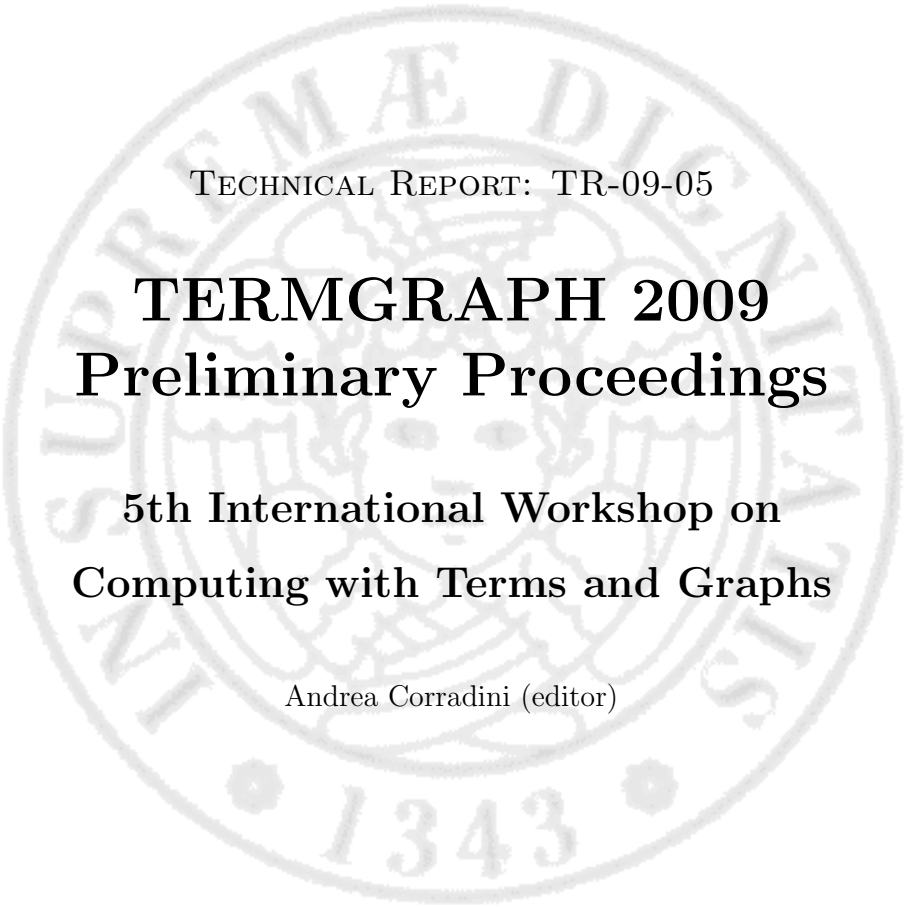
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-09-05

TERMGRAPH 2009
Preliminary Proceedings

5th International Workshop on Computing with Terms and Graphs

Andrea Corradini (editor)

March 10, 2009

address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. tel: +39 050 2212700 fax: +39 050 2212726

Preface

TERMGRAPH 2009 took place in York (UK) on March 22, 2009, as a one-day satellite event of ETAPS 2009. Previous editions of the TERMGRAPH workshops series took place in Barcelona (2002), in Rome (2004), in Vienna (2006), and in Braga (2007).

The advantage of computing with graphs rather than terms (strings or trees) is that common subexpressions can be shared, which improves the efficiency of computations in space and time. Sharing is ubiquitous in implementations of programming languages: many implementations of functional, logic, object-oriented and concurrent calculi are based on term graphs. Term graphs are also used in symbolic computation systems and automated theorem proving.

Research in term and graph rewriting ranges from theoretical questions to practical implementation issues. Many different research areas are included, for instance: the modelling of first- and higher-order term rewriting by (acyclic or cyclic) graph rewriting, the use of graphical frameworks such as interaction nets and sharing graphs to model strategies of evaluation (for instance, optimal reduction in the lambda calculus), rewrite calculi on cyclic higher-order term graphs for the semantics and analysis of functional programs, graph reduction implementations of programming languages, graphical calculi modelling concurrent and mobile computations, object-oriented systems, graphs as a model of biological or chemical abstract machines, and automated reasoning and symbolic computation systems working on shared structures.

The aim of TERMGRAPH 2009 was to bring together researchers working in these different domains and 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 term graph rewriting. Topics of interest for the workshop are all aspects of term graphs and sharing of common subexpressions in rewriting, programming, automated reasoning and symbolic computation. This includes (but is not limited to) term rewriting, graph transformation, programming languages, models of computation, graph-based languages, semantics and implementation of programming languages, compiler construction, pattern recognition, databases, bioinformatics, and system descriptions.

This report contains the eight contributions presented during the workshop: they were selected by the Program Committee according to originality, significance, and general interest. In addition to these presentations, the programme included two invited lectures, by Fabio Gadducci and Hélène Kirchner.

After the workshop selected authors will be invited to submit a revised version of their contribution for the electronic post-proceedings which will be published as a volume of the Electronic Notes in Theoretical Computer Science (ENTCS) series, which is published electronically through the facilities of Elsevier Science B.V.

The Program Committee consisted of

- Andrea Corradini (chair, Italy)
- Rachid Echahed (France)
- Marko van Eekelen (The Netherlands)
- Maribel Fernández (UK)
- Ian Mackie (France)
- Detlef Plump (UK)

For their help in reviewing and selecting the submitted abstract, we are grateful to the Programme Committee members and to Bahareh Badban, Clara Bertolissi, Roberto Bruni, Jean Goubault-Larrecq, Alberto Lluch Lafuente, Maarten de Mol, and Jorge Sousa Pinto.

Pisa, March 2009 Andrea Corradini

Table of Contents

Invited Speakers

Submitted Contributions

Some Properties of an Old-Fashioned Algebra for Graphs

(ABSTRACT)

Fabio Gadducci[a]

[a] *Dipartimento di Informatica, Pisa, Italy*

Hyper-Graphs with interfaces consider an hyper-graph G equipped with a pair of morphisms $j : J \to G$ and $i : I \to G$, representing the possible connections of G with the environment. Indeed, these "handles" have been used for defining suitable graphical operators, and exploited for e.g. the encoding of process calculi.

A recent survey by Selinger illustrates the algebraic presentations of (variants of) hyper-graphs with *discrete* interfaces, i.e., such that I, J are just sets of nodes. Using a graph transformation jargon, the key reasoning is plainly told. Consider a signature Σ as an hyper-graph G_Σ (sorts/nodes, operators/hyper-edges) and focus on the hyper-graphs typed over G_Σ, i.e., where nodes (edges) are labelled by sorts (operators): Any such hyper-graph with discrete interfaces is uniquely characterized by an arrow of a monoidal category $DGS(\Sigma)$, (almost) freely generated from Σ.

This characterisation allows for providing an inductive presentation of DPO rewriting, and it can be specialised to structures other than hyper-graphs. However, the restriction to discrete interfaces is unfortunate, since it boils down to have rewriting rules with no "read only" component, thus forbidding to properly recast in the algebraic framework the results about parallelism for DPO rewriting. This talk shows how, starting from a signature Σ, to define a new signature Σ_u where the arrows of $DGS(\Sigma_u)$ correspond to hyper-graphs, typed over Σ, with *disconnected* interfaces, i.e., such that I, J are just sets of isolated hyper-edges, sharing no node.

The construction of Σ_u recalls the presentation of hyper-graphs as bipartite, simple graphs. As a paradigmatic example, we focus on hyper-graphs typed over the signature Σ^g, with sort N and unary operator $e : N \to N$, corresponding to standard graphs: Each graph is identified by an algebra over the signature Σ^g_u, with sorts E, N and operators $s, t : E \to N$; and we show how graphs with disconnected interfaces, typed over Σ^g, are uniquely characterized by the arrows of $DGS(\Sigma^g_u)$.

The talk further highlights some properties of $DGS(\Sigma_u)$, in the light of the use of adhesive categories as a framework for the generalization of DPO rewriting.

Layout based on the macro package of the
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

A Port Graph Calculus and its Application to Autonomic Computing

(ABSTRACT)

Hélène Kirchner[a] Oana Andrei[b]

[a] *INRIA Bordeaux - Sud-Ouest Research Center, France*

[b] *Department of Computing Science, University of Glasgow, UK*

Autonomic computing refers to self-manageable systems initially provided with some high-level instructions from administrators. This is a biologically inspired computation model that gained much interest with the recent development of large scale distributed systems such as service infrastructures and grids. For such systems, there is a crucial need for theories and formal frameworks to model computations, to define languages for programming and to establish foundations for verifying important properties of these systems.

From our previous work on biochemical applications, the structure of *port graph* (or multigraph with ports) and a rewriting calculus have emerged to model interactions between molecules or proteins. We propose port graphs as a formal model for distributed resources and grid infrastructures, where each resource is modeled by a node with ports. The lack of global information and the autonomous and distributed behavior of components are modeled by a multiset of port graphs and rewrite rules which are applied locally, concurrently, and non-deterministically. Some computations take place wherever it is possible and in parallel, while others may be controlled by strategies.

