence and termination of VR.

As we show in Section 5, devoted to execution examples, although we are forced to introduce a particular sequential evaluation pattern, the good properties on scalability and speedup of DVR are preserved, and so DVR is capable to exploit the available parallelism coming from functional specification of the program. Let us note that with our technique, we can exploit parallelism emerging from the functional part of the code, but we do not enter in x-functions which are treated as black boxes.

4 Supporting the $\mathsf{L}^{\star}$ Extension within PELCR

In the original version of PELCR, only dealing with pure lambda calculus, the basic operation executed while performing the reduction is the composition of pair of edges coincident on a same node, see Figure 3.d). This is supported through adequate data structures maintained by PELCR to represent the weights of the edges. These data structures are mostly based on string representation of the weights, and those strings constitute the essential part of the payload of each application message exchanged between distinct processes to notify each other of the existence of new edges in the graph originated by DVR steps.

To support the $\mathsf{L}^{\star}$ extension with no substantial modification of the basic mechanisms employed by PELCR for the support to parallelism (i.e. message aggregation and load balancing mechanisms whose benefits on the run-time

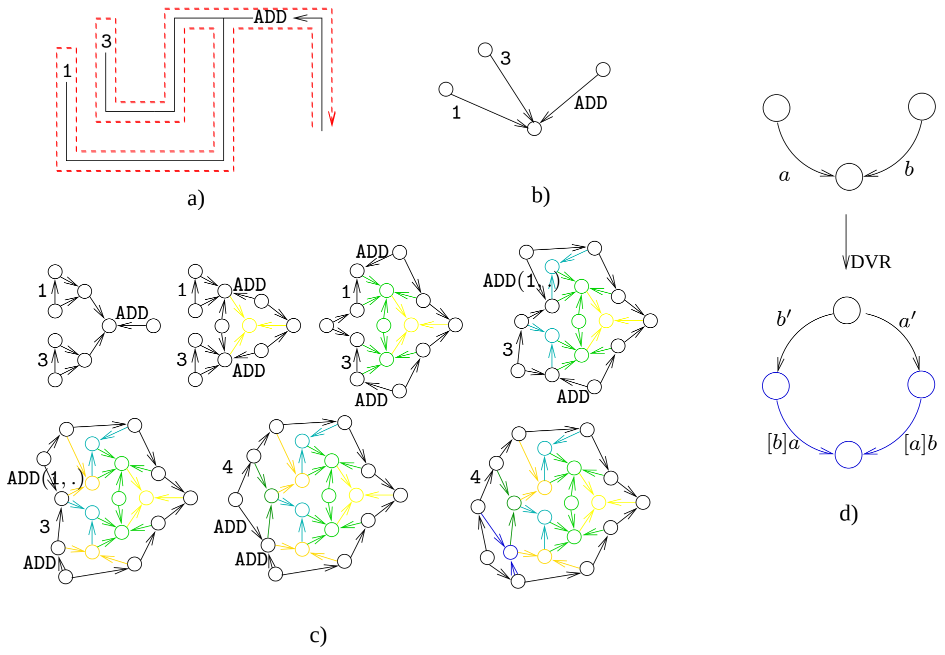


Fig. 3. The reduction of (ADD 1 3)

behaviour have already been extensively tested in the context of pure lambda calculus), and with no variation in the HC strategy, which performs DVR in a parallel effective manner, we have extended the data structures maintained by PELCR in a way allowing a compact representation of x-function needs to be eventually evaluated. The representation is based on a structured data type, namely `function_descriptor`, which implements the state associated to x-generators by maintaining: (i) a function pointer, allowing the retrieve of the code associated with the function when the evaluation needs to be performed, and (ii) a vector of structured entries, namely `parameters`, that, for each parameter to be passed to the function, indicate whether the parameter has already been stored and, in the negative case, stores the corresponding value.

The structured type `function_descriptor` has two additional fields storing the number of arguments for the function, and the number of arguments which have already been stored within the `parameters` vector. Hence, the evaluation of the function takes place as soon as the vector records the whole set of parameters required by the function itself. In accord with Equation (6) in Definition 2.3, this occurs when an edge carrying a `function_descriptor` for a function with n parameters, which already stores $n-1$ of these parameters, is composed through DVR with an edge carrying an additional parameter to be passed to the function (hence the whole set of parameters for the function gets completed).

The type of the return value for the function and the type of the parameters can be also selected by the user through a proper macro, namely `USERTYPE`. However, the current implementation of PELCR cannot cope with parameters of pointer type. This is because, while performing DVR steps, the

```
#uselib "./shared.so"

#def double = \l.\k.\x.((l)k)((l)k)x
#def map = \f.\l.\k.\x.((l)\y.(k)(f)y)x
#def F1 =\x.xfunction(foo)(x)
#def FF = \l.((map)F1)(double)l
#def AA = \k.\x.((k)123)((k)10)((k)3)x

#def five = \f.\x.(f)(f)(f)(f)(f)x
((five)FF)AA
#quit
```

Fig. 4. Benchmark code

`function_descriptor` and the function parameters might migrate across the different processes, so that the function might eventually be evaluated by a process that does not really host the buffer pointed by the parameter passed to the function. This is the same problem appearing in standard Remote Procedure Calls (RPCs), which require proper solutions we plan to introduce within next releases of PELCR.

Actually, the fact that the `function_descriptor` carries a function pointer, instead of the whole code for the function is an optimisation allowing a reduced size for the application messages, especially when dealing with functions having large size of the corresponding modules. On the other hand, this approach imposes the constraint that the function code needs to be available at all the processes so that the function can be evaluated by any of them while performing the reduction.

However, with respect to this point we have implemented the optimisation of avoiding to maintain the code of all the functions possibly involved in the reduction within main memory. Instead, a process dynamically loads the function code whenever required in the form of a Dynamic Linking Library (DLL).

5 Experimental Results

The experiments have been performed by using the code reported in Figure 4, which also gives an idea of the language that can be used within PELCR. One of the main motivations for the use of PELCR is that parallelism comes transparently for the end-user, which writes programs in a mixed functional/imperative language, taking advantage of the parallelism emerging from the functional structure and without having to explicitly use message passing primitives. For this application, we have used an SMP computing system, namely an IBM machine with 16 SP Power3 CPUs. The application works on lists of integers. The list structure is encoded in lambda calculus, whilst integers are 32bit unsigned integers, natively supported by the C compiler.

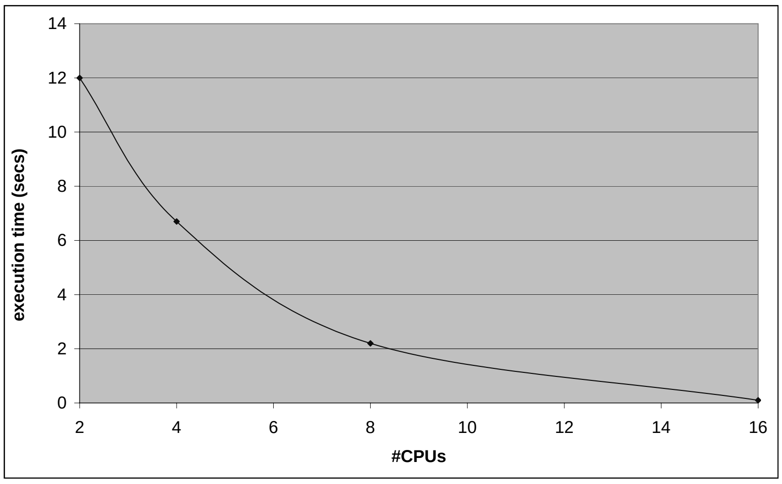


Fig. 5. Execution Time Results.

The library (`shared.so`) provides the x-function and contains the definition of the numerical function `foo` programmed in C language. The functional program takes a list as input, and iterates, by means of an opportune Church numeral, the application of the function `FF` to the initial list `AA`. The lambda term `FF` gets a list l, builds a list obtained by concatenating l with itself and maps the x-function `foo` to the list.

Note that at each iteration the list doubles in size and so the application of the function may rise a good degree of parallelism as proved by the scalability shown in Figure 5. The reported data are very encouraging and essentially confirm good scalability for the execution parallelism achievable by PELCR even when using the computational effects hereby presented.

References

[1] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in TCS.* Cambridge University Press, 1998.

[2] V. Danos, M. Pedicini, and L. Regnier. Directed virtual reductions. In M. Bezem D. van Dalen, editor, LNCS 1258, pages 76–88. EACSL, Springer Verlag, 1997.

[3] V. Danos and L. Regnier. Local and asynchronous beta-reduction (an analysis of Girard's EX-formula). LICS, pages 296–306. IEEE Computer Society Press, 1993.

[4] V. Danos and L. Regnier. Proof nets and the Hilbert space. In J.-Y. Girard,

Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic.* Cambridge University Press, 1995.

[5] J.-Y. Girard. Geometry of interaction 1: Interpretation of system F. In R. Ferro, et al. editors *Logic Colloquium '88*, pages 221–260. North-Holland, 1989.

[6] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. of 17th Annual ACM Symposium on Principles of Programming Languages.* ACM, San Francisco, California, pages 16–30, 1990.

[7] I. Mackie. The geometry of interaction machine. In *POPL*, pages 198–208, 1995.

[8] I. Mackie. YALE: yet another lambda evaluator based on interaction nets In ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming, pages 117–128, ACM, 1998.

[9] M. Pedicini. *Exécution et Programmes.* PhD thesis, Équipe de Logique Mathématiques, Université de Paris 7, 1999.

[10] M. Pedicini and F. Quaglia. A parallel implementation for optimal lambda-calculus reduction PPDP '00: Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming, pages 3–14, ACM, 2000.

[11] M. Pedicini and F. Quaglia. PELCR: Parallel environment for optimal lambda-calculus reduction. *CoRR*, cs.LO/0407055, accepted for publication on TOCL, ACM, 2005.

[12] J. S. Pinto. Parallel implementation models for the lambda-calculus using the Geometry of Interaction. In *TLCA*, pages 385–399, 2001.

[13] J. S. Pinto. *Parallel Implementation with Linear Logic (Applications of Interaction Nets and of the Geometry of Interaction).* PhD thesis, École Polytechnique, 2001.

[14] L. Regnier. *Lambda-Calcul et réseaux.* PhD thesis, Université Paris VII, 1992.

Type Theory and Language Constructs for Objects with States

H. Xu [1] and S. Yu [2]

Department of Computer Science
The University of Western Ontario
London, Ontario, Canada

Abstract

Most class-based Object-Oriented Programming Languages (OOPLs) are strongly typed languages, which means every object created in a program is associated with a type. However, how to add object dynamic behaviors modeled by Harel's statecharts into object types is a challenging task. We propose adding states and state transition functions, which are largely unstated in object type theory, into object type definitions and typing rules. We argue that in order to ensure the correctness of the type system in OOPLs, the state changes of objects during their execution should be properly defined and enforced. As a consequence, we propose our type theory of the τ-calculus, which refines Abadi and Cardelli's ς-calculus, in modeling objects with their dynamic behaviors. In our proposed type theory, we also explain that a subtyping relation between object types should imply the inclusion of their dynamic behaviors. By adding states and state transition functions into object types, we propose modifying programming language constructs for state tracking. We argue that the τ-calculus with modified class definitions can be implemented efficiently in current object-oriented programming languages.

Key words: Object types, states and state transition functions, ς-calculus, τ-calculus, language constructs.

1 Introduction

Type theories for OOPLs have been proposed by many authors. In *A Theory of Objects* [1], Abadi and Cardelli developed their object calculi (ς-calculus) which were stated as a method loosely modeling object-based languages. The type abstraction of object in ς-calculus is conceptually simple and basically

[1] Email: hxu@csd.uwo.ca
[2] Email: syu@csd.uwo.ca

reflects the objects in current OOPLs. However, it lacks necessary expressiveness in certain situations. For example, we may have a class *Transmission* in which the method *void turnoff (void)* is included. An instance (object) of *Transmission* may be in one of the following three states (status): *Neutral*, *Reverse*, and *Forward*, and we assume that the method *turnoff* of the object should be called legally only when the object is in the state *Neutral*. It would be wrong for *turnoff* to be called when the object is in the state *Forward* or *Reverse*. However, neither ς-calculus nor any other current type system can handle such basic problems. In current type systems, any method of an object can be legally called at any time as long as the object is still alive. The state or status of an object is not a consideration of the type systems. Corresponding to the real world, the set of methods of an object that can be called at a certain time may not include all the methods of the object. It depends on what state the object is in, which again depends on the state transitions that define the dynamic behaviors of the object. We argue that these characteristics of object dynamics should be reflected in object type theory. The problems related to states and state transitions of an object may be solved in other ways, but clearly they are part of the type system and better to be solved in the type system.

States and state transitions in object have already been introduced in object-oriented modeling. Statecharts were introduced by David Harel in 1987 [5] and then incorporated into object-oriented modeling methods and languages such as OMT [10] and UML [2] to describe the dynamic behaviors of objects. Finite state machines (FSMs), in a form directly mapped from statecharts, have become a standard model for representing object behaviors.

Several other models also concern states and state machines in types or object types. In a relatively early paper [14], states (not object states) were introduced in programming languages for enhancing software reliability. In [4], the authors proposed language features which allow objects to change class membership dynamically and then developed a type system for their language. However, the states mentioned in the paper are irrelevant to the states and state transitions in statecharts. In [3], a programming model of typestates for objects was developed. However, their type definition contains all of an object's fields and, thus, their object type is an implementation-dependent entity. In [8], the authors proposed that the behavior of objects of a subtype should also satisfies the behavior of supertype objects. But the properties described in the paper do not directly connect to states and transitions. In [9], the author proposed to integrate state machines and OOPLs, in which a state of an object is represented as a set of virtual bindings rather than being a clearly defined entity. In [11], the authors proposed typing objects with states, but focused on formalizing non-uniform concurrent objects. Several papers presented type-based general methods (not methods in a class) for resource usage analysis [7] or resource usage analysis via scoped methods [15]. There are also papers aiming to specify state machines in OOPLs [12] or to initialize some kind of state-oriented programming in implementing hierarchical state machines [13]. However, all these ideas are very different from introducing states and state transition functions into object types.

We propose our τ-calculus for the typed system which comprises formal system fragments. The most fundamental formal system fragments are the object typing and subtyping rules with formally defined states and state transition functions. Our τ-calculus is viewed as an improvement of ς-calculus. States and state transitions are being introduced as an essential part of a class. That is, each class has its own states and state transition functions. We also introduce programming language constructs for implementing states and state transitions. The syntax developed for class is easy to understand and suitable for most of OOPLs. The OOPL type checking system can then include state tracking algorithm and provide a higher degree of program correctness excluding many object behavior errors.

The idea of introducing states and state functions into object types was motivated by David Harel's statecharts. However, they have become different entities. Statecharts have been used for modeling the behaviors of objects normally in the modeling stage and before the programming is done, but objects with states are defined in the programs which are in the implementation stage. More importantly, a statechart models the whole status of an object, but the states of an object defined by a programmer may reflect only a small part of the total behaviors of the object, when states are relevant.

2 Object Types and States

During the life time of an object, the object may change into different states and it may have different behaviors when it is in different states. As we have described in the introduction, the set of methods of an object that can be legally called may be different when the object is in a different state. After a method is called, the object may be transformed into another state. The state transition of an object (a class) can usually be described by a state diagram.

The state diagram of an object or a system is essentially a Deterministic Finite Automaton (DFA) [6]. Hence, the state diagram of an object can be described as $M = (Q, \Sigma, s_0, \delta)$ where Q is the set of states; Σ is the set of methods; s_0 is the starting state - the state when an object is created; δ is the set of transitions defined by the function $\delta(p, a) = q$ for $p, q \in Q$ and $a \in \Sigma$.

Assume that the state diagram of *Dryer* and its abstract DFA model are provided in Figure 1, where $Q = \{0, 1, 2, 3\}$ and 0,1,2,3 represent the named states of *OffSlow*, *OffFast*, *SlowHeating*, and *FastHeating*, respectively; $\Sigma = \{l_1 = chg2Fast,\ l_2 = chg2Slow,\ l_3 = turnOff, l_4 = turnOn\}$ and the labels l_1, l_2, l_3, l_4 are the method names of the object *Dryer* in lexicographical order; the transition function set δ is defined as: $\delta(0, l_4) = 2$, $\delta(0, l_1) = 1$, $\delta(1, l_4) = 3$, $\delta(1, l_2) = 0$, $\delta(2, l_3) = 0$, $\delta(2, l_1) = 3$, $\delta(3, l_3) = 1$, and $\delta(3, l_2) = 2$.

For an object *Dryer* in a state $q \in \{0, 1, 2, 3\}$, if a message $e \in \Sigma$ is received

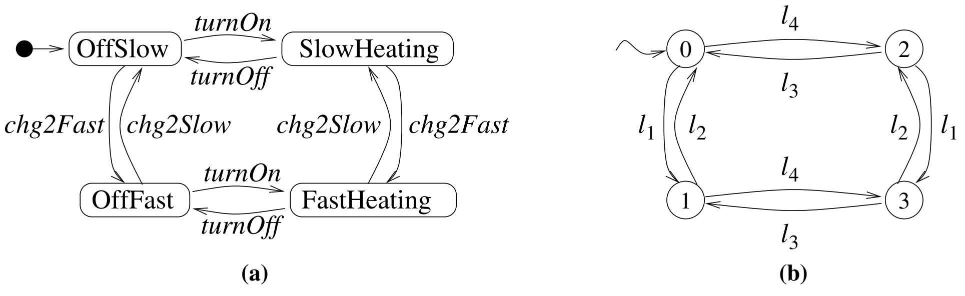


Fig. 1. (a) State diagram of *Dryer* and (b) its abstract DFA model

(a method e is called or invoked) and $\delta(q,e)=p$ is well defined for some $p\in Q$, we consider that the transition to p is a valid transition. Otherwise, $\delta(q,e)$, e.g., $\delta(2,l_4)$, is NOT defined - a *Dryer* can not be turned on while it is in state *SlowHeating*. Then the set denoted by $S_i=\{(q,p)|\ p,q\in Q\ \wedge\ \delta(q,l_i)=p\}$ contains all the proper state changes for a *Dryer* in receiving a message labeled by l_i during its execution. The attempt of state changes for *Dryer* may not be satisfactory when a corresponding transition is not defined although the method is predefined in the class. The state changes of a *Dryer*, resulted from a sequence of method invocations, should be in coherence with the transitions defined in object dynamics. Otherwise, type errors may occur and they should be recognized by the type system.

We observe that an object type is irrelevant to the states and state transition functions of an object according to the rules in ς-calculus. As a result, an algorithm, which is based on the rules for weak reduction in ς-calculus and constitutes an interpreter for ς-terms, cannot exclude type errors of such object misbehavior. For example, the algorithm used for method invocation check in ς-calculus is defined recursively as:

```
$Outcome(a.l_j) \triangleq$
    $let\ \ o{=}Outcome(a)$ .................................... (1)
    $in\ \ if\ o$ has form $[l_i=\varsigma(x_i)b_i\{x_i\}^{\,i\in 1..n}]$ and $j\in 1..n$
        $then\ \ Outcome(b_j\{\!\{o\}\!\})$
        $else\ \ wrong$
```

But in our proposed type theory, a method invocation $a.l_j$ reduces to the result of the substitution of the host object for the *self* (or *this*) parameter in the body of the method named l_j only when a state transition for the host object in the current state exists. In another word, the reduction of $a.l_j \rightarrowtail b_j\{\!\{x_j\leftarrow o\}\!\}$ for l_j $(j\in 1..n)$ is subject to the condition that there exists a valid state change defined in state functions for the object in the current state. Afterwards, the object may transform into another one. On line(1), the statement *let o=Outcome(a)* implies that there exists a valid state transition for object a to be transformed into o while the method l_j is called.

We consider that the state machines composed of states and state transition functions are essential entities in construction of object types. Note that the states used in this paper are clearly different from the concept of typestates [3,14]. Object types are different from object classes. Object types are abstract entities which can be constructed from class definitions but leave out the implementation details.

3 τ-Calculus for Object Types and Class Types

We start with a type system, which is composed of several formal system fragments for object types. We give some important properties of our type system in contrast to what have been defined in ς-calculus. Some fundamental properties of a type system, such as the reduction theorem, are supported by both type theories.

First, we list the syntax fragment for Δ_{Ob} in τ-calculus. This syntax is in fact implicit in the rules of Δ_{Ob} and for the later fragments we do not display the syntax explicitly.

Syntax fragment for Δ_{Ob}

$A, B ::=$	*types*
$[Q, (l_i : B_i) :: S_i^{i \in 1..n}]$	object type (l_i distinct, Q is the set of all states, S_i is the set of valid state transitions for method l_i)
$a, b ::=$	*terms*
$[\, l_i = \tau(x_i : A_i) b_i^{i \in 1..n}]$	object (l_i *distinct*)
$a.l$	method call

Similar to what in ς-calculus, fields are just a special kind of methods and, thus, represented uniformly as methods. The method update feature in OOPLs allows an object dynamically change its behavior in execution. This makes the type of an object hard to trace in a type system. We do not put this feature into discussion in this paper because it has very limited usage in class-based OOPLs.

Definition 3.1 The states and state transitions that are associated with an object are formally described as a triple $A = (Q, \Sigma, S_\Sigma)$:

(i) Q is the set of states. Each node in the state diagram is a state $q \in Q$.

(ii) Σ is the set of methods, i.e., $\Sigma = \{l_i \mid i = 1 \ldots n\}$, where n is the total number of methods.

(iii) $S_\Sigma = \cup_{i=1}^{n} S_i$, where $S_i = \{(p, q) \mid p, q \in Q$ and there is a transition from p by l_i to $q\}$.

In contrast to what appeared in ς-calculus, the enforcement of association between a method and its transition set in object types of τ-calculus is important, but sometimes in a hidden form and underestimated. In some cases, we can model an object with only one state p ($Q = \{p\}$) and all the transitions cause no state change ($S_i = \{(p, p)\}$ for *i=1...n*). In these cases, the states of the objects can be ignored.

Four kinds of judgements are used in fragment Δ_{Ob} in τ-calculus: (1) state set and transition set judgment $E \vdash Q, S$ for $Q \curvearrowright S$, stating that Q is a well-formed set of states and S is a set of state transitions and $S \subseteq \prod_{p,q\in Q}(p,q)$ (i.e., $S \subseteq Q \times Q$) for $q, p \in Q$ is well defined in E, (2) a type judgement $E \vdash B$, stating that B is a well-formed type in the environment E, (3) a value type judgement $E \vdash (b : B) :: S$, stating that b has type B which is bound by state transition set S in E, and (4) state activation correctness judgement $E \vdash a@q \in Q$ and $\exists(q,p) \in S$, stating that an object a is in the state q and there exists a state transition (q,p) for a is a well-formed environment in E.

Δ_{Ob} (τ-calculus):

- Type Object
  $$\frac{E \vdash Q, B_i, S_i \ \ for\ Q{\curvearrowright}S_i \ \ \forall i{\in}1..n}{E \vdash [\,Q,\ (l_i{:}B_i){::}S_i^{\,i\in 1..n}]} \text{ where } Q{\curvearrowright}S_i \text{ means } S_i \subseteq \textstyle\prod_{p,q\in Q}(p,q)$$
- Val Object
  $$\frac{E, a{:}A \vdash (b_i{:}B_i){::}S_i \ \ \forall i{\in}1..n}{E \vdash [\,l_i{=}\tau(a{:}A)\, b_i^{\,i\in 1..n}]{:}A} \text{ where } A \equiv [\,Q,\ (l_i : B_i) :: S_i^{\,i\in 1..n}]$$
- Val Select
  $$\frac{E \vdash a{:}[\,Q,\, (l_i{:}B_i){::}S_i^{\,i\in 1..n}] \ \ E \vdash a@q{\in}Q \wedge \exists(q{,}p){\in}S_j \ \ j{\in}1..n}{E \vdash (a.l_j) \wedge (a@q{\uparrow}{=}a@p)}$$
  where $a@q \uparrow$ means update value of q for object a by a state transition defined in S_j for $q \in Q$.

Rule Type Object states that the object type $[\,Q,\ (l_i : B_i) :: S_i^{\,i\in 1..n}]$ is well-formed in E, provided that there exist a well-formed state set Q and a set of transitions S satisfying $Q \curvearrowright S$ ($S \subseteq \prod_{p,q\in Q}(p,q)$) in E. We always assume that, when writing $[\,Q,\ (l_i : B_i) :: S_i^{\,i\in 1..n}]$, that the labels l_i must be distinct. We identify object types $[Q,\ (l_i : B_i) :: S_i^{\,i\in 1..n}]$ by sorting components $(l_i : B_i) :: S_i$ in certain order, e.g., lexicographical order of l_i.

Rule Val Object states that an object type $[\,Q,\ (l_i : B_i) :: S_i^{\,i\in 1..n}]$ can be formed from a collection of n methods whose *self* parameter (e.g. *this* pointer referring to host object in C++ and Java) has type $[Q,\ (l_i : B_i) :: S_i^{\,i\in 1..n}]$ and whose bodies have type $B_1 :: S_1, ..., B_n :: S_n$. Note that *this* pointer is embedded in every method body and the circularity is used by the *self* parameter.

Rule Val Select describes how to enforce type correctness to a method invocation $a.l_j$. If there is an well-formed object type $[\,Q,\ (l_i : B_i) :: S_i^{\,i\in 1..n}]$ and method l_j indexed by j, $1 \leq j \leq n$, can be correctly invoked only when there exists a valid transition $(q,p) \in S_j$ for the object a in current state q ($a@q$). The state value of object a is updated to the state p.

Correspondingly, we can represent class types for an object type based on our τ-calculus. Let $A \equiv [Q, (l_i : B_i) :: S_i^{\,i\in 1..n}]$ be an object type, then $Class(A) \triangleq [new : A,\ l_i : A \rightarrow (B_i :: S_i)^{i\in 1..n}]$ is a class that can generate objects of type A. These classes have the form of $[new = \tau(z{:}Class(A))[q = \tau(P{:}Q,\ q_0{:}P)z@q_0,\ l_i = \tau(x{:}A, q{:}P, s_i{:}S_i)z.\ l_i(x) :: s_i(q)\ ^{i\in 1..n}],\ l_i :: s_i =$

$\lambda(x{:}A).\lambda(q{:}P)(s_i(q))(b_i)\ ^{i\in 1..n}]$. An object type is an implementation independent entity in contrast to an object class which is an implementation dependent entity [16]. Therefore, $P : Q$ stands for that a particular choice of the state set P in class implementation is an instance of the formally described state set Q in the triple, e.g., the state set $P = \{\mathit{HasJob},\ \mathit{NoJob}\}$ may be chosen to implement the class *Person* whose type contains the state set $Q = \{0_{\mathit{Employed}}, 1_{\mathit{Unemployed}}\}$. So there are the mappings of states from *HasJob* to 0_{Employed} and from *NoJob* to $1_{\mathit{Unemployed}}$ for *Person*. Similarly, $s_i : S_i$ stands for that the sets of state transition functions ${s_i}^{i=1..n}$ associated with the state set P are the instances of those ${S_i}^{i=1..n}$ associated with the state set Q. Note that q_0 is the starting state when an object is created. The state transition functions, denoted by $\lambda(q : P)(s_i(q))^{i\in 1..n}$, are a part of methods defined in a class. As a result, a method invocation will first check the state correctness and then do the rest computation. Similar to what in ς-calculus, an ad hoc inheritance relation on class types "$\mathit{Class}(A')$ may inherit from $\mathit{Class}(A)$ iff $A' <: A$" is set to follow the principle of method reuse for objects with states.

4 Inheritance and Subtyping

Although an object class is not an object type, a class is often taken as a type-defining construct in OOPLs. An insidious problem with inheritance in OOPLs is how to distinguish subtyping relation between object types, which indicates the inclusion of behaviors, from other purposes such as code reuse for classes. Objects of the same class are of the same type, but object of the same type may not belong to the same class. Inheritance may indicate a subtyping relation or code reuse or both. If subtyping relation is sound between A and B ($A <: B$), an object o_A of subclass A can emulate the behaviors of any object o_B of superclass B. Assume an arbitrary valid computation in terms of a sequence of state changes for o_B is represented by $o_B@q_0 \uparrow= q_1$, $o_B@q_1 \uparrow= q_2, \ldots, o_B@q_{m-1} \uparrow= q_m$. This property $\varphi(B)$ of o_B must be properly inherited by o_A in a form of behavioral inclusion polymorphism. As a result, there should be no type errors for o_A to simulate the computation: $o_A@q_0 \uparrow= q_1$, $o_A@q_1 \uparrow= q_2, \ldots, o_A@q_{m-1} \uparrow= q_m$. To enforce that the objects of a subtype ought to behave the same as those of its supertype for the same sequence of method invocations, the subtyping relation between two objects indicates a relation between two state machines, denoted by $M_A \preceq M_B$ where $M_A = (Q_A, \Sigma_A, S_A)$ and $M_B = (Q_B, \Sigma_B, S_B)$ satisfying $Q_A \supseteq Q_B$, $\Sigma_A \supseteq \Sigma_B$, and for each method $l_i \in \Sigma_B$, $S_{A_i} \supseteq S_{B_i}$ $(i = 1 \ldots |\Sigma_B|)$.

We provide that $\Delta_{<:Ob1}$ supports the basic behavioral inclusion polymorphism with single inheritance for subtyping derivation in our τ-calculus:

- Sub Object 1 (τ) (l_i distinct):

$$\frac{E \vdash (Q \subseteq \hat{Q}) \quad E \vdash B_i \wedge (\hat{Q} \curvearrowright \hat{S}_i) \quad E \vdash (Q \curvearrowright S_j) \wedge (S_j \subseteq \hat{S}_j) \ \ \forall i \in 1..n{+}m\ \forall j \in 1..n}{E \vdash [\,\hat{Q},\ (l_i{:}B_i){::}{\hat{S}_i}^{\,i\in 1..n+m}]\ <:\ [\,Q,\ (l_j{:}B_j){::}{S_j}^{\,j\in 1..n}]}$$

Rule Sub Object 1 (τ) states a general subtyping relation between object types. Let $[\hat{Q}, (l_i : B_i) :: \hat{S}_i^{i \in 1..n+m}]$ and $[Q, (l_j : B_j) :: S_j^{j \in 1..n}]$ be object types for $\hat{o}$ and o respectively. If there exist $Q \subseteq \hat{Q}$, and $\hat{Q} \curvearrowright \hat{S}_i$ for each method l_i indexed by i ($i = 1..n{+}m$), and $(Q \curvearrowright S_j) \wedge (S_j \subseteq \hat{S}_j)$ for each method l_j indexed by j ($j = 1..n$), then the object type of $\hat{o}$ is a subtype of type of o.

The subtyping rule is necessary to be extended when multiple inheritance is considered. In this case, the states and state transition functions associated with the subclass can be obtained by cross product of the states and state transition functions associated with its superclasses (see Appendix A).