In this talk, we first introduce port graphs, that are graphs with multiple edges and loops, with nodes having explicit connection points, called ports, and edges attaching to ports of nodes. We then define a rewrite calculus on these graphs where rules and strategies are themselves port graphs, i.e. first-class objects of the calculus. As a consequence, they can be rewritten as well, and rules can create new rules, providing a way of modeling emergence in a system. This approach also provides a formal framework to reason about computations and to verify desirable properties. We give some suggestions on expressing properties of a modeled system as strategies. This work in progress opens the way to many further research topics.

Layout based on the macro package of the
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

A Term Rewriting Technique for Decision Graphs

Bahareh Badban[1]

Department of Software Engineering
University of Konstanz
Germany

Abstract

We provide an automatic verification for a fragment of FOL quantifier-free logic with zero, successor and equality. We use BDD representation of such formulas and to verify them, we first introduce a (complete) *term rewrite system* to generate an equivalent *Ordered* $(0, S, =)$-BDD from any given $(0, S, =)$-BDD. Having the ordered representation of the BDDs, one can verify the original formula in constant time. Then, to have this transformation automatically, we provide an algorithm which will do the whole process.

Keywords: Term Rewrite System; First order logic; Decision Procedure; Verification.

1 Introduction

In this article we consider the satisfiability and tautology problem for boolean combinations over the equational theory of zero and successor in the natural numbers. The atoms are equations between terms built from variables, zero (0) and successor (S). Formulas are built from atoms by means of negation ($\neg$) and conjunction ($\wedge$). The formulas are quantifier-free, except for the implicit outermost quantifier ($\forall$ when considering tautology checking, and $\exists$ when considering satisfiability).

In general, the decision problem for plain equational theories is unsolvable already, so we must restrict to particular theories. The decision problem for boolean combinations over equational theories can be approached in several ways. *Binary Decision Diagrams* (BDDs) represent boolean functions as directed acyclic graphs [5]. They are of value for validating formulas in propositional logic. In [5] OBDDs (*Ordered* BDDs) are reduced BDDs which accept some ordering on boolean variables. A boolean function is satisfiable if and only if its unique OBDD representation does not correspond to 0. In the BDD-method, a formula is transformed to a propositionally equivalent *Ordered Binary Decision Diagram* (OBDD) which can be seen as a large if-then-else (ITE) tree with shared subterms (see Section 2).

[1] Email: badban@inf.uni-konstanz.de

Layout based on the macro package of the
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Although in principle also OBDD representations are exponentially big, it appears that in practice many formulas have a succinct OBDD-representation. Furthermore, boolean operations, such as negation and conjunction, can be computed on OBDDs very cheaply. Together with the fact that (due to sharing) many practical boolean functions have a small OBDD representation, OBDDs are very popular in verification of hardware design, and play a major role in symbolic model checking.

In order to solve the satisfiability or tautology problem, each path in the OBDD has to be checked for consistency, with respect to the underlying equational theory. A path represents a conjunction of (negated) equations, on which the aforementioned decision procedures can be applied. All inconsistent paths can be removed, resulting in an OBDD with only consistent paths. However, due to sharing subterms, an OBDD can have exponentially many paths, so still there is a computational bottleneck. In the *Encoding method* these steps are reversed. First the formula is transformed to a purely propositional formula. In this translation, facts from the equational theory (e.g. congruence of functions, transitivity of equality and orderings) are encoded into the formula. Then a *finite model property* is used to obtain a finite upperbound on the cardinality of the model. Finally, variables that range over a set of size n are encoded by $log(n)$ propositional variables. The resulting formula can be checked for satisfiability with any existing SAT-technique, for instance based on resolution [7] or on BDDs [5]. An early example is Ackermann's reduction [1], by which second order variables can be eliminated. More optimal versions are in [10,17,6].

To date, several methods have been proposed to reduce different logics into propositional logic, which captures boolean functions. Goel et al. [10] and Bryant et al. [6]. present methods to transform the logic of Equality with Uninterpreted Functions (EUF) into propositional logic. In [18] the theory of *separation predicates* is reduced to propositional logic. In [16] the EUF extended with constrained lambda expressions, ordering, and successor and predecessor functions, is translated to propositional logic. The idea of extending the theory of BDDs was recognized earlier by Groote and van de Pol [11], who presented an algorithm to transform EQ-BDDs to EQ-OBDDs, where EQ-BDDs represent the extension of BDDs with equalities. We extend the method for EQ-BDDs from [11] to a fragment of quantifier free logic FOL. We make a terminating set of rewrite rules on $(0, S, =)$-BDDs, resulting in a $(0, S, =)$-R-OBDD, such that all paths in the $(0, S, =)$-R-OBDD are satisfiable. This property enables us to check tautology, contradiction and satisfiability on $(0, S, =)$-R-OBDDs in constant time. At the end we present an algorithm through which any formula of the logic above is translated to an $(0, S, =)$-R-OBDD.

We define the set of terms as the closure of $\bar{V} = V \cup \{0\}$ (union of the sets of variables and zero) under successor. To be able to have an ordering on BDDs, we will need to define an ordering on terms of the logic. What is the appropriate ordering on terms? The answer, unfortunately, is not obvious. In [2] Chapter 3, two orderings which resulted in failed attempts are explained. One of them does not provide termination, and the other does not omit all unsatisfiable paths.

The approach introduced here is in a sence a variant of [3], though our new ordering yields some simpler representation of the terms in the proof settings. Besides, it provides an alternative technique for the OBDD transformation. This, as

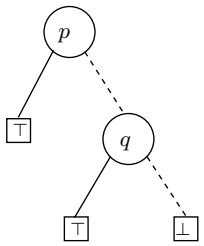


Figure 1. $ITE(p, \top, ITE(q, \top, \bot))$. *Solid* lines denote the left branch of the ITE (when their corresponding guard holds) and *dashed* lines represents their counterpart.

a result can offer a different method for possible extensions of the background logic (notice that as mentioned above, finding the right order is not easy, and this problem would remain for bigger theories as well). In the current work, substitution rules are certainly different than those of the previous work. In addition, we also introduce an automatic way for transforming any formula (in our FOL fragment) into some Ordered BDD. We do this by means of a so called `sort` algorithm.

Road map. In Section 2, we describe BDDs, and give a formal syntax and semantics of $(0, S, =)$-BDDs. In Section 3 our transformation is presented, leading to the set of $(0, S, =)$-R-OBDDs. First a total and well-founded order on variables is assumed, and extended to a total well-founded order on equalities. Then the rewrite system is presented. Finally, we prove termination and satisfiability over all paths. Section 4 presents an algorithm with the same result as the given term rewrite system. Finally, Section 5 concludes with some remarks on implementation and possible applications.

2 Binary Decision Diagrams

A *binary decision diagram* [5] (BDD) represents a boolean function as a finite, rooted, binary, ordered, directed acyclic graph. The leaves of this graph are labeled $\bot$ and $\top$, and all internal nodes are labeled with boolean variables. A node with label p, left child L and right child R, written $ITE(p, L, R)$, represents the formula *if* p *then* L *else* R.

Given a fixed total order on the propositional variables, a BDD can be transformed to an *Ordered* binary decision diagram (OBDD), in which the propositions along all paths occur in increasing order, redundant tests ($ITE(p, x, x)$) don't occur, and the graph is maximally shared. For a fixed order, each boolean function is represented by a unique reduced OBDD (in the sequel we simply use OBDD to denote a reduced OBDD). For more information on that, one can see [5].

Example 2.1 Figure 1 illustrates a BDD representation of the following formula: $ITE(p, \top, ITE(q, \top, \bot))$ where p and q are propositional variables.

2.1 BDDs with Equality, Zero and Successor

In this section we introduce some basic notations and definitions. We also provide the syntax and semantics of BDDs extended with zero, successor and equality. For our purpose, the sharing information present in the graph is immaterial, so we

formalize BDDs by terms (i.e. trees). We show that every formula is representable as a BDD.

We assume V is a set of variables, and define $\bar{V} = V \cup \{0\}$. Sets of terms, formulas, guards and BDDs are defined below:

Definition 2.2 Terms $t \in W$, formulas $\varphi \in \Phi$, guards $g \in G$ and $(0, S, =)$-BDDs $T \in B$ are defined by the following grammar (with $x \in V$):

$$\begin{aligned} t &::= 0 \mid x \mid S(t) \\ \varphi &::= \bot \mid \top \mid t = t \mid \neg\varphi \mid \varphi \wedge \varphi \mid ITE(\varphi, \varphi, \varphi) \\ g &::= \bot \mid \top \mid t = t \\ T &::= \bot \mid \top \mid ITE(g, T, T) \end{aligned}$$

A guard is *trivial* if it is $\bot$ or $\top$, and otherwise it is *non-trivial*. Here are some notations that we will use in this paper: In order to avoid confusion with the $=$-symbol in guards, we use $\equiv$ to identify syntactic equality between terms or formulas. Symbols $x, y, z, u, \ldots$ denote variables; $r, s, t, \ldots$ range over W; $\varphi, \psi, \ldots$ range over Φ; $f, g, \ldots$ range over guards. $\texttt{Var}(t)$ represents the variable occurring in term t. Furthermore, we will write $x \neq y$ instead of $\neg(x = y)$ and $S^m(t)$ for the m-fold application of S to t, so $S^0(t) \equiv t$ and $S^{m+1}(t) \equiv S(S^m(t))$. Note that each $t \in W$ is of the form $S^m(u)$, for some $m \in \mathbb{N}$ and $u \in \bar{V}$.

We will use some fixed interpretation for the above formulas: Terms are interpreted over the natural numbers ($\mathbb{N}$) and for formulas we use the classical interpretation over $\{0, 1\}$. Given a valuation $v : V \to \mathbb{N}$, we extend v homomorphically to terms and formulas as:

$$\begin{aligned} v(0) &= 0 \\ v(S(t)) &= 1 + v(t) \\ v(\bot) &= 0 \\ v(\top) &= 1 \\ v(s = t) &= 1, \text{ if } v(s) = v(t), \text{ 0, otherwise.} \\ v(\neg\varphi) &= 1 - v(\varphi) \\ v(\varphi \wedge \psi) &= \min(v(\varphi), v(\psi)) \\ v(ITE(\varphi, \psi, \chi)) &= v(\psi) \text{ if } v(\varphi) = 1, \, v(\chi) \text{ otherwise.} \end{aligned}$$

It is trivial that the value of a formula under any valuation function is either 0 or 1.

Given a formula φ, we say it is *satisfiable* if there exists a valuation $v : V \to \mathbb{N}$ such that $v(\varphi) = 1$; it is a *contradiction* otherwise. If for all $v : V \to \mathbb{N}$, $v(\varphi) = 1$, then φ is a *tautology*. Finally, if $v(\varphi) = v(\psi)$ for all valuations $v : V \to \mathbb{N}$, then φ and ψ are called *equivalent*. v satisfies φ (or equivalently φ holds under v) is denoted as: $v \models \varphi$.

Lemma 2.3 *Every formula in Φ is equivalent to at least one $(0, S, =)$-BDD.*

3 Representant-Ordered $(0, S, =)$-BDDs

The first step to make a BDD ordered, is to simplify all its guards, in isolation. Here, simplification on guards will be done by Definition 3.2. In Section 3.1 we present a new order on terms. In Definition 3.8 we define an ordering on guards. Notice that these definitions are different from those of [3]. Thereafter, we will introduce a term rewrite system. Using this system we simplify BDDs to their most reduced form, denoted as $(0, S, =)$-R-OBDD.

3.1 Definition of $(0, S, =)$-R-OBDDs

We consider a fixed total and well-founded ordering on V. Below we assume that the variables x, y and z are ordered as $x \prec y \prec z$.

Definition 3.1 [ordering definition] We extend $\prec$ to a total order on W:

- $0 \prec u$ for each element u of V
- $S^m(x) \prec S^n(y)$ if and only if $x \prec y$ or $(x \equiv y$ and $m < n)$ for each two elements $x, y \in \bar{V}$

As of now, we may use the term OBDD instead of $(0, S, =)$-R-OBDD, for simplicity.

Definition 3.2 Suppose g is a guard. By $g\downarrow$ we mean the normal form of g obtained after applying the following rewrite rules on it:

$$\begin{aligned} x = x &\to \top \\ S(x) = S(y) &\to x = y \\ 0 = S(x) &\to \bot \\ x = S^{m+1}(x) &\to \bot \qquad \text{for all } m \in \mathbb{N} \\ t = r &\to r = t \quad \text{for all } r, t \in W \text{ such that } r \prec t. \end{aligned}$$

We call g simplified if it cannot be further simplified, i.e. $g \equiv g\downarrow$. A $(0, S, =)$-BDD T is called simplified if all guards in it are simplified.

Lemma 3.3 *If $g \in G$ is simplified to g' using Definition 3.2, then g and g' are equivalent.*

Next lemma shows possible shapes of a simplified guard. In contrast to [3] the smaller term sits on the left.

Lemma 3.4 *If g is a simplified guard, then it has one of the following shapes:*

- $S^m(0) = x$ *for some* $x \in V$
- $S^m(x) = S^n(y)$ *for some* $x, y \in V$, $x \prec y$, $m = 0$ *or* $n = 0$
- $\top$ *or* $\bot$

It is worth mentioning that as a result each guard has only one simplified form.

In order to be able to substitute one term by another, we may often need to upraise the atom which includes the term by applying some few additional successors, and then to do the replacement. Below, we explain our strategy for doing this:

Definition 3.5 Let $m \in \mathbb{N}$. For terms $r, t \in W$, a variable $y \in V$ and a guard $g \in G$ we define:

$$(r = t)\uparrow^m := S^m(r) = S^m(t) \quad \textit{(lifting)}$$

Definition 3.6 Suppose g is a simplified non-trivial guard, $y \in V$ and $t, r \in W$. We define:

$$g|_{r=S^m(y)} := \begin{cases} (g\uparrow^m [S^m(y) := r]) \downarrow & \textit{if } y \textit{ occurs in } g \\ g & \textit{otherwise} \end{cases}$$

$$g|_{t \neq r} := \begin{cases} \bot & \textit{if } g \equiv (t = r) \downarrow \\ g & \textit{otherwise} \end{cases}$$

The following lemma shows the soundness of the operations above:

Lemma 3.7 *For any guard g and a positive natural number m, $g\uparrow^m$ and g are equivalent terms. Moreover, for a guard f, if $v \models f$ for some valuation v then $v(g) = v(g|_f)$.*

To have ordered BDDs, we need to impose some order on simplified guards. Below is what we use as the ultimate order on such guards:

Definition 3.8 [order] We define a total order $\prec$ on simplified guards as:

- $\bot \prec \top \prec g$, for all simplified guards g different from $\top$, $\bot$.
- $(S^p(x) = S^q(y)) \prec (S^m(u) = S^n(v))$ iff:

$$\begin{aligned} &i)\ x \prec u \text{ or} \\ &ii)\ x \equiv u,\ p < m \text{ or} \\ &iii)\ x \equiv u,\ p \equiv m\ , y \prec v \text{ or} \\ &iv)\ x \equiv u,\ p \equiv m,\ y \equiv v,\ q < n \end{aligned}$$

According to this definition $(r_1 = t_1) \prec (r_2 = t_2)$ iff $(r_1, t_1) \prec_{lex} (r_2, t_2)$, in which $\prec_{lex}$ is a lexicographic order on quadruples of the total, well-founded orders $(\bar{V}, \prec) \times (\mathbb{N}, <) \times (\bar{V}, \prec) \times (\mathbb{N}, <)$, and therefore it is well-founded and total. This way without getting into the structures of the involved terms, only by knowing the order between them, one could determine the order of the guards.

Now we can build the term rewrite system, which will be applied on $(0, S, =)$-BDDs and will generate an ordered version of them.

Definition 3.9 [$(0, S, =)$-R-OBDD] An $(0, S, =)$-R-OBDD (Representant-Ordered $(0, S, =)$-BDD) is a simplified $(0, S, =)$-BDD (i.e. all its guards are simplified) which is a normal form with respect to the following term rewrite system:

(i) $ITE(\top, T_1, T_2) \to T_1$

(ii) $ITE(\bot, T_1, T_2) \to T_2$

(iii) $ITE(g, T, T) \to T$

(iv) $ITE(g, ITE(g, T_1, T_2), T_3) \to ITE(g, T_1, T_3)$

(v) $ITE(g, T_1, ITE(g, T_2, T_3)) \to ITE(g, T_1, T_3)$

(vi) $ITE(g_1, ITE(g_2, T_1, T_2), T_3) \to ITE(g_2, ITE(g_1, T_1, T_3), ITE(g_1, T_2, T_3))$ *if* $g_1 \succ g_2$

(vii) $ITE(g_1, T_1, ITE(g_2, T_2, T_3)) \to ITE(g_2, ITE(g_1, T_1, T_2), ITE(g_1, T_1, T_3))$ *if* $g_1 \succ g_2$

(viii) *for every simplified* $(0, S, =)$*-BDD* C*, if* y *occurs in* g *and* $S^n(x) = S^m(y) \prec g$ *then:*
$ITE(S^n(x) = S^m(y),\ C[g], T\ ) \to ITE(S^n(x) = S^m(y),\ C[\ g|_{S^n(x)=S^m(y)}], T\ )$

In the viiith rule, one of m or n must be 0, since according to the assumption $S^n(x) = S^m(y)$ is a simplified guard (Lemma 3.4).

Obviously the result of applying any rule (of Definition 3.9) on a simplified BDD is a simplified BDD. The BDD which can no longer be simplified is called of *normal* form. In the sequel we show that each BDD has a normal form. The next lemma in immediate from this definition:

Lemma 3.10 *Suppose* $T \in B$ *is a* $(0, S, =)$*-BDD which becomes* T' *after applying any arbitrary rule of Definition 3.9 on it. Then* T *and* T' *are equivalent. As a result each* $(0, S, =)$*-BDD is equivalent with any of its normal forms.*

Example 3.11 Let $x \prec y \prec z$. Below we present our simplification technique over $ITE(S(y) = z, ITE(x = S^2(y), \top, \bot), \bot)$ (Figure 2).

$$
\begin{aligned}
& ITE(S(y) = z, ITE(x = S^2(y), \top, \bot), \bot) \\
\xrightarrow{6} \quad & ITE(x = S^2(y), ITE(S(y) = z, \top, \bot), ITE(S(y) = z, \bot, \bot)) \\
\xrightarrow{3} \quad & ITE(x = S^2(y), ITE(S(y) = z, \top, \bot), \bot) \\
\xrightarrow{8} \quad & ITE(x = S^2(y), ITE(\{S^3(y) = S^2(z)[S^2(y) := x]\}\downarrow, \top, \bot), \bot) \\
\overset{substitution}{\equiv} \quad & ITE(x = S^2(y), ITE(\{S(x) = S^2(z)\}\downarrow, \top, \bot), \bot) \\
\equiv \quad & ITE(x = S^2(y), ITE(x = S(z), \top, \bot), \bot)
\end{aligned}
$$

3.2 Termination

To show that our system is terminating we first prove some properties on $\prec$.