5 Program Language Constructs and State Tracking

We first provide an example of our new programming language constructs for *states* and *state changes* in a *class* definition for class *Dryer* (see Figure 2(a)). We also use the current class structure (see Figure 2(b)) to implement states and state transitions for the purpose of comparison. At the end of a constructor heading in the new construct, the initial state is specified. On line 1 of Figure 2, it is an explicit declaration of state set for the class. The syntax used in our class *Dryer* definition is C++ syntax except the added syntax for states and state transitions.

```
     Class Dryer {                /* (a) Our proposed class construct */
1.     state: {OffSlow, OffFast, SlowHeating, FastHeating};
2.     public:
3.          Dryer(int voltage)::{ −>OffSlow } :_voltage(voltage) {}
4.          ~Dryer() {}
5.          void turnOn()  ::{OffSlow−>SlowHeating, OffFast−>FastHeating} {...}
6.          void turnOff()  ::{SlowHeating−>OffSlow, FastHeating−>OffFast} {...}
7.          void chg2Fast() ::{OffSlow−>OffFast, SlowHeating−>FastHeating} {...}
8.          void chg2Slow()::{OffFast−>OffSlow, FastHeating−>SlowHeating} {...}
9.          ... /* Other methods if necessary */
10.    private:
11.         int _voltage;  };       /* End of new class construct */

     Class Dryer {         /* (b) Traditional C++ class construct implementing states */
12.    enum state {OffSlow, OffFast, SlowHeating, FastHeating};
13.    public:
14.         Dryer(int voltage) : _voltage(voltage) {}
15.         ~Dryer() {}
16.         void turnOn()   {switch (_mystate) {case OffSlow: _mystate=SlowHeating;
17.                          break;  case OffFast: _mystate=FastHeating; break; default:} ...}
18.         void turnOff()   {switch (_mystate) {case SlowHeating: _mystate=OffSlow;
19.                          break;  case FastHeating: _mystate=OffFast; break; default:} ...}
20.         void chg2Fast() {switch (_mystate) {case OffSlow: _mystate=OffFast;
21.                          break;  case SlowHeating: _mystate=FastHeating; break; default:} ...}
22.         void chg2Slow() {switch (_mystate) {case OffFast: _mystate=SlowFast;
23.                          break;  case FastHeating: _mystate=SlowHeating; break; default:} ...}
24.    private:
25.         state _mystate;  int _voltage;  };      /* End of C++ class construct */
```

Fig. 2. (a) Our proposed class construct for *Dryer* and (b) the traditional C++ class construct for *Dryer* with states

One may argue that the above class definitions can be implemented as what is shown in Figure 2(b), so programmers must implement the states and state transitions as part inside method code. The advantage of using our proposed class construct is obvious. First, it is in a much simpler form. The states and state transition functions associated with each method reflect a more intuitive mapping from statechart. Second, inheritance (subtyping) clearly indicates the dynamic behavioral inclusion, e.g., if "*class* SmartDryer : *public* Dryer" is declared, then the state set and the methods together with their associated state transition functions in class *Dryer* are inherited to class *SmartDryer*. Moreover, we can add new state transitions for the same state set in the subclass. Third, the new constructs for states and state transition functions are easy to implement at the compilation stage.

The syntax of states and state transition functions added into the class construct can be easily related to our proposed calculi. Based on our proposed object typing and subtyping fragments and programming language constructs, we can detect type errors of object misbehavior. If we do not consider parallelism at this stage, the methods of objects are called in sequence, which allows us to trace the state changes step by step.

6 Conclusion

We propose that object dynamic behavior, in respect to the general existence of states and state changes in objects, should be as an essential component in object type theory and subtyping modeling. This is an amendment to the object typing framework that appeared in ς-calculus by Abadi and Cardelli. The object type with states is more precise, and can ensure a higher level correctness of type systems built for objects. As a result, type errors caused by method calls at improper state can be detected. We believe that this approach in building our object type system is generally applicable in OOPLs. More work, such as a precise definition of semantics for states, flow control optimization in guarded transitions, and a formal translation of our proposed class-based language and τ-calculus, should be continued in the near future.

References

[1] Abadi, M. and L. Cardelli, "A Theory of Objects", Springer, New York, 1996.

[2] Booch, G., J. Rumbaugh, and I. Jacobson, "The Unified Modeling Language User Guide", Addison-Wesley, 1999.

[3] Deline, R. and M. Fahndrich, "Typestates for objects", ECOOP 2004, 465-90.

[4] Drossopoulou, S., F. Damiani, M. Dezani-Ciancaglini, and P. Giannini, "Fickle: dynamic object re-classification", ECOOP 2001, 130-49.

[5] Harel, D. and E. Grey, *Executable Object Modeling with Statecharts*, IEEE Computer, **30**, issue 7(1997), 31-42.

[6] Hopcroft, J.E., R. Motwani, and J.D. Ullman, "Introduction to Automata Theory, Languages, and Computation", 2nd Edition, Addison-Wesley, 2001.

[7] Igarashi, A. and N. Kobayashi, “Resource Usage Analysis”, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2002, 331-42.

[8] Liskov, B.H. and J.M. Wing, *A behavioral notion of subtyping*, ACM Transactions on Programming Languages and Systems, **16**, issue 6(1994), 1811-41.

[9] Madsen, O.L., “Towards Integration of State Machines and object-oriented Languages”, Proceedings of TOOLS Europe ’99: Technology of Object Oriented Languages and Systems. 29th International Conference, 1999, 261-74.

[10] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, “Object-Rriented Modeling and Design”, Prentice Hall 1991.

[11] Ravara, A. and V.T. Vasconcelos, “Typing non-uniform concurrent objects”, Proceedings of CONCUR 2000. 11th International Conference on Concurrency Theory, 2000, 474-88.

[12] Sakharov, A., “State Machine Specification Directly in Java and C++”, addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications, OOPSLA, 2000, 103-104.

[13] Samek, M. and P. Montgomery, *State-Oriented Programming*, Embedded Systems Engineering, **13**, issue 8(2000), 22-43.

[14] Strom, R.E. and S. Yemini, *Typestate: A Programming Language Concept for Enhancing Software Reliability*, IEEE Transactions on Software Engineering, **SE-12**, issue 1(1986), 157-171.

[15] Tan, G., X. Ou, and D. Walker, “Resource Usage Analysis Via Scoped Methods”, Foundations of Object-Oriented Languages (2003), URL: http://www.cs.princeton.edu/~dpw/papers/smethods.pdf.

[16] Yu, S., *Class-is-type is inadequate for object reuse*, ACM Sigplan Notice, **36**, No. 6(2001), 50-59.

Appendix A: $\Delta_{<:Ob2}$ (τ-calculus)

Let $A \equiv [Q, (l_k : B_k) :: S_k^{k\in 1..\ell}]$ and $\ell = |\Sigma_1 \cup \Sigma_2|$:

SUB OBJECT 2 (τ) (l_i distinct, $Q \ddot{\curvearrowright} S_k$ means $\prod_{p,p' \in Q_1, q,q' \in Q_2}((p,q),(p',q'))$):

$$\frac{E \vdash (Q_1 \wedge Q_2) \wedge (B^1_i \wedge (Q_1 \curvearrowright S^1_i)) \wedge (B^2_j \wedge (Q_2 \curvearrowright S^2_j)) \quad E \vdash (Q = Q_1 \times Q_2) \wedge Q \ddot{\curvearrowright} (S_k) \wedge (S = S^1 \bigotimes S^2) \ \forall i \in 1..n, j \in 1..m}{E \vdash (A <: [Q_1, (l_i : B^1_i) :: {S^1}_i{}^{i\in 1..n}]) \wedge (A <: [Q_2,\ (l_j : B^2_j) :: {S^2}_j{}^{j \in 1..m}]) \ \textit{where}\ (max(n,m) < \ell \leq n+m)}$$

Rule SUB OBJECT 2 (τ) describes the object subtyping rule for multiple inheritance in addition to the $\Delta_{<:Ob1}$ of τ-calculus. Let $A_{S^1} \equiv [Q_1, (l_i : {B^1}_i) :: {S^1}_i{}^{i\in 1..n}]$ and $A_{S^2} \equiv [Q_2,\ (l_j : {B^2}_j) :: {S^2}_j{}^{j\in 1..m}]$ be the two supertypes (or superclasses). If there exist the conjunctive conditions of $(Q = Q_1 \times Q_2) \wedge (Q_1 \curvearrowright {S^1}_i) \wedge (Q_2 \curvearrowright {S^2}_j) \wedge (Q \ddot{\curvearrowright} S_k) \wedge (S = S^1 \bigotimes S^2)$ (where $i = 1..n$ and $j = 1..m$ and $(max(n,m) < \ell \leq n+m)$), then the object type of $A \equiv [Q, (l_k : B_k) :: {S_k}^{k\in 1..\ell}]$ is a subtype of both the type A_{S^1} and type A_{S^2}. $S = S^1 \bigotimes S^2$ describes construction of the state transition functions ${S_k}^{k=1..\ell}$ for the triple (Q, Σ, S) associated with the subtype from the sets S^1 and S^2.

DCM 2005 Preliminary Version

SCHOOL: a Small Chorded Object-Oriented Language

S. Drossopoulou, A. Petrounias, A. Buckley, S. Eisenbach

{ *s.drossopoulou, a.petrounias, a.buckley, s.eisenbach* } @ *imperial.ac.uk*

Department of Computing, Imperial College London, United Kingdom

Abstract

Chords are a declarative synchronisation construct based on the *Join*-Calculus, available in the programming language C_ω. To our knowledge, chords have no formal model in an object-oriented setting.

In this paper we suggest SCHOOL, a formal model for an imperative, object-oriented language with chords. We give an operational semantics and type system, and can prove soundness of the type system.

1 Introduction

A *chorded program* [1] consists of class definitions, each class defining one or more chords. A chord has a signature and a body. A chord's signature is an aggregate that comprises at most one *synchronous* method and zero or more *asynchronous* methods.

A chord body is executed when an object has received at least one message for *each* of the chord's synchronous and asynchronous method signatures. Potentially multiple method calls are needed to invoke a chord's body. This reflects the notion of *join* from the *Join*-Calculus [2], where the *join-pattern* consists of the methods comprising the chord signature.

For instance, the following chord, an unbounded buffer:

```
int get() & async put(int x) { return x }
```

will execute the body and return *x* only when there is a *simultaneous* presence of invocations to both of the methods in its signature.

The method *get* is synchronous, and hence will block its caller until there is a message present for method *put* and hence it can join. The latter method is asynchronous, a subtype of *void*, and returns immediately to its caller; thus messages sent to it must be queued by the receiving object until consumed by the joining of the chord.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Those chords whose signatures contain a *synchronous* method are called *synchronous* chords. Chords with only *asynchronous* methods in their signature are called *asynchronous* chords.

2 SCHOOL

We present SCHOOL (see overview in figure 1) in the form of structural operational semantics (found in figure 2) and an accompanying type system (in figure 3). An extended version of this paper with additional material, a more thorough coverage of chords in general, and hand-written proofs of soundness can be found from the following website: `slurp.doc.ac.uk/school`.

Expressions and Programs

The syntax of SCHOOL expressions is: method call, sequence of expressions, the receiver (*this*), a parameter (x), and the values *null* (for the null pointer) and $voidVal$ (for the result of an execution that returns *void* or for the result of a call to an *asynchronous* method).

We also define SCHOOL programs, which are tuples of mappings. We do not give a syntax for programs, and therefore can omit rather mechanical definitions of derived functions which lookup methods and superclasses.

A program consists of 1) a mapping from a class and method name to the method's signature in that class, 2) a mapping from a class and method name to all chords in which the method name is the *synchronous* part, 3) a mapping from a class name to the set of the class's *asynchronous* chords, and 4) a mapping from a class name to the name of its superclass.

A method signature contains a return type, a method name and a parameter type. The name of the formal parameter is derived from the name of the method: for a method called `mth`, the parameter will be called `mth_x`. These restrictions are, of course, inconvenient for programming but are not essential to our study of chords and types, and they allow a considerably more succinct presentation.

We represent a chord as a set of asynchronous method names along with the expression representing the chord's body. Thus, the distinction between a *synchronous chord* and an *asynchronous chord* is whether the chord appears in the image of the second or the third component of a SCHOOL program. A method name can appear in any number of chords.

For ease of notation we also define the following four lookup functions: the function $\mathcal{M}\ (\ P,\ c,\ m\ )$ is the projection of the first component of P, and returns m's signature in class c; the function $\mathcal{SC}hs\ (\ P,\ c,\ m\ )$ is the projection of the second component of P, and finds the *synchronous* chords to which m belongs, returning their *asynchronous* method names plus their bodies; the function $\mathcal{AC}hs\ (\ P,\ c\ )$ is the projection of the third component of P, and returns the set of *asynchronous* chords for class c; finally, $\mathcal{M}^a\ (\ P,\ c\ )$ gives all *asynchronous* method names present in class c's chord definitions.

Objects, Messages and the Heap

One can view chord invocation as message-passing between objects. A caller object sends a message comprising of a name and an argument to a receiver object. A call to an asynchronous method returns immediately, but the corresponding chord body may not yet be ready to run. Therefore, asynchronous method calls are queued within the receiver object.

Consequently an object comprises 1) the name of its defining class and 2) one queue for each *asynchronous* method signature in its class. Thus, the state of an object is represented by its queues.

Queues are modelled as mappings from method identifiers to *multisets of values* representing the actual parameter passed when the asynchronous method was called. The use of multisets allows a natural presentation of the non-deterministic nature of handling asynchronous methods call, whereby asynchronous calls are not guaranteed to be handled in the order they were made, even if they were made consecutively from the same thread [1]. We need to have *multisets* rather than sets in order to model the situation where an asynchronous method was called twice with the same parameter.

An interesting observation is that any object that can access another object can write to its queues by calling asynchronous methods. However, only the chord body associated with an asynchronous method signature can read from the method's queue. Reading from a queue consumes one of its elements.

The heap maps addresses (in $\mathbb{N}$) to objects. Once an object is allocated at an address, there is no way to remove it. Thus, in terms of address-to-value mappings, the heap grows monotonically. However, the queues within each object grow and shrink as messages are sent to and consumed from queues, as described earlier.

Operational Semantics

SCHOOL operational semantics are found in figure 2. The aim of the rules is to abstract as much away from scheduling as possible. Hence, we welcome non-determinism whenever there is choice, thus maximising the possible behaviours of programs. There are three rules of particular interest: Async, Join, and Strung. These three rules capture the essence of chord invocation in SCHOOL.

Async describes invocation of an *asynchronous* method: the value representing void is immediately returned, and the actual argument is placed in the appropriate queue of the receiving object.

Join describes invocation of a *synchronous* method. The caller will block until all the *asynchronous* methods present in the chord containing the invoked method have at least one message each in their respective queues at the receiving object. Notice that the choice of chord is non-deterministic, as is the choice of participating queue elements. Once the chord joins, the messages are consumed from the queues and the current expression becomes the body of the chord.

Strung describes execution of *asynchronous* chords. Essentially, this rule exhibits non-deterministic choice at three levels: the selection of object in the heap, the selection of *asynchronous* chord, and the selection of elements from the participating queues. The body of the chord will execute in a new thread (we call this *spawning*).

Type Judgements

The judgement $\mathrm{P},\ \Gamma\ \vdash\ e : t$ describes the static type of a source level expression e, while the judgement $\mathrm{P},\ h\ \vdash\ e : t$ describes the dynamic type of a runtime expression e. The complete SCHOOL type system can be found in figure 3.

Well-Formed Programs

A well-formed SCHOOL source program (WF-Prgm) is comprised of well-formed class declarations (WF-Class). A class declaration is well-formed if its superclass is a class, *i.e.,* `Object` or a class defined in the program, any method overridden from the superclass has the same signature up to `async` or `void`, all *synchronous* chords are well-formed, and all *asynchronous* chords are well-formed.

A *synchronous* chord is well-formed when the return type of the chord's *synchronous* method signature coincides with the type of the chord body. The chord body is typed in a context where formal parameters take the types mentioned in *synchronous* and *asynchronous* method signatures, and *this* takes the type of the current class. Any other method signatures in the chord's signature must have a return type of `async`.

An *asynchronous* chord is well-formed when the chord body has type `void`, when typed in a context where the formal parameters take the types mentioned in the *asynchronous* method signatures.

With regard to method overriding, our system allows a method that returns `void` to be overridden in a subclass by a method that returns `async`. It also allows a method that returns `async` to be overridden in a subclass by a method that returns `void`. C_ω only allows a method that returns `void` to be overridden by a method that returns `async`. While overriding such as we allow may not be good programming practice, it does not affect the soundness of the type system, and so is allowed.

Furthermore, C_ω imposes restrictions on the overriding of methods when involved in chords, in order to avoid the inheritance anomaly [3]. The inheritance anomaly, however, is concerned with preservation of synchronisation properties and is unrelated to type soundness. Therefore, our system does not impose similar restrictions.

Finally, we do not require the class hierarchy to be acyclic. Although this property is useful for a compiler, it is not essential for type soundness.

Well-Formed Heaps

A well-formed heap (WF-Heap) requires that every value in an object's queues must have a type according to the parameter type in the corresponding asynchronous method signature.

Soundness

The evaluation rules for SCHOOL preserve types throughout execution. We prove this property through a subject-reduction theorem [4]. The proof technique is standard.

We first define appropriate substitutions, σ, which map identifiers onto addresses in a type preserving way.

Definition 2.1 [Appropriate Substitution]
For a substitution $\sigma = Id \cup \{ this \} \rightarrow Addr$, a heap h, and an environment Γ, we have:

$$\Gamma, h \vdash \sigma$$

iff:

$$dom(\Gamma) = dom(\sigma)$$
$$\Gamma(id) = c \implies \_, h \vdash \sigma(id) : c$$

We can easily prove that an appropriate substitution, σ, when applied to an expression e turns it into a runtime expression, of the same type as the original expression.

Lemma 2.2 (Substitution)

$$\left.\begin{array}{l} \mathrm{P}, \Gamma \vdash e : t \\ \mathrm{P}, h \vdash \sigma \end{array}\right\} \implies \mathrm{P}, h \vdash [\, e\,]_{\sigma} : t$$

Proof. By induction on expression e. □

Furthermore, if a runtime expression has a certain type in a heap h, then it preserves its type in any heap h' where the objects have the same classes as the corresponding objects in h.

Lemma 2.3 (Preservation)
If

$$\forall \iota \in dom(h) : h(\iota) = [\![\, c \parallel \_ \,]\!] \implies h'(\iota) = [\![\, c \parallel \_ \,]\!]$$

then:

$$\mathrm{P}, h \vdash e : t \implies \mathrm{P}, h' \vdash e : t$$

Proof. By structural induction on expression e. □

We can prove subject reduction for the sequential case:

Lemma 2.4 (Subject Reduction - Sequential)

$$\left.\begin{array}{l} \mathrm{P} \vdash h \\ \vdash \mathrm{P} \\ \mathrm{P},\, h \vdash e : t \\ e,\, h \rightsquigarrow e',\, h' \end{array}\right\} \Longrightarrow \begin{array}{l} \mathrm{P} \vdash h' \\ \mathrm{P},\, h' \vdash e' : t \end{array}$$

Proof. By structural induction on the derivation $\rightsquigarrow$. □

Finally, we can prove subject reduction for the multithreaded case:

Theorem 2.5 (Subject Reduction - Threads)
For SCHOOL heaps h and h', program P, *expressions $e_1, \ldots e_n, e'_1, \ldots e'_m$, types $t_1, \ldots t_n$, if*

- $\vdash \mathrm{P}$ *and* $\mathrm{P} \vdash h$,
- $e_1, \ldots, e_n, h \rightsquigarrow e'_1, \ldots, e'_m, h'$
- $\mathrm{P}, h \vdash e_i : t_i \qquad \forall i \in 1..n$

then, there exist types $t'_1, \ldots, t'_m$ so that:

- $\mathrm{P} \vdash h'$
- $\mathrm{P}, h' \vdash e'_i : t'_i \qquad \forall i \in 1..m$
- $\{ t_1, \ldots, t_n \} \cup \{ void \} = \{ t'_1, \ldots, t'_m \} \cup \{ void \}$

Proof. By case analysis on $\rightsquigarrow$ and application of lemma 2.4. □

3 Conclusions and Future Work

We designed SCHOOL with the aim of studying the features essential to an understanding of chords in an imperative, object-oriented setting. We made various design decisions to keep our language simple and the description minimal. Consequently, only classes and chords were necessary; the operational semantics requires only half a page, and ten rules!

One design decision involves fields: in the extended paper available online, we describe the language SCHOOL+F, an extension of SCHOOL with fields. We show how fields can be emulated using only chords, and hence how programs in SCHOOL+F can be translated into programs in SCHOOL. We describe the translation function and show that translated programs in SCHOOL have equivalent behaviours with their original programs in SCHOOL+F. Hence SCHOOL is as expressive as SCHOOL+F and fields are not a necessary feature of the original language.

We have also incorporated subclasses in SCHOOL, and were thus able to formally confirm that although inheritance and synchronisation do not generally mix well [3], the issues are unrelated to type soundness. Thus, in SCHOOL, we allow a method returning `void` to be overridden by a method returning async, and vice-versa. We also allow a method defined in one chord to be part of another chord in a subclass. The restrictions on overriding and method declaration in C_ω are thus unrelated to typing issues; rather, they attempt to preserve how a method is synchronised in subclasses.

In further work, we would like to extend SCHOOL to study interesting interactions with other language features. Although features like generics, packages, inner classes, overloading, and various control structures are probably orthogonal to chords, we expect that the introduction of delegates and exceptions may throw some interesting questions.

More interesting will be the study of the combination of SCHOOL and explicit synchronisation mechanisms as in Java and $C^\sharp$, like locks and monitors. Furthermore, we would like to design extensions of chords to incorporate more advanced features, such as preemptions, priorities and transactions. We will use SCHOOL to express our designs.

It would also be interesting to consider issues around the scheduling for chords. The semantics of SCHOOL is non-deterministic, and thus abstract away one important property of chords as in Polyphonic $C^\sharp$; namely that any chord which *can* run (*i.e.,* whose queues are not empty), will *eventually* run. One could try to characterise such *fair* execution strategies through a further refinement of the operational semantics. More interesting would be a formal understanding of particular scheduling mechanisms, and proof of their properties.

Finally, another challenging direction is the use of chords in program understanding and verification. In [1], asynchronous methods correspond to states, and some synchronous method calls correspond to state change. Although this analogy cannot be expected to always hold, it would be interesting and useful to study how state transition diagrams can be mapped into chorded programs, and vice-versa. Such an approach would then allow the application of model-checkers.

References

[1] Benton, N., Cardelli, L., and Fournet, C. Modern concurrency abstractions for $C^\sharp$. *ACM Trans. Program. Lang. Syst. 26*, 5 (2004), 769–804.

[2] Fournet, C., and Gonthier, G. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages* (1996), ACM Press, pp. 372–385.

[3] Matsuoka, S., and Yonezawa, A. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research directions in concurrent object-oriented programming* (1993), MIT Press, pp. 107–150.

[4] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. *Inf. Comput. 115*, 1 (1994), 38–94.

Abstract Syntax

$$\begin{array}{rcl} e^s \in Expr^S & ::= & \texttt{null} \mid \text{voidVal} \mid \texttt{this} \mid x \\ & \mid & \texttt{new}\,c \mid e^s.m\,(\,e^s\,) \mid e^s\ ;\ e^s \\ MethSig & ::= & t\;m\,(\,c\,) \\ t \in Type & ::= & \texttt{void} \mid \texttt{async} \mid c \\ x,\ c,\ m \in Id & & \end{array}$$

Lookup Functions

$$\begin{array}{rcl} \mathcal{M}\,(\,\mathrm{P},\ c,\ m\,) & = & P{\downarrow_1}\,(\,c,\ m\,) \\ \mathcal{SC}hs\,(\,\mathrm{P},\ c,\ m\,) & = & P{\downarrow_2}\,(\,c,\ m\,) \\ \mathcal{AC}hs\,(\,\mathrm{P},\ c\,) & = & P{\downarrow_3}\,(\,c\,) \\ \mathcal{M}^a & : & Program \times Id^c \to \mathcal{P}\,(\,Id^m\,) \\ \mathcal{M}^a & = & \{\,m \mid \mathcal{M}\,(\,P,\ c,\ m\,) = \texttt{async}\,m\,(\,\_\,)\,\} \end{array}$$

Program Representation

$$\begin{array}{rcl} Program & = & Id^c \times Id^m \to MethSig \\ & & \times \\ & & Id^c \times Id^m \to \mathcal{P}\,(\,Chord\,) \\ & & \times \\ & & Id^c \to \mathcal{P}\,(\,Chord\,) \\ & & \times \\ & & Id^c \to Id^c \\ Chord & = & \mathcal{P}\,(\,Id^m\,) \times Expr \end{array}$$

Runtime Entities

$$\begin{array}{rcl} Heap & = & \mathbb{N} \to Object \\ Object & = & Id^c \times Queues \\ Queues & = & Id^m \to multiset(Val) \\ e \in Expr & ::= & voidVal \mid nullPtrEx \mid v \\ & \mid & \texttt{new}\,c \mid e.m\,(\,e\,) \mid e\ ;\ e \\ v \in Val & ::= & null \mid \iota \\ \iota \in \mathbb{N} & & \end{array}$$

Well-Formedness

$$\frac{\mathrm{P} \vdash c \diamond_{cl} \implies \mathrm{P} \vdash c}{\vdash \mathrm{P}}\ \text{WF-Prgm}$$

$$\frac{\begin{array}{l} \mathrm{P} \vdash \mathrm{P}{\downarrow_4}\,(\,c\,) \diamond_{cl} \\ \mathcal{M}\,(\,\mathrm{P},\ \mathrm{P}{\downarrow_4}\,(\,c\,),\ m\,) = t\;m\,(\,t'\,) \implies \mathcal{M}\,(\,\mathrm{P},\ c,\ m\,) = t''\;m\,(\,t'\,) \\ \quad \text{where } t'' = t \text{ or } t,\ t'' \in \{\,void,\ async\,\} \\ \mathcal{SC}hs\,(\,\mathrm{P},\ c,\ m\,) \ni (\,\{\,m_1,\ \ldots,\ m_n\,\},\ e\,) \implies \\ \quad \forall i \in 1..n : \exists\, t_i : \mathcal{M}\,(\,\mathrm{P},\ c,\ m_i\,) = \texttt{async}\,m_i\,(\,t_i\,) \\ \qquad\qquad \exists\, t,\ t' : \mathcal{M}\,(\,\mathrm{P},\ c,\ m\,) = t\;m\,(\,t'\,) \\ \qquad\qquad \mathrm{P},\ (\,m_1\_x \mapsto t_1,\ \ldots,\ m_n\_x \mapsto t_n,\ m\_x \mapsto t',\ this \mapsto c\,) \vdash e : t \\ \mathcal{AC}hs\,(\,\mathrm{P},\ c\,) \ni (\,\{\,m_1,\ \ldots,\ m_n\,\},\ e\,) \implies \\ \quad \forall i \in 1..n : \exists\, t_i : \mathcal{M}\,(\,\mathrm{P},\ c,\ m_i\,) = \texttt{async}\,m_i\,(\,t_i\,) \\ \qquad\qquad \mathrm{P},\ (\,m_1\_x \mapsto t_1,\ \ldots,\ m_n\_x \mapsto t_n,\ this \mapsto c\,) \vdash e : void \end{array}}{\mathrm{P} \vdash c}\ \text{WF-Class}$$

$$\frac{h\,(\,\iota\,) = [\![\,c \parallel qs\,]\!],\ \mathcal{M}\,(\,\mathrm{P},\ c,\ m\,) = \texttt{async}\,m\,(\,t\,),\ v \in qs\,(\,m\,) \implies \mathrm{P},\ h \vdash v : t}{\mathrm{P} \vdash h}\ \text{WF-Heap}$$

Fig. 1. SCHOOL Overview

Contexts

$$E[\,.\,] \;::=\; E[\,.\,].m\,(\,e\,) \mid \iota.m\,(\,E[\,.\,]\,) \mid E[\,.\,]\ ;\ e$$

Evaluation Rules

$$\frac{e,\ h\ \rightsquigarrow\ e',\ h'}{E[\,e\,],\ h\ \rightsquigarrow\ E[\,e'\,],\ h'}\ \textsc{Cntx}
\qquad
\frac{r\ \in\ \mathbb{N}\ \cup\ \{\,null,\ voidVal\,\}}{r\ ;\ e,\ h\ \rightsquigarrow\ e,\ h}\ \textsc{Seq}$$

$$\frac{\{\,e_1,\ \ldots,\ e_n\,\}\ =\ \{\,e'_1,\ \ldots,\ e'_n\,\}}{e_1,\ \ldots,\ e_n,\ h\ \rightsquigarrow\ e'_1,\ \ldots,\ e'_n,\ h}\ \textsc{Perm}
\qquad
\frac{e_n,\ h\ \rightsquigarrow\ e'_n,\ h'}{e_1,\ \ldots,\ e_n,\ h\ \rightsquigarrow\ e_1,\ \ldots,\ e_{n-1},\ e'_n,\ h'}\ \textsc{Run}$$

$$\frac{}{null.\mathrm{m}\,(\,v\,),\ h\ \rightsquigarrow\ nullPtrEx,\ h}\ \textsc{Ex}
\qquad
\frac{}{E[\,nullPtrEx\,],\ h\ \rightsquigarrow\ nullPtrEx,\ h}\ \textsc{Ex-Prop}$$

$$\frac{\begin{array}{l} h\,(\,\iota\,)\ =\ \mathcal{U}df \\ \mathcal{M}^a\,(\,P,\ C\,)\ =\ \{\,m_1,\ \ldots,\ m_n\,\} \end{array}}{\mathrm{new}\ \mathrm{C},\ h\ \rightsquigarrow\ \iota,\ h[\,\iota\ \mapsto\ [\![\,\mathrm{C} \parallel m_1\ \mapsto\ \emptyset,\ \ldots,\ m_n\ \mapsto\ \emptyset\,]\!]\,]}\ \textsc{New}$$

$$\frac{\begin{array}{l} h\,(\,\iota\,)\ =\ [\![\,c \parallel qs\,]\!] \\ \mathcal{M}\,(\,P,\ c,\ \mathrm{m}\,)\ =\ \mathrm{async}\ \mathrm{m}\,(\,\_\,) \end{array}}{\iota.\mathrm{m}\,(\,v\,),\ h\ \rightsquigarrow\ voidVal,\ h[\,\iota\ \mapsto\ [\![\,c \parallel qs[\,\mathrm{m}\ \mapsto\ \{\,v\,\}\ \cup\ qs\,(\,\mathrm{m}\,)\,]\,]\!]\,]}\ \textsc{Async}$$

$$\frac{\begin{array}{l} h\,(\,\iota\,)\ =\ [\![\,c \parallel qs\,]\!] \\ \mathcal{SC}hs\,(\,P,\ c,\ \mathrm{m}\,)\ \ni\ (\{\,m_1,\ \ldots,\ m_n\,\},\ e) \\ \forall i\ \in\ 1..n : qs\,(\,m_i\,)\ =\ \{\,v_i\,\}\ \cup\ q_i \end{array}}{\begin{array}{l} \iota.\mathrm{m}\,(\,v\,),\ h\ \rightsquigarrow\ e[\,{}^{v_1}/_{m_1\_x},\ \ldots,\ {}^{v_n}/_{m_n\_x},\ {}^{v}/_{m\_x},\ {}^{\iota}/_{this}\,], \\ \qquad h[\,\iota\ \mapsto\ [\![\,c \parallel qs[\,m_1\ \mapsto\ q_1,\ \ldots,\ m_n\ \mapsto\ q_n\,]\,]\!]\,] \end{array}}\ \textsc{Join}$$

$$\frac{\begin{array}{l} h\,(\,\iota\,)\ =\ [\![\,c \parallel qs\,]\!] \\ \mathcal{AC}hs\,(\,P,\ c\,)\ \ni\ (\{\,m_1,\ \ldots,\ m_n\,\},\ e\,) \\ \forall i\ \in\ 1..n : qs\,(\,m_i\,)\ =\ \{\,v_i\,\}\ \cup\ q_i \end{array}}{\begin{array}{l} e_1,\ \ldots,\ e_k,\ h\ \rightsquigarrow\ e_1,\ \ldots,\ e_k,\ e[\,{}^{v_1}/_{m_1\_x},\ \ldots,\ {}^{v_k}/_{m_k\_x},\ {}^{\iota}/_{this}\,], \\ \qquad h[\,\iota\ \mapsto\ [\![\,c \parallel qs[\,m_1\ \mapsto\ q_1,\ \ldots,\ m_n\ \mapsto\ q_n\,]\,]\!]\,] \end{array}}\ \textsc{Strung}$$

Fig. 2. SCHOOL Operational Semantics

Class and Type Declarations

$$\frac{P{\downarrow_4}(c) \neq \mathcal{U}df}{P \vdash c \diamond_{cl}}\ \textsc{Def-Class-1} \qquad \frac{}{P \vdash \mathit{Object} \diamond_{cl}}\ \textsc{Def-Class-2}$$

$$\frac{t \in \{\mathit{void}, \mathit{async}\}}{P \vdash t \diamond_{tp}}\ \textsc{Def-Type-1} \qquad \frac{P \vdash c \diamond_{cl}}{P \vdash c \diamond_{tp}}\ \textsc{Def-Type-2}$$

Source-Level Type Judgements

$$\frac{P \vdash c \diamond_{cl}}{P, \Gamma \vdash \mathit{null} : c}\ \text{ST-}\textsc{Null} \qquad \frac{}{P, \Gamma \vdash \mathit{voidVal} : \mathit{void}}\ \text{ST-}\textsc{Void} \qquad \frac{z \in \{\mathit{this}\} \cup x}{P, \Gamma \vdash z : \Gamma(z)}\ \text{ST-}\textsc{This}\text{X}$$

$$\frac{P \vdash c \diamond_{cl}}{P, \Gamma \vdash \texttt{new c} : \texttt{c}}\ \text{ST-}\textsc{New} \qquad \frac{\begin{array}{l} P, \Gamma \vdash e_1 : c \\ P, \Gamma \vdash e_2 : t \\ \mathcal{M}(P, c, \mathrm{m}) = t_r\ \mathrm{m}(t) \end{array}}{P, \Gamma \vdash e_1.\mathrm{m}(e_2) : t_r}\ \text{ST-}\textsc{Inv} \qquad \frac{\begin{array}{l} P, \Gamma \vdash e_1 : t_1 \\ P, \Gamma \vdash e_2 : t_2 \end{array}}{P, \Gamma \vdash e_1\ ;\ e_2 : t_2}\ \text{ST-}\textsc{Seq}$$

Run-Time Type Judgements

$$\frac{P \vdash c \diamond_{cl}}{P, h \vdash \mathit{null} : c}\ \text{RT-}\textsc{Null} \qquad \frac{}{P, h \vdash \mathit{voidVal} : \mathit{void}}\ \text{RT-}\textsc{Void} \qquad \frac{h(\iota) = [\![c \,\|\, \_]\!]}{P, h \vdash \iota : c}\ \text{RT-}\textsc{Addr}$$

$$\frac{P \vdash c \diamond_{cl}}{P, h \vdash \texttt{new c} : \texttt{c}}\ \text{RT-}\textsc{New} \qquad \frac{\begin{array}{l} P, h \vdash e_1 : c \\ P, h \vdash e_2 : t \\ \mathcal{M}(P, c, \mathrm{m}) = t_r\ \mathrm{m}(t) \end{array}}{P, h \vdash e_1.\mathrm{m}(e_2) : t_r}\ \text{RT-}\textsc{Inv} \qquad \frac{\begin{array}{l} P, h \vdash e_1 : t_1 \\ P, h \vdash e_2 : t_2 \end{array}}{P, h \vdash e_1\ ;\ e_2 : t_2}\ \text{RT-}\textsc{Seq}$$

$$\frac{\begin{array}{l} P, h \vdash e : c \\ P{\downarrow_4}(\mathrm{c}) = c' \end{array}}{P, h \vdash e : c'}\ \text{RT-}\textsc{Sub-Cls} \qquad \frac{P, h \vdash e : \mathit{void}}{P, h \vdash e : \mathit{async}}\ \text{RT-}\textsc{Sub-Async} \qquad \frac{P \vdash t \diamond_{tp}}{P, h \vdash \mathit{nullPtrEx} : t}\ \text{RT-}\textsc{Ex}$$

Fig. 3. SCHOOL Type System

Coalgebraic Description of Generalized Binary Methods [1]

Furio Honsell [2]

DIMI, Università di Udine, ITALY

Marina Lenisa [3]

DIMI, Università di Udine, ITALY

Rekha Redamalla [4]

DIMI, Università di Udine, ITALY, and
B.M. Birla Science Centre, Hyderabad, INDIA.

Abstract

We extend Reichel-Jacobs coalgebraic account of specification and refinement of objects and classes in Object Oriented Programming to (*generalized*) *binary methods*. These are methods which take more than one parameter of a class type. Class types include sums and (possibly infinite) products type constructors. We study and compare two solutions for modeling generalized binary methods, which use purely covariant functors. In the first solution, which applies when we have already a class implementation, we reduce the behaviour of a generalized binary method to that of a bunch of unary methods. These are obtained by *freezing* the types of the extra class parameters to constant types. The *bisimulation behavioural equivalence* induced on objects by this model amounts to the *greatest congruence* w.r.t method application. Alternatively, we treat binary methods as *graphs* instead of functions, thus turning contravariant occurrences in the functor into covariant ones.

Key words: OO-programming, Binary methods, Coalgebraic semantics.

Introduction

In [10,7,8], a categorical semantics for *objects* and *classes* based on *coalgebras* is given. The idea underpinning this approach is that coalgebras, duals of algebras, allow to focus on the behaviour of objects while abstracting from the concrete representation of the state of the objects.

[1] Work partially supported by the UE Project IST-510996 TYPES.
[2] Email: `honsell@dimi.uniud.it`
[3] Email: `lenisa@dimi.uniud.it`
[4] Email: `redamall@dimi.uniud.it`

In the coalgebraic approach of [10,7,8], a class is modelled as an F-coalgebra $(A, f : A \to F(A))$ for a suitable functor F. The carrier A represents the space of *attributes*, or *fields*, and the coalgebra operation f represents the *public* methods of the class, i.e. the methods which are accessible from outside the class. Thus the objects of a class are modelled as the elements of the carrier. Their behaviour under application of public methods, viewed as functions acting on objects, is then captured by the coalgebra map f. Thus the coalgebraic model induces exactly the behavioural equivalence on objects, whereby two objects are equated if, for each public method, the application of the method to the two objects, for any list of parameters, produces equivalent results. A benefit of the coalgebraic model is a coinduction principle for establishing the behavioural equivalence.

Following [7], we distinguish between class *specifications* and class *implementations* (or simply classes). A class specification is like an abstract class, in which only the signatures of constructors and (public) methods are given, without their actual code. Assertions enforce behavioural constraints on constructors and methods. Implementation of constructors and methods is given in a class implementation. In the *bialgebraic* approach, a class specification induces a pair of functors, determined by the signature of constructors and methods, respectively. A class implementation is any bialgebra satisfying the assertions. Here we will focus only on the coalgebraic part, which is the problematic one. For a complete bialgebraic treatment, see [6].

Binary methods, i.e. methods with more than one class argument, apparently escape the coalgebraic approach. The extra class parameters produce contravariant occurrences in the functor modelling methods, and hence cannot be dealt with a straightforward application of the coalgebraic methodology.

We extend Reichel-Jacobs coalgebraic description to generalized binary methods, i.e. methods whose type parameters include sums and possibly infinite products type constructors. Our focus of interest are equivalences on objects which are “well-behaved”, in the sense that they induce a canonical non-redundant model on the quotient of the given class. Therefore, such equivalences must be *congruences* w.r.t. method application. In this paper we show that canonical models can be built also for generalized binary methods using purely covariant tools. We propose two solutions. Our first solution applies to the case where we have already a class implementation. It is based on the observation that the behaviour of a generalized binary method can be captured by a bunch of unary methods obtained by *“freezing”*, in turn, the types of the class parameters to the states of the class implementation given at the outset, i.e. by viewing them as constant types. Our second solution is based on a set-theoretic understanding of functions, whereby binary methods in a class specification are viewed as *graphs* instead of functions. Thus contravariant function spaces in the functor are rendered as covariant sets of relations.

We prove that the behavioural equivalence induced by the *“freezing ap-*

proach" amounts to the *greatest congruence* w.r.t method application on the given class. As a by-product, we gain a (coalgebraic) coinduction principle for reasoning about such greatest congruence.

As far as the graph model, the behavioural equivalence is not a congruence, in general. Remarkably, we show that a necessary and sufficient condition for this to hold is that the graph and freezing equivalence coincide. As a consequence, when this is the case, we obtain a spectrum of coinduction principles for reasoning on the greatest congruence.

The interest of the graph approach goes beyond coalgebraic semantics, since it suggests a new way for solving the well-known problem of typing binary methods when subclasses are viewed as subtypes, see *e.g* [3].

In this paper, we work on a set-theoretic category, denoted by $\mathcal{C}$. For basic definitions and results on coalgebras we refer to [9].

In the literature, various authors have been considered the problem of the coalgebraic description of binary methods, see e.g. [12]. For an extensive comparison with the literature, see [6].

1 Generalized Binary Methods and Behavioural Equivalences

We call a method $m : X \times T_1 \times \ldots \times T_q \to T_0$ *generalized binary* if T_i ranges over the following grammar of types:

$$(\mathcal{T} \ni)\ T ::= \ X \mid K \mid T \times T \mid T + T \mid \Pi_K T,$$

where $X \in TVar$, is a variable for class types, and K is any constant type. Notice that the product type $\Pi_K T$ corresponds to the function space $K \to T$. That is, in a generalized binary method, we allow functional parameters, where variable types can appear only in *strictly positive* positions. For simplicity, in this paper we will consider only one class. There would be no additional conceptual difficulty in dealing with the general case.

A preliminary step in discussing equivalences induced on objects by generalized binary methods consists in extending the behavioural equivalence on objects of a class X to the whole structure of types in $\mathcal{T}$ over X. This is achieved through the *relational lifting* of [5]. In the definition below, by abuse of notation, we denote by X and T also their set-theoretic semantic counterparts.

Definition 1.1 [Relational Lifting] Let R^X be a relation on X, let $T \in \mathcal{T}$ be such that $Var(T) \subseteq \{X\}$. We define the extension $R^T \subseteq T \times T$ by induction on T as follows:

- if $T = K$, then $R^T = Id_{K \times K}$,
- if $T = T_1 \times T_2$, then $R^T = \{((a_1, a_2), (a'_1, a'_2)) \mid a_1 R^{T_1} a'_1 \wedge a_2 R^{T_2} a'_2\}$,
- if $T = T_1 + T_2$, then $R^T = \{((1, a), (1, a')) \mid a R^{T_1} a'\} \cup \{((2, a), (2, a')) \mid a R^{T_2} a'\}$,
- if $T = \Pi_K T_1$, then $R^T = \{(f, f') \in \Pi_K T_1 \mid \forall a \in K \implies fa R^{T_1} f' a\}$.

```
class spec : $Register$                    class R
    methods :                                  attributes :
      set : $X \times N \to X$                   $val : int$
      get : $X \to N$                          methods :
      eq  : $X \times X \to \mathbb{B}$           r.get = $\langle$r.$val$, r$\rangle$
    assertions :                                 r.set($n$) = $\langle$r$'$, r$'\rangle$
      r.set($n$).get = $n$                           where r$'$.$val = n$
      r$_1$.get = r$_2$.get $\Leftrightarrow$                r$_1$.eq(r$_2$) = if (r$_1$.get = r$_2$.get)
        r$_1$.eq(r$_2$) = $true$                            then $\langle true$, r$_1\rangle$
end class spec                                       else  $\langle false$, r$_1\rangle$
                                               end class
```

Table 1
Example of Class Specification and Class.

Definition 1.2 [Congruence] Let $\approx^X$ be an equivalence on objects of a class X and let $m : X \times T_1 \times \ldots \times T_q \to T_0$ be a method in X, then $\approx^X$ is a *congruence w.r.t.* m if $x \approx^X x'$ and $a_1 \approx^{T_1} a'_1 \ldots a_q \approx^{T_q} a'_q \Rightarrow x.m(\boldsymbol{a}) \approx^{T_0} x'.m(\boldsymbol{a'})$, where $\approx^{T_i}$ denotes the extension of $\approx^X$ to the type T_i, according to the definition above. (Notice the use of the "dot-notation" for method calls.)

2 Class Specifications and Class Implementations

Definition 2.1 A class specification S is a structure consisting of
- A finite set of *method declarations*
$$m : X \times T_1 \times \ldots \times T_q \to T_0 \ ,$$
- A finite set of *assertions*, regulating the behaviour of the objects belonging to the class.

The language for assertions is any first order language with constant symbols and function symbols for denoting constructors, methods and (extensions of) behavioural equivalences at all types. Typical assertions are equations, see e.g. [11] for more details.

In the lefthand part of Table 1, we present an example of a class specification, *Register*, which features the binary method *eq* for comparing the content of two registers.

A class (implementation) consists of attributes (fields), constructors and methods. Attributes and methods of a class can be either private or public. For simplicity, we assume all attributes to be private, and all methods to be public. We do not use a specific programming language to define classes, since we are working at a semantic level. Any programming language would do. In this perspective, the code corresponding to a method declaration $m : X \times \prod_{j=1}^{q} T_j \to T_0$ is given by a set-theoretic function $\alpha : X \times \prod_{j}^{q} T_j \to T_0 + Excp + 1$, since a method can possibly terminate with an exception or not terminate.

Definition 2.2 A class C *implements* a specification S if method declarations correspond, and their implementations satisfy the assertions in S.

In the righthand part of Table 1, we present the class R implementing the class specification, *Register*.

3 Coalgebraic Description of Objects and Classes: unary case

In this section, we illustrate the coalgebraic description of class specifications and class implementations in the case of unary methods. Following [10,7], we associate a functor to a class specification as follows:

Definition 3.1 Let S be a class specification with method declarations $m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}, \quad i = 1, \ldots, k$, where all methods are unary (i.e. $X \notin T_{ij}\ \forall j = 1, \ldots, q_i$). Then using currification the method declarations in S induce the functor $H : \mathcal{C} \to \mathcal{C}$ defined by

$$H \triangleq \prod_{i=1}^{k} \prod_{j=1}^{q_i} T_{ij} \to (T_{i0} + Excp + 1),$$

where $Excp$ denotes a set of exceptions/errors.

Notice that the functor H is covariant only if the method m is unary. Generalized binary methods, such as the method *eq* in the class specification *Register*, produce contravariant occurrences of X in the corresponding functor. In Section 4, we discuss how to overcome this problem.

The class implementations can be viewed as coalgebras as follows:

Definition 3.2 Let S be a class specification inducing a functor H. A *class* implementing S is an H-coalgebra satisfying the assertions in S.

On the other hand, given a concrete class, this induces a coalgebra for the functor determined by its method declarations, as follows:

Definition 3.3 i) A class $C = \langle \{f_i : T_i\}_{i=1}^{n}, \{m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}\}_{i=1}^{k} \rangle$ induces a coalgebra (X, α) for the functor H determined by the declarations of methods m_i, defined as follows:
- The carrier X is the set of *states* determined by the fields f_i.
- The coalgebra map $\alpha : X \to HX$ is defined by $\alpha \triangleq \langle \alpha_i \rangle_{i=1}^{k}$, where $\alpha_i : X \to H_i X$ is the function implementing the method m_i.

ii) An object of a class C is an element of the set of states X of C.

In the following lemma we characterize the behavioural equivalence on objects induced by the coalgebraic description of a class implementation. Such behavioural equivalence equates objects with the same behaviour under application of methods:

Lemma 3.4 *Let S be a class specification with method declarations $m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i_0}$, $i = 1, \ldots, k$, inducing the functor $H = \prod_{i=1}^{k} H_i$, let $(X, \langle \alpha_i \rangle_{i=1}^{k})$ be an H-coalgebra implementing S. Then the greatest H-bisimulation on $(X, \langle \alpha_i \rangle_i)$, $\approx_H$, can be characterized as follows:*

$$x \approx_H x' \iff \forall i.\ \forall \boldsymbol{a}.\ \alpha_i(x)(\boldsymbol{a}) \approx_H \alpha_i(x')(\boldsymbol{a}) \ ,$$

where, by abuse of notation, $\alpha_i(x)(\boldsymbol{a}) \approx_H \alpha_i(x')(\boldsymbol{a})$ denotes the extension of $\approx_H$ to the type T_{i0}, i.e. $\approx_H^{T_{i0}}$, according to Definition 1.1 of relational lifting.

4 Coalgebraic Description of Generalized Binary Methods

In this section, we show how to extend the coalgebraic model to generalized binary methods. Our first proposal (Section 4.1) applies when a concrete coalgebra (i.e. class implementation) is given. It is based on the observation that the behaviour of a generalized binary method can be simulated by a bunch of unary methods, each one determined by "freezing" all the occurrences of X in the parameter types and object type, but one. "Freezing" an occurrence of X means that X is replaced by the carrier, i.e. the set of states, of the given class. The behavioural equivalence thus obtained turns out to be the greatest congruence w.r.t. the original generalized binary method.

In Section 4.2, we present an alternative solution to the freezing functor. Here we turn contravariant occurrences in the type of parameters of a generalized binary method m into covariant ones simply by interpreting m as a *graph* instead of a function. To this aim, we introduce a new functor G (*graph functor*), where the function space is substituted by the corresponding space of *graph relations*.

The advantage of this latter solution w.r.t. the previous one is that this approach directly applies to specifications. Moreover, we do not have to use the intermediate step of the unary methods. The drawback is that the graph behavioural equivalence is not a congruence w.r.t. method application in general. However, there are many interesting situations where it is. In these cases a rich spectrum of conceptually independent coinduction principles is available. We discuss this issue in Section 4.3.

4.1 The Freezing Functor

We proceed in two passes.
First, we reduce a generalized binary method to a bunch of purely binary methods with the same observable behaviour. Let m be a generalized binary method. In particular, for each parameter of type $T_1 + T_2$, we can duplicate the method. In the first version, we will have a parameter of type T_1. In the second version the parameter will be of type T_2. Moreover, each parameter of type $\Pi_K T$ can be viewed as the product of $|K|$ parameters of type T. Thus,

by applying the above transformations to a generalized binary method, we get a (possibly infinite) set of purely binary methods $m : X \times \prod_{j \in J} T_j \to T_0$, where J is a possibly infinite set of indexes.

In the second pass, we reduce each purely binary method to a bunch of unary methods. Let C be a class implementation with set of states $\bar{X}$, including a purely binary method $m : X \times \prod_{j \in J} T_j \to T_0$, implemented by the function α. In order to recover the observable behaviour of the original method m, we need to consider a bunch of unary methods m_l, one for each class parameter, where m_l describes the behaviour of an object when it is used as l^{th} class parameter. Let I be the set of indexes corresponding to class parameters, including the object, for all $l \in I$, we define :

$$m_l : X \times (\prod_{j \in J} T_j[\bar{X}/X]) \to T_0, \quad \alpha_l : X \times (\prod_{j \in J} T_j[\bar{X}/X] \to T_0)$$

$$\alpha_l(x)(a_1, \dots, a_q) \triangleq \alpha(a_l)(a_1, \dots, a_{l-1}, x, a_{l+1}, \dots, a_q) \ .$$

Now we can define a coalgebraic model of the class implementation using purely covariant tools, as in Section 3, using the freezing functor F defined by:

Definition 4.1 [Freezing Functor] Let C be a class implementation with purely binary methods and set of states $\bar{X}$. The freezing functor determined by C is defined by

$$F \triangleq \prod_{i=1}^{k} F_i \ ,$$

where, for each unary method $m_i, F_i \triangleq H_i$, and for each binary method $m_i : X \times \prod_{j \in J_i} T_{ij} \to T_{i0}$ with class parameters in I_i, $F_i \triangleq \prod_{l \in I_i} F_{il}$, where $F_{il}X \triangleq (\prod_{j \in J} T_{ij})[\bar{X}/X] \to (T_{i0} + Excp + 1)$, for all $l_i \in I_i$.

The following definition of *1-ary method context* will be useful to characterize the behavioural equivalence induced by the freezing model:

Definition 4.2 [1-ary Method Context] Let C be a class. *A 1-ary method context*, $D[\ ]$, is a context with exactly one hole, whose top operator is a method in C, where the hole either corresponds to the object or to a parameter

The behavioural equivalence on objects induced by the freezing functor can then be characterized as follows:

Lemma 4.3 (Freezing Bisimulation and Coinduction Principle) *Let C be a class implementation with set of states $\bar{X}$, and let F be the freezing functor induced by C. Then the greatest F-bisimulation $\approx_F$, on the coalgebra determined by the methods in C, can be characterized as follows:*

$$x \approx_F x' \iff \forall D[\ ] \text{ 1-ary context. } D[x] \approx_F D[x']$$

We can now establish the result which motivates our treatment:

Theorem 4.4 *Let C be a class. Then for all methods $m_i : X \times \prod_{j \in J} T_{ij} \to T_{i0}$ in $C, \forall x\boldsymbol{a}, x'\boldsymbol{a'} \in X \times \prod_{j \in J} T_{ij}$, $x\boldsymbol{a} \approx_F x\boldsymbol{a'} \implies \alpha_i(x)(\boldsymbol{a}) \approx_F \alpha_i(x')(\boldsymbol{a'})$.*

Moreover, since any congruence is an F-bisimulation, by coinduction, one can then show that $\approx_F$ is the greatest congruence.

Theorem 4.5 *Let C be a class. Then the freezing behavioural equivalence $\approx_F$ induced on C is the greatest congruence w.r.t method application.*

4.2 The Graph Functor

Definition 4.6 [Graph Functor] The method declarations in S induce the *graph functor* $G : \mathcal{C} \to \mathcal{C}$ defined by

$$G \triangleq \prod_{i=1}^{k} G_i \ ,$$

where, for each unary method $m_i, G_i \triangleq H_i$ (see Definition 3.1), and for each generalized binary method $m_i : X \times \prod_{j\in J} T_{ij} \to T_{i0}$, $G_i \triangleq \mathcal{P}(\prod_{j\in J} T_{ij} \times (T_{i0} + Excp + 1))$.

Definition 3.3, which gives the coalgebra induced by a given class, extends immediately to the case of the graph functor. On the contrary, the extension to the graph functor of the definition of class implementation (Definition 3.2) requires more care. Namely class implementations shall be taken to be *functional* G-coalgebras.

The graph behavioural equivalence can be characterized in terms of *n-ary method contexts*, which are method contexts with holes for any class parameter.

Definition 4.7 [n-ary Method Context] Let C be a class, and let m be a method of C with n (generalized) class parameters including the object, implemented by α. The method m induces an *n-ary context*

$$D[\ ] = [\ ].\alpha(b_1, \ldots, b_k) \ ,$$

where $b_i = [\ ]$, if T_i is a (generalized) class type, otherwise, if T_i is a constant type, b_i is any argument of type T_i.

Lemma 4.8 (Graph Bisimulation and Coinduction Principle) *Let $G = \prod_{i=1}^{k} G_i$ be the functor induced by the method declarations in S, and let $(X, \langle\alpha_i\rangle_{i=1}^{k})$ be a G-coalgebra implementing S. Then the greatest G-bisimulation on $(X, \langle\alpha_i\rangle_{i=1}^{k})$, $\approx_G$, can be characterized as follows:*

$$x \approx_G x' \iff \forall D[\ ] n\text{-ary context.}\ \ \forall \boldsymbol{a}\exists\boldsymbol{a'}.(\boldsymbol{a} \approx_G \boldsymbol{a'}\ \&\ D[x,\boldsymbol{a}] \approx_G D[x,\boldsymbol{a'}])$$
$$\&\ \forall\boldsymbol{a'}\exists\boldsymbol{a}.(\boldsymbol{a} \approx_G \boldsymbol{a'}\ \&\ D[x,\boldsymbol{a}] \approx_G D[x,\boldsymbol{a'}]) \ .$$

Notice the alternation of quantifiers $\forall\exists$ in the definition of graph behavioural equivalence, due to the presence of the powerset in the graph functor.

The functor G has always a final coalgebra, see e.g. [1]. In general, it is not functional, and moreover the functionality property of a coalgebra is not preserved by the unique morphism into the final coalgebra. Therefore, the image of a class implementation under the final morphism is not guaranteed to be a class implementation. Thus we can lack minimal class implementations. In Section 4.3, we study conditions for the final morphism to preserve the functionality property, thus recovering minimal implementations.

4.3 Comparing Graph and Freezing Behavioural Equivalences

One can easily check that $\approx_F$ is a graph bisimulation, using reflexivity of $\approx_F$. Thus $\approx_F \subseteq \approx_G$. The converse inclusion does not hold in general. For example, this is the case for the class R' obtained from the class R of registers when we drop methods *get* and *set*, and we consider only method *eq*. Namely, for R', $\approx_G$ equates all pairs of registers, while $\approx_F$ is the identity relation on registers. Moreover, notice that in this case $\approx_G$ is *not* a congruence w.r.t. *eq*.

The following result is a fundamental tool for recovering $\approx_F = \approx_G$:

Theorem 4.9 *$\approx_G = \approx_F$ iff $\approx_G$ is a congruence w.r.t. the methods in the class.*

The equality $\approx_G = \approx_F$ on a functional G-coalgebra is equivalent to the fact that its image into the final coalgebra is still a functional coalgebra. Thus Theorem 4.9 above gives an answer to the problem of minimal class implementations for the graph functor, raised at the end of Section 4.2.

Another relevant consequence of the fact that $\approx_G = \approx_F$ is a simplified coinductive characterization of $\approx_F$, in terms of "head" contexts, where the hole is in head position, i.e. it corresponds to the target object:

Proposition 4.10 *If $\approx_G = \approx_F$, then*

$$x \approx_F x' \iff \forall D[\ ] \text{ head context. } D[x] \approx_F D[x'].$$

Theorem 4.9 above is all that we might want. However, in practice, it is useful to have also alternative sufficient conditions. The interested reader can see [6].

5 Relational types

The idea of treating binary methods as graphs, rather than as functions, can be fruitfully pursued to overcome the well-known problem arising when inheritance is combined with subtyping, see e.g. [3]. Namely, if we type binary methods with the usual arrow type, which is contravariant, we lose the property that subclasses are subtypes. We propose to introduce a new type constructor, i.e. the *relation type*, and use this to type binary methods in class declarations. Since relation types are purely covariant, the subtyping property is maintained by subclasses. Binary methods can still be typed also with the standard arrow type, which is a subtype of the corresponding relation type, see Table 2. To preserve safety, contrary to arrow types, we assume

$$\frac{x:\alpha \quad M:\beta}{\lambda x^{\alpha}.M:\alpha\otimes\beta} \qquad \frac{\alpha\leq\alpha' \quad \beta\leq\beta'}{\alpha\otimes\beta\leq\alpha'\otimes\beta'} \qquad \frac{}{\alpha\to\beta\leq\alpha\otimes\beta}$$

Table 2
Typing rules for *Relational Types* $\otimes$

relation types not to be "applicable" i.e. there is no relational counterpart to the rule : $\frac{M:\alpha\to\beta \quad N:\alpha}{MN:\beta}$. This solution to the problem of typing binary methods is quite simple, and it allows for *single dispatching* in method calls. Moreover, contrary to other proposals, our proposal allows for "future code extensions" without losing the subtyping property of classes. We will study this proposal in a future paper.

References

[1] P.Aczel. *Non-wellfounded sets*, CSLI Lecture Notes **14**, Stanford 1988.

[2] Aczel P., N.Mendler. A Final Coalgebra Theorem, *CTCS*, D.H.Pitt et al. eds., Springer LNCS **389**, 1989, 357–365.

[3] Bruce K.B., Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, Benjamin C. Pierce On Binary Methods, *TAPOS* **1**(3), 1995, 221-242.

[4] Forti M., F.Honsell. Set-theory with free construction principles, *Ann. Scuola Norm. Sup. Pisa*, Cl. Sci. (4)**10**, 1983, 493–522.

[5] Hermida C., B.Jacobs. Structural induction and coinduction in a fibrational setting, Information and Computation, 1998, 145(2):107-152.

[6] Honsell F., M.Lenisa, R.Redamalla. Coalgebraic Description of Generalized Binary Methods, Technical Report, University of Udine.

[7] Jacobs B.. Objects and Classes, co-algebraically, *Object-Orientation with Parallelism and Book Persistence*, B.Freitag et al. eds., Kluwer Academic Publishers, 1996, 83–103.

[8] Jacobs B. Behaviour-refinement of object-oriented specifications with coinductive correctness proofs, *TAPSOFT'97*, M.Bidoit et. al. eds., Springer LNCS **1214**, 1997, 787–802.

[9] Jacobs B., J.Rutten. A tutorial on (co)algebras and (co)induction, *Bulletin of the EATCS* **62**, 1996, 222–259.

[10] Reichel H. An approach to object semantics based on terminal co-algebras, *MSCS* **5**, 1995, 129-152.

[11] Rothe J., H. Tews and B. Jacobs. The Coalgebraic Class Specification Language CCSL, Journal of Universal Computer Science,**7**(2001), pp.175-193.

[12] Tews H. Coalgebraic Methods for Object-Oriented Specifications, Ph.D. thesis, Dresden Univ. of technology, 2002.