Lemma 3.12 *Let* $f \equiv S^n(x) = S^m(y)$ *and* $g \equiv S^k(v) = S^l(w)$*. If* $f \prec g$ *and* $f \equiv f\downarrow$ *and* $g \equiv g\downarrow$ *and* $y \in \{v, w\}$*, then* $g|_f \prec g$*.*

Definition 3.13 [recursive path order for BDDs] Let S and T be simplified BDDs. Then $S \equiv f(S_1, S_2) \succ_{rpo} g(T_1, T_2) \equiv T$ if and only if

(I) $S_1 \succeq_{rpo} T$ or $S_2 \succeq_{rpo} T$; or

(II) $f \succ g$ and $S \succ_{rpo} T_1, T_2$; or

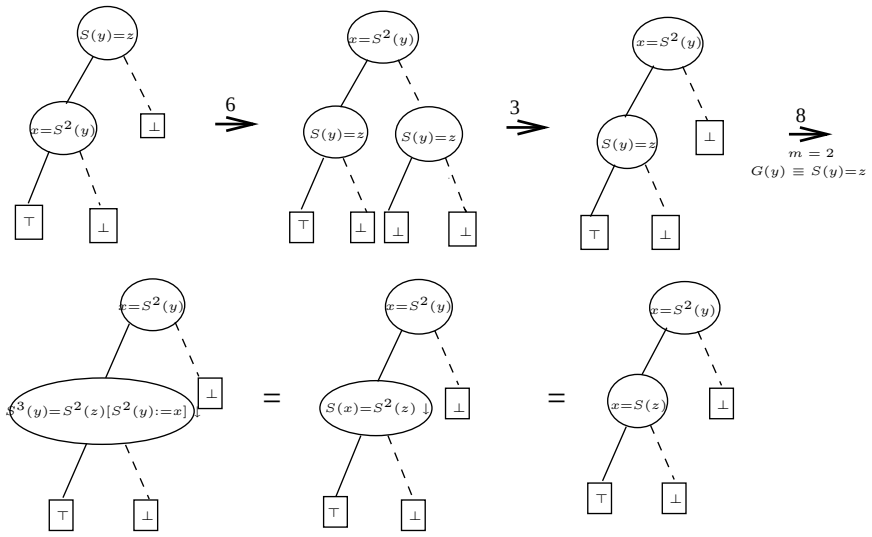


Figure 2. Derivation in Example 3.11

(III) $f \equiv g$ and $S \succ_{rpo} T_1, T_2$ and either $S_1 \succ_{rpo} T_1$, or ($S_1 \equiv T_1$ and $S_2 \succ_{rpo} T_2$).

Here $x \succeq_{rpo} y$ means that $x \succ_{rpo} y$ or $x \equiv y$, and $S \succ_{rpo} T_1, T_2$ is shorthand for $S \succ_{rpo} T_1$ and $S \succ_{rpo} T_2$.

This definition forces an order on BDDs, as shown in [4] Chapter 6.

Lemma 3.14 *Let f, g be two simplified guards, such that $f \prec g$, and C is a $(0, S, =)$-BDD. If g occurs at least once in C, then $C[g] \succ_{rpo} C[f]$.*

Proof. This holds because of the monotonical behaviour of $\succ_{rpo}$ ([4] Chapter 6). ■

Next lemma shows that if a sub-tree of a BDD is replaced by a smaller tree, then the whole tree will become smaller.

Lemma 3.15 *If T is a simplified BDD, and S a sub BDD of it, and S' is another simplified BDD where $S \succ_{rpo} S'$, then if we replace S by S' in T and derive T', we will have $T \succ_{rpo} T'$*

Proof. It is easy by induction on the structure of T and Definition 3.13(III). ■

Applying any rewrite rule on a BDD results in a smaller BDD with respect to the $\succ_{rpo}$ order:

Lemma 3.16 *Each rewrite rule is contained in $\succ_{rpo}$.*

Proof. The only non straightforward case is rule 8, wherefore we use Lemmas 3.12 and 3.14. ■

Now, we can prove that our term rewrite system (Definition 3.9) always terminates:

Theorem 3.17 (Termination) *The rewrite system defined in Definition 3.9 is terminating on simplified $(0, S, =)$-BDDs.*

Proof. According to the previous lemma all rewrite rules are contained in $\succ_{rpo}$. This implies termination, because $\succ_{rpo}$ is a reduction order, i.e. well-founded, and closed under substitutions and contexts [4] Chapter 6. ■

As an immediate result of termination, each BDD has a normal form:

Corollary 3.18 *Every $(0, S, =)$-BDD is equivalent to at least one $(0, S, =)$-R-OBDD.*

3.3 Satisfiability of paths in $(0, S, =)$-R-OBDDs

We consider α, β, γ to represent finite sequences of (possibly negated) guards. Let us denote the empty sequence by ε and the concatenation of sequences α and β by $\alpha.\beta$.

Definition 3.19 We define the set of *Paths* in a $(0, S, =)$-BDD by:

- $Pat(\top) = Pat(\bot) = \varepsilon$
- $Pat(ITE(g, T_1, T_2)) = \{g.\alpha \mid \alpha \in Pat(T_1)\} \cup \{\neg g.\beta \mid \beta \in Pat(T_2)\}$

α is an *ordered* path if it occurs in some $(0, S, =)$-OBDD. We are going to prove that al paths in an OBDD are satisfiable.

The next two lemmas give syntactical properties on OBDDs, which can be used for proving satisfiability of each path in an OBDD.

Lemma 3.20 *Let $T \equiv ITE(S^m(x) = S^n(z), T_1, T_2)$ be a $(0, S, =)$-R-OBDD. Let α be a path in T_2 and $H = \{S^{j_i}(x) = r_i \mid 1 \leq i \leq k\}$ be the set of all positive guards on α which have x as their left-hand side variable. Then for each positive guard on α with a variable which occurs in an atom in H, we can conclude that the guard belongs to H.*

The next lemma says that in an OBDD, the left-most variable of each guard will not occur at the right-hand side of any guard underneath it.

Lemma 3.21 *Let $T \equiv ITE(S^m(x) = r, T_1, T_2)$ be a $(0, S, =)$-R-OBDD. Then for all guards $s = t$ occurring in T_1 or T_2 we have $x \not\equiv \mathtt{Var}(t)$ (i.e. $t \not\equiv S^k(x)$ for any k).*

Now we prove the second main theorem, which is satisfiability of each path in an OBDD.

Theorem 3.22 (Paths are satisfiable) *Each path in a $(0, S, =)$-R-OBDD is satisfiable.*

Satisfiability of paths in OBDDs results in:

Corollary 3.23

- *$\top$ is the only tautological $(0, S, =)$-R-OBDD.*
- *$\bot$ is the only contradictory $(0, S, =)$-R-OBDD.*
- *Every other $(0, S, =)$-R-OBDD is satisfiable.*

Proof. Each path in a tautological OBDD should end in a $\top$. Because if T is a tautological OBDD, containing a path α which ends in a $\bot$, then according to Theorem 3.22, there is a valuation v which satisfies α. But then $v(T) = 0$, which is impossible since T is a tautology. Therefore, if T has more than one leaf, rule 3 of Definition 3.9 is applicable on a tautological OBDD which is not $\top$, and this contradicts the orderedness. So $T \equiv \top$. Similarly for a contradictory one. ■

4 The Transformer Algorithm

In this section we present an algorithm to transform any formula in our logic into an equivalent OBDD. One could consider an algorithm which applies the rules of our term rewrite system one by one, on the given formula, until it reaches an OBDD. Although this is possible, it is not efficient, since in the process a lot of unnecessary cases will be checked on the formula, until it can reach a normal form. We instead extend the algorithm in [12], which is based in Shannon's expansion with the smallest equation $x = y$:

$$\varphi \Longleftrightarrow (x = y \wedge \varphi|_{x=y}) \vee (x \neq y \wedge \varphi|_{x \neq y}).$$

Since the set of BDDs is a subset of the set of formulas,so in this section we may use BDDs wherever we work with set of formulas. In order to simplify formulas, we extend the reducing method in Definition 3.6 over all formulas:

Definition 4.1 For any formula φ and any simplified literal (guard) l, we define:

$$\begin{aligned}(\neg\varphi)|_l &:= \neg(\varphi|_l)\\ (\varphi \wedge \psi)|_l &:= (\varphi|_l) \wedge (\psi|_l)\\ ITE(\varphi_1, \varphi_2, \varphi_3)|_l &:= ITE((\varphi_1)|_l, (\varphi_2)|_l, (\varphi_3)|_l)\end{aligned}$$

As a result the corresponding lemma (i.e. Lemma 3.7) is extendible to all formulas, as well:

Lemma 4.2 *If $v \models l$ for a literal l and a valuation v, then $v(\varphi) \equiv v(\varphi|_l)$.*

Proof. By induction on the structure of φ and using Lemma 3.7, it is straightforward. ■

Applying an $|_l$ operation on a BDD will not increase the size of the BDD.

Lemma 4.3 *Let T be a simplified BDD. Suppose l is a simplified guard possibly occurring on T. If l is no bigger than the guards occurring on T then $T \succeq_{rpo} T|_l$.*

Proof. According to Lemma 3.12 and Definition 3.6, the guards do not get bigger. Now, by using induction over the structure of T and Definitions 4.1, the proof is trivial. ■

Intuitively, the operation $|_l$ replaces the rightmost variable occurring in the (positive) literal l with the left most term sitting in l. So, one would expect that the rightmost variable would not appear in the formula after this operation is applied:

Lemma 4.4 *Let l be a simplified guard of the form $r = S^m(y)$ in which $y \in V$. Then $y \notin T|_l$.*

Proof. It is trivial by Definitions 4.1 and 3.6. ■

In the next definition we generalize the simplification method over guards in Definition 3.2 to all formulas, because in practice we also need to make the guards, occurring inside BDDs and formulas, smaller if possible:

Definition 4.5 We extend the simplification rules of Definition 3.2 to all formulas below:

$$g \longrightarrow g\downarrow \quad \textit{(if } g \textit{ is not simplified)}$$
$$\neg g \longrightarrow \neg(g\downarrow) \quad \textit{(if } g \textit{ is not simplified)}$$

$$(\varphi \wedge \bot) \longrightarrow \bot \qquad (\bot \wedge \varphi) \longrightarrow \bot$$
$$(\varphi \wedge \top) \longrightarrow \varphi \qquad (\top \wedge \varphi) \longrightarrow \varphi$$
$$(\neg\top) \longrightarrow \bot \qquad (\neg\bot) \longrightarrow \top$$
$$ITE(\top,\ \varphi,\ \psi) \longrightarrow \varphi \qquad ITE(\bot,\ \varphi,\ \psi) \longrightarrow \psi$$
$$ITE(g,\ \psi,\ \psi) \longrightarrow \psi$$

$\varphi\Downarrow$ represents a most simplified version of φ.

Similar to the Lemma 3.3, it can be proved that:

Lemma 4.6 *If φ is simplified to φ' using Definition 4.5, then φ and φ' are equivalent.*

Lemmas 4.2 and 4.6 together will lead to:

Lemma 4.7 *If $v \models l$ for a literal l and a valuation v, then $v(\varphi) \equiv v((\varphi|_l)\Downarrow)$.*

Theorem 4.8 *Let T be a simplified BDD. If g is the smallest guard occurring in T, then $T \succ_{rpo} (T|_l)\Downarrow$ for $l \in \{g, \neg g\}$.*

Proof. According to the last case of Definition 4.1, in order to calculate $T|_l$ we could apply the operation $|_l$ to its sub-trees. Let us consider a case distinction over l:

- $l \equiv g$. g occurs in T. Hence there is a sub-tree of T of the form $ITE(g, T_1, T_2)$. Let T' be one of these. Therefore:

$$\begin{aligned}
(T'|_g)\Downarrow &\equiv ITE(g|_g, T_1|_g, T_2|_g)\Downarrow && \text{(Definition 4.1)}\\
&\equiv (T_1|_g)\Downarrow && \text{(Definition 4.5)}\\
&\preceq_{rpo} T_1|_g && \text{(using Lemma 3.16 for Definition 4.5)}\\
&\preceq_{rpo} T_1 && \text{(Lemma 4.3)}\\
&\prec_{rpo} T' && \text{(Lemma 3.13(I))}
\end{aligned}$$

  Above, $A \preceq_{rpo} B$ means $B \succeq_{rpo} A$. Now according to Lemma 3.15, our original tree is bigger. Meaning that $T \succ_{rpo} (T|_l)\Downarrow$. In this last conclusion we also used Lemma 3.16 and Lemma 4.3 implicitly.

- $l \equiv \neg g$. Similar. ■

The following function, called `sort`, is introduced to take the smallest guard occurring in a formula and bring it to the topmost place, and sort and simplify the formula afterwards.

Definition 4.9 We define a function `sort` on simplified formulas, which sorts and simplifies the formulas with respect to their smallest guard.

- $\texttt{sort}(\bot) \equiv \bot$
- $\texttt{sort}(\top) \equiv \top$
- Let g be the smallest guard occurring (positively or negatively) in φ. Then

$$\texttt{sort}(\varphi) \equiv \begin{cases} \texttt{sort}(\varphi|_g \Downarrow) & \text{if } \texttt{sort}(\varphi|_g \Downarrow) \equiv \texttt{sort}(\varphi|_{\neg g} \Downarrow) \\ ITE(g,\ \texttt{sort}(\varphi|_g \Downarrow),\ \texttt{sort}(\varphi|_{\neg g} \Downarrow)) & \text{otherwise} \end{cases}$$

Since each BDD is a formula, therefore the function `sort` can be recursively used. The following lemmas are immediately derived from this definition.

Theorem 4.10 $\texttt{sort}(\varphi)$ *is terminating over any formula* φ*.*

Proof. Using induction over the structure of φ. ■

Lemma 4.11 *The set of all variables in* $\texttt{sort}(\varphi)$ *is a subset of the set of all variables in* φ*.*

Lemma 4.12 $\texttt{sort}(\varphi)$ *is a BDD for any formula* φ*. Moreover* φ *is equivalent to* $\texttt{sort}(\varphi)$*, i.e.* $v(\varphi) \equiv v(\texttt{sort}(\varphi))$ *for any valuation* v*.*

One application of `sort` does not always yield an OBDD:

Example 4.13 Let $\varphi \equiv ITE(x = S(z), ITE(y = z, \top, \bot), \bot)$; we show how the `OBDD` algorithm finds an equivalent OBDD for this φ.

φ is simplified already, so that $\psi = \varphi \Downarrow = \varphi$. Now $\psi \neq \bot$, hence we must enter the **while***-loop: we first need to calculate* $\texttt{sort}(\varphi)$*. $x = S(z)$ is the smallest guard.* $\texttt{sort}(\psi|_{x=S(z)}) \equiv ITE(x = S(y), \top, \bot)$ *and* $\texttt{sort}(\psi|_{x\neq S(z)}) \equiv \bot$*. Hence*

$$\begin{aligned} \texttt{sort}(\psi) &= ITE(x = S(z),\ \texttt{sort}(\psi|_{x=S(z)}),\ \texttt{sort}(\psi|_{x\neq S(z)})) \\ &= ITE(x = S(z),\ ITE(x = S(y), \top, \bot), \bot) \qquad \textit{(above)} \end{aligned}$$

Now $\psi \neq \texttt{sort}(\psi)$, hence we must repeat the **while***-loop: $x = S(y)$ is the smallest guard.* $\texttt{sort}(\psi|_{x=S(y)}) \equiv ITE(x = S(z), \top, \bot)$ *and* $\texttt{sort}(\psi|_{x\neq S(y)}) \equiv \bot$*. Hence*

$$\begin{aligned} \texttt{sort}(\psi) &= ITE(x = S(y),\ \texttt{sort}(\psi|_{x=S(y)}),\ \texttt{sort}(\psi|_{x\neq S(y)})) \\ &= ITE(x = S(y),\ ITE(x = S(z), \top, \bot), \bot) \qquad \textit{(above)} \end{aligned}$$

Again $\psi \neq \texttt{sort}(\psi)$*, hence we must repeat the* **while***-loop:* $x = S(y)$ *is the smallest guard.* $\texttt{sort}(\psi|_{x=S(y)}) \equiv ITE(x = S(z), \top, \bot)$ *and* $\texttt{sort}(\psi|_{x\neq S(y)}) \equiv \bot$*. Hence*

$$\begin{aligned}\texttt{sort}(\psi) &= ITE(x = S(y),\ \texttt{sort}(\psi|_{x=S(y)}),\ \texttt{sort}(\psi|_{x\neq S(y)}))\\ &= ITE(x = S(y),\ ITE(x = S(z), \top, \bot), \bot) \qquad \textit{(above)}\end{aligned}$$

This time $\psi = \texttt{sort}(\psi)$*, hence we must leave the* **while***-loop, and stop with* $\psi = ITE(x = S(y),\ ITE(x = S(z), \top, \bot), \bot)$ *as the outcome.*

As a result, we need to have some algorithm that can recursively apply `sort` until a fixed point is reached. This is what we look for with the next algorithm:

Definition 4.14 The following algorithm, generates a $(0, S, =)$-R-OBDD for any formula:

`OBDD(`φ`)`
  ψ:= $\varphi\Downarrow$;
  φ:= $\bot$;
  while $\varphi \neq \psi$ **do**
    φ:= ψ ;
    ψ:= `sort`(ψ) ;
  od
  return ψ

Later, in Theorem 4.17, we will prove that this algorithm always returns an OBDD. The following two lemmas describe some properties of the `sort` function which will be used to prove termination of the `OBDD` algorithm.