DCM 2005 Preliminary Version

Quantum Communication and Cryptography: Introductory Concepts and State-of-the-Art

Raja Nagarajan

Department of Computer Science
The University of Warwickz
Coventry CV4 7AL

Abstract

The novel field of quantum computing and quantum information has gathered significant impetus in the last few years, and it has the potential to radically impact the future of information technology. While the successful construction of a large-scale computer may be some years away, secure communication involving quantum cryptography has recently been demonstrated in a scenario involving banking transactions in Vienna, and practical equipment for quantum cryptography is commercially available.

In this tutorial, I shall introduce a few basic concepts of quantum information processing, and discuss quantum communication and cryptographic protocols. I will also give an overview of the state-of-the-art in practical quantum cryptography.

The major selling point of quantum cryptography is unconditional security. But can this be guaranteed in practice? Even when protocols have been mathematically proved to be secure, it is notoriously difficult to achieve robust and reliable implementations of secure systems. I will highlight techniques that can be used for formal modelling and analysis of quantum protocols and their implementations.

On Reversible Combinatory Logic

Alessandra Di Pierro[1]

Dipartimento di Informatica, Universitá di Pisa, Italy

Chris Hankin and Herbert Wiklicky[1,1]

Department of Computing, Imperial College London, United Kingdom

Abstract

The λ-calculus is destructive: its main computational mechanism – beta reduction – destroys the redex and makes it thus impossible to replay the computational steps. Recently, reversible computational models have been studied mainly in the context of quantum computation, as (without measurements) quantum physics is inherently reversible. However, reversibility also changes fundamentally the semantical framework in which classical computation has to be investigated. We describe an implementation of classical combinatory logic into a reversible calculus for which we present an algebraic model based on a generalisation of the notion of group.

1 Introduction

It has been suggested, e.g. [11], that the standard model for computation as embodied in Turing Machines answers the problem of what constitutes a "computational procedure" in Hilbert's 10th Problem by reference to mental arithmetic as practised in previous times by European school children, accountants and waiters. This "waiter's arithmetic" is non-reversible and destructive. It is open to speculation whether a culture based on reversible computation like an abacus would have developed a different basic computational model. Quantum computation and the need for minimal energy loss make reversible computation once again interesting, see e.g. [15]. This has been the motivation for van Tonder [14] who presents a reversible applied lambda calculus (with quantum constants); his operational semantics provided the inspiration for the operational semantics of our reversible version **rCL** of Combinatory Logic. On the other hand, the set of combinators that we consider here have

[1] The authors are partly funded by the EPSRC project S77066A "Quantitative Analysis of Computational Resources".

also been studied by Abramsky [1,2], although with a different motivation, namely the links between reversible calculus and linear logic.

Our main motivation for investigating a reversible version of Combinatory Logic is ultimately the development of a denotational semantics of (probabilistic versions of) the λ-calculus reflecting the operational semantics we introduced in [10]. This kind of semantics is based on linear operator algebras and aims to support a compositional approach to (probabilistic) program analysis. The close relationship between reversibility and certain important classes of linear operators – in particular unitary and normal operators – was the starting point of a deeper investigation of the structure of reversible computation.

Reversibility naturally introduces a notion of symmetry into computation and is therefore strongly related to the theory of groups and their action, which are considered by most mathematicians as being virtually synonymous of symmetry [16]. Starting from the work on invertible lambda terms we could then use the group of permutations for the classical λ-calculus as our mathematical base for investigating compositionality issues in the static analysis of complex systems. However, the notion of automorphism associated to group is in some sense too "trivial" to characterise the symmetry involved in a reversible computation; it turns out that the structure of these objects can be better characterised algebraically by using *groupoids* and not just groups. In fact, the extension from groups to groupoids was formally introduced to describe reversible processes which may traverse a number of states. These cannot be easily captured by using group theory as this only allows us to characterise processes which start from one point and (possibly after a number of steps) come back to the same point. On the contrary, in groupoid theory processes can have different start and end points but they can be composed if and only if the starting point of one process is the end point of the previous one. Intuitively, a groupoid can be thought of as a group with many identities [6]. It is interesting to note that according to Connes, quantum mechanics was discovered by considering the groupoid of quantum transitions rather than the group of symmetry [7].

2 The Groupoid Structure of Reversible Computations

Our approach to reversible computation is based on a particular algebraic model of computation which naturally reflects the operational meaning of term reduction and its reverse process. This model is based on the notion of a *groupoid*. A groupoid, also known as a *virtual group*, is an algebraic structure introduced by Brandt [5] (for further details see e.g. [13,16,12,6]).

Definition 2.1 A *groupoid* with *base* $\mathcal{B}$ is a set $\mathcal{G}$ with mappings α and β from $\mathcal{G}$ onto $\mathcal{B}$, a partially defined binary operation $(g,h) \mapsto g \cdot h = gh$, and an inverse map $g \mapsto g^{-1}$ from $\mathcal{G}$ to $\mathcal{G}$ satisfying the following conditions:

(i) gh is defined whenever $\beta(g) = \alpha(h)$, and in this case $\alpha(gh) = \alpha(g)$ and $\beta(gh) = \beta(h)$.

(ii) if gh and hk are defined then so are $(gh)k$ and $g(hk)$ and they are equal (associativity).

(iii) For each $g \in \mathcal{G}$, there are left- and right-identity elements λ_g and ϱ_g satisfying $\lambda_g g = g = g\varrho_g$.

(iv) Each $g \in \mathcal{G}$ has an inverse g^{-1} satisfying $g^{-1}g = \varrho_g$ and $gg^{-1} = \lambda_g$.

An important property of groupoids is that $\alpha(g)$ and λ_g ($\beta(g)$ and $\rho(g)$) determine one another, that is there is a bijection between the base $\mathcal{B}$ and the set of all the identity elements of $\mathcal{G}$. This implies that identities are essentially unique, or in other words there is only one object which is "composable" on the left (right) with a given object g.

Consider a set of 'computational processes' $\mathcal{C} = \{C_i\}_i$, such as for example those specified via some programs in a formal programming language. Among the elements in $\mathcal{C}$ we would usually expect to find a 'neutral' or 'identity computation', that is intuitively a computational process which does not change any input state, such as for example a **skip** statement in a **while** language. We would also usually be able to define a way to compose two processes sequentially, in the sense that we can feed the result of one computation C_1 into another computation C_2 by obtaining a new one usually denoted by $C_1; C_2$ or $C_2 \circ C_1$. Moreover, In order for the computation be reversible we need a notion of symmetry. These general ideas which essentially identify reversible computation with some kind of (symbolic) dynamical process establish a groupoid structure on $\mathcal{C}$. In order to see this more precisely, consider a calculus with a notion of "reduction" i.e. a transition relation $\longrightarrow$ between terms $T, \ldots \in \mathcal{T}$, and its transitive and reflexive closure $\twoheadrightarrow$. Define a computational groupoid $\mathcal{G} = \mathcal{G}(\mathcal{T}, \twoheadrightarrow)$ as follows:

- $\mathcal{G} \subseteq \mathcal{T} \times \mathcal{T}$ with $(T, T') \in \mathcal{G}$ iff $T \twoheadrightarrow T'$.
- $\mathcal{B} = \mathcal{T}$
- $\alpha((T, T')) = T$ and $\beta((T, T')) = T'$
- $(T, T') \cdot (T', T'') = (T, T'')$
- $\lambda_{(T,T')} = (T, T)$ and $\varrho_{(T,T')} = (T', T')$
- $(T, T')^{-1} = (T', T)$.

Intuitively, we can reverse a computation for a term $T \in \mathcal{T}$ if we keep information about its 'history', i.e. information about the transition steps that have been performed during the computation. A highly expensive way to make the transition relation $\longrightarrow$ reversible is to use as history all strings $H \in \mathcal{H} = \mathcal{T}^*$ of terms in $\mathcal{T}$ and replace each transition $T_1 \longrightarrow T_2$, by $\langle T_1 \mid H \rangle \rightarrowtail \langle T_2 \mid HT_1 \rangle$ for all $H \in \mathcal{H}$. In this way we record the complete history of the previous terms and it is easy to see that computation is now reversible, i.e. we can reverse a derivation path until we reach the initial term.

In general, we might be interested in a more "efficient" way of recording the derivation history of a term. However, it depends on the nature and structure of the original calculus what information the history has to record; for example, in Van Tonder's λ-calculus [14] the history keeps track only of the substitutions which take place in each β-reduction step.

3 Combinatory Logic

Combinatory Logic (**CL**) is a formalism which (similarly to the λ-calculus) was introduced to describe functions and certain primitive ways to combine them to form other functions. With respect to the λ-calculus it has the advantage that is variable free; this allows one to avoid all the technical complications concerned with substitutions and congruence. It has on the other hand the disadvantage of being less intuitive than the λ-notation. For the purpose of this work we have opted for this more involved formalism because it allows for a more agile treatment and definition of our notion of reversible computation.

Definition 3.1 [Combinatory Logic Terms] The set of combinatory logic terms, **CL**-terms, over a finite or infinite set of constants containing K and S and an infinite set of variables is defined inductively as follows:

(i) all variables and constants are **CL**-terms,

(ii) if X and Y are **CL**-terms, then (XY) is a **CL** term.

The two combinators S and K form a common basis for combinatory logic. However, other sets of basic combinators can be defined. We will use the base consisting of four basic operations encoded in the combinators B (implementing bracketing), C (elementary permutations), W (duplication), and K (for deletion) which are defined as follows (cf [8, p379]):

$$\mathsf{K} \equiv \lambda xy.x, \ \mathsf{W} \equiv \lambda xy.xyy, \ \mathsf{C} \equiv \lambda xyz.xzy, \ \mathsf{B} \equiv \lambda xyz.x(yz).$$

Importantly, we can use B, W and C to implement the common combinator S (cf [8, p155]):

$$\mathsf{S} \equiv \mathsf{B}(\mathsf{B}(\mathsf{B}\mathsf{W})\mathsf{C})(\mathsf{B}\mathsf{B}).$$

In order to generate equalities provable in this calculus we use a notion of reduction similar to the *weak reduction* for the SK-calculus [3]. This is defined as the smallest extension of the relation on **CL**-terms induced by the basic operators which is compatible with application.

Definition 3.2 The reduction relation $\rightarrowtail$ on **CL**-terms is defined by the following rules:

(i) $\mathsf{K}XY \rightarrowtail X$,	(iv) $\mathsf{B}XYZ \rightarrowtail X(YZ)$,
(ii) $\mathsf{W}XY \rightarrowtail XYY$,	(v) $X \rightarrowtail X'$ implies $XY \rightarrowtail X'Y$,
(iii) $\mathsf{C}XYZ \rightarrowtail XZY$,	(vi) $X \rightarrowtail X'$ implies $YX \rightarrowtail YX'$,

We will denote by $\twoheadrightarrow$ the reflexive transitive closure of $\rightarrowtail$.

The relation between the λ-calculus and **CL** is a standard result (cf. [3, p156]). With reference to the standard base $\{\mathsf{S}, \mathsf{K}\}$ there is a canonical encoding $(\ )_{\mathbf{CL}}$ of λ terms in **CL** terms. It is a well known result that in presence of a rule for extensionality the two theories λ-calculus and **CL**(which are in general not equivalent) become equivalent (cf.[3, Def 7.3.14]).

3.1 Invertible Terms

The assumption of extensionality is also essential in the investigation of invertibility, as shown in [9,4] in the context of λ-calculus.

Within the theory **CL+ext** that is **CL** extended with the rule (cf [3, Def 7.1.10]):

$$Px = P'x \text{ with } x \notin FV(PP') \text{ implies } P = P',$$

we can characterise the *invertible* combinatory logic terms. We first observe that a semi-group structure on the extended theory **CL+ext** is given by defining a composition of terms by means of the B combinator as

$$X \cdot Y = \mathsf{B}XY$$

as for all Z we get $(X \cdot Y)Z = \mathsf{B}XYZ = X(YZ)$. This operation is associative and can be seen as implementing 'sequential' or 'functional composition'. In the λ-calculus it is defined by

$$M \cdot N = \lambda z.M(Nz)$$

for any two λ-terms M, N.

Moreover, we can take the I combinator as the identity; in the λ-calculus this is given, for example, by the term $\lambda x.x$.

Naturally, the question arises which terms of a calculus like **CL+ext** form a group, i.e. for which terms X we have an element X^{-1} (the inverse) such that

$$X \cdot X^{-1} = X^{-1} \cdot X = \mathsf{I}.$$

The classically invertible **CL** terms are all those terms X for which there is a Y such that $\mathsf{B}XY = \mathsf{B}YX = \mathsf{I}$ holds (cf also [8, Sect 5.D.5 and Def 5.D.1]). A very simple example of an invertible term is the identity combinator I which is its own inverse. In fact, we have that $\mathsf{I} \cdot \mathsf{I} = \mathsf{BII} = \mathsf{I}$. However, in calculi without extensionality this might be about the only example of an invertible term. According to [3, Section 21.3] the invertible terms in the λ-calculus (without extensionality) form the trivial group $\{\mathsf{I}\}$. Extensionality is therefore needed to obtain some non-trivial invertible elements. It allows us to show for example that $\mathsf{C} = \mathsf{C}^{-1}$, i.e. C is its own inverse. This is intuitively clear as the combinator C is essentially representing a transposition of its 2nd and 3rd argument and permutations are reversible. Dezani [9] and Bergstra and Klop [4] have studied the problem of how to describe the invertible elements in different calculi and theories. This also resulted in a description of the group of all invertible elements in the $\lambda\eta$-calculus (cf. [3, Ch 21]).

Contrary to the classical approach we will define a calculus which is reversible in the sense that all reductions in the calculus are invertible. The new reversible calculus will be an extension of the **CL+ext** theory, so that all classical **CL+ext** reductions will still be reductions in the new calculus.

4 Reversible Combinatory Logic

Providing a mechanism to record the computational history of a term allows us to define a reversible version of **CL**, which we will call **rCL**.

Formally, we define a reversible combinatory logic term, or **rCL**-term, as a pair $\langle M \mid H\rangle$, where M is a classical **CL**-term, which we refer to as the *proper term*, and H is a list of elements S which record the reduction steps S (forward execution) and their expansion steps $\overline{S}$ (backward execution). We refer to H as the *history term* and define its syntax by

$$H ::= \varepsilon \mid S : H \mid \overline{S} : H$$
$$S ::= T\mathsf{K}_n^m \mid \mathsf{W}_n^m \mid \mathsf{B}_n^m \mid \mathsf{C}_n^m$$

with T a classical **CL**-term and $n, m \in \mathbb{N}$. We denote by $\mathcal{H}$ be the set of all history terms. The meaning of the two numbers n and m is to record the exact point in the term in which the combinator, i.e. its corresponding reduction rule, is applied, and the length of prefix of the reduced term, respectively. This information is important to guarantee a unique replay of all reduction steps. We will often omit ε and use blank to represent the empty history. We will denote by $S + l$ with $l \in \mathbb{N}$ a history step where the position reference is increased by l, e.g. $T\mathsf{K}_n^m + l = T\mathsf{K}_{n+l}^m$ and by $H + l$ a position shift applied to a whole history, i.e. $H + l = S_1 + l : S_2 + l : \ldots : S_k + l$.

Formally, we define the function *len* on classical **CL**-terms by:

$$len(X) = \begin{cases} 1 & \text{if } X \text{ is a constant or variable} \\ n + m & \text{if } X = (YZ) \text{ with } len(Y) = n \text{ and } len(Z) = m. \end{cases}$$

The reversible (forward) reduction relation on **rCL** is defined by:

(i) $\langle \mathsf{K}XY \mid \rangle \rightarrowtail \langle X \mid Y\mathsf{K}_0^{len(X)}\rangle$, (iii) $\langle \mathsf{C}XYZ \mid \rangle \rightarrowtail \langle XZY \mid \mathsf{C}_0^{len(X)}\rangle$,

(ii) $\langle \mathsf{W}XY \mid \rangle \rightarrowtail \langle XYY \mid \mathsf{W}_0^{len(X)}\rangle$, (iv) $\langle \mathsf{B}XYZ \mid \rangle \rightarrowtail \langle X(YZ) \mid \mathsf{B}_0^{len(X)}\rangle$,

The reversible (backward) reduction relation on **rCL** is defined by:

(i) $\langle X \mid \rangle \rightarrowtail \langle \mathsf{K}XY \mid \overline{Y\mathsf{K}_0^{len(X)}}\rangle$, (iii) $\langle XZY \mid \rangle \rightarrowtail \langle \mathsf{C}XYZ \mid \overline{\mathsf{C}}_0^{len(X)}\rangle$,

(ii) $\langle XYY \mid \rangle \rightarrowtail \langle \mathsf{W}XY \mid \overline{\mathsf{W}}_0^{len(X)}\rangle$, (iv) $\langle X(YZ) \mid \rangle \rightarrowtail \langle \mathsf{B}XYZ \mid \overline{\mathsf{B}}_0^{len(X)}\rangle$,

Additionally the assume the following structural rules:

(i) $\langle X \mid \rangle \rightarrowtail \langle X' \mid H'\rangle$ implies $\langle XY \mid \rangle \rightarrowtail \langle X'Y \mid H'\rangle$,

(ii) $\langle X \mid \rangle \rightarrowtail \langle X' \mid H'\rangle$ implies $\langle YX \mid \rangle \rightarrowtail \langle YX' \mid H' + len(Y)\rangle$,

(iii) $\langle X \mid \rangle \rightarrowtail \langle X' \mid H'\rangle$ implies $\langle X \mid H\rangle \rightarrowtail \langle X' \mid H : H'\rangle$.

(iv) $\langle X \mid \overline{H} : H\rangle \rightarrowtail \langle X \mid \rangle$ and $\langle X \mid H : \overline{H}\rangle \rightarrowtail \langle X \mid \rangle$.

The last two rules allows us to go back to the starting point by reversing the history. For example:

$$\langle \mathsf{W} \mid \rangle \rightarrowtail \langle \mathsf{KWB} \mid \overline{\mathsf{BK}_0^1}\rangle \rightarrowtail \langle \mathsf{W} \mid \overline{\mathsf{BK}_0^1} : \mathsf{BK}_0^1\rangle \rightarrowtail \langle \mathsf{W} \mid \rangle, \text{ and}$$
$$\langle \mathsf{KWB} \mid \rangle \rightarrowtail \langle \mathsf{W} \mid \mathsf{BK}_0^1\rangle \rightarrowtail \langle \mathsf{KWB} \mid \mathsf{BK}_0^1 : \overline{\mathsf{BK}_0^1}\rangle \rightarrowtail \langle \mathsf{KWB} \mid \rangle$$

This also shows that the histories $\overline{\mathsf{BK}_0^1}$ and BK_0^1 are (right and left) inverses of each other (cf. Section 4.1). We denote by $\twoheadrightarrow$ the reflexive and transitive closure of $\rightarrowtail$.

Example 4.1 Without the position references the following two terms reduce to the same term:

$$\langle \mathsf{K(CW)C} \mid \rangle \rightarrowtail \langle \mathsf{CW} \mid \mathsf{CK}\rangle \text{ and } \langle \mathsf{KCCW} \mid \rangle \rightarrowtail \langle \mathsf{CW} \mid \mathsf{CK}\rangle$$

It is therefore impossible to tell where $\langle \mathsf{CW} \mid \mathsf{CK}\rangle$ came from. However with position information we have

$$\langle \mathsf{K(CW)C} \mid \rangle \rightarrowtail \langle \mathsf{CW} \mid \mathsf{CK}_0^2\rangle \text{ and } \langle \mathsf{KCCW} \mid \rangle \rightarrowtail \langle \mathsf{CW} \mid \mathsf{CK}_0^1\rangle$$

The position information also allows us to encode different reduction strategies ($n = 0$ indicates left-most reduction) as in the following example.

Example 4.2 Let us consider the classical term $\mathsf{W}(\mathsf{B}XYZ)\mathsf{K}$. It has two possible reduction paths which are reflected in the history terms:

$$\langle \mathsf{W}(\mathsf{B}XYZ)\mathsf{K} \mid \rangle \rightarrowtail \langle (\mathsf{B}XYZ)\mathsf{KK} \mid \mathsf{W}_0^4\rangle \rightarrowtail \langle (X(YZ))\mathsf{KK} \mid \mathsf{W}_0^4 : \mathsf{B}_0^1\rangle \text{ and}$$
$$\langle \mathsf{W}(\mathsf{B}XYZ)\mathsf{K} \mid \rangle \rightarrowtail \langle (\mathsf{W}(X(YZ))\mathsf{K} \mid \mathsf{B}_1^1\rangle \rightarrowtail \langle (X(YZ))\mathsf{KK} \mid \mathsf{B}_1^1 : \mathsf{W}_0^3\rangle$$

Classical combinatory logic can be embedded in **rCL** by representing any **CL**-term M with a **rCL**-term T of the form $\langle M \mid \varepsilon\rangle$. We can show that the weak reduction relation for **CL**-terms can be simulated by the reversible reduction relation on **rCL**. This is implied by the following more general result.

Proposition 4.3 *For every $M \in$ **CL** we have:*

$$M \twoheadrightarrow N \textit{ or } N \twoheadrightarrow M \textit{ iff } \forall H \in \mathcal{H}\ \exists H' \in \mathcal{H} :\ \langle M \mid H\rangle \twoheadrightarrow \langle N \mid H'\rangle.$$

4.1 The History Group $\mathcal{H}$

For a history $H = S_1 : S_2 : \ldots : S_{n-1} : S_n \in \mathcal{H}$, we define its formal inverse

$$\overline{H} = \overline{S_1 : S_2 : \ldots : S_{n-1} : S_n} = \overline{S_n} : \overline{S_{n-1}} : \ldots : \overline{S_2} : \overline{S_1}$$

with the following properties: (i) $\overline{\overline{H}} = H$ and (ii) $H : \overline{H} = \varepsilon$.

It is easy to see that by construction the set of histories $\mathcal{H}$ forms a group with with respect to the composition operation ":".

The inverse of a history and the inverse of a classical **CL** term, if it exists, are closely related. The inverse history can, to a certain degree, simulate the effects of the inverse term. In order to establish this relation, we first show how the group structure of the history terms interacts with the reversible reduction rules introduced before.

Lemma 4.4 *Let X be a classical* **CL** *term, and let $H \in \mathcal{H}$. Then*

$$\langle X \mid \rangle \rightarrowtail\!\!\!\twoheadrightarrow \langle X' \mid H \rangle \text{ iff } \langle X' \mid \rangle \rightarrowtail\!\!\!\twoheadrightarrow \langle X \mid \overline{H} \rangle.$$

Proof. As $\overline{H} : H \equiv H : \overline{H} = \varepsilon$, we have

$$\langle X \mid \overline{H} \rangle \rightarrowtail\!\!\!\twoheadrightarrow \langle X' \mid \overline{H} : H \rangle \equiv \langle X' \mid \rangle$$

and thus by the reversible backward reduction rules

$$\langle X' \mid \rangle \rightarrowtail\!\!\!\twoheadrightarrow \langle X \mid \overline{H} \rangle.$$

□

We can now show that for classical invertible terms M, histories can be used to simulate a reduction for the inverse M^{-1} given a reduction for M.

Proposition 4.5 *Let M be an invertible term in* **CL**. *Given a history $H \in \mathcal{H}$ and two* **CL** *terms N_1 and N_2 such that*

$$\langle M N_1 \mid \rangle \rightarrowtail\!\!\!\twoheadrightarrow \langle N_2 \mid H \rangle.$$

Then there exist $H', H'' \in \mathcal{H}$ such that

$$\langle M^{-1} N_2 \mid H'' \rangle \rightarrowtail\!\!\!\twoheadrightarrow \langle N_1 \mid H' \rangle.$$

Proof. By Lemma 4.4 and the hypothesis we have that $\langle N_2 \mid \rangle \rightarrowtail\!\!\!\twoheadrightarrow \langle M N_1 \mid \overline{H} \rangle$. Using the history $\overline{H}$ and again Lemma 4.4, we get

$$\langle M^{-1} N_2 \mid \overline{H} + \mathit{len}(M^{-1}) \rangle \rightarrowtail\!\!\!\twoheadrightarrow \langle M^{-1} M N_1 \mid \rangle.$$

Therefore

$$\langle M^{-1} N_2 \mid \overline{H} + \mathit{len}(M^{-1} + 1 : \mathsf{B}_0^{\mathit{len}(M^{-1}} \rangle \rightarrowtail \langle \mathsf{B} M^{-1} N_2 \mid \overline{H} + \mathit{len}(M^{-1} + 1 \rangle$$
$$\rightarrowtail\!\!\!\twoheadrightarrow \langle \mathsf{B} M^{-1} M N_1 \mid \rangle \overset{\text{def}}{=} \langle (M^{-1} \cdot M) N_1 \mid \rangle \rightarrowtail\!\!\!\twoheadrightarrow \langle N_1 \mid H' \rangle.$$

□

4.2 The Groupoid of Reversible Computations

Given a group G and a set X, a *group action* of G on X is defined as a homomorphism π of G into the automorphism group of X, i.e. $\pi(g) \in \mathrm{Aut}(X)$ such that $\pi(e) = \mathrm{id}$, and $\pi(gh)(x) = \pi(g)(\pi(h)(x))$. Given a group action π of G on X we can define a groupoid $\mathcal{G} = \mathcal{G}(X, G, \pi)$ as follows:

- $\mathcal{G} \subseteq X \times G \times X$ with $(x, g, y) \in \mathcal{G}$ iff $\pi(g)(x) = y$.
- $\mathcal{B} = X$
- $\alpha((x, g, y)) = x$ and $\beta((x, g, y)) = y$
- $(x, g, y) \cdot (y, h, z) = (x, hg, z)$
- $\lambda_{(x,g,y)} = (x, e, x)$ and $\varrho_{(x,g,y)} = (y, e, y)$
- $(x, g, y)^{-1} = (y, g^{-1}, x)$.

We show that the set of reversible computations is the groupoid defined by the action of the history group $\mathcal{H}$ on the set of **rCL** terms. Intuitively, this

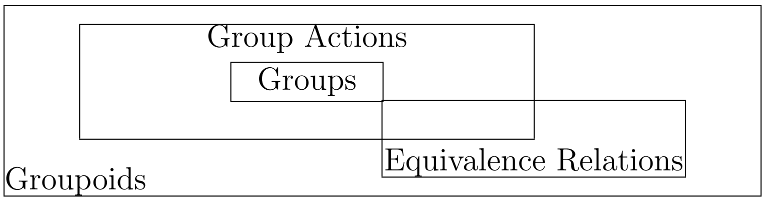


Fig. 1. Groups, Group Actions and Equivalence Relations

means that each history term determines a permutation on **rCL** corresponding to a reversible computation, and vice versa.

Consider the groupoid $\mathcal{G}$ defined by the action π of $\mathcal{H}$ on **rCL** given by

$$\pi(H)(\langle M \mid H' \rangle) = \langle M \mid H' : H \rangle$$

The computational groupoid $\mathcal{G}(\mathbf{rCL}, \rightarrowtail\!\!\!\twoheadrightarrow)$ constructed as in Section 2 using the transition relation $\rightarrowtail\!\!\!\twoheadrightarrow$ on **rCL** terms, and the group action groupoid $\mathcal{G}(\mathbf{rCL}, \mathcal{H}, \pi)$ defined above are isomorphic. The isomorphism is given by simply forgetting about the "connecting history".

Proposition 4.6 *The map* $\delta : \mathcal{G}(\mathbf{rCL}, \mathcal{H}, \pi) \to \mathcal{G}(\mathbf{rCL}, \rightarrowtail\!\!\!\twoheadrightarrow)$ *which is defined as* $\delta(\langle T, H, T' \rangle) = \langle T, T' \rangle$*, i.e.* $\delta = (\alpha, \beta)$*, is a groupoid isomorphism.*

5 Conclusion

We have introduced a reversible version **rCL** of Combinatory Logic where terms are enriched with a "history" part which allows us to uniquely "replay" every computational step. We have utilised the structure of a *groupoid* to model computation in **rCL**.

Groupoids can bee seen as a generalisation of several mathematical structures, such as *groups*, *group actions* and *equivalence relations*, as shown in Figure 1 (cf. [12]). The last two structures are particularly relevant for our treatment of **rCL**. In fact, the computational paths of a reversible calculus can be seen as the orbits of a group acting on some space, in our case the history group acting on the space of **rCL** terms. On the other hand, the equational theory of a calculus introduces an equivalence relation on the terms. Groupoids are therefore able to accommodate the operational semantics as well as the equational theory of **rCL**.

Further work will concentrate on constructing a denotational semantics for **rCL** based on the groupoid structure presented here. Our aim is in a compositional definition of transition operators which serves as a basis for semantics-based analysis techniques for the λ-calculus. For this we hope to exploit well-established results on the relation between operator algebras (in particular C* algebras) and groupoids [13]. Furthermore, we believe that reversible combinatory logic can in principle be used for a (maybe highly inefficient) translation of classical into quantum computation.

References

[1] Abramsky, S., *A structural approach to reversible computation*, in: *Proceedings of LCCS 2001*, 2001, pp. 1–16.

[2] Abramsky, S., E. Haghverdi and P. Scott, *Geometry of interaction and linear combinatory algebras*, Mathematical Structures in Computer Science **12** (2002), pp. 625–665.

[3] Barendregt, H. P., "The Lambda Calculus," Studies in Logic and the Foundations of Mathematics **103**, North-Holland, 1984, revised edition.

[4] Bergstra, J. and J. W. Klop, *Invertible terms in the lambda calculus*, Theoretical Computer Science **11** (1980), pp. 19–37.

[5] Brandt, W., *Über eine Verallgemeinerung des Gruppengriffes*, Mathematische Annalen **96** (1926), pp. 360–366.

[6] Brown, R., *From groups to groupoids: a brief survey*, Bull. London Math. Soc. **19** (1987), pp. 113–134.

[7] Connes, A., "Noncommutative Geometry," Academic Press, San Diego, 1994.

[8] Curry, H. B. and R. Feys, "Combinatory Logic," North-Holland, 1958.

[9] Dezani-Ciancaglini, M., *Characterization of normal forms possesing inverse in the $\lambda-\beta-\eta$-calculus*, Theoretical Computer Science **2** (1976), pp. 323–337.

[10] Di Pierro, A., C. Hankin and H. Wiklicky, *Probabilistic lambda-calculus and quantitative program analysis*, Journal of Logic and Computation **15** (2005), pp. 159–179.

[11] Mundici, D. and W. Sieg, *Paper machines*, Philosophica Mathematica **Series III, 3** (1995), pp. 5–30.

[12] Ramsay, A. and J. Renault, editors, "Groupoids in Analyis, Geometry, and Physics," Contemporary Mathematics **282**, AMS, Providence, RI, 2001.

[13] Renault, J., "A Groupoid Approach to C*-Algebras," Lecture Notes in Mathematics **793**, Springer Verlag, Berlin – Heidelberg – New York, 1980.

[14] van Tonder, A., *A lambda calculus for quantum computation*, SIAM Journal of Computation **33** (2004), pp. 1109–1135.

[15] Vitanyi, P., *Time, space, and energy in reversible computing*, in: *Proceedings of the ACM International Conference on Computing Frontiers*, 2005.