Lemma 4.15 *Let* T *be any simplified BDD. Then:*

(i) $\texttt{sort}(T)\Downarrow \equiv \texttt{sort}(T)$.

(ii) $T \succeq_{rpo} \texttt{sort}(T)$.

One can easily check that the `OBDD` algorithm (i.e. Definition 4.14) terminates as soon as $T \equiv \texttt{sort}(T)$. Below, we claim that such a T is ordered.

Theorem 4.16 *If* T *is a simplified BDD for which* $T \equiv \texttt{sort}(T)$*, then* T *is an ordered BDD.*

Now we can prove the main theorem which is termination of the algorithm:

Theorem 4.17 (Termination of the algorithm) *The algorithm given in Definition 4.14 is terminating, and* `OBDD`(φ) *is a* $(0, S, =)$*-R-OBDD equivalent to* φ*, for any given formula* φ*.*

Proof. According to the Lemma 4.12 $\texttt{sort}(\varphi)$ will be a BDD equivalent to φ. The $\succeq_{rpo}$ ordering is well-founded, therefore using Lemma 4.15(ii) we know that after finitely many steps we will reach a fixed point of $\texttt{sort}(\psi) \equiv \psi$, for some ψ.

Now, using Theorem 4.16, this ψ is an ordered BDD, which on the other-hand is the outcome of the $\texttt{OBDD}(\varphi)$ (by definition), and is equivalent to φ (by Lemma 4.12). ■

Below we show, on a simple example, how the OBDD algorithm operates of formulas.

Example 4.18 We consider the formula of Example 3.11:

$$\varphi \equiv ITE(S(y) = z, ITE(x = S^2(y), \top, \bot), \bot).$$

φ is simplified already, therefore $\varphi \Downarrow \equiv \varphi$ and $\psi := \varphi$. We enter the loop since $\bot \neq \varphi$. Here, $\texttt{sort}(\psi)$ is:

$$ITE(x = S^2(y),\ \texttt{sort}(\varphi|_{x=S^2(y)}),\ \texttt{sort}(\varphi|_{x\neq S^2(y)}).$$

The innermost formulas are to be computed here. We will have:

$$\texttt{sort}(\varphi|_{x=S^2(y)}) \equiv ITE(x = S(z), \top, \bot) \ \text{ and } \texttt{sort}(\varphi|_{x\neq S^2(y)}) \equiv \bot.$$

Substituting these two in the formula, we obtain

$$\texttt{sort}(\psi) \equiv ITE(x = S^2(y), ITE(x = S(z), \top, \bot), \bot).$$

Once more applying the sort function over this formula will result in $\texttt{sort}(\texttt{sort}(\psi)) \equiv \texttt{sort}(\psi)$. This is a fixed point, therefore $\texttt{OBDD}(\varphi)$ is $ITE(x = S^2(y), ITE(x = S(z), \top, \bot), \bot)$. This is an OBDD for the original formula $ITE(S(y) = z, ITE(x = S^2(y), \top, \bot), \bot)$.

5 Conclusion

In this paper we provided another sound and complete method for verification of a fragment of quantifier-free FOL. This fragment contains equality with zero and successors. We introduced an algorithm which transforms each formula into (one of) its equivalent ordered BDD (s). In an Ordered BDD all paths are satisfiable so a formula is a tautology (contradictory) if and only if the derived OBDD is q $\top$ ($\bot$), and is satisfiable otherwise.

Although the logic that we tackled is rather small, many verification problems can be expressed in this logic. A lot of research has directed towards model checking techniques for verification of large systems, with huge state spaces. In this plot, BDDs have also been of use [8,9]. Among these, symbolic model checking [15,14] have been of much interest. The idea is to use OBDDs for reducing the size of the representive state space. Although propositional logic have been much into considerations, our technique has the ability to provide a more expressive logic. This would prevent necessary obligations for finding proper transformation functions from bigger languages into propositional logic [13]. SMT solvers or also UPPAAL which is a tool used for verification of real time systems, seem closer to our purpose. UPPAAL uses separation logic for modeling real time systems. Formulas are sets of constraints over expressions like $x > y$, $x \leq 6 + z$ or $2 < x \leq 6$, etc.

Another line of research could be extension of BDD-method (current results) to other algebras. Some interesting extensions are incorporation of addition (+), or

an investigation of other free algebras (such as LISP-list structures based on null and cons).

6 Acknowledgement

I would like to thank Jaco van de Pol for his comments on the paper.

References

[1] W. Ackermann. *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1954.

[2] B. Badban. *Verification Techniques for Extensions of Equality Logic*. PhD thesis, Amsterdam Vrij University, 2006.

[3] B. Badban and J.C. van de Pol. Zero, successor and equality in binary decision diagrams. *Annals of Pure and Applied Logic*, 133(1-3):101–123, 2005.

[4] M.A. Bezem, J.W. Klop, and R.C. de Vrijer, editors. *Term Rewriting Systems*. Cambridge University Press, 2003.

[5] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[6] R.E. Bryant, S. German, and M.N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2001.

[7] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[8] Stefan Edelkamp. Heuristic search planning with bdds. In *PuK*, 2000.

[9] Stefan Edelkamp and Peter Kissmann. Limits and Possibilities of BDDs in State Space Search. In *AAAI*, pages 1452–1453, 2008.

[10] A. Goel, K. Sajid, H. Zhou, and A. Aziz. BDD based procedures for a theory of equality with uninterpreted functions. In *Proc. CAV*, 1998.

[11] J.F. Groote and J.C. van de Pol. Equational binary decision diagrams. In *Proc. of LPAR 2000*, pages 161–178, 2000.

[12] J.F. Groote and J.C. van de Pol. Equational binary decision diagrams. In *Proc. Conference on Logic for Programming and Automated Reasoning*, 2000.

[13] Shuvendu K. Lahiri, Randal E. Bryant, and Byron Cook. A symbolic approach to predicate abstraction. In *CAV*, pages 141–153, 2003.

[14] S.K. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. *CoRR'06*, abs/cs/0612003.

[15] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, 1992.

[16] G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[17] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In *Proc. CAV*, pages 455–469, 1999.

[18] O. Strichman, S.A. Seshia, and R.E. Bryant. Deciding separation formulas with SAT. In *Proc. CAV*, pages 209–222, 2002.

Σ Decision Diagrams

Didier Buchs and Steve Hostettler[1]

Software Modeling and Verification Laboratory
University of Geneva,
route de Drize 7, CH-1227 Carouge Switzerland,

Abstract

Encoding and rewriting of large set of terms is very useful in a number of domains, such as model checking and theorem proving. The challenge of encoding several billions of states requires efficient ways of representing and manipulating terms. *Term Graph Rewriting* is a well-known technique to share common subterms and thus save both memory and processing time. However, this does not always fit well to the operational framework since it destroys the original structure and replaces it by a new one. This paper introduces a new kind of Decision Diagrams (DD), especially designed to handle set of terms in an efficient way. Based on the *Set Decision Diagrams(SDD)*, an evolution of the well-known *Binary Decision Diagrams(BDD)*, we propose the *Sigma Decision Diagrams (ΣDD)*, a new approach to perform *Term Rewriting* on a set of terms in order to compute efficiently the image of that set.

Keywords: Term Rewriting; Set Decision Diagrams; Σ Decision Diagrams; Set of Terms.

1 Introduction

When performing model checking [1] using formalisms such as *High Level Petri Nets* [2,3] or the *Chemical Abstract Machine* [4] one has to manipulate large sets (usually billions) of terms expressing the state (or the set of states) of the system. This is a challenge both in terms of memory footprint and CPU consumption.

Term Graph Rewriting is an approach to perform efficient rewriting, however it alters the current term and thus the current state. This drawback becomes prohibitive when performing model checking since we need to keep all the states of the system. Therefore, we need another approach to rewrite large sets of states.

This paper proposes the Sigma Decision Diagrams (ΣDD), a new data structure that is an extension of the *Set Decision Diagrams* (SDD) [5] which are themselves an extension of the *Data Decision Diagrams* (DDD) [6] and the well-known *Binary Decision Diagrams* (BDD) [7]. Roughly speaking, those structures handle sets of sequences of assignments in a symbolic way, by sharing common sub-graphs.

[1] This project was partially funded by the COMEDIA project of the Hasler foundation, ManCom initiative project number 2107.

The SDD framework provides us a basis for performing Order-Sorted Term Rewriting [8,9] on set of terms. It also enables us to share common sub-terms and rewriting steps reducing both memory and processing time. To do that, we have extended the SDD framework to handle *Algebraic Abstract Data Types* (AADT) [10].

Ultimately the goal is to provide a complete framework based on *DDD*, *SDD* and ΣDD, in order to perform model checking of *High Level Petri Nets* and particularly on *Algebraic Petri Nets* [2,3] and their extensions as described in [11].

The article is organized as follows: first we establish the prerequisites to formally encode a *Term* and the *Rewriting Rules* using the *Decision Diagrams Framework*. Then we define the ΣDD itself and its implementation. After what we analyze benchmarks. Then we compare our work to other approaches and particularly to the more traditional *Term Graph Rewriting*. Finally, we conclude and discuss the open issues and future work.

2 Modeling formalism

Here we recall some usual definitions required to formally define *Order-Sorted Term Rewriting* of *Ordered Algebraic Specification*. We constructively define the required concepts, preceded by an informal explanation if necessary. For more details see [8].

2.1 Abstract Algebraic Data Types

Informally AADT consists of describing domain names called sorts and defining operators on them. These operators are described syntactically by their names, domains and co-domains and their semantics are constrained by conditional equations. It must be noted that our theory and our implementation are compatible with the powerful extension of AADT called Order-Sorted Algebraic Data Types. In this extension, domains are no longer disjoint but follow inclusion relations based on the given ordering. We suppose that $\mathbb{SORT}$, $\mathbb{FUNC}$ are disjoint universes of respectively sort and function names.

Definition 2.1 [Sort & S-Sorted Set] Let $S \subseteq \mathbb{SORT}$ be a finite set of sorts . A *S-sorted set* A is a union of a family of sets indexed by S ($A = \bigcup_{s \in S} A_s$), noted as $A = (A_s)_{s \in S}$. Informally an S-Sorted Set is a partitioned set in which the partitions are determined by the sorts names.

A *Signature* defines the names of the operations between the sorts. This will enable composition of said operations to build terms.

Definition 2.2 [Order-Sorted Signature and Terms] Let $\leq \subseteq (S \times S)$ be a partial order [2] . An *order-sorted signature* is a triple $\Sigma = \langle S, \leq, F \rangle$, where $S \subseteq \mathbb{SORT}$ is a finite set of sorts, $\langle S, \leq \rangle$ is a partially ordered set of sorts and $F = (F_{w,s})_{w \in S^*, s \in S}$ is a $(S^* \times S)$-sorted set of function names of $\mathbb{FUNC}$.

We often denote a function name $f \in F_{s_1,\ldots,s_n,s}$ by $f : s_1, \ldots, s_n \rightarrow s$ and $f :\rightarrow s$ if $f \in F_{\epsilon,s}$ where ϵ denotes the empty word.

[2] We extend the ordering $\leq$ on S to words of equal length of S^* by $s_1, \ldots, s_n$ iff $\forall i\ s_i \leq s'_i$ with $1 \leq i \leq n$. Similarly we extend $\leq$ to pairs $S^* \times S$ by $(w, s) \leq (w', s')$ iff $w \leq w'$ and $s \leq s'$.

The set of terms of Σ over X is a S-sorted set $T_{\Sigma,X}$, where each set $(T_{\Sigma,X})_s$ is inductively defined as follows:

- $x \in (T_{\Sigma,X})_s, \forall x \in X_s$.
- $f \in (T_{\Sigma,X})_s, \forall f :\rightarrow s'$ such that $s' \leq s$ and is called a constant.
- for all operations that are not a constant : $f(t_1, \ldots, t_n) \in (T_{\Sigma,X})_s$, $\forall f : s_1, \ldots, s_n \rightarrow s'$ such that $s' \leq s$ and $\forall t_i \in (T_{\Sigma,X})_{s_i}$ with $1 \leq i \leq n$

We also define $\tau : (T_{\Sigma,X})_s \rightarrow S$, the typing function s.t $\forall t \in (T_{\Sigma,X})_s$, $\tau(t) = s$.

Definition 2.3 [Σ-Equation] Given an order-sorted signature $\Sigma = \langle S, \leq, F\rangle$ and X a S-sorted set of variables. The set E of equations is the set of pairs $\langle t, t'\rangle$, denoted $t = t'$ with $t, t' \in (T_{\Sigma,X})_{s \in S}$. Equations are easily extended with conditions into conditional equations. Conditions are conjunctions of equations.

A term algebra is an algebra that is freely generated from a signature which specifies names and arities of the operations on it. Applying equations E to the term algebra partitions it into equivalence classes ($0 + 2$, $1 + 3 - 2$, $1 + 1$ or 2).

The term algebra quotiented by the equations (noted $T_{\Sigma_{\equiv E}}$) into equivalence classes is called the *Initial Algebra*. We call *generators* the minimal set of operators of F that can be combined to build any value of the *Initial Algebra*. The *Initial Algebra* is then said to be finitely generated by the *generators*.

2.2 Order-Sorted Term Rewriting

Operationally we use a technique called Term-Rewriting [8,9] to find the equivalence classes and thus to evaluate the terms. Giving a term, we are interested in its normal form. The stepwise application of so called rewrite rules derived from the axioms is called term rewriting. Term rewriting is a Turing complete computational model.

Definition 2.4 [Context and Subterms] Let $\Sigma = \langle S, \leq, F\rangle$ be an order-sorted signature and X be a S-sorted variable set, let also $\Box \notin F \cup X$ be a special constant symbol called a placeholder. A context C of a term $t \in T_{\Sigma,X}$ is a term $(T_{\Sigma \cup \Box,X})_s$ such that if $C_t[\Box_1, \ldots, \Box_n]$ is a context with n occurrences of $\Box$ and $t_1, \ldots, t_n$ are terms $\in (T_{\Sigma \cup \Box,X})_s$, then $C_t[t_1, \ldots, t_n]$ is the result of replacing the $\Box_i$ by the t_i.

A term $st \in (T_{\Sigma,X})_s$ is a *subterm* of $t \in (T_{\Sigma,X})_s$ noted $st \subseteq t$ if there exists a context C of term t denoted $C_t[\_]$ such that $t = C_t[st]$.

Definition 2.5 [Substitution] Let $\Sigma = \langle S, \leq, F\rangle$ be an order-sorted signature and X be a S-sorted variable set. A substitution σ is mapping $\sigma : X_s \rightarrow (T_{\Sigma,X})_{s'}$ where $s, s' \in S$ and $s' \leq s$. Every substitution σ extends uniquely to a morphism $\sigma^{\#} : (T_{\Sigma,X})_s \rightarrow (T_{\Sigma,X})_{s'}$, where $s, s' \in S$ and $s' \leq s$:

- $\sigma^{\#}(f(t_1, \ldots, t_n)) = f(\sigma^{\#}(t_1), \ldots, \sigma^{\#}(t_n)))$
- $\sigma^{\#}(f_s) = f_s$ with $f_s \in F_{\epsilon,s}$
- $\sigma^{\#}(x_s) = \sigma(x_{s'})$ if $s' \leq s$

Definition 2.6 [Rewriting Rule] Let $\Sigma = \langle S, \leq, F\rangle$ be an order-sorted signature and X be an S-sorted set of variables. Let also $e = \langle l, r, cond\rangle \in E$ be a Σ-Equation. The rewriting rule r is derived from e by orienting the relation and is noted $l \rightsquigarrow_{cond} r$

or simply $l \rightsquigarrow r$ if there is no condition. We note $Rew_{\Sigma,X} \subseteq T_{\Sigma,X} \times T_{\Sigma,X}$ the set of rewrite rules w.r.t Σ and X.

- As axioms may be conditional, rewriting rules may also be conditional.
- In the sequel, if two rewriting rules have the same left-hand side (l), they must have different conditions of application.

Definition 2.7 [Rewriting step] Let $\Sigma = \langle S, \leq, F\rangle$ be an order-sorted signature, X be a S-sorted set of variables and $l \rightsquigarrow r$ with $l, r \in T_{\Sigma,X}$ a rewrite rule. Let also t and $t' \in T_{\Sigma,X}$ be two terms. The pair $\langle t, t'\rangle$ is called a rewriting step if there exists a context C_t of term t and a substitution σ such that : $t = C_t[\sigma^{\#}(l)]$ and $t' = C_t[\sigma^{\#}(r)]$.

Let a term $t \in T_{\Sigma,X}$, t is said to be in *normal form* (irreductible) iff:

$$\nexists t' \in T_{\Sigma,X} \text{ such that } t \rightsquigarrow_{Rew_{\Sigma,X}} t'.$$

We will use the *Innermost Rewriting* strategy in the sequel. It is a bottom-up algorithm that proceeds by first normalizing the subterms of a term. When all subterms are reduced to a normal form, the term itself is considered for reduction.

3 Encoding formalism

These basic definitions will help us to formally define a state space representation and operational semantics using the *Decision Diagrams*(DD) [5,6].

Data Decision Diagrams (DDD) and Set Decision Diagrams (SDD) are both evolutions of the well-known Binary Decision Diagrams (BDD) [7]. While BDD is often seen as representing a Boolean function, it can also be seen as a set of sequences of assignments of Boolean values to variables. DDD (resp. SDD) are similar but for any kind of values (resp. sets) of the form $(var_1 := val_1).(var_2 := val_2)\ldots(var_n := val_n)$. In the sequel, $\mathbb{E}$ is the set of variable and $\forall e \in \mathbb{E}$, $Dom(e)$ is the set of values that can be taken by the variable e.

0 represents the empty Decision Diagrams, namely a sequence that finishes with 0 does not exist (like in ZBDD), 1 represents an existing sequence of assignments, and T represents the undefined sequence. T is usually obtained whenever operations are performed on incompatible DDD/SDD sequences (cf. Def. 3.2). $0, 1$ and T are called terminals since they may end a Decision Diagram.