[16] Weinstein, A., *Groupoids: Unifying internal and external symmetry*, Notices of the AMS **43** (1996), pp. 744–752.

Classically-controlled Quantum Computation

Simon Perdrix [1] Philippe Jorrand [2]

Leibniz Laboratory
IMAG-INPG
Grenoble, France

Abstract

It is reasonable to assume that quantum computations take place under the control of the classical world. For modelling this standard situation, we introduce a Classically-controlled Quantum Turing Machine (CQTM) which is a Turing machine with a quantum tape for acting on quantum data, and a classical transition function for a formalized classical control. In CQTM, unitary transformations and quantum measurements are allowed. We show that any classical Turing machine is simulated by a CQTM without loss of efficiency. Furthermore, we show that any k-tape CQTM is simulated by a 2-tape CQTM with a quadratic loss of efficiency. The gap between classical and quantum computations which was already pointed out in the framework of measurement-based quantum computation (see [14]) is confirmed in the general case of classically-controlled quantum computation. In order to appreciate the similarity between programming classical Turing machines and programming CQTM, some examples of CQTM will be given in the full version of the paper. Proofs of lemmas and theorems are omitted in this extended abstract.

Key words: Classically-Controlled Quantum Computation, Quantum Turing Machine

1 Introduction

Quantum computations operate in the quantum world. For their results to be useful in any way, by means of measurements for example, they operate under the control of the classical world. Quantum teleportation [1] illustrates the importance of classical control: the correcting Pauli operation applied at the end is classically controlled by the outcome of a previous measurement. Another example of the importance of classical control is measurement-based quantum computation [10,12,15,14,16,5], where classical conditional structures are required for controlling the computation. This classical control may be described

[1] Email: simon.perdrix@imag.fr
[2] Email: philippe.jorrand@imag.fr

as follows: *"if the classical outcome of measurement number i is λ, then measurement number $i+1$ is on qubit q_a according to observable O_a, otherwise measurement number $i+1$ is on qubit q_b according to observable O_b"*. A particularly elegant formalization of measurement-based quantum computation is the measurement calculus [5].

The necessity of integrating the classical control in the description of quantum computations is a now well understood requirement in the design of high level languages for quantum programming [7,17]. There are also some propositions of lower level models of computation integrating classical control, like the quantum random access machines (QRAM[9,2]). However there exist no formal and abstract model of quantum computation integrating classical control explicitly. This paper aims at defining such an abstract model of classically-controlled quantum computation.

One of the main existing abstract models of quantum computation is the Quantum Turing Machine (QTM) introduced by Deutsch [4], which is an analogue of the classical Turing machine (TM). It has been extensively studied by Bernstein and Vazirani [3]: a quantum Turing machine is an abstract model of quantum computers, which expands the classical model of a Turing machine by allowing a *quantum* transition function. In a QTM, superpositions and interferences of configurations are allowed, but the classical control of computation is not formalized and inputs and outputs of the machine are still classical. This second point means that the model of QTM explores the computational power of quantum mechanics for solving classical problems, without considering *quantum* problems, i.e. quantum input/output.

While models dealing with quantum states like quantum circuits [8,19] and QRAM, are mainly used for describing specific algorithms, the development of complexity classes, like QMA [18], which deal with quantum states, points out the necessity of theoretical models of quantum computation acting on quantum data.

The recently introduced model of Linear Quantum Turing Machine (LQTM) by S. Iriyama, M. Ohya, and I. Volovich [6] is a generalization of QTM dealing with mixed states and allowing irreversible transition functions which allow the representation of quantum measurements without classical outcomes. As a consequence of this lack of classical outcome, the classical control is not formalized in LQTM, and, among others, schemes like teleportation cannot be expressed. Moreover, like QTM, LQTM deals with classical input/output only.

We introduce here a Classically-controlled Quantum Turing Machine (CQTM) which is a TM with a quantum tape for acting on quantum data, and a classical transition function for a formalized classical control. In CQTM, unitary transformations and quantum measurements are allowed. Notice that the model of CQTM restricted to projective measurements is equivalent to the model of measurement-based quantum Turing machines (MQTM) introduced in [14]. *Theorem 2.1* shows that any TM is simulated by a CQTM without loss of efficiency. In section 3, CQTM with multiple tapes is introduced. *Theorem 3.1* shows that any k-tape CQTM is simulated by a 2-tape CQTM with a quadratic loss of efficiency. Moreover, the gap between classical and quantum computations which was already pointed out in the framework of measurement-based quantum computation (see [14]) is confirmed in the general case of classically-controlled quantum computation. A perspective is to make the CQTM not only a well defined theoretical model but also a bridge to practical models of quantum computations like QRAM, by relying on the fact that natural models of quantum computations are classically controlled.

2 Classically-controlled Quantum Turing Machines

2.1 Quantum states and admissible transformations

The quantum memory of a CQTM is composed of quantum cells. A quantum cell is a d-level quantum system [11], its state is a normalized vector in a d-dimensional Hilbert space. A basis of this Hilbert space is described by a finite alphabet of symbols Σ_Q such that $|\Sigma_Q| = d$. The state $|\phi\rangle \in \mathcal{H}_{\Sigma_Q}$ of a quantum cell is

$$|\phi\rangle = \sum_{\tau \in \Sigma_Q} \alpha_\tau |\tau\rangle ,$$

with $\sum_{\tau\in\Sigma_Q} |\alpha_\tau|^2 = 1$.

General quantum measurements operate according to the corresponding postulate of quantum mechanics: quantum measurements are described by a collection $\{M_{\tau_1}, \dots, M_{\tau_k}\}$ of measurement operators acting on the state space of the system being measured. The index τ refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ immediately before the measurement then the probability that the classical result τ occurs is given by

$$p(\tau) = \langle\psi| M_\tau^\dagger M_\tau |\psi\rangle ,$$

and the state of the system after the measurement is

$$\frac{M_\tau |\psi\rangle}{\sqrt{p(\tau)}}.$$

The measurement operators satisfy the completness equation,

$$\sum_\tau M_\tau^\dagger M_\tau = I.$$

General quantum measurements are also called *admissible transformations.* Notice that admissible transformations which are composed of only one operator M_τ are nothing but unitary transformations since $p(\tau) = 1$, the state after the transformation is $M_\tau |\psi\rangle$ and the completeness equation reduces to

$M_\tau^\dagger M_\tau = I$. Conversely, any unitary transformation A is an admissible transformation.

For a given Hilbert space $\mathcal{H}_{\Sigma_Q}$, we exhibit some admissible transformations with classical results belonging to a finite set $\Sigma_C = \Sigma_Q \cup \overline{\Sigma_Q} \cup \{\lambda\}$, where $\overline{\Sigma_Q} = \{\overline{\tau} : \tau \in \Sigma_Q\}$ and $\lambda \notin \Sigma_Q$:

- $Std = \{M_\tau\}_{\tau \in \Sigma_Q}$ is a projective measurement in the standard basis: $\forall \tau \in \Sigma_Q, M_\tau = |\tau\rangle\langle\tau|$,
- $\mathcal{T}_\tau = \{M_\tau, M_{\overline{\tau}}\}$ is a test for the symbol τ: $M_\tau = |\tau\rangle\langle\tau|$ and $M_{\overline{\tau}} = I - |\tau\rangle\langle\tau|$,
- $\mathcal{P}_{[\tau_a,\tau_b]} = \{M_\lambda\}$ is a unitary transformation with outcome λ, and $M_\lambda = (\sum_{\tau\in\Sigma_Q-\{\tau_a,\tau_b\}} |\tau\rangle\langle\tau|) + |\tau_a\rangle\langle\tau_b| + |\tau_b\rangle\langle\tau_a|$ is a permutation of the symbols τ_a and τ_b.
- $\mathcal{U}_V = \{M_\lambda\}$ is the unitary transformation $M_\lambda = V$, with classical outcome λ.
- $\mathcal{O}_O = \{P_k\}_k$, is a projective measurement according to the observable $O = \Sigma_k P_k$.

2.2 *Defining a CQTM*

For completeness, definition 2.1 is the definition of a deterministic TM [13]. A classically-controlled quantum Turing machine (definition 2.2) is composed of a quantum tape of quantum cells, a set of classical internal states and a head for applying admissible transformations to cells on the tape. The role of the head is crucial because it implements the interaction across the boundary between the quantum and the classical parts of the machine.

Definition 2.1 A deterministic (classical) Turing Machine is defined by a triplet $M = (K, \Sigma, \delta)$, where K is a finite set of states with an identified initial state s, Σ is a finite alphabet with an identified "blank" symbol #, and δ is a deterministic transition:

$$\delta : K \times \Sigma \to (K \cup \{\text{"}yes\text{"}, \text{"}no\text{"}, h\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}.$$

We assume that h (the halting state), "*yes*" (the accepting state) and "*no*" (the rejecting state) are not in K.

Definition 2.2 A Classically-controlled Quantum Turing Machine is a quintuple $M = (K, \Sigma_C, \Sigma_Q, \mathcal{A}, \delta)$. Here K is a finite set of classical states with an identified initial state s, Σ_Q is a finite alphabet which denotes basis states of quantum cells, Σ_C is a finite alphabet of classical outcomes, $\mathcal{A}$ is a set of one-quantum cell admissible transformations, and δ is a classical transition function:

$$\delta : K \times \Sigma_C \to (K \cup \{\text{"}yes\text{"}, \text{"}no\text{"}, h\}) \times \{\leftarrow, \rightarrow, -\} \times \mathcal{A}.$$

We assume that h (the halting state), "*yes*" (the accepting state) and "*no*" (the rejecting state) are not in K, and that all possible classical outcomes of each measurement of $\mathcal{A}$ are in Σ_C. Moreover we assume that Σ_Q always contains a "blank" symbol #, Σ_C always contains a "blank" symbol # and a "non-blank" symbol $\overline{\#}$, and $\mathcal{A}$ always contains the admissible "blank test" transformation $\mathcal{T}_\#$.

The function δ is a formalization of the classical control of the quantum computation and can also be viewed as the "program" of the machine. It specifies, for each combination of current state $q \in K$ and last obtained classical outcome $\tau \in \Sigma_C$, a triplet $\delta(q,\tau) = (p, D, A)$, where p is the next classical state, $D \in \{\leftarrow, \rightarrow, -\}$ is the direction in which the head will move, and $A \in \mathcal{A}$ is the admissible transformation to be performed next. The *blank test* admissible transformation $\{M_\#, M_{\overline{\#}}\}$ establishes a correspondence between the quantum blank symbol (#) and the classical blank (#) and non-blank ($\overline{\#}$) symbols: if the state $|\phi\rangle$ of the measured quantum cell is $|\#\rangle$, the outcome of the measurement is # whereas if $|\phi\rangle$ is orthogonal to $|\#\rangle$ ($\langle\phi\,|\#\rangle = 0$) then the outcome is $\overline{\#}$.

How does the program start? The quantum input of the computation $|\phi\rangle = \sum_{\tau\in(\Sigma_Q-\{\#\})^n} \alpha_\tau\,|\tau\rangle$, which is in general unknown, is placed on n adjacent cells of the tape, while the state of all other quantum cells of the tape is $|\#\rangle$. The head is pointing at the blank cell immediately located on the left of the input. Initially, the classical state of the machine is s and # is considered as the last classical outcome, thus the first transition is always $\delta(s,\#)$.

How does the program halt? The transition function δ is total on $K \times \Sigma_C$ (irrelevant transitions will be omitted from its description). There is only one reason why the machine cannot continue: one of the three halting states h, "*yes*", and "*no*" has been reached. If a machine M halts on input $|\phi_{in}\rangle$, the output $M(|\phi_{in}\rangle)$ of the machine M on $|\phi_{in}\rangle$ is defined. If states "*yes*" or "*no*" are reached, then $M(|\phi_{in}\rangle) =$ "*yes*" or "*no*" respectively. Otherwise, if halting state h is reached then the output is the state $|\phi_{out}\rangle$ of the tape of M at the time of halting. Since the computation has gone on for finitely many steps, only a finite number of cells are not in the state $|\#\rangle$. The output state $|\phi_{out}\rangle$ is the state of the finite register composed of the quantum cells from the leftmost cell in a state which is not $|\#\rangle$ to the rightmost cell in a state which is not $|\#\rangle$. Naturally, it is possible that M never halts on input $|\phi_{in}\rangle$. If this is the case we write $M(|\phi_{in}\rangle) = \nearrow$.

A *configuration* of a CQTM M is intuitively a complete description of the current state of the computation. Formally, a configuration is a triplet $(q, \tau, |\psi\rangle)$, where $q \in K \cup \{h, \text{"}yes\text{"}, \text{"}no\text{"}\}$ is the internal state of M, $\tau \in \Sigma_C$ is the last obtained outcome, and $|\psi\rangle \in \mathcal{H}_{\Sigma'_Q}$ represents the state of the tape and the position of the head. Here $\Sigma'_Q = \Sigma_Q \cup \underline{\Sigma}_Q$, where $\underline{\Sigma}_Q = \{\underline{\tau} : \tau \in \Sigma_Q\}$ is a set of *pointed* versions of the symbols in Σ_Q. From a state $|\phi\rangle \in \mathcal{H}_{\Sigma_Q}$ of the tape, the state $|\psi\rangle \in \mathcal{H}_{\Sigma'_Q}$ is obtained by replacing the symbol of Σ_Q by the corresponding symbol of $\underline{\Sigma}_Q$ for the quantum cell pointed at by the head. For instance, if $K = \{q_1, q_2\}$, $\Sigma_C = \{\#, \overline{\#}, t, u, v\}$ and $\Sigma_Q = \{\#, a, b\}$, the configuration

$$(q_1, u, \frac{1}{\sqrt{2}}(|a\#\underline{b}b\rangle + |b\#\underline{a}b\rangle))$$

means that the internal state of the machine is q_1, the last outcome is u, the state of the tape is $\frac{1}{\sqrt{2}}(|a\#bb\rangle + |b\#ab\rangle)$, and the head is pointing at the third cell from the right.

3 CQTM and TM

The following theorem shows that any TM is simulated by a CQTM without loss of efficiency.

Theorem 3.1 *Given any TM M_C operating in time $f(n)$, where n is the input size, there exists a CQTM M_Q operating in time $O(f(n))$ and such that for any input x, $M_C(x) = M_Q(|x\rangle)$*

Since any TM is simulated by a CQTM without loss of efficiency, the model of CQTM is classically universal (see [14] for definitions of classical and quantum universalities), but, as will be shown in *Lemma 4.4*, CQTM with one tape are not quantum universal, because only one-cell admissible transformations are allowed. In order to allow transformations on more than one cell, we introduce multiple tapes CQTMs. Intuitively, with k heads, k-cell admissible transformations can be performed.

4 CQTM with multiple tapes

We introduce a generalization of the CQTM, the classically-controlled Turing machine with multiple tapes. We show that any k-tape CQTM is simulated by a 2-tape CQTM with an inconsequential loss of efficiency. Moreover, by showing that 1- and 2-tape CQTM are not equivalent, we point out a *gap* between classical and quantum computations.

Definition 4.1 A k-tape Classically-controlled Quantum Turing Machine where $k > 0$, is a quintuple $M = (K, \Sigma_C, \Sigma_Q, \mathcal{A}, \delta)$, where K is a finite set of classical states with an identified initial state s, Σ_Q is a finite alphabet which denotes basis states of each quantum cell. $\mathcal{A}$ is a set of k-cell admissible transformations, Σ_C is a finite alphabet of classical outcomes of k-cell admissible transformations and δ is a classical transition function

$$\delta : K \times \Sigma_C \to (K \cup \{\text{“}yes\text{”}, \text{“}no\text{”}, h\}) \times (\{\leftarrow, \rightarrow, -\})^k \times \mathcal{A}.$$

We assume that all possible classical outcomes of each measurement of $\mathcal{A}$ are in Σ_C and that $\mathcal{A}$ always contains the k admissible “blank test” transformations, one for each tape of the machine.

Intuitively, $\delta(q, \tau) = (q', (D_1, \ldots D_k), A)$ means that, if M is in state q and the last classical outcome is τ, then the next state will be q', the k heads of the machine will move according to $D_1, \dots, D_k$ and the next k-quantum cell admissible transformation will be A. This admissible transformation will be performed on the k quantum cells pointed at by the heads of the machine after they have moved. A k-cell admissible transformation A can be defined directly, for instance by use of a k-cell unitary transformation V ($A = \mathcal{U}_V$). A can also be defined as a composition of two admissible transformations A_1, A_2 respectively on j and l cells such that $j + l = k$, then $A = [A_1, A_2]$ means that the first j heads apply A_1 and, simultaneously, the last l heads apply A_2. The classical outcome is the concatenation of the outcomes of A_1 and A_2, where λ is the unit element of the concatenation (i.e. $\tau.\lambda = \tau$).

A k-tape CQTM starts with an input state $|\phi\rangle$ on a specified tape T_1, and if the halting state h is reached, the machine halts and the output is the state of the specified tape T_1.

Theorem 4.2 *Given any k-tape CQTM M operating in time $f(n)$, where n is the input size, there exists a 2-tape CQTM M' operating in time $O(f(n)^2)$ and such that for any input $|\psi\rangle$, $M(|\psi\rangle) = M'(|\psi\rangle)$.*

Theorem 4.1 is a strong evidence of the power and stability of CQTMs: adding a bounded number of tapes to a 2-tape CQTM does not increase their computational capabilities, and impacts their efficiency polynomially only. This stability makes 2-tape CQTMs a good candidate for quantum universality, i.e. the ability to simulate any quantum computation. This ability is proved with the following two lemmas:

Lemma 4.3 *Any pattern of the measurement calculus [5] can be simulated in a time polynomial in the size of the pattern by a 2-tape CQTM.*

Lemma 4.4 *Any quantum circuit can be simulated by a 2-tape CQTM in polynomial time.*

The following lemma shows that some 2-tape CQTMs cannot be simulated by 1-tape CQTMs:

Lemma 4.5 *There exists a 2-tape CQTM M such that no 1-tape CQTM simulates M.*

To sum up, two tapes are enough for quantum computation (*Lemma 4.3*), whereas one tape is enough for classical computation (*Theorem 3.1*) but not for quantum computation (*Lemma 4.4*). Thus a *gap* between classical and quantum computations appears. Notice that this result does not contradict the equivalence, in terms of decidability, between classical and quantum computations: the gap appears iff quantum data are considered.

One may wonder why 1-tape CQTMs are not quantum universal whereas Briegel and Raussendorf have proved, with their One-way quantum computer, that one-qubit measurements are universal [16]. The proof by Briegel and Raussendorf is given with a strong assumption which is that there exists a grid of auxiliary qubits which have been initially prepared, by some unspecified external device, in a globally entangled state (the cluster state), whereas *creation* of entanglement is a crucial point in the proof of *Lemma 4.4*. Moreover, another strong assumption of one-way quantum computation is that the input state $|\varphi\rangle$ has to be classically known (i.e. a mathematical description of $|\varphi\rangle$ is needed), whereas the manipulation of unknown states (i.e. manipulation of qubits in an unknown state) is usual in quantum computation (e.g. teleportation [1]). Since none of these assumptions are verified by 1-tape CQTM, the previous results do not contradict the results of Briegel and Raussendorf.

5 Conclusion

This paper introduces a new abstract model for quantum computations, the model of classically-controlled quantum Turing machines (CQTM). This model allows a rigorous formalization of the inherent interactions between the quantum world and the classical world during a quantum computation. Any classical Turing machine is simulated by a CQTM without loss of efficiency, moreover any k-tape CQTM is simulated by a 2-tape CQTM affecting the execution time only polynomially.

Moreover the gap between classical and quantum computations which was already pointed out in the framework of measurement-based quantum computation (see [14]) is confirmed in the general case of classically-controlled quantum computation.

The classically-controlled quantum Turing machine is a good candidate for establishing a bridge between, on one side, theoretical models like QTM, CQTM, MQTM [14] and on the other side practical models of quantum computation like quantum random access machines.

References

[1] C. Bennett et al. *Teleporting an unknown quantum state via dual classical and EPR channels*, Phys Rev Lett, 1895-1899, 1993.

[2] S. Bettelli, L. Serafini, and T. Calarco. *Toward an architecture for quantum programming*, arXiv, cs.PL/0103009, 2001.

[3] E. Bernstein and U. Vazirani, *Quantum complexity theory*, SIAM J. Compt. 26, 1411-1473, 1997.

[4] D. Deutsch. *Quantum theory, the Church-Turing principle and the universal quantum computer*, Proceedings of the Royal Society of London A 400, 97-117, 1985.

[5] V. Danos, E. Kashefi, P. Panangaden *The Measurement Calculus* , e-print arXiv, quant-ph/0412135.

[6] S. Iriyama, M. Ohya, I. Volovich. *Generalized Quantum Turing Machine and its Application to the SAT Chaos Algorithm*, arXiv, quant-ph/0405191, 2004.

[7] Ph. Jorrand and M. Lalire. *Toward a Quantum Process Algebra*, Proceedings of the first conference on computing frontiers, 111-119, 2004, e-print arXiv, quant-ph/0312067.

[8] A. Y. Kitaev, A. H. Shen and M. N. Vyalyi. *Classical and Quantum Computation*, American Mathematical Society, 2002.

[9] E. H. Knill. *Conventions for Quantum Pseudocode*, unpublished, LANL report LAUR-96-2724

[10] D. W. Leung. *Quantum computation by measurements*, arXiv, quant-ph/0310189, 2003.

[11] A. Muthukrishnan and C. R. Stroud. *Multi-valued logic gates for quantum computation*, arXiv, quant-ph/0002033, 2000.

[12] M. A. Nielsen. *Universal quantum computation using only projective measurement, quantum memory, and preparation of the 0 state*, arXiv, quant-ph/0108020, 2001.

[13] C. M. Papadimitriou. *Computational Complexity*, Addisson-Wesley Publishing Compagny, 1994.

[14] S. Perdrix and Ph. Jorrand. *Measurement-Based Quantum Turing Machines and their Universality*, arXiv, quant-ph/0404146, 2004.

[15] S. Perdrix. *State Transfer instead of Teleportation in Measurement-based Quantum Computation*, arXiv, quant-ph/0402204, 2004.

[16] R. Raussendorf, D. E. Browne and H. J. Briegel. *Measurement-based quantum computation with cluster states*, arXiv, quant-ph/0301052, 2003.

[17] P. Selinger. *Towards a quantum programming language.* To appear in Mathematical Structures in Computer Science, 2003.

[18] J. Watrous, *Succinct quantum proofs for properties of finite groups*, Proc. 41st Annual Symposium on Foundation of Computer Science, pp. 537-546, 2000.

[19] A. C. Yao, *Quantum circuit complexity*, Proc. 34th IEEE Symposium on Foundation of Computer Science, pp. 352-361, 1993.

A Calculus for Reconfiguration

(Extended abstract)

Sonia Fagorzi [2] and Elena Zucca [3]

DISI
University of Genova
Genova, Italy

Abstract

We present a simple calculus, called R-calculus (for "reconfiguration"), intended to provide a kernel model for a computational paradigm in which standard execution (that is, execution of a single computation described by a fragment of code) can be interleaved with operations at the meta-level which can manipulate in various ways the context in which this computation takes place. Formally, this is achieved by introducing as basic terms of the calculus "configurations", which are, roughly speaking, pairs consisting of an (open, mutually recursive) collection of named components and a term representing a "program" running in the context of these components. The R-calculus has been originally developed as a formal model for programming-in-the large, where computations correspond to applications running in some context of software components, and operations at the meta-level correspond to the possibility of dynamically loading, updating or in general manipulating these software components without stopping the application. However, the calculus can also be seen as useful for programming-in-the-small issues, because configurations combine the features of lambda-abstractions (first-class functions), records, environments with mutually recursive definitions, and modules.

We state confluence of the calculus and define a call-by-need strategy which leads to a generalization, including reconfiguration features, of call-by-need lambda-calculi.

Key words: Module calculi, call-by-need strategy.

[1] Partially supported by APPSEM II - Thematic network IST-2001-38957, and MIUR EOS - Extensible Object Systems.
[2] Email: `fagorzi@disi.unige.it`
[3] Email: `zucca@disi.unige.it`

Introduction

In the last years considerable effort has been invested in developing kernel module/fragment calculi [5,13,12,6,11] providing foundations for manipulation and combination of software components. However, these calculi are based on a *static* view of software composition, in the sense that open code fragments can be flexibly combined together, but before actually starting execution of a computation we must have obtained a fully reduced and closed piece of code. In module calculi this is formally reflected by the fact that *selection*, denoted $e.X$, where e is a module expression and X is the name of a module component, can only be performed when e is a basic module (no module operators remain to be reduced) and, moreover, there are no components which still need to be imported.

However, modern programming environments increasingly include *dynamic reconfiguration* features, in the sense that interleaving is allowed between *reconfiguration* steps and standard *execution* steps. Examples of reconfiguration features are dynamic loading as in Java and C#, where single code fragments are dynamically linked to an already executing program, dynamic rebinding [9], that is, the ability of changing the meaning of names used by an application at execution time, marshaling/unmarshaling of values or computations from a running application to another.

Here, we present a simple calculus, called *R*-calculus (for "reconfiguration"), intended to provide a kernel model for a computational paradigm in which standard execution (that is, execution of a single computation described by a possibly open fragment of code) can be interleaved with operations at the meta-level which can manipulate in various ways the context in which this computation takes place.

Formally, this is achieved by defining as basic terms of the calculus "configurations", which are, roughly speaking, pairs consisting of an (open, mutually recursive) collection of named components (formally, a basic module) and a term representing a "program" running in the context of these components. Configurations can be combined by classical module/fragment operators (in particular, we take as underlying module calculus *CMS* [5,6]) and, hence, reduction steps can be either execution steps of the program or steps which perform module operators (reconfiguration steps).

As discussed above, the initial motivation for the *R*-calculus has been the search for formal models for programming-in-the large. In this respect, the calculus is part of a stream of work [1,3,2,4] on foundations for systems supporting reconfiguration features, in which the system structure can dynamically change after starting execution of an application.

However, the calculus can also be seen as useful for programming-in-the-small issues, because configurations combine the features of lambda-abstractions (first-class functions), records, environments with mutually recursive definitions, and modules.

$$
\begin{array}{llll}
e \in \mathsf{Exp} ::= & & & \textbf{expression} \\
 & \mid\ x & & \text{variable} \\
 & \mid\ [\iota;\ o;\ \rho] & (\mathsf{dom}(\iota)\cap\mathsf{dom}(\rho)=\emptyset) & \text{basic module} \\
 & \mid\ [\iota;\ o;\ \rho \mid e] & (\mathsf{dom}(\iota)\cap\mathsf{dom}(\rho)=\emptyset) & \text{basic configuration} \\
 & \mid\ e_1 + e_2 & & \text{sum} \\
 & \mid\ {}_{\sigma^\iota}|e|_{\sigma^o} & & \text{reduct} \\
 & \mid\ \mathsf{freeze}_\sigma e & & \text{freeze} \\
 & \mid\ e{\downarrow}_X & & \text{run} \\
 & \mid\ e{\uparrow} & & \text{result} \\
\iota & := x_i \overset{i\in I}{\mapsto} X_i & & \textbf{input}\ \text{assignment} \\
o & := X_i \overset{i\in I}{\mapsto} e_i & & \textbf{output}\ \text{assignment} \\
\rho & := x_i \overset{i\in I}{\mapsto} e_i & & \textbf{local}\ \text{assignment} \\
\sigma & := X_i \overset{i\in I}{\mapsto} Y_i, Y_j{}^{j\in J} & & \textbf{renaming}
\end{array}
$$

Fig. 1. Syntax

In this extended abstract, we focus on this aspect. In the first section, we present the R-calculus as a pure calculus (no reduction strategy) and state its confluence. Then, we outline a call-by-need strategy which should lead to a generalization, including reconfiguration features, of call-by-need lambda-calculi as in [8]. This second part will be worked out in a forthcoming full version of this paper, which will also include a more extended investigation on different strategies for the calculus, and on how they can be used to encode/generalize other primitive calculi, such as lambda calculi and module calculi.

1 The R-calculus

In this section we provide an introduction to the R-calculus by examples, which is organized as follows: first, we illustrate the module fragment of the calculus, which is a variant of (mixin) module calculi such as [6,13]; then we introduce configurations and illustrate the interleaving of program execution and reconfiguration; finally, we discuss higher-order features of the calculus, showing how reconfiguration can take place at many levels. The formal syntax and reduction rules are given in Fig.1 and Fig.2, respectively.

Module operators Terms of the calculus denote either *modules* (collections of components which are either *input* or *output* or *local*) or *configurations* (pairs consisting of a module and a *program* running in the context of the components offered by the module). The subset of the calculus constituted by terms denoting modules is a standard (mixin) module calculus. In particular, basic modules and module operators are those of *CMS* [6]. In reduction rules, we give somewhere a slightly different version w.r.t. those in [6], more suitable for our technical development; in particular, we relax the applicability of reduct and freeze operators, and we include rules (**m-subst**) and (**m-subst-output**) in the style of the m-calculus [13] (see explanations below).

Module expressions are constructed on top of *basic modules* by the three operators of *sum*, *reduct* and *freeze*.

A **basic module** has form $[\iota; o; \rho]$ where: ι is a map from *deferred* variables into *input* names, o is a map from *output* names into expressions and ρ is a map from *local* variables into expressions. Names $X, Y, Z, \ldots$ are used to refer to a component from outside the module (hence they are used by module operators), while variables $x, y, z, \ldots$ are used to refer to a component from the inside. For instance, denoting by $e[x_1, \ldots, x_n]$ an expression possibly containing $x_1, \ldots, x_n$ as free variables, the expression
$e_0 \triangleq [x \mapsto X, z \mapsto Z; Y \mapsto e_1[x, z, y]; y \mapsto e_2[x, z, y]]$
is a basic module with two input, one output and one local component. Local components can be mutually recursive.
The **sum** operator allows to combine two modules by performing the union of input components and the disjoint union of output and local components (see rule (**m-sum**)). Input components with the same name in the two modules are shared in the resulting module, while conflicts among deferred or local variables are solved by α-renaming. For instance, since the sets of output names are disjoint, we can perform the sum of the basic module e_0 above with

$e_5 \triangleq [y \mapsto Y, w \mapsto Z; X \mapsto e_3[y, w, x]; x \mapsto e_4[y, w, x]]$

We obtain the following basic module, that we call e_6:
$[x \mapsto X, z \mapsto Z, y' \mapsto Y, w \mapsto Z; Y \mapsto e_1[x, z, y], X \mapsto e_3[y', w, x']; y \mapsto e_2[x, z, y], x' \mapsto e_4[y', w, x']]$
where the input name Z is shared by the two variables z and w, while variables y and x in e_5 are α-renamed into y' and x', respectively, since they conflict with those in e_0.

The **reduct** operator allows to perform a renaming of component names, where input and output names are renamed independently (see rule (**m-reduct**)). The input renaming is a map whose domain and codomain are old and new input names, respectively, whereas the output renaming is a map whose domain and codomain are new and old output names, respectively. For instance,

$${}_{X_1 \mapsto X, X_2 \mapsto X, \_ \mapsto W}|[x_1 \mapsto X_1, x_2 \mapsto X_2, x_3 \mapsto X_3; Y \mapsto e_1, Z \mapsto e_2; x \mapsto e_3]|_{Y_1 \mapsto Y, Y_2 \mapsto Y}$$

reduces to:

$[x_1 \mapsto X, x_2 \mapsto X, x_3 \mapsto X_3, w \mapsto W; Y_1 \mapsto e_1, Y_2 \mapsto e_1; x \mapsto e_3]$.

A non-injective input renaming allows merging two input names (like X_1 and X_2 into X in the example), whereas a non-surjective one is used for adding dummy input names (W in the example), which are bound to fresh variables

One hole contexts and contextual closure

$$\begin{array}{lll}
\mathcal{E} & ::= & \Box \mid [\iota;\, \mathcal{O};\, \rho] \mid [\iota;\, o;\, \mathcal{L}] \mid [\iota;\, \mathcal{O};\, \rho \mid e] \mid [\iota;\, o;\, \mathcal{L} \mid e] \mid [\iota;\, o;\, \rho \mid \mathcal{E}] \\
 & \mid & \mathcal{E} + e \mid e + \mathcal{E} \mid {}_{\sigma^\iota|}\mathcal{E}_{|\sigma^o} \mid \mathsf{freeze}_\sigma \mathcal{E} \mid \mathcal{E}{\downarrow_X} \mid \mathcal{E}{\uparrow} \\
\mathcal{R} & ::= & \Box \mid \mathcal{R} + e \mid {}_{\sigma^\iota|}\mathcal{R}_{|\sigma^o} \mid \mathsf{freeze}_\sigma \mathcal{R} \\
\mathcal{O} & ::= & X \mapsto \mathcal{E}, o \\
\mathcal{L} & ::= & x \mapsto \mathcal{E}, \rho \\
v \in \mathsf{Val} & ::= & [\iota;\, o;\, \rho] \mid [\iota;\, o;\, \rho \mid v]
\end{array}$$

$$(\mathcal{E})\ \frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}$$

Module simplification

$$(\textbf{m-sum})\ \frac{}{[\iota_1;\, o_1;\, \rho_1] + [\iota_2;\, o_2;\, \rho_2] \longrightarrow [\iota_1, \iota_2;\, o_1, o_2;\, \rho_1, \rho_2]}\ \begin{array}{l}\mathsf{dom}(\iota_1, \rho_1) \cap \mathsf{FV}([\iota_2;\, o_2;\, \rho_2]) = \emptyset \\ \mathsf{dom}(\iota_2, \rho_2) \cap \mathsf{FV}([\iota_1;\, o_1;\, \rho_1]) = \emptyset\end{array}$$

$$(\textbf{m-reduct})\ \frac{}{{}_{\sigma^\iota|}[\iota_1, \iota_2;\, o;\, \rho]_{|\sigma^o} \longrightarrow [\sigma^\iota \circ \iota_1, \iota_2;\, o \circ \sigma^o;\, \rho]}\ \mathsf{cod}(\iota_2) \cap \mathsf{dom}(\sigma^\iota) = \emptyset$$

$$(\textbf{m-freeze})\ \frac{}{\mathsf{freeze}_\sigma \Big[x_i \overset{i \in I}{\mapsto} X_i, \iota;\, o;\, \rho \Big] \longrightarrow \Big[\iota;\, o;\, \rho, x_i \overset{i \in I}{\mapsto} o(\sigma(X_i)) \Big]}\ \mathsf{cod}(\iota) \cap \mathsf{dom}(\sigma) = \emptyset$$

$$(\textbf{m-subst})\ \frac{}{[\iota;\, o;\, x_1 \mapsto \mathcal{E}[x_2], \rho] \longrightarrow [\iota;\, o;\, x_1 \mapsto \mathcal{E}\{\rho(x_2)\}, \rho]}\ \begin{array}{l} x_2 \notin \mathsf{HB}(\mathcal{E}) \\ x_2 \in \mathsf{dom}(\rho) \\ x_2 \overset{*}{\underset{\iota,o,\rho}{\not\Longrightarrow}} x_1 \end{array}$$

$$(\textbf{m-subst-output})\ \frac{}{[\iota;\, X \mapsto \mathcal{E}[x], o;\, x \mapsto e, \rho] \longrightarrow [\iota;\, X \mapsto \mathcal{E}\{e\}, o;\, x \mapsto e, \rho]}\ x \notin \mathsf{HB}(\mathcal{E})$$

Reconfiguration and substitution

$$(\textbf{reconf/subst})\ \frac{\mathcal{R}[\iota;\, o;\, \rho] \underset{\alpha}{\longrightarrow} \mathcal{R}'[\iota';\, o';\, \rho']}{\mathcal{R}[\iota;\, o;\, \rho \mid e] \longrightarrow \mathcal{R}'[\iota';\, o';\, \rho' \mid \alpha(e)]}$$

$$(\textbf{subst-prg})\ \frac{}{[\iota;\, o;\, \rho \mid \mathcal{E}[x]] \longrightarrow [\iota;\, o;\, \rho \mid \mathcal{E}\{\rho(x)\}]}\ \begin{array}{l} x \notin \mathsf{HB}(\mathcal{E}) \\ x \in \mathsf{dom}(\rho) \end{array}$$

Run and result

$$(\textbf{run})\ \frac{}{[\iota;\, o;\, \rho]{\downarrow_X} \longrightarrow [\iota;\, o;\, \rho \mid o(X)]} \qquad (\textbf{res})\ \frac{}{(\mathcal{R}[\iota;\, o;\, \rho \mid e]){\uparrow} \longrightarrow e}\ \mathsf{FV}(e) \cap \mathsf{dom}(\iota, \rho) = \emptyset$$

Dependency relation on module variables

Given ι, o and ρ, $x \overset{*}{\underset{\iota,o,\rho}{\Longrightarrow}} y$, meaning "$x$ depends on y through ι, o and ρ", is the least transitive and reflexive relation on $\mathsf{dom}(\iota, \rho)$ induced by:

$$\frac{}{x_1 \overset{*}{\underset{\iota,o,\rho}{\Longrightarrow}} x_2}\ x_2 \in \mathsf{FV}(\rho(x_1)) \qquad \frac{}{x_1 \overset{*}{\underset{\iota,o,\rho}{\Longrightarrow}} x_2}\ x_1 \in \mathsf{dom}(\iota) \wedge x_2 \in \mathsf{FV}(o)$$

We write $x_2 \overset{*}{\underset{\iota,o,\rho}{\not\Longrightarrow}} x_1$ if $x_2 \overset{*}{\underset{\iota,o,\rho}{\Longrightarrow}} x_1$ is not derivable.

Fig. 2. *R*-calculus reduction rules

(w in the example). A non-injective output renaming allows duplications of definitions (in the example, the definition of Y is used for both Y_1 and Y_2), whereas a non-surjective one is used for deleting output components (in the example Z).

The **freeze** operator allows linking of input and output names inside a module (see rule (**m-freeze**)). That is, this operator resolves input names, so that input components are transformed into local. For instance, the expression $\mathsf{freeze}_{X\mapsto X, Z\mapsto Y}(e_6)$ reduces to:

$$\begin{aligned}&[\ y' \mapsto Y;\\ &\quad Y \mapsto e_1[x,z,y], X \mapsto e_3[y',w,x'];\\ &\quad y \mapsto e_2[x,z,y], x' \mapsto e_4[y',w,x'], x \mapsto e_3[y',w,x'], z \mapsto e_1[x,z,y], w \mapsto e_1[x,z,y]\ ].\end{aligned}$$

Rules (**m-subst**) and (**m-subst-output**) allow to substitute local variables with their definitions in module components . In rule (**m-subst**) the side-condition $x_2 \not\xrightarrow[\iota,o,\rho]{*} x_1$ means that the (local) component x_2 does not depend on the (local) component x_1. Without this condition, the R-calculus would not be confluent (see [7,13]).

Configurations A **basic configuration** is a pair $[\iota;\ o;\ \rho \mid e]$, consisting of a basic module and an expression, called *program*. A basic configuration can evolve by reduction steps of the program; moreover, local variables can be replaced by their defining expressions (see rule (**subst-prg**)). This is illustrated by the reduction sequence below:

$[;\ X \mapsto x;\ y \mapsto 1, x \mapsto 2+y \mid x] \longrightarrow [;\ X \mapsto x;\ y \mapsto 1, x \mapsto 2+y \mid 2+y] \longrightarrow$
$[;\ X \mapsto x;\ y \mapsto 1, x \mapsto 2+y \mid 2+1] \longrightarrow [;\ X \mapsto x;\ y \mapsto 1, x \mapsto 2+y \mid 3]$,

Moreover, module operators described in previous section can be applied to configurations as well, and act as reconfiguration operators, in the sense that they allow to modify the context of a program during its execution. This is performed by rule (**reconf/subst**), which also allows local variable substitution inside configuration components. Note that in the premise of rule (**reconf/subst**) the reduction step at the module level is labelled by α. This is just to keep track and propagate to the program possible α-renaming happened during the module evaluation step. In this way, a needed input component can become available, as shown below:

$\mathsf{freeze}_{Z\mapsto Z}[z \mapsto Z;\ ;\ y \mapsto 1, x \mapsto 2+z \mid x] + [;\ Z \mapsto 3;\ ] \longrightarrow$
$\mathsf{freeze}_{Z\mapsto Z}[z \mapsto Z;\ ;\ y \mapsto 1, x \mapsto 2+z \mid 2+z] + [;\ Z \mapsto 3;\ ] \longrightarrow$
$\mathsf{freeze}_{Z\mapsto Z}[Z \mapsto z;\ Z \mapsto 3;\ y \mapsto 1, x \mapsto 2+z \mid 2+z] \longrightarrow$
$[;\ Z \mapsto 3;\ y \mapsto 1, x \mapsto 2+z, z \mapsto 3 \mid 2+z] \longrightarrow$
$[;\ Z \mapsto 3;\ y \mapsto 1, x \mapsto 2+z, z \mapsto 3 \mid 2+3] \longrightarrow$
$[;\ Z \mapsto 3;\ y \mapsto 1, x \mapsto 2+z, z \mapsto 3 \mid 5]$.

Higher-order features and configuration levels Since a program can be in turn a configuration, both local variable resolution steps and reconfiguration steps can take place at an outer configuration level (the innermost where the needed variable is bound). For instance, the expression

$\mathsf{freeze}_{Z\mapsto Z}([z \mapsto Z;\ ;\ y \mapsto 2 \mid [;\ X \mapsto x;\ x \mapsto y+z \mid x]] + [;\ Z \mapsto 3;\ ])$
can reduce as follows:
$\longrightarrow\ \mathsf{freeze}_{Z\mapsto Z}([z \mapsto Z;\ ;\ y \mapsto 2 \mid [;\ X \mapsto x;\ x \mapsto y+z \mid y+z]] + [;\ Z \mapsto 3;\ ])$
$\longrightarrow$
$\mathsf{freeze}_{Z\mapsto Z}([z \mapsto Z;\ ;\ y \mapsto 2 \mid [;\ X \mapsto x;\ x \mapsto y+z \mid y+z]] + [;\ Z \mapsto 3;\ ])\ \longrightarrow$
$\mathsf{freeze}_{Z\mapsto Z}[z \mapsto Z;\ Z \mapsto 3;\ y \mapsto 2 \mid [;\ X \mapsto x;\ x \mapsto 2+z \mid 2+z]]\ \longrightarrow$
$[;\ Z \mapsto 3;\ y \mapsto 2, z \mapsto 3 \mid [;\ X \mapsto x;\ x \mapsto 2+z \mid 2+z]]\ \longrightarrow\ \ldots\ \longrightarrow$
$[;\ Z \mapsto 3;\ y \mapsto 2, z \mapsto 3 \mid [;\ X \mapsto x;\ x \mapsto 2+z \mid 5]]$.

The **run** operator allows to obtain a basic configuration from a (basic) module, by starting the execution of one of its output components (see rule (**run**)). For instance, the expression $[;\ X \mapsto x;\ y \mapsto 2, x \mapsto 2+y]\downarrow_X$ reduces in one step to the basic configuration seen before.

The **result** operator allows to extract the program from a configuration (see rule (**res**)). Formally, a configuration level is modeled by an expression of the form $\mathcal{R}[[\iota;\ o;\ \rho \mid e]]$ where $\mathcal{R}$ is a context consisting only of reconfiguration operators. The result operator can be safely applied only if the program does not refer to any variable bound in the basic configuration of the considered configuration level. This condition prevents scope extrusion of variables, which would lead to dynamic errors. For instance, in the expression $(\mathsf{freeze}_{Z\mapsto Z}([z \mapsto Z;\ ;\ \mid [;\ X \mapsto x;\ x \mapsto 2+z \mid x]\uparrow] + [;\ Z \mapsto 3;\ ]))\uparrow$, the innermost result operator cannot be executed since the program still refers to the variable x bound in the enclosing basic configuration. Hence, we first need to resolve this variable, that is,
$\longrightarrow\ (\mathsf{freeze}_{Z\mapsto Z}([z \mapsto Z;\ ;\ \mid [;\ X \mapsto x;\ x \mapsto 2+z \mid 2+z]\uparrow] + [;\ Z \mapsto 3;\ ]))\uparrow$.

At this point, we can either apply reconfiguration steps, or the innermost result operator (indeed, the program does no longer refer to any variable bound at this level). By choosing the latter reduction alternative, we obtain $\longrightarrow$ $(\mathsf{freeze}_{Z\mapsto Z}[z \mapsto Z;\ ;\ \mid 2+z] + [;\ Z \mapsto 3;\ ])\uparrow$, which eventually reduces to 5.

The R-calculus enjoys the Church-Rosser property.

Theorem 1.1 (Church-Rosser) *The R-calculus reduction relation* $\longrightarrow$ *is confluent.*

2 A call-by-need strategy

The R-calculus can be equipped with different strategies, which can be used to encode/generalize other primitive calculi, such as lambda calculi and module calculi. In this extended abstract, we outline a call-by-need strategy which should lead to a generalization, including reconfiguration features, of call-by-need lambda-calculi as in [8].

Rule ($\mathcal{E}$) now deals with (one hole) evaluation contexts rather than (arbitrary) contexts. Contexts $\mathcal{D}[x, x_n]$ are used to denote a set of declarations s.t. evaluation of x transitively depends on evaluation of x_n, as in [8]. Values also contain a new constant $\bullet$, which arises due to cycles. In particular, we

Evaluation contexts, values and answers

$$\begin{array}{lll}
\mathcal{E}^{\mathsf{ev}} & ::= & \Box \mid [\iota;\, o;\, \rho \mid \mathcal{E}^{\mathsf{ev}}] \mid [\iota;\, o;\, \mathcal{D}[x,x_n]\,, x_n \mapsto \mathcal{E}^{\mathsf{ev}}_n, \rho \mid \mathcal{E}^{\mathsf{ev}}[x]] \mid \mathcal{E}^{\mathsf{ev}} + e \\
 & \mid & [\iota, x \mapsto X;\, o;\, \rho \mid \mathcal{E}^{\mathsf{ev}}_1[x]] + \mathcal{E}^{\mathsf{ev}}_2 \mid {}_{\sigma^\iota|}\mathcal{E}^{\mathsf{ev}}{}_{|\sigma^o} \mid \mathsf{freeze}_\sigma \mathcal{E}^{\mathsf{ev}} \mid \mathcal{E}^{\mathsf{ev}}{\downarrow_X} \mid \mathcal{E}^{\mathsf{ev}}{\uparrow} \\
\mathcal{D}[x,x_n] & ::= & x \mapsto \mathcal{E}^{\mathsf{ev}}[x_1]\,, x_1 \mapsto \mathcal{E}^{\mathsf{ev}}_1[x_2]\,, \ldots, x_{n-1} \mapsto \mathcal{E}^{\mathsf{ev}}_{n-1}[x_n] \\
v \in \mathsf{Val} & ::= & [\iota;\, o;\, \rho] \mid \mathcal{R}[\iota;\, o;\, \rho \mid v] \mid \bullet \\
a \in \mathsf{Ans} & ::= & v \mid (\mathcal{R}[\iota;\, o;\, \rho \mid a]){\uparrow}
\end{array}$$

Reconfiguration and substitution

$$\textbf{(reconf)}\ \frac{\mathcal{R}[\iota;\, o;\, \rho] \xrightarrow[\alpha]{} \mathcal{R}'[\iota';\, o';\, \rho']}{\mathcal{R}[\iota;\, o;\, \rho \mid \mathcal{E}^{\mathsf{ev}}[x]] \xrightarrow[\mathsf{need}]{} \mathcal{R}'[\iota';\, o';\, \rho' \mid \alpha\,(\mathcal{E}^{\mathsf{ev}}[x])]}\quad \begin{array}{l} x \notin \mathsf{HB}\,(\mathcal{E}^{\mathsf{ev}}) \\ x \in \mathsf{dom}(\iota) \end{array}$$

$$\textbf{(subst-prg)}\ \frac{}{[\iota;\, o;\, \rho, x \mapsto v \mid \mathcal{E}^{\mathsf{ev}}[x]] \xrightarrow[\mathsf{need}]{} [\iota;\, o;\, \rho, x \mapsto v \mid \mathcal{E}^{\mathsf{ev}}\,\{v\}]}\quad x \notin \mathsf{HB}\,(\mathcal{E}^{\mathsf{ev}})$$

$$\textbf{(subst)}\ \frac{}{[\iota;\, o;\, \mathcal{D}[x,x_n]\,, x_n \mapsto v, \rho \mid \mathcal{E}^{\mathsf{ev}}[x]] \xrightarrow[\mathsf{need}]{} [\iota;\, o;\, \mathcal{D}[x,x_n\,\{v\}]\,, x_n \mapsto v, \rho \mid \mathcal{E}^{\mathsf{ev}}[x]]}\quad x \notin \mathsf{HB}\,(\mathcal{E}^{\mathsf{ev}})$$

Binding re-association

$$\textbf{(assoc-input)}\ \frac{\mathcal{R}[\iota_2;\, o_2;\, \rho_2] \xrightarrow[\alpha]{} \mathcal{R}'[\iota'_2;\, o'_2;\, \rho'_2]}{\begin{array}{c}[\iota_1;\, o_1;\, \rho_1, x \mapsto (\mathcal{R}[\iota_2;\, o_2;\, \rho_2 \mid a]){\uparrow} \mid \mathcal{E}^{\mathsf{ev}}[x]] \xrightarrow[\mathsf{need}]{} \\ [\iota_1;\, o_1;\, \rho_1, x \mapsto (\mathcal{R}'[\iota'_2;\, o'_2;\, \rho'_2 \mid \alpha\,(a)]){\uparrow} \mid \mathcal{E}^{\mathsf{ev}}[x]]\end{array}}\quad \begin{array}{l} x \notin \mathsf{HB}\,(\mathcal{E}^{\mathsf{ev}}) \\ (\mathsf{FV}(a) \cup \mathsf{FV}(\rho_2)) \cap \mathsf{dom}(\iota_2) \neq \emptyset \end{array}$$

$$\textbf{(assoc)}\ \frac{}{\begin{array}{c}[\iota_1;\, o_1;\, \rho_1, x \mapsto (\mathcal{R}[\iota_2;\, o_2;\, \rho_2 \mid a]){\uparrow} \mid \mathcal{E}^{\mathsf{ev}}[x]] \xrightarrow[\mathsf{need}]{} \\ [\iota_1;\, o_1;\, \rho_1, x \mapsto a, \rho_2 \mid \mathcal{E}^{\mathsf{ev}}[x]]\end{array}}\quad \begin{array}{l} x \notin \mathsf{HB}\,(\mathcal{E}^{\mathsf{ev}}) \\ (\mathsf{FV}(a) \cup \mathsf{FV}(\rho_2)) \cap \mathsf{dom}(\iota_2) = \emptyset \\ \mathsf{dom}(\rho_2) \cap (\mathsf{FV}(o_1) \cup \mathsf{FV}(\rho_1)) = \emptyset \end{array}$$

Lifting

$$\textbf{(sum-lift)}\ \frac{}{[\iota_1;\, o_1;\, \rho_1 \mid \mathcal{E}^{\mathsf{ev}}[x]]{\uparrow} + [\iota_2;\, o_2;\, \rho_2] \xrightarrow[\mathsf{need}]{} [\iota_1;\, o_1;\, \rho_1 \mid \mathcal{E}^{\mathsf{ev}}[x] + [\iota_2;\, o_2;\, \rho_2]]{\uparrow}}\quad \begin{array}{l} x \notin \mathsf{HB}\,(\mathcal{E}^{\mathsf{ev}}) \\ x \in \mathsf{dom}(\iota_1) \end{array}$$

$$\textbf{(reduct-lift)}\ \frac{}{{}_{\sigma^\iota|}[\iota;\, o;\, \rho \mid \mathcal{E}^{\mathsf{ev}}[x]]{\uparrow}_{|\sigma^o} \xrightarrow[\mathsf{need}]{} \left[\iota;\, o;\, \rho \mid {}_{\sigma^\iota|}\mathcal{E}^{\mathsf{ev}}[x]_{|\sigma^o}\right]{\uparrow}}\quad \begin{array}{l} x \notin \mathsf{HB}\,(\mathcal{E}^{\mathsf{ev}}) \\ x \in \mathsf{dom}(\iota) \end{array}$$

$$\textbf{(freeze-lift)}\ \frac{}{\mathsf{freeze}_\sigma([\iota;\, o;\, \rho \mid \mathcal{E}^{\mathsf{ev}}[x]]{\uparrow}) \xrightarrow[\mathsf{need}]{} [\iota;\, o;\, \rho \mid \mathsf{freeze}_\sigma(\mathcal{E}^{\mathsf{ev}}[x])]{\uparrow}}\quad \begin{array}{l} x \notin \mathsf{HB}\,(\mathcal{E}^{\mathsf{ev}}) \\ x \in \mathsf{dom}(\iota) \end{array}$$

Fig. 3. Call-by-need reduction rules

consider a sequence of declarations of the shape $\mathcal{D}[x, x]$ to be equivalent to $\bullet$. Hence, $\bullet$ represents a program that provably diverges, as in [8].

Rules (**m-sum**), (**m-reduct**) and (**m-freeze**) are as in Fig.2 (hence are not reported). We do not allow module component simplification, hence contexts $[\iota;\ \mathcal{O};\ \rho]$ and $[\iota;\ o;\ \mathcal{L}]$ are removed, and also rules (**m-subst**) and (**m-subst-out**). Indeed, the intuition is that simplification of module components only takes place when triggered by the executing program (see rule (**subst**) below).

In rule (**reconf**), reconfiguration now only takes place when the execution of the program needs a deferred variable. Indeed, in this case reconfiguration steps could possibly make the variable local. Note also that, since there are no longer rules (**m-subst**) and (**m-subst-out**), this rule can only be applied with a premise which is a step of application of a module operator, hence only deals with reconfiguration steps. In rule (**subst-prg**), substitution of an instance of a local variable x with its defining expression inside the program now only takes place when the execution of the program needs this variable, and the defining expression has already been reduced to a value. If this is not the case, then the evaluation of the definition of x is triggered, which can trigger, recursively, evaluation of other definitions, until a local variable x_n whose evaluation does not depend on any other is found (evaluation context $[\iota;\ o;\ \mathcal{D}[x, x_n]\,, x_n \mapsto \mathcal{E}^{\mathsf{ev}}_n, \rho \mid \mathcal{E}^{\mathsf{ev}}[x]]$). When the definition of a local variable x_n is reduced to a value, this value can be used to replace a needed reference to x_n in another definition (rule (**subst**)).

Rules (**assoc-input**) and (**assoc**) are new rules used in the call-by-need strategy, inspired to the (**assoc**) rule in [8]. These rules are used when the program needs a variable x whose definition is neither a value, nor an expression which can be further reduced, but an *answer* (roughly speaking, a values containing free variables inserted in a context which can provide definitions for these variables). In this case, a call-by-need strategy should avoid to evaluate these definitions if not necessary. If there are no references to deferred variables (rule (**assoc**)), then it is possible to eliminate a reconfiguration level and to rearrange local bindings. In the other case (rule (**assoc-input**)), it is necessary to first perform reconfiguration steps until there are no longer references to deferred variables.

Rules (**sum-lift**), (**reduct-lift**) and (**freeze-lift**) are also new rules used in the call-by-need strategy, inspired to the (**lift**) rule in [8]. These rules are used when the program needs a deferred variable x which cannot be provided at the current configuration level, which is a basic configuration (hence, no more reconfiguration steps can be performed), but could be possibly provided after extracting the program by the outer configuration level. In this case, it is possible to "move inside" the enclosing module operator.

Finally, rule (**run**) is like in the calculus, whereas rule (**res**) is removed (replaced by the (**assoc-input**) and (**assoc**) rules).

We conjecture that the following properties of the call-by-need strategy hold.

Theorem 2.1 (Soundness and completeness of $\xrightarrow[\text{need}]{}$)
If $e \xrightarrow[\text{need}]{} e'$, then $e \longrightarrow^{*} e'$.
If $e \longrightarrow^{*} v$, then there exists a such that $e \xrightarrow[\text{need}]{}^{*} a$.

References

[1] D. Ancona, S. Fagorzi, and E. Zucca. A calculus for dynamic linking. In *ICTCS 2003*, LNCS 2841, pages 284–301, 2003.

[2] D. Ancona, S. Fagorzi, and E. Zucca. A calculus for dynamic reconfiguration with low priority linking. *Electonical Notes in Theoretical Computer Science*, 2004. In WOOD'04: Workshop on Object-Oriented Developments. To appear.

[3] D. Ancona, S. Fagorzi, and E. Zucca. A calculus with lazy module operators. In *TCS 2004 (IFIP Int. Conf. on Theoretical Computer Science)*, pages 423–436. Kluwer Academic Publishers, 2004.

[4] D. Ancona, S. Fagorzi, and E. Zucca. Mixin modules for dynamic rebinding. In *TGC 2005 -Symposium on Trustworthy Global Computing*, LNCS. Springer, April 2005. To appear.

[5] D. Ancona and E. Zucca. A primitive calculus for module systems. In *PPDP'99*, LNCS 1702, pages 62–79. Springer, 1999.

[6] D. Ancona and E. Zucca. A calculus of module systems. *Journ. of Functional Programming*, 12(2):91–132, 2002.

[7] Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Information and Computation*, 139(2):154–233, Dec. 1997.

[8] Z. M. Ariola and M.Felleisen. The call-by-need lambda calculus. *Journ. of Functional Programming*, 7(3):265–301, 1997.

[9] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *ICFP 2003*, pages 99–110. ACM Press, 2004.

[10] S. Fagorzi. *Module Calculi for Dynamic Reconfiguration.* PhD thesis, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 2005.

[11] T. Hirschowitz, X. Leroy, and J. B. Wells. Call-by-value mixin modules: Reduction semantics, side effects, types. In *ESOP 2003*, LNCS 2986, pages 64–78. Springer, 2004.

[12] E. Machkasova and F.A. Turbak. A calculus for link-time compilation. In *ESOP 2000*, LNCS 1782, pages 260–274. Springer, 2000.

[13] J. B. Wells and R. Vestergaard. Confluent equational reasoning for linking with first-class primitive modules. In *ESOP 2000*, LNCS 1782, pages 412–428. Springer, 2000.

Splitting Mobility and Communication in Boxed Ambients

Pablo Garralda [1,2] Adriana Compagnoni [1,3]

Computer Science Department
Stevens Institute of Technology
Hoboken, NJ - U.S.

Abstract

Stemming from our previous work on **BACI**, a boxed ambients calculus with communication interfaces, we define a new calculus that further enhances communication mechanisms and mobility control by introducing multiple communication ports, access control lists, and port hiding.

The development of the calculus is mainly focused on three objectives: separation of concerns between mobility and communication, fine-grained controls, and locality.

Communication primitives use ports to establish communication channels between ambients, while ambient names are only used for mobility. Port names are used in communications with children ambients as well as in communications with parent ambients, providing extra information to the communicating parties. The introduction of multiple ports allows for extra control in communications and a direct implementation of dedicated channels such as those used for ftp, ssh, or other services.

In order to control mobility, the calculus includes co-capabilities *à la* Safe Ambients, but with the addition of *access control lists.* These lists contain the names of the ambients that are allowed to enter or exit the ambient with that co-capability.

The resulting calculus not only provides more flexibility and expressiveness than **BACI**, but also enables simpler implementations using more powerful constructs for communication and mobility. We establish the basic meta-theory of the calculus by proving a subject reduction result.

Key words: AMBIENTS, MOBILITY, COMPUTING MODELS

[1] This research was supported in part by the National Science Foundation project No. CCR-0220286 ITR: Secure Electronic Transactions. The funding bodies are not responsible for any use that might be made of the results presented here.
[2] Email: `pablo@cs.stevens.edu`
[3] Email: `abc@cs.stevens.edu`

1 Introduction

In Cardelli and Gordon's Mobile Ambients (MA)[6], ambients represent nested computational environments containing data and live computation. Ambients are capable of moving under the influence of the process they enclose and can dissolve their perimeter with an *open* operation. Mobile Ambients provide a direct characterization of computational processes as well as computational devices.

Boxed Ambients (BA) [3] evolved from MA, by removing the ability of an ambient of dissolving its boundary. In BA, an ambient is a "box" that cannot be opened. This notion of closed ambient provides complete encapsulation of the agents they contain. To enable the communication lost by disabling the open operation, ambients are equipped with communication channels to exchange information with adjacent ambients (parent and children ambients).

Both in MA and BA, ambient mobility is commanded by processes inside the ambient. The commands for mobility are called *capabilities*. The capabilities tell an ambient to open or move inside or outside another ambient. Unrestricted mobility, however, can lead to undesired *interferences* between two concurrent processes. Addressing this concern, control over capabilities was first introduced in Safe Ambients [10] and later used in New Boxed Ambients (NBA) [4] in the form of *co-capabilities*. A capability can be exercised only in the presence of a matching co-capability. Hence, in order to enter an ambient using the in capability, that ambient must contain a matching $\overline{\mathsf{in}}$ co-capability authorizing that access; similarly for exiting using the out capability.

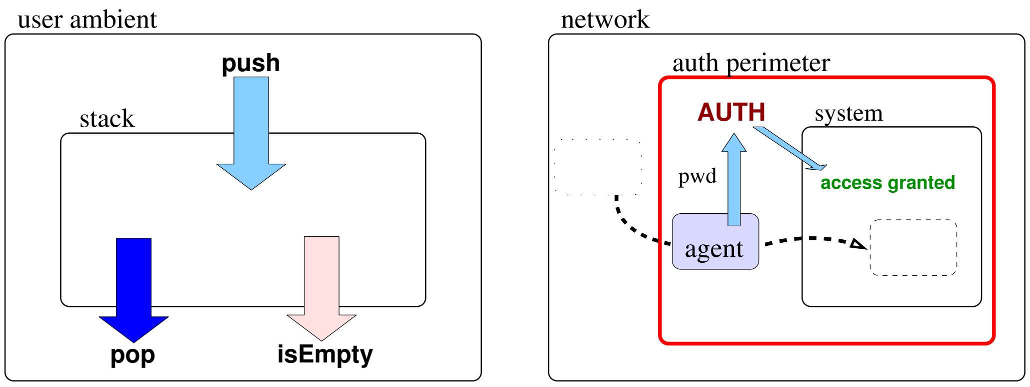
Bonelli *et al.* [1] introduced the notion of *local views* in the calculus of *Boxed Ambients with Communication Interfaces* (**BACI**). In this calculus, each ambient has an associated communication *port* and a *local view*. The communication port is used for sending and receiving messages to and from other ambients, and the local view represents the communication types that are used by the processes enclosed inside the ambient. **BACI** is flexible enough to allow an ambient to communicate with different parents using different types. However, this flexibility came with the price of a rather complex syntax and some run-time type checking required to guarantee type safety.

In this paper, we present an enhanced and simplified version of **BACI** called **BACIv2**. In this version, we share the same goals present in the original version: to stress the separation between communication and mobility and to reduce the amount of global information in the calculus. However, this time, we achieve a better trade-off between locality, expressiveness and the calculus complexity.

In **BACIv2**, processes inside an ambient can use multiple ports –each one with its own communication type– to communicate with other neighboring processes within or outside the ambient. This allows, for instance, the straightforward specification of a host exposing several services like `ssh` or ftp, using a different port for each service.

Notice that ports can be encoded using dedicated ambients. However, having multiple ports as primitives has the advantage of not requiring the co-capabilities necessary for the mobility of such ambients.

Another application of multiple ports is the implementation of data structures such as a `stack`. We can encapsulate the implementation of the stack with a single ambient, using different ports for each of the stack operations as depicted in Figure 1(a).



(a) Stack ambient using multiple ports (b) Dynamic ambient access control

Fig. 1. Some examples in **BACIv2**.

BACIv2 also enhances the mobility control by using fine-grained co-capabilities, where each co-capabilities contains an *access control list* to restrict access to those ambients in the list, enabling dynamic access control. Access control lists can be used to implement access control using passwords similar to the mechanisms found in NBA [5].

Figure 1(b) depicts an ambient that, in order to enter a restricted ambient, needs to authenticate itself by sending the password to an authenticator process located outside the restricted ambient. Next, the authenticator process validates the password and instructs the restricted ambient to grant access to the validated ambient. Finally, the restricted ambient allows the access of the incoming ambient, and the operation is completed.

BACIv2 also introduces port name restriction. This restriction is used to create truly private communication channels, preventing undesired communication interferences. Moreover, new ports can be created dynamically using a special primitive called connect. connect binds two different ports: one from the parent ambient and one from a child ambient, using a new (private) port name. This construct creates new communication channels without requiring any previous knowledge of the parent or child ambient names or the ports they use.

Messages:			
M, N	$::=$	α	name
	$\mid$	C	capability
Capabilities:			
C, D	$::=$	$\mathsf{in}\ \alpha$	enter
	$\mid$	$\mathsf{outTo}\ \alpha$	exit
	$\mid$	$C.D$	path
	$\mid$	x	variable
Prefixes:			
π	$::=$	C	capabilities
	$\mid$	$(\tilde{x} : \tilde{\varphi})^{\eta}$	receive
	$\mid$	$\langle \tilde{M} \rangle^{\eta}$	send
	$\mid$	$\overline{\mathsf{in}}\{\chi\}$	co-enter
	$\mid$	$\overline{\mathsf{outTo}}\{\chi\}$	co-exit
	$\mid$	$\mathsf{connect}{\downharpoonleft}(c : \tilde{\varphi})$	connect down
	$\mid$	$\mathsf{connect}{\upharpoonleft}(c : \tilde{\varphi})$	connect up
Access control list:			
χ	$::=$	any	unrestricted
	$\mid$	$\tilde{\alpha}$	list of names
Ports:			
c, u, v			constant
Basic types			
φ	$::=$	amb	ambient
	$\mid$	cap	capability

Processes:			
P, Q	$::=$	$\mathbf{0}$	nil process
	$\mid$	$P \mid Q$	composition
	$\mid$	$!\pi.P$	replication
	$\mid$	$(\boldsymbol{\nu}\, n : \mathsf{amb})P$	restriction
	$\mid$	$(\boldsymbol{\nu}_p\, c : \tilde{\varphi})P$	port restr.
	$\mid$	$\pi.P$	prefixing
	$\mid$	$\alpha[\, P\,]$	ambient
	$\mid$	$[M = N]\{P\}\{Q\}$	equality
Names:			
α, β	$::=$	n	constant
	$\mid$	x	variable
Locations:			
η	$::=$	$\uparrow c$	upward
	$\mid$	$\downarrow c$	downward
	$\mid$	$\star c$	local
Typing environments			
Σ	$::=$	$\emptyset$	empty
	$\mid$	$\Sigma, x : \varphi$	variable
Process interface type			
Γ	$::=$	$\emptyset$	empty
	$\mid$	Γ, τ	interface
Communication types			
τ	$::=$	$c : \tilde{\varphi}$	typed port

Table 1
Syntax of **BACIv2**

2 BACIv2

2.1 Syntax and Semantics

The complete syntax of the calculus is summarized in Table 1. It includes two main syntactic categories: *processes* and *messages*. Messages, ranged over by M and N, include ambient *names* and *capabilities*. Ambient names, ranged over by α and β, can be either constant ambient names or name variables. Capabilities, ranged over by C and D, can be either the capabilities for entering and exiting an ambient, variables or a "path" which is a sequence of capabilities describing a mobility path. In addition to the sets of *variables* and *ambient names*, we also have the set of *ports* used for communication.

Processes, ranged over by P, Q, R, and S, are built from the constructors of *inactivity*, showing the end of a process; *parallel composition* of two processes; *replication*, used for recursion; ambient and port name *restriction*; *prefixing*, where π is an operation that is followed by a continuation process P; a named *ambient* encapsulating a process; and, finally, *message equality testing* for branching to either P or Q depending on whether M is equal to N or not.

The syntax for name and port restriction includes the name of the hidden ambient or port paired with the appropriate type. This is done in order to unify the syntax for the structural congruence rules (omitted here).

Process prefixes can be divided into four different groups: capabilities, message send and receive, restricted co-capabilities and connects. *Capabilities* command an ambient to move inside or outside another ambient. *Send* and *receive* exchange messages between processes possibly at different locations. A *location* is merely a port located in the parent or a child ambient or locally in the same ambient. *Co-capabilities* allow the entrance or exit of a particular set of ambients depending on whether the ambient name is included in its *access control list*. Finally, *connect* can be used to dynamically create a new port (of type $\tilde{\varphi}$) between child and parent ambients.

Capabilities in **BACIv2** are slightly different from the ones in Safe Ambients or NBA. Here, co-capabilities are not included as capabilities.[4] Moreover, the out(To) capability refers to the target ambient instead of referring to the ambient that is being exited. This implies that the moving ambient must know its destination. However, after executing outTo, a process can be certain of its current location. This is more difficult to assert with the standard out capability found in other ambient calculi.

Send and receive use locations to address a particular port in a parent or a child. In order to establish a communication, both send and receive must have matching port names.

This can be seen in the INPUT reduction rule:

(INPUT $\downarrow$-$\uparrow$)

$$(\tilde{x} : \tilde{\varphi})^{\downarrow c}.P \mid n[\, \langle \tilde{M} \rangle^{\uparrow c}.Q \mid R \,] \longrightarrow P\{\tilde{x} := \tilde{M}\} \mid n[\, Q \mid R \,]$$

where $\downarrow c$ matches $\uparrow c$.

Only capabilities and ambient names can be sent as messages. However, the syntax can be easily extended to allow other kinds of messages, such as integers or booleans.

Co-capabilities $\overline{\mathsf{in}}\{\}$ and $\overline{\mathsf{outTo}}\{\}$ have a list of ambients that are allowed to enter (or exit) using that co-capability. Additionally, the label any can be used to denote unrestricted access. Here is the reduction rule for EXIT:

(EXIT)

$$k[\, m[\, n[\, \mathsf{outTo}\ k.P_1 \mid P_2 \,] \mid Q_1 \,]\ \overline{\mathsf{outTo}}\{\chi\}.Q_2 \mid Q_3 \,] \longrightarrow$$
$$k[\, m[\, Q_1 \,] \mid n[\, P_1 \mid P_2 \,] \mid Q_2 \mid Q_3 \,]$$

$$\text{where } \chi \text{ is } \{\tilde{n}_1, n, \tilde{n}_2\} \text{ or } \mathsf{any}$$

Notice that the outTo refers to the grand-parent ambient instead of the parent ambient. In this way, the moving ambient can specify the target ambient.

[4] This means that co-capabilities cannot appear as messages so, they cannot be sent or received.

Finally, the connect prefix allows a process to create a new private channel shared with another process located in the parent or a child ambient.

(connect)

$$\mathsf{connect}{\downarrow}(u : \tilde{\varphi}).P \mid n[\,\mathsf{connect}{\uparrow}(v : \tilde{\varphi}).Q \mid R\,] \quad \longrightarrow$$

$$(\boldsymbol{\nu}_p\, c : \tilde{\varphi})(P\{u := c\} \mid n[\,Q\{v := c\} \mid R\,])$$

where c is a fresh port name.

The complete set of reduction rules (omitted here due to lack of space) include standard stuctural rules as well as rules for (output) and (enter) which are analogous to the ones presented above.

3 Typing Judgments

Typing environments are defined by the following grammar.

$$\begin{array}{rll} \Sigma ::= & \emptyset & \text{empty environment} \\ \mid & \Sigma, x : \varphi & \text{extension} \end{array}$$

Typing environments are assumed to assign a unique type to each name in its domain.

There exist two different typing judgments, one for messages and one for processes:

- $\Sigma \vdash M : \varphi$, read "$M$ is a well-formed message of type φ in Σ", and
- $\Sigma \rhd P : \Gamma$, read "P is a well-formed process of type is Γ in Σ".

In contrast to other systems, there is no communication type associated with an ambient name, instead an ambient name has the constant type amb.

In the judgment $\Sigma \rhd P : \Gamma$, the process interface Γ exhibits the communication types used by P. Γ assigns communication types to each free port c used in P. Ports hidden using port restriction $(\boldsymbol{\nu}_p\, c : \tilde{\varphi})$ do not appear in Γ, since their type is declared in the restriction operation. This fact is reflected in the corresponding typing rule:

(proc-p-res)

$$\frac{\Sigma \rhd P : \Gamma, c : \tilde{\varphi}}{\Sigma \rhd (\boldsymbol{\nu}_p\, c : \tilde{\varphi})P : \Gamma}$$

Similarly, the connect operation also abstracts port names, hiding them from the resulting Γ. The complete set of typing rules were omitted.

The type system guarantees that communication inside ambients and across ambient boundaries never leads to type mismatches. This is formalized as the Subject Reduction theorem:

Theorem 3.1 (Subject Reduction)
If $\Sigma \rhd P : \Gamma$ and $P \longrightarrow Q$, then $\Sigma \rhd Q : \Gamma$.

Proof. By induction on the derivation of $P \longrightarrow Q$. □

4 Examples

4.1 Stack

In this example, we model a stack for storing names using multiple ports. We consider three primitive operations applied to stacks: *push*, *pop*, *isEmpty*. *Push* takes an element and inserts it on the top of the stack; *pop*, on the other hand, removes the element on top and returns it as a result of the operation. Finally, *isEmpty* is used to query the stack whether it is empty or not.

For the implementation of these operations we use different ports. In fact, we use two ports per operation: one for receiving the request and the other to deliver the response (possibly excepting the *push* operation that does not require a response). In general, we name each port using the name of the operation along with a subindex indicating whether it is receiving a request or submitting a response.

The stack is represented by the following ambient.

$$STACK = stack[\, INTERNALS \mid !PUSH \mid !POP \mid !ISEMPTY \,]$$

The processes inside the stack can be divided into four parts: *INTERNALS* which keeps the internal state and manages some internal operations, and the processes that manage each operation.

The implementation uses a linked list of nodes. Each node is an ambient containing a “stacked” value. A local port called *top* is used as a variable to store the name of the node that holds the top value of the stack.

The *INTERNALS* part holds the current state of the stack. Initially, the stack is empty, so *INTERNALS* only contains the initial state: *INITSTATE*.

$INITSTATE$[5] $= \langle empty \rangle^{\star top}$

We use the name *empty* to denote that the stack is empty; therefore, *empty* is “stored” as the *top*.

In order to manipulate this internal state, we introduce two syntactic definitions: **gettop** and **settop**.

$$\textbf{gettop}(x).P = (x)^{\star top}.(P \mid \langle x \rangle^{\star top})$$
$$\textbf{settop}(n).P = P \mid (x)^{\star top}.\langle n \rangle^{\star top}$$

Gettop retrieves the name of the node ambient that holds the top value of the stack and binds it to a variable. On the other hand, **settop** takes an ambient name (*i.e.* a node name) as an argument and sets that name to be the top of the stack.

Using these macros, we can define the *ISEMPTY* operation:

$ISEMPTY$[6] $=$

5 In general, we omit the nil process at the end of a process expression for sake of readability.

6 We use a “_” instead of a variable name to denote that the value received on this port is a dummy value just used to establish the communication.

$$(\_)^{\Uparrow isEmpty_{req}}.\mathbf{gettop}(t).[t = empty]\{\langle true\rangle^{\Uparrow isEmpty_{res}}\}\{\langle false\rangle^{\Uparrow isEmpty_{res}}\}$$

First, a request is received on $isEmpty_{req}$ port. Then, after retrieving the top name, the process checks if it is equal to *empty* or not, returning the names *true* or *false* over the port $isEmpty_{res}$.

The *PUSH* operation receives a value v over the port $push_{req}$ and creates a new node with that value.

$$PUSH = (v)^{\Uparrow push_{req}}.NEWNODE(v)$$

The newly created node stores the value at the top of the stack and records the name of the old top of the stack as the next node in the linked list.

$$NEWNODE(x) =$$
$$(\boldsymbol{\nu}\, node : \mathsf{amb})\mathbf{gettop}(t).(node[\, NODE\_INT(x,t)\,] \mid settop(node))$$

Internally, each node has a process that is ready to release the stored value after it is triggered by the entrance of a special messenger ambient called *popper*. As its name suggests, the sole purpose of the *popper* ambient is as a signal to the node that is being popped from the stack.

$$NODE\_INT(v,t) = \overline{\mathsf{in}}\{popper\}.\langle(v,t)\rangle^{\Uparrow release}$$

After this signal, the node retrieves the stored value and also the name of the following node in the stack. This feature of the nodes is used by the *POP* operation. After receiving a request via the pop_{req} port, the *POP* operation sends a messenger ambient to the top node to start the removing process.

$$POP = (\_)^{\Uparrow pop_{req}}.\mathbf{gettop}(t).(popper[\,\mathsf{in}\ t\,] \mid (v,t')^{\Downarrow release}.\mathbf{settop}(t').\langle v\rangle^{\Uparrow pop_{res}})$$

Simultaneously, a parallel process waits for the response from the top node through the *release* port. After that response is received, the next node in the stack is placed at the top of the stack; finally, the value that was on top is retrieved from the stack ambient using the pop_{res} port.

4.2 Authentication Using Passwords

This example shows a simplified implementation of a mechanism similar to the capabilities in NBA [5] that use passwords as an access control to enter or exit into other ambients. The example corresponds to the situation depicted in Figure 1(b).

An *agent* request access to a system *sys* by sending its credentials to *AUTH*, the process in charge of the authentication procedure.

$$agent[\,\mathsf{in}\{sys\ \mathbf{with}\ passwd\}.P\,] \ \mid\ AUTH \ \mid\ sys[\,ACCESS \mid SYS\,]$$

AUTH is continuously listening for incoming requests. On each request, it tests the given password, either granting the requested access or informing that the access was denied.

$$AUTH = !(m, m_{pwd})^{\Downarrow auth_{req}}.[m_{pwd} = good_{pwd}]\{GRANT\}\{DENY\}$$

$GRANT = \langle granted\rangle^{\downarrow auth_{res}}.\langle m\rangle^{\downarrow grant}$
$DENY = \langle denied\rangle^{\downarrow auth_{res}}$

The agent uses the macro $\mathsf{in}\{n$ **with** $passwd\}$ to request access.

$\mathsf{in}\ \{n\ \textbf{with}\ passwd\}.P = REQUEST_{agent}(sys, passwd) \mid (\_)^{\star go}.\mathsf{in}\ n.P)$

The $REQUEST$ process continuously tries to get access by sending its credentials and waiting for a response each time. Therefore, the continuation process P is blocked until the access is granted.

$$REQUEST_m(n, pwd) = (\boldsymbol{\nu}_p\ do\_auth : \mathsf{amb})(\langle try\rangle^{\star do\_auth} \mid !(\_)^{\star do\_auth}.\langle m, pwd\rangle^{\uparrow auth_{req}}.(res)^{\uparrow auth_{res}}[res = granted]\{\langle ok\rangle^{\star go}\}\{\langle retry\rangle^{\star do\_auth}\})$$

where m is the name of the ambient requesting permission to enter the ambient n.

Finally, after $AUTH$ authorizes the access, it communicates with the system to allow the access of the authenticated process via a port named $grant$.

$ACCESS = !(a)^{\uparrow grant}.\overline{\mathsf{in}}\{a\}$

5 Summary and Conclusions

Continuing our earlier work on **BACI**[1], the calculus presented here aims at further decoupling communication from mobility. This is achieved by the introduction of multiple ports that are exclusively used for communication. The addition of access control lists to co-capabilities enables better mobility control. These fine-grained co-capabilities are useful in encoding specific mechanisms that can be used transparently, without any side-effects on the ambient mobility control. Multiple ports reduce the need of complex encodings to implement several communication channels, rendering specifications closer to the reality being modeled. However, excessive use of ports could be reduced by extending the calculus with session types [9] for ports. A type system including session types would enforce stronger type safety guarantees. Moreover, the introduction of co-capabilities that also bind the name of the entering ambient, similar to NBA [5] co-capabilities, could give each ambient additional control over its child ambients.

Acknowledgement

We are grateful to Mariangiola Dezani-Ciancaglini for enlightening discussions and to Elsa Gunter, Joëlle Despeyroux, Eduardo Bonelli, Ricardo Medel, and Alejandro Russo for many useful comments and constructive criticism.

References

[1] Eduardo Bonelli, Adriana B. Compagnoni, Mariangiola Dezani-Ciancaglini, and Pablo Garralda. Boxed ambients with communication interfaces. In Jirí Fiala, Václav Koubek, and Jan Kratochvíl, editors, *MFCS*, volume 3153 of *Lecture Notes in Computer Science*, pages 119–148. Springer, 2004.

[2] M. Bugliesi and G. Castagna. Secure Safe Ambients. In *POPL'01, ACM Symposium on Principles of Programming Languages*, pages 222–235. ACM Press, 2001.

[3] M. Bugliesi, G. Castagna, and S. Crafa. Access Control for Mobile Agents: the Calculus of Boxed Ambients. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):57 – 124, Jan 2004.

[4] Michele Bugliesi, Silvia Crafa, Massimo Merro, and Vladimiro Sassone. Communication and Mobility Control in Boxed Ambients. To appear in *Information and Computation.* Extended and revised version of M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication Interference in Mobile Boxed Ambients. In FSTTCS'02, volume 2556 of LNCS, pages 71-84. Springer-Verlag, 2002.

[5] Michele Bugliesi, Silvia Crafa, Massimo Merro, and Vladimiro Sassone. Communication interference in mobile boxed ambients. In *Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science, FST&TCS 2002*, volume 2556 of *LNCS*, pages 71–84. Springer, 2002.

[6] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. Special Issue on Coordination, Daniel Le Métayer Editor.

[7] Giuseppe Castagna and Jan Vitek. Seal: A Framework for Secure Mobile Computations. In Henri E. Bal, Boumediene Belkhouche, and Luca Cardelli, editors, *Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*, pages 47–77, Berlin, 1999. Springer-Verlag.

[8] Mario Coppo, Mariangiola Dezani-Ciancaglini, Elio Giovannetti, and Ivano Salvo. M3: Mobility Types for Mobile Processes in Mobile Ambients. In James Harland, editor, *CATS'03*, volume 78 of *ENTCS*. Elsevier, 2003.

[9] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *Lecture Notes in Computer Science*, pages 22–138. Springer-Verlag, 1998.

[10] Francesca Levi and Davide Sangiorgi. Controlling Interference in Ambients. *Transactions on Programming Languages and Systems*, 25(1):1–69, 2003.

[11] Andrew Phillips, Nobuko Yoshida, and Susan Eisenbach. A distributed abstract machine for boxed ambient calculi. In *ESOP'04*, LNCS. Springer, April 2004.

DCM 2005 Preliminary Version

Abstract Effective Models

Udi Boker [1,2]

School of Computer Science
Tel Aviv University
Tel Aviv 69978, Israel

Nachum Dershowitz [3]

School of Computer Science
Tel Aviv University
Tel Aviv 69978, Israel

Abstract

We modify Gurevich's notion of abstract machine so as to encompass computational models, that is, sets of machines that share the same domain. We also add an effectiveness requirement. The resultant class of "Effective Models" includes all known Turing-complete state-transition models, operating over any countable domain.

Key words: Computational models, Turing machines, ASM, Abstract State Machines, Effectiveness

1 Sequential Procedures

We first define "sequential procedures", along the lines of the "sequential algorithms" of [3]. These are abstract state transition systems, whose states are algebras.

Definition 1.1 [States]

- A *state* is a structure (algebra) s over a (finite-arity) vocabulary $\mathcal{F}$, that is, a domain (nonempty set of elements) D together with interpretations $[\![f]\!]_s$ over D of the function names $f \in \mathcal{F}$.
- A *location* of vocabulary $\mathcal{F}$ over a domain D is a pair, denoted $f(\overline{a})$, where f is a k-ary function name in $\mathcal{F}$ and $\overline{a} \in D^k$.

[1] This work was carried out in partial fulfillment of the requirements for the Ph.D. degree of the first author.
[2] Email: udiboker@tau.ac.il
[3] Email: nachumd@tau.ac.il

- The *value* of a location $f(\overline{a})$ in a state s, denoted $[\![f(\overline{a})]\!]_s$, is the domain element $[\![f]\!]_s(\overline{a})$.
- We sometimes use a term $f(t_1,\ldots,t_k)$ to refer to the location $f([\![t_1]\!]_s,\ldots,[\![t_k]\!]_s)$.
- Two states s and s' over vocabulary $\mathcal{F}$ with the same domain *coincide* over a set T of $\mathcal{F}$-terms if $[\![t]\!]_s = [\![t]\!]_{s'}$ for all terms $t \in T$.
- An *update* of location l over domain D is a pair, denoted $l := v$, where $v \in D$.
- The *modification* of a state s into another state s' over the same vocabulary and domain is $\Delta(s,s') = \{l := v' \mid [\![l]\!]_s \neq [\![l]\!]_{s'} = v'\}$.
- A *mapping* $\rho(s)$ of state s over vocabulary $\mathcal{F}$ and domain D via injection $\rho : D \to D'$ is a state s' of vocabulary $\mathcal{F}$ over D', such that $\rho([\![f(\overline{a})]\!]_s) = [\![f(\rho(\overline{a}))]\!]_{s'}$ for every location $f(\overline{a})$ of s.
- Two states s and s' over the same vocabulary with domains D and D', respectively, are *isomorphic* if there is a bijection $\pi : D \leftrightarrow D'$, such that $s' = \pi(s)$.

A "sequential procedure" is like Gurevich's [3] "sequential algorithm", with two modifications for computing a specific function, rather than expressing an abstract algorithm: the procedure vocabulary includes special constants "*In*" and "*Out*"; there is a single initial state, up to changes in *In*.

Definition 1.2 [Sequential Procedures]

- A *sequential procedure* A is a tuple $\langle \mathcal{F}, In, Out, D, \mathcal{S}, \mathcal{S}_0, \tau \rangle$, where: $\mathcal{F}$ is a finite vocabulary; *In* and *Out* are nullary function names in $\mathcal{F}$; D, the procedure *domain*, is a domain; $\mathcal{S}$, its *states*, is a collection of structures of vocabulary $\mathcal{F}$, closed under isomorphism; $\mathcal{S}_0$, the *initial states*, is a subset of $\mathcal{S}$ over the domain D, containing equal states up to changes in the value of *In* (often referred to as a single state s_0); and $\tau : \mathcal{S} \to \mathcal{S}$, the *transition function*, such that:
  - · **Domain invariance**. The domain of s and $\tau(s)$ is the same for every state $s \in \mathcal{S}$.
  - · **Isomorphism constraint**. $\tau(\pi(s)) = \pi(\tau(s))$ for some bijection π.
  - · **Bounded exploration**. There exists a finite set T of "critical" terms, such that $\Delta(s,\tau(s)) = \Delta(s',\tau(s'))$ if s and s' coincide over T.

  Tuple elements of a procedure A are indexed $\mathcal{F}_A$, τ_A, etc.
- A *run* of a procedure A is an infinite sequence $s_0 \leadsto_\tau s_1 \leadsto_\tau s_2 \leadsto_\tau \cdots$, where s_0 is an initial state and every $s_{i+1} = \tau_A(s_i)$.
- A run $s_0 \leadsto_\tau s_1 \leadsto_\tau s_2 \leadsto_\tau \cdots$ *terminates* if $s_i = s_{i+1}$ from some point on.
- The *terminating state* of a terminating run $s_0 \leadsto_\tau s_1 \leadsto_\tau s_2 \leadsto_\tau \cdots$ is its stable state. If there is a terminating run beginning with state s and terminating in state s', we write $s \leadsto^!_\tau s'$.
- The *extensionality* of a sequential procedure A over domain D is the partial

function $f : D \to D$, such that $f(x) = [\![Out]\!]_{s'}$ whenever there's a run $s \rightsquigarrow_{\tau}^{!} s'$ with $[\![In]\!]_s = x$, and is undefined otherwise.

Domain invariance simply ensures that a specific "run" of the procedure is over a specific domain. The isomorphism constraint reflects the fact that we are working at a fixed level of abstraction. See [3, p. 89]. The bounded-exploration constraint ensures that the behavior of the procedure is effective. This reflects the informal assumption that the program of an algorithm can be given by a finite text [3, p. 90].

2 Programmable Machines

The transition function of a "programmable machine" is given by a finite "flat program":

Definition 2.1 [Programmable Machines]

- A *flat program* P of vocabulary $\mathcal{F}$ has the following syntax:

```
if $x_{11} \doteq y_{11}$ and $x_{12} \doteq y_{12}$ and $\ldots$ $x_{1k_1} \doteq y_{1k_1}$
   then $l_1 := v_1$
if $x_{21} \doteq y_{21}$ and $x_{22} \doteq y_{22}$ and $\ldots$ $x_{2k_2} \doteq y_{2k_2}$
   then $l_2 := v_2$
$\vdots$
if $x_{n1} \doteq y_{n1}$ and $x_{n2} \doteq y_{n2}$ and $\ldots$ $x_{nk_n} \doteq y_{nk_n}$
   then $l_n := v_n$
```

  where each $\doteq$ is either '$=$' or '$\neq$', $n, k_1, \ldots, k_n \in \mathbf{N}$, and all the x_{ij}, y_{ij}, l_i, and v_i are $\mathcal{F}$-terms.

- Each line of the program is called a *rule*.

- The *activation* of a flat program P on an $\mathcal{F}$-structure s, denoted $P(s)$, is a set of updates $\{l := v \mid \texttt{if } p \texttt{ then } l := v \in P, [\![p]\!]_s\}$ (under the standard interpretation of $=$, $\neq$, and conjunction), or the empty set $\emptyset$ if the above set includes two values for the same location.

- A *programmable machine* is a tuple $\langle \mathcal{F}, In, Out, D, \mathcal{S}, \mathcal{S}_0, P\rangle$, where all but the last component is as in a sequential procedure (Definition 1.2), and P is a flat program of $\mathcal{F}$.

- The *run* of a programmable machine and its *extensionality* are defined as for sequential procedures (Definition 1.2), where the transition function τ is given by $\tau(s) = s' \in \mathcal{S}$ such that $\Delta(s, s') = P(s)$.

To make flat programs more readable, we combine rules, as in

```
% comment
if cond-1
   stat-1
```

```
      stat-2
   else
      stat-3
```

Analogous to the the main lemma of [3], one can show that every programmable machine is a sequential procedure, and every sequential procedure is a programmable machine.

In contradistinction to those Abstract Sequential Machines (ASMs), we do not have built in equality, booleans, or an undefined in the definition of procedures: The equality notion is not presumed in the procedure's initial state, nor can it be a part of the initial state of an "effective procedure", as defined below. Rather, the transition function must be programmed to perform any needed equality checks. Boolean constants and connectives may be defined like any other constant or function. Instead of a special term for undefined values, a default domain value may be used explicitly.

3 Effective Models

We define an "effective procedure" as a sequential procedure satisfying an "initial-data" postulate (Axiom 3.3 below). This postulate states that the procedures may have only finite initial data in addition to the domain representation ("base structure"). An "effective model" is, then, any set of effective procedures that share the same domain representation.

We formalize the finiteness of the initial data by allowing the initial state to contain an "almost-constant structure". Since we are heading for a characterization of effectiveness, the domain over which the procedure actually operates should have countably many elements, which have to be nameable. Hence, without loss of generality, one may assume that naming is via terms.

Definition 3.1 [Almost-Constant and Base Structures]

- A structure S is *almost constant* if all but a finite number of locations have the same value.
- A structure S of finite vocabulary $\mathcal{F}$ over a domain D is a *base structure* if all the domain elements are the value of a unique $\mathcal{F}$-term. That is, for every element $e \in D$ there exists a unique $\mathcal{F}$-term t such that $[\![t]\!]_S = e$.
- A structure S of vocabulary $\mathcal{F}$ over domain D is the *union* of structures S' and S'' of vocabularies $\mathcal{F}'$ and $\mathcal{F}''$, respectively, over D, denoted $S = S' \uplus S''$, if $\mathcal{F} = \mathcal{F}' \uplus \mathcal{F}''$, $[\![l]\!]_S = [\![l]\!]_{S'}$ for every location l of S', and $[\![l]\!]_S = [\![l]\!]_{S''}$ for every location l of S''.

A base structure is isomorphic to the standard free term algebra (Herbrand universe) of its vocabulary.

Proposition 3.2 *Let S be a base structure over vocabulary G and domain D. Then:*

- *Vocabulary G has at least one nullary function.*
- *Domain D is countable.*
- *Every domain element is the value of a unique location of S.*

Axiom 3.3 (Initial Data) *The procedure's initial states consist of an infinite base structure and an almost-constant structure. That is, for some infinite base structure BS and almost-constant structure AS, and for every initial state s_0, we have $s_0 = BS \uplus AS \uplus \{In\}$ for some In.*

Definition 3.4 [Effective Procedures and Models]

- An *effective procedure* A is a sequential procedure satisfying the initial-data postulate. An effective procedure is, accordingly, a tuple $\langle \mathcal{F}, In, Out, D, \mathcal{S}, \mathcal{S}_0, \tau, BS, AS \rangle$, adding a base structure BS and an almost-constant structure AS to the sequential procedure tuple, defined in Definition 1.2.
- An *effective model* E is a set of effective procedures that share the same base structure. That is, $BS_A = BS_B$ for all effective procedures $A, B \in E$.

A computational model might have some predefined complex operations, as in a RAM model with built-in integer multiplication. Viewing such a model as a sequential algorithm allows the initial state to include these complex functions as oracles [3]. Since we are demanding effectiveness, we cannot allow arbitrary functions as oracles, and force the initial state to include only finite data over and above the domain representation (Axiom 3.3). Hence, the view of the model at the required abstraction level is accomplished by "big steps", which may employ complex functions, while these complex functions are implemented by a finite sequence of "small steps" behind the scenes. That is, (the extensionality of) an effective procedure may be included (as an oracle) in the initial state of another effective procedure. (Cf. the "turbo" steps of [2].)

4 Effective Includes Computable

Turing machines, and other computational methods, can be shown to be effective. We demonstrate below how Turing machines and counter machines can be described by effective models.

Turing Machines.
We consider Turing machines (TM) with two-way infinite tapes. The tape alphabet is $\{0, 1\}$. The two edges of the tape are marked by a special \$ sign. As usual, the state (instantaneous description) of a Turing machine is $\langle Left, q, Right \rangle$, where *Left* is a finite string containing the tape section left of the reading head, q is the internal state of the machine, and *Right* is a finite

string with the tape section to the right to the read head. The read head points to the first character of the *Right* string.

TMs can be described by the following effective model E:

Domain: Finite strings ending with a \$ sign. That is the domain $D = \{0,1\}^*\$$.

Base structure: Constructors for the finite strings (*name*/*arity*): \$/0, *Cons*_0/1, and *Cons*_1/1.

Almost-constant structure:

- Input and Output (nullary functions): *In*, *Out*. The value of *In* at the initial state is the content of the tape, as a string over $\{0,1\}^*$ ending with a \$ sign.
- Constants for the alphabet characters and TM-states (nullary): 0, 1, *q*_0, *q*_1, …, *q*_*k*. Their initial value is irrelevant, as long it is a different value for each constant.
- Variables to keep the current status of the Turing machine (nullary): *Left*, *Right*, and *q*. Their initial values are: *Left* = \$, *Right* = \$, and *q* = *q*_0.
- Functions to examine the tape (unary functions): *Head* and *Tail*. Their initial value, at all locations, is \$.

Transition function: For each Turing machine $m \in \mathrm{TM}$, define an effective procedure $m' \in E$ via a flat program looking like this:

```
if q = q_0  % TM's state q_0
   if Head(Right) = 0
      % write 1, move right, switch to q_3
      Left := Cons_1(Left)
      Right := Tail(Right)
      q := q_3
      % Internal operations
      Tail(Cons_1(Left)) := Left
      Head(Cons_1(Left)) := 1
   if Head(Right) = 1
      % write 0, move left, switch to q_1
      Left := Tail(Left)
      Right := Cons_0(Right)
      q := q_1
      % Internal operations
      Tail(Cons_0(Right)) := Right
      Head(Cons_0(Left)) := 0
if q = q_1   % TM's state q_1
        ...
if q = q_k   % the halting state
   Out := Right
```

The updates for Head and Tail are bookkeeping operations that are really part of the "behind-the-scenes" small steps.

The procedure also requires some initialization, in order to fill the internal functions *Head* and *Tail* with their values for all strings up to the given input string. It sequentially enumerates all strings, assigning their *Head* and *Tail* values, until encountering the input string. The following internal variables (nullary functions) are used in the initialization (Name = initial value): New = \$, Backward = 0, Forward = 1; AddDigit = 0, and Direction = \$.

```
% Sequentially constructing the Left variable
% until it equals to the input In, for filling
% the values of Head and Tail.
% The enumeration is $, 0$, 1$, 00$, 01$, ...
if Left = In % Finished
   Right := Left
   Left := $
else % Keep enumerating
   if Direction = New % default val
      if Head(Left) = $ % $ -> 0$
         Left := Cons_0(Left)
         Head(Cons_0(Left)) := 0
         Tail(Cons_0(Left)) := Left
      if Head(Left) = 0 % e.g. 110$ -> 111$
         Left := Cons_1(Tail(Left))
         Head(Cons_1(Tail(Left)) := 1
         Tail(Cons_1(Tail(Left)) := Tail(Left)
      if Head(Left) = 1 % 01$->10$; 11$->000$
         Direction := Backward
         Left := Tail(Left)
         Right := Cons_0(Right)
   if Direction = Backward
      if Head(Left) = $ % add rightmost digit
         Direction := Forward
         AddDigit := True
      if Head(Left) = 0 % change to 1
         Left := Cons_1(Tail(Left))
         Direction := Forward
      if Head(Left) = 1 % keep backwards
         Left := Tail(Left)
         Right := Cons_0(Right)
   if Direction = Forward % Gather right 0s
      if Head(Right) = $ % finished gathering
         Direction := New
         if AddDigit = 1
            Left := Cons_0(Left)
            Head(Cons_0(Left)) := 0
            Tail(Cons_0(Left)) := Left
            AddDigit = 0
      else
            Left := Cons_0(Left)
            Right := Tail(Right)
            Head(Cons_0(Left)) := 0
            Tail(Cons_0(Left)) := Left
```

Counter Machines.
Counter machines (CM) can be described by the following effective model E: The domain is the natural numbers $\mathbf{N}$. The base structure consists of a nullary function *Zero* and a unary function *Succ*, interpreted as the regular successor over $\mathbf{N}$. The almost-constant structure has the vocabulary (*name*/*arity*): $Out/0, CurrentLine/0, Pred/1, Next/1, Reg\_0, \ldots, Reg\_n/0$, and $Line\_1, \ldots, Line\_k/0$. Its initial data are $True = 1$, $Line\_i = i$, and all other locations are 0. The same structure applies to all machines, except for the number of registers (*Reg_i*) and the number of lines (*Line_i*). For every counter machine $m \in$ CM define an effective procedure $m' \in E$ with the following flat program:

```
% Initialization: fill the values of the
% predecessor function up to the value
% of the input
if CurrentLine = Zero
   if Next = Succ(In)
      CurrentLine := Line_1
   else
      Pred(Succ(Next)) := Next
      Next := Succ(Next)
% Simulate the counter-machine program.
% The values of a,b,c and d are as in
% the CM-program lines.
if CurrentLine = Line_1
   Reg_a := Succ(Reg_a) % or Pred(Reg_a)
   Pred(Succ(Reg_a)) := Reg_a
   if Reg_b = Zero
      CurrentLine := c
   else
      CurrentLine := d
if CurrentLine = Line_2
   ...
% Always:
Out := Reg_0
```

5 Discussion

In [3], Gurevich proved that any algorithm satisfying his postulates can be represented by an Abstract State Procedure. But an ASM is designed to be "abstract", so is defined on top of an arbitrary structure that may contain *non-effective* functions. Hence, it may compute non-effective functions. We have adopted Gurevich's postulates, but added an additional postulate (Axiom 3.3) for effectivity: an algorithm's initial state may contain only finite data in addition to the domain representation. Different runs of the same procedure share the same initial data, except for the input; different procedures of the same model share a base structure.

Here, we showed that Turing machines and counter machines are effective models. In [1], we prove the flip side, namely that Turing machines can simulate all effective models. To cover hypercomputational models, one would need to relax the effectivity axiom or the bounded exploration requirement.

References

[1] Udi Boker and Nachum Dershowitz, A formalization of the Church-Turing Thesis, submitted.

[2] N. G. Fruja and R. F. Stärk. The hidden computation steps of Turbo Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines — Advances in Theory and Applications, 10th International Workshop, ASM 2003, Taormina, Italy*, pages 244–262. Springer-Verlag, Lecture Notes in Computer Science 2589, 2003.

[3] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1:77–111, 2000.

iRho: the Software
[System Description]

Luigi Liquori

INRIA, France

Abstract

This paper describes the first implementation of an interpreter for iRho, an imperative version of the Rewriting-calculus, based on pattern-matching, pattern-abstractions, and side-effects. The implementation contains a parser and a call-by-value evaluator in Natural Semantics; everything is written using the programming language Scheme. The core of this implementation (evaluator) is *certified* using the proof assistant Coq.

Performances are honest compared to the minimal essence of the implementation. This document describes, by means of examples, how to use and to play with iRho. The final objective is to make iRho a, so called, *agile programming language*, in the vein of some useful scripts languages, like, *e.g*. Python and Ruby, where proof search is not only feasible but easy.

1 Introduction to the Rewriting Calculus

One of the main advantages of the *rewriting-based* languages, like Elan [16], Maude [14], ASF+SDF [19, 2], OBJ* [10], Stratego [18] is *pattern-matching*. Pattern-matching allows to discriminate between alternatives: once a pattern is recognized, a pattern is associated with an action. The corresponding pattern is thus rewritten in an appropriate instance of a new one.

Another advantage of rewriting-based languages (in contrast with ML or Haskell) is the ability to handle *non-determinism* in the sense of a collection of results: pattern matching need not to be exclusive, *i.e*. multiple branches can be "fired" simultaneously. An empty collection of results represents an application failure, a singleton represents a deterministic result, and a collection with more than one element represents a non-deterministic choice between the elements of the collection. This feature makes the calculus quite close to logic languages too; this means that it is possible to make a product of two patterns, thus applying "in parallel" both patterns.

Optimistic/pessimistic semantics can then be imposed to the calculus by defining successful results as products that have at least a component (respectively all the components) different from error values. It should be possible to obtain a logic language on top of it by redefining appropriate strategy for backtracking.

Useful applications lie in the field of pattern recognition, and strings/trees manipulation. Pattern-matching has been widely used in functional and logic programming, as ML [15,7], Haskell [11], Scheme [17], or Prolog [9]; generally, it is considered a convenient mechanism for expressing complex requirements about the function's argument, rather than a basis for an *ad hoc* paradigm of computation.

The *Rewriting-calculus* (Rho) [4, 5] integrates in a uniform way, matching, rewriting, and functions; its abstraction mechanism is based on the rewrite rule formation: in a term of the form $P \to A$, one abstracts over the pattern P. Note that the Rewriting-calculus is a generalization of the Lambda-calculus if the pattern P is a variable. If an abstraction $P \to A$ is applied to the term B, then the evaluation mechanism is based on the binding of the free variables present in P to the appropriate subterms of B applied to A. Indeed, this binding is achieved by matching P against B. One of the advantages of matching is that it is "customizable" with more sophisticated matching theories, *e.g*. the associative-commutative one.

This year, an imperative extension enhancing the (functional) Rho, was presented in [13]; shortly, we introduced imperative features like referencing (*i.e*. "malloc-like ops", ref expr), dereferencing (*i.e*. "goto-memory ops", ! expr), and assignments operators ($\mathsf{X} := \mathsf{expr}$). The associated type system was enriched with dereferencing-types (*i.e*. pointer-types, int ref), and product-types (*e.g*. $\mathsf{int} \to \mathsf{int} \wedge \mathsf{nat} \to \mathsf{nat}$). The mathematical content of this extension was validated by the help of the semiautomatic proof assistant Coq. A toy software prototype, mimicking the mathematical behavior of the dynamic semantics was also implemented in Scheme. This paper introduces shortly the first LGPL release of the software; a parser has been implemented and more syntactic sugar has been added to make the interpreted easier to use. The core kernel is conform to the semantics specification of [13]; future releases will also come with a machine assisted "certificate" that the design choices are correct. We may envisage also proof extraction of the main kernel routine, in case of a "port" of the software in Caml or in Haskell[1]. This paper presents the syntax of the iRho language and some examples that can be run directly by cut and paste in the interpreter. The current distribution can be found in: `http://www-sop.inria.fr/mirho/Luigi.Liquori/iRho/`. It contains: two software releases `iRho-1.0.scm`, and `iRho-1.1.scm`, a precompiled binary version for Linux architecture[2], a file `demo.rho` containing many examples, and a copy of the [13] paper (journal version).

We conclude with a table showing future releases and evolutions of the present software, like (polymorphic) type inference, powerful matching and unification algorithms, exceptions handlers, strategies, calling external languages, objects, etc.

[1] The extraction mechanism in Coq can currently target Caml or Haskell code.

[2] `ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5, dynamically linked (uses shared libs), stripped.`

2 Playing with iRhoSW

The interpreter iRhoSW greets you as follows:

```
 ----------------------------------------------
|               ----------\                     |
|               | i R h o  >                    |
|               ----------/                     |
| An Imperative Rewriting Calculus Interpreter |
| Kernel Certified by the Proof Assistant Coq  |
|            Powered by Bigloo Scheme           |
|              Copyright Inria 2005             |
|                Version 1.1alpha               |
|              NoEffect Theory Loaded           |
|              $ = Switch Theory                |
|              # = Clean  Namespace             |
|              @ = Exit   iRho                  |
 ----------------------------------------------
```

As usual, the first thing to learn is how to exit from the *read-eval-print* loop: just evaluate "`@;;`" to exit. Evaluating "`$;;`" moves the interpreter to the *empty* (or *syntactic*) matching theory to the *no-stuck* matching theory, introduced firstly in [5]: we will be more precise about this theory in a moment, after presenting the syntax and a sketch of the reduction semantics. Evaluate "`#;;`" allows to clean a *global namespace*, *i.e.* a space where constants, functions, and term rewriting systems can be names and globally reused.

Syntax

The untyped (abstract) syntax of iRho is as follows:

```
key   ::= "(" | ")" | "," | "^" | "!" | ":=" | "->" |
          "<-?" | "?->" | ";" | "=" | "[" | "]" | "|"  Keywords
var   ::=  "any sequence of capital alphadigit"        Variables
const ::=  "any sequence of non capital alphadigit"    Constants
patt  ::= var    | const | const patt   |
          patt,patt      | ^ patt                       Patterns
expr  ::= const | var    | patt -> expr | expr expr |
          expr,expr      | ^ expr       | ! expr    |
          expr:=expr     | expr <-? expr ?-> expr   |
          var=expr       | [(var=expr |)*]              Expressions
```

One important point is that *linearity* in pattern is not enforced in the syntax; the solution we adopt in this formalization and implementation of the Rewriting-calculus was influenced by the choice of the implementation language of our operational semantics, namely Scheme and the matching algorithm adopted [12]. As such, the specification of the matching algorithm in iRho accepts non-linear patterns, and compares subparts of the datum (through $\equiv$, implemented via the primitive `equiv?` in Scheme). Confluence is preserved, thanks to the call-by-value strategy of the operational semantics. Examples of legal terms are:

```
iRho  IN > 12;;
iRho OUT > 12
iRho  IN > dummy;;
iRho OUT > dummy
iRho  IN > x;;
iRho OUT > x
iRho  IN > X;;
iRho OUT > (Effect: Unbound Variable X)
iRho  IN > 12;;
iRho OUT > 12
iRho  IN > (12->13 12);;
iRho OUT > 13
iRho  IN > (12->12 14);;
iRho OUT > (Effect: Pattern Mismatch)
iRho  IN > ((a->b,a->d) a);;
iRho OUT > (b , d)
iRho  IN > ((a->b,c->d) a);;
iRho OUT > b
iRho  IN > (f X Y)->X;;
iRho OUT > (Fun ((f X) Y) -> X)
iRho  IN > ((f X X)->X (f 3 4));;
iRho OUT > (Effect: Pattern Mismatch)
iRho  IN > $;;
iRho OUT > Switching_to_empty_theory
iRho  IN > ((a->b,c->d) a);;
iRho OUT > (b , (Effect: Pattern Mismatch))
```

The last example can help to understand that the no-stuck theory absorbs pattern-matching failures, while the empty theory is not. This is perhaps a good point to introduce the reduction semantics.

Reduction Semantics

The semantics behaves as follows (see [13] for a detailed presentation):

```
 (patt -> expr exprnf) => sigma(expr)  where sigma=patt<<exprnf
((expr1,expr2) exprnf) => ((expr1 exprnf),(expr2 exprnf))
```

The first rule fires an application if the argument is in normal-form (call-by-value semantics) and if it *matches* with the pattern, while the second rule distributes the application to all elements of a structure. That's all you need to do if you want to play just with the functional fragment of the Rho. In a nutshell, the functional fragment is "just" a Lambda-calculus with patterns, records, and non exclusive pattern-matching (*i.e.* multiple branches can be fired simultaneously). The possibility to fire, in parallel, multiple matching branches is one of the biggest peculiarity of the Rewriting-calculus w.r.t. other languages featuring (exclusive and sequential) pattern-matching.

Adding imperative features causes to introduce a store, *i.e.* an global partial mapping `s` from locations to expressions in normal forms (*i.e.* values), and to add the following reduction rules:

```
  ^ exprnf   /s =>    loc/(s,loc=exprnf)  where loc not in Dom(s)
   !loc      /s => s(loc)/ s              where loc     in Dom(s)
loc:=exprnf /s => exprnf/(s,loc=exprnf)  where loc     in Dom(s)
exprs1;expr2/s => ((X->expr2) expr1)/s   where  X   fresh in  expr2
```

In a nutshell: the first rule allocates a new fresh location `loc` in the store and binds it to the value `exprnf`; the second rule reads the content of the location `loc`; the third rule writes in the location `loc` the value `exprnf`. The last rule (sequence) is just a macro for a dummy function application; the call-by-value strategy ensures that `expr1` will be evaluated (possibly with a store modification) before `expr2`. Examples of legal terms are:

```
iRho  IN > ^ 1,2;;
iRho OUT > ((Ref 1) , 2)
iRho  IN > ^ (1,2);;
iRho OUT > (Ref (1 , 2))
iRho  IN > (!^(X->X)  4);;
iRho OUT > 4
iRho  IN > ((X,Y)->(X,Y) (^3,^4));;
iRho OUT > ((Ref 3) , (Ref 4))
iRho  IN > ((X,Y)->(Y:=!X;(!X,!Y)) (^3,^4));;
iRho OUT > (3 , 3)
iRho  IN > ((X,Y)->(Y:=!X;(X,!X,Y,!Y)) (^3 , ^4));;
iRho OUT > ((Ref 3) , (3 , ((Ref 3) , 3)))
iRho  IN > ((f X Y)->(Z->(X:=!Z) X) (f ^3 ^4));;
iRho OUT > 3
iRho  IN > (X->((^ Y->Y) X) ^ 4);;
iRho OUT > 4
iRho  IN > ((XREF->((X->XREF:=X) 5)) ^dummy);;
iRho OUT > 5
```

The macro "="

This simple macro allows to modify a global namespace; it is also useful to define quickly constants values, functions, and term rewriting systems with built-in fix-points.

```
iRho  IN > ID = (X->X);;
iRho OUT > (Fun X -> X)
iRho  IN > IDID = (X->X X->X);;
iRho OUT > (Fun X -> X)
iRho  IN > ID;;
iRho OUT > (Fun X -> X)
iRho  IN > (ID 4);;
iRho OUT > 4
iRho  IN > IDID;;
iRho OUT > (Fun X -> X)
iRho  IN > (IDID 4);;
iRho OUT > 4
iRho  IN > MATCHPAIR = ((f(X,Y))->X);(MATCHPAIR (f(2,3)));;
iRho OUT > 2
iRho  IN > MATCHCURRY = (f X Y)->X;(MATCHCURRY (f 2 3));;
iRho  IN > SWAP=((X,Y)->((AUX->(AUX:=!X;X:=!Y;Y:=!AUX;
                                   (!X,!Y,!AUX)))(^0)));;
iRho OUT > (Fun (X , Y) -> ((Fun AUX ->
           ((Fun FRESH1005 -> ...
           ((Bang X) , ((Bang Y) , (Bang AUX))))
           (Ass Y (Bang AUX)))) (Ass X (Bang Y))))
           (Ass AUX (Bang X)))) (Ref 0)))      Swapping two variables
iRho  IN > (SWAP(^4,^5));;
iRho OUT > (5 , (4 , 4))
iRho  IN > FIXV = FUN->VAL->(FUN (FIXV FUN) VAL);;
iRho OUT > (Fun FUN -> (Fun VAL ->
           ((FUN (FIXV FUN)) VAL)))           A call-by-value fix point
iRho  IN > (FIXV ID 3);;
Segmentation fault                           Sorry, reload everything ...
iRho  IN > LETRECPLUS = ((PLUS ->
           (PLUS ((succ (succ 0)),(succ (succ 0)))))
           (FIXV (PLUS -> VAL ->
           (((0,N)       -> N  ,
            ((succ M),N) -> (succ (PLUS (M,N)))) VAL))));;
                          letrec PLUS = ‘‘Peano’s plus’’ in (PLUS (2,2))
```

If-then-else

Control structures can be easily be defined as follows:

```
iRho  IN > NEG = (true -> false, false -> true);;
iRho OUT > ((Fun true -> false) , (Fun false -> true))
iRho  IN > (NEG true);;
iRho OUT > false
iRho  IN > AND = ((true, true)  -> true,
                  (true, false) -> false,
                  (false,true)  -> false,
                  (false,false) -> false);;
iRho OUT > ((Fun (true  , true)  -> true) ,
           ((Fun (true  , false) -> false) ,
           ((Fun (false , true)  -> false) ,
            (Fun (false , false) -> false))))
iRho  IN > OR = ((true, true)  -> true,
                 (true, false) -> true,
                 (false,true)  -> true,
                 (false,false) -> false);;
iRho OUT > ((Fun (true  , true)  -> true) ,
           ((Fun (true  , false) -> true) ,
           ((Fun (false , true)  -> true) ,
           (Fun (false  , false) -> false))))
iRho  IN > OMG = (X->(X X));;
iRho OUT > (Fun X -> (X X))
iRho  IN > ((OMG OMG) <-? (AND (true,true)) ?-> 4);; Happy syntax
iRho OUT > 4                                 Don't try with false :-)
```

Defining Term Rewriting Systems

One may wonder a simpler way to define a term rewriting system and a fix-point operator allowing to use a term rewriting system; the iRhoSW offers two ways to do it in a simpler and efficient way. The first is by using the macros "=" while the latter is by using the macros "[...]". The main difference between those two alternatives is in efficiency (the former being faster the the latter). We first introduce some macros for Peano's numbers

```
iRho  IN > ZERO     = 0;;
           ONE      = (succ 0);;
           TWO      = (succ ONE);;
           THREE    = (succ TWO);;
           ...
```

Then we simply define our PLUS term rewriting system as follows:

```
iRho  IN > PLUS = ((0,N)        -> N,
                  ((succ N),M) -> (succ (PLUS (N,M))));;
iRho OUT > ((Fun        (0 , N) -> N) ,
            (Fun ((succ N) , M) -> (succ (PLUS (N , M)))))
iRho  IN > (PLUS (THREE,THREE));;
iRho OUT > (succ (succ (succ (succ (succ (succ 0))))))
```

or as follows:

```
iRho  IN > [PLUS = ((0,N)       -> N,
                  ((succ N),M) -> (succ (PLUS (N,M))))];;
iRho OUT > Term Rewriting System Definition
iRho  IN > [PLUS = ((0,N)       -> N,
                  ((succ N),M) -> (succ (PLUS (N,M))))];
           (PLUS (THREE,THREE));;
iRho OUT > (succ (succ (succ (succ (succ (succ 0))))))
```

Note that in the two encodings (using "=" or "[...]") one term rewriting system can "call" another term rewriting system as follows (using sequencing):

```
iRho  IN > PLUS = ((0,N)        -> N ,
                  ((succ N),M) -> (succ (PLUS (N,M))));
           FIB  = (0            -> (succ 0) ,
                  (succ 0)     -> (succ 0) ,
                (succ (succ X)) -> (PLUS ((FIB (succ X)),
                                          (FIB X))));
          (FIB FOUR);;                              First encoding
```

```
iRho  IN > [PLUS = ((0,N)        -> N ,
                   ((succ N),M)  -> (succ (PLUS (N,M))))
            |
             FIB = (0            -> (succ 0) ,
                     (succ 0)    -> (succ 0) ,
                 (succ (succ X)) -> (PLUS ((FIB (succ X)),
                                          (FIB X))))];
           (FIB FOUR);;                              Second encoding
iRho OUT > (succ (succ (succ (succ (succ 0)))))
iRho  IN > [PLUS = ((0,N)         -> N ,
                    ((succ N),M)  -> (succ (PLUS (N,M))))
            |
            MULT = ((0,M)         -> 0,
                    ((succ N),M)  -> (PLUS (M,(MULT (N,M)))))
            |
            POW  = ((N,0)         -> (succ 0),
                    (N,(succ M))  -> (MULT (N,(POW (N,M)))))];;
           (POW (TWO,TEN));;                                 Power
iRho  IN > [ACK  =
            ((0,N)               ->(succ N),
             ((succ M),0)        ->(ACK(M,(succ 0))),
             ((succ M),(succ N))->(ACK(M,(ACK((succ M),N)))))];
           (ACK  (THREE,FOUR));;                         Ackermann
iRho  IN > LIST = (10,11,12,13,15,16,nil);;
iRho  In > [FINDN = ((0,nil)          -> fail,
                    ((succ N),nil)    -> fail,
                    ((succ 0),(X,Y))  -> X,
                    ((succ N),(X,Y))  -> (FINDN (N,Y)))];
           (FINDN (THREE,LIST));;            Find an element in a list
iRho  In > [KILLM = ((m,(n,nil))  -> (n,nil),
                     (m,(m,X))    -> X,
                     (m,(n,X))    -> (n,(KILLM (m,X))))];
           (KILLM (13,LIST));;               Kill an element in a list
```

A More Tricky Example: Negation Normal Form

This function is used in implementing *decision procedures*, present in almost all model checkers. The processed input is an implication-free language of formulas with generating grammar:

$$\phi ::= \mathsf{p} \mid \mathsf{and}(\phi,\phi) \mid \mathsf{or}(\phi,\phi) \mid \mathsf{not}(\phi)$$

We present three encodings, the first uses the "=" macro, the second uses the "[...]" macro and the last is just the macro-expansion of the second one (some outputs are omitted).

```
iRho  IN > PHI = (and ((not (and (p,q))),(not (and (p,q)))));;
iRho  IN > NNF = (  p ->  p,
                  q ->  q,
    (not (not  X))    -> (NNF X),
    (not (or  (X,Y))) -> (and ((not (NNF X)),(not (NNF Y)))),
    (not (and (X,Y))) -> (or  ((not (NNF X)),(not (NNF Y)))),
    (and (X,Y))       -> (and ((NNF X),(NNF Y))),
    (or  (X,Y))       -> (or  ((NNF X),(NNF Y)));
           (NNF PHI);;                              First encoding
iRho  IN > [NNF = (  p ->  p,
                     q ->  q,
     (not (not  X))    -> (NNF X),
     (not (or  (X,Y))) -> (and ((not (NNF X)),(not (NNF Y)))),
     (not (and (X,Y))) -> (or  ((not (NNF X)),(not (NNF Y)))),
     (and (X,Y))       -> (and ((NNF X),(NNF Y))),
     (or  (X,Y))       -> (or  ((NNF X),(NNF Y))))];
           (NNF PHI);;                             Second encoding
iRho OUT > (and((or((not p),(not q))),(or((not p),(not q)))))
```

Certification: the DIMPRO *pattern*

In [13] we experimented with an interesting "pattern (in the sense of *"The Gang of Four"* [8]) called DIMPRO, a.k.a. Design-IMplement-PROve, to design safe software, which respects *in toto* its mathematical and functional specifications. The iRhoSW is a direct derivative of such a methodology.

Intuitively, we started from a clean and elegant mathematical design, from which we continued with an implementation of a prototype satisfying the design (using a functional language), and finally we completed it with a mechanical certification of the mathematical properties of the design, by looking for the simplest "adequacy" property of the related software implementation. These three phases are strictly coupled and, very often, one particular choice in one phase induced a corresponding choice in another phase, very often forcing backtracking.

The process refinement is done by iterating cycles until all the global properties wanted are reached (the process is reminiscent of a fixed-point computation, or of a B-refinement [1]). All three phases have the same status, and each can influence the other.

Our recipe probably suggests a new schema, or "pattern", in the sense of *"The Gang of Four"* [8], for design-implement-certify safe software. This could be subject of future work. A small software interpreter for our core-calculus is surely a good test of the "methodology". More generally, this methodology could be applied in the setting of raising quality software to the highest levels of the *Common Criteria, CC* [6] (from EAL5 to EAL7), or level five of the *Capability Maturity Model, CMM*. We schedule in our agenda our novel DIMPRO, in the folklore of "design pattern", hoping that it would be useful to the community developing safe software for crucial applications.

Agenda

Our iRhoSW is really young: the table below sketch some possible improvements planned in the next two future releases.

MAJOR IMPROVEMENTS/RELEASE	2.0	3.0
exceptions on pattern-matching failure	✓	
first-order type inference	✓	
more control structures and strategies	✓	
simple objects and object-based inheritance	✓	
type-inference *à la* Damas-Milner		✓
unification and AC matching theory		✓
rewriting-rule as patterns [3]		✓
calling externals Scheme/Java/C		✓
I/O (files)		✓
certification using Coq	✓	??

References

[1] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*.
[2] Asf+Sdf Team. The Asf+Sdf Meta-Environment Home Page, 2005.
[3] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Pattern Type Systems. In *Proc. of POPL*. 2003.
[4] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In *Proc. of FOSSACS*, volume 2030 of *LNCS*, pages 166–180, 2001.
[5] H. Cirstea, L. Liquori, and B. Wack. Rho-calculus with Fixpoint: First-order system. In *Proc. of TYPES*. Springer-Verlag, 2004.
[6] Common Criteria Consortium. The Common Criteria Home Page, 2005.
[7] Cristal Team. OCaml Home page, 2005.
[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides (The Gang of Four). *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[9] GNU Prolog Team. The Prolog Home Page, 2005.
[10] J. Goguen. The OBJ Family Home Page, 2005.
[11] Haskell Team. The Haskell Home Page, 2005.
[12] G. Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Ph.d. thesis, Université de Paris 7 (France), 1976.
[13] Liquori, L. and Serpette, B. iRho, An Imperative Rewriting Calculus. In *Proc. of PPDP*, pages 167–178. The ACM Press, 2004.
[14] Maude Team. The Maude Home Page, 2005.
[15] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
[16] Protheo Team. The Elan Home Page, 2005.
[17] Scheme Team. The Scheme Language, 2005.
[18] Stratego Team. The Stratego Home Page, 2005.
[19] A. van Deursen, J. Heering, and P. Klint. *Language Prototyping*, 1996.