Only the DDD definitions are covered in detail. We only give minimal definitions for the SDDs. The reader should be able to intuitively understand how they work and should refer to [5,6] for more details.

Definition 3.1 [Data Decision Diagrams] The DDD set $\mathbb{D}$ is the least set:

- The terminals $\{0, 1, T\} \subseteq \mathbb{D}$
- $\langle e, \alpha\rangle \in \mathbb{D}$ with :
  - $e \in \mathbb{E}$ with $\mathbb{E}$ the set of DDD variables.
  - $Dom(e)$ represents the domain of the variable $e \in \mathbb{E}$.
  - $\alpha : Dom(e) \to \mathbb{D}$, s.t $x \in Dom(e)$ and $\{\alpha(x) \neq 0\}$ is finite.

Notation : $e \xrightarrow{x} d$ denotes the $DDD\langle e, \alpha\rangle$ with $\alpha(x) = d$ and $\forall y \in Dom(e)$ s.t $x \neq y$, $\alpha(y) = 0$.

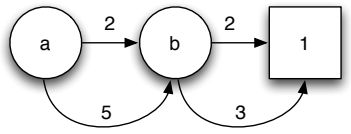


Fig. 1. DDD

The DDD with $\mathbb{E} = \{a, b\}$ and $\cup_{e\in\mathbb{E}} Dom(e) = \{2, 3, 5\}$ on the left side represents following union: $a \xrightarrow{2} b \xrightarrow{2} 1 + a \xrightarrow{2} b \xrightarrow{3} 1 + a \xrightarrow{5} b \xrightarrow{2} 1 + a \xrightarrow{5} b \xrightarrow{3} 1$
This illustrates the sharing among the encoded states. The Cartesian product of the variables' domains (Dom(a) = {2, 5}, Dom(b) = {2, 3}) is encoded in an efficient way.

Definition 3.2 [DDD compatibility] Two DDD are said compatible iff their sequences are compatible. Two sequences $s = e_1 \xrightarrow{x_1} \ldots 1$ and $s' = e'_1 \xrightarrow{x'_1} \ldots 1$ are compatible (noted $s_1 \approx s_2$) iff:

- $s = s' = 1$
- $s = e \xrightarrow{x} d \wedge s' = e' \xrightarrow{x'} d'$ such that:
  - $e = e'$ and
  - $d \approx d'$ if $x = x'$

Since DDD represent sets, we can define the usual set operations on them such as $\cup_{ddd}$, $\cap_{ddd}$, $\setminus_{ddd}$ or $\cup$, $\cap$, $\setminus$ if there is no possible confusion. For a definition of the set operations on DDD, see [6]. Concatenation $d_1 \otimes_{ddd} d_2$ concats d_2 to every terminal of d_1.

Unlike work on binary decision diagrams, operators are not limited to those previously defined. Indeed one of the strengths of the DD-like structure is their support of so-called inductive homomorphisms. Namely, operations that are inductively defined on the structure of the DD and that are compatible with the union operator. This compatibility induces a high efficiency of user defined operations. An homomorphism is a mapping ϕ from $\mathbb{D}$ to itself s.t $\phi(0) = 0$ and $\phi(d \cup d') = \phi(d) \cup \phi(d')$, $\forall d, d' \in \mathbb{D}$.

The union ($\cup$) and the composition ($\circ$) of two homomorphisms are homomorphisms. Since a decision diagram is inductively defined, operations on them can also be inductively defined. This allows the user to give a local definition of the homomorphism i.e. what it should do with a given pair $\langle variable, value\rangle$.

Definition 3.3 [Inductive Homomorphisms on DDD] Let $\phi_{e,x_{e\in E, x\in Dom(e)}}$ be a familly of homomorphisms and d_1 a DDD :

$$\forall d \in \mathbb{D}, \phi(d) = \begin{cases} 0 & \text{if } d = 0 \\ d_1 & \text{if } d = 1 \\ T & \text{if } d = T \\ \bigcup_{x\in Dom(e)} \phi_{e,x}(e \xrightarrow{x} \alpha(x)) & \text{if } d = \langle e, \alpha\rangle \end{cases}$$

is an inductive homomorphism.

As for the set operations, inductive homomorphism can be evaluated lazily saving both memory and processing time.
Example: Let suppose we want to define a user-defined function $\phi_{add_{v1}}$ that adds v_1 to every variable greater than zero and returns 1 when reaching the terminal.

$$\phi_{add_{v_1}}(e \xrightarrow{x} d) = \begin{cases} e \xrightarrow{x+v_1} \phi_{add_{v_1}}(d) & \text{, if } x > 0 \\ e \xrightarrow{x} \phi_{add_{v_1}}(d) & \text{otherwise} \end{cases},$$

$$\phi_{add_{v_1}}(1) = 1$$

$\phi^*(d)$ represents the fixpoint application of ϕ on d. That is, applying $d' = \phi(d)$ where $\phi^* = \phi^n$ with n the smallest integer such that $\phi^n(S) = \phi^{n-1}(S)$.

In order to handle more complex structures, being able to assign single values to variables is not enough. Set Decision Diagrams (SDD) solve that problem by allowing assignments to be sets. Arcs of the SDD represent a set instead of a value.

Definition 3.4 [Set Decision Diagrams] The SDD set $\mathbb{S}$ is the least set:

- $\{0, 1, T\} \subseteq \mathbb{S}$
- $\langle e, \alpha \rangle \in \mathbb{S}$ with :
  - $e \in \mathbb{E}$ with $\mathbb{E}$ the set of all SDD variables.
  - $\alpha : \pi \to \mathbb{S}$, with $\pi = \{a_0, \ldots a_i, \ldots, a_n\}$ a partition of Dom(e) s.t $\forall a_i, a_j \in \pi$, with $i \neq j$, $\alpha(a_i) \neq \alpha(a_j)$.

As for the DDD, $e \xrightarrow{x} d$ denotes the $SDD\langle e, \alpha \rangle$ with $\alpha(x) = d$. Compatible sequences, concatenation operator ($\otimes_{sdd}$) and set operators ($\cup_{sdd}$, $\cap_{sdd}$ and $\setminus_{sdd}$) are defined on SDD (for a definition see [5]). One can define SDD homomorphisms that are similar to their DDD equivalents. Since it is possible to embed DDD into SDD, it is also possible to embed DDD homomorphisms into SDD homomorphisms.

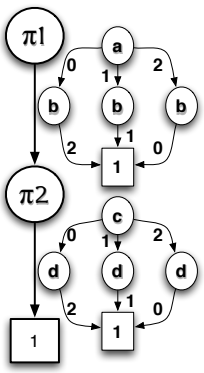


Fig. 2. SDD

The SDD on the left side represents the Cartesian product of π_1 and π_2 that is 9 paths or states. SDD ($e_{sdd} = \{\pi_1, \pi_2\}$) embed DDD ($e_{ddd} = \{a, b, c, d\}$):

$$\pi_1 \xrightarrow{a\xrightarrow{1}b\xrightarrow{1}1} \pi_2 \xrightarrow{c\xrightarrow{1}d\xrightarrow{1}1} 1 + \pi_1 \xrightarrow{a\xrightarrow{0}b\xrightarrow{2}1} \pi_2 \xrightarrow{c\xrightarrow{1}d\xrightarrow{1}1} 1 +$$
$$\pi_1 \xrightarrow{a\xrightarrow{2}b\xrightarrow{0}1} \pi_2 \xrightarrow{c\xrightarrow{1}d\xrightarrow{1}1} 1 + \pi_1 \xrightarrow{a\xrightarrow{1}b\xrightarrow{1}1} \pi_2 \xrightarrow{c\xrightarrow{0}d\xrightarrow{2}1} 1 +$$
$$\pi_1 \xrightarrow{a\xrightarrow{0}b\xrightarrow{2}1} \pi_2 \xrightarrow{c\xrightarrow{0}d\xrightarrow{2}1} 1 + \pi_1 \xrightarrow{a\xrightarrow{2}b\xrightarrow{0}1} \pi_2 \xrightarrow{c\xrightarrow{0}d\xrightarrow{2}1} 1 +$$
$$\pi_1 \xrightarrow{a\xrightarrow{1}b\xrightarrow{1}1} \pi_2 \xrightarrow{c\xrightarrow{2}d\xrightarrow{0}1} 1 + \pi_1 \xrightarrow{a\xrightarrow{0}b\xrightarrow{2}1} \pi_2 \xrightarrow{c\xrightarrow{2}d\xrightarrow{0}1} 1 +$$
$$\pi_1 \xrightarrow{a\xrightarrow{2}b\xrightarrow{0}1} \pi_2 \xrightarrow{c\xrightarrow{2}d\xrightarrow{0}1} 1$$

Again, the power of the SDD lies in the Cartesian product symbolic encoding. Using SDD, thanks to the sets, we end up with a two-dimensional symbolic encoding.

From an implementation point of view, we can leverage on the canonicity (thanks to the DD creation and DD union operator) of the representation in order to implement constant time equality between DD and thus implement efficient caching. Every set operation or homomorphism is applied on an inductive structure and thus each processing step can be put in the cache for further use. This is very useful to save computing time and it allows to efficiently implement fix-point computations.

4 ΣDD

In this section, we will formally define the ΣDD and operations on them. Since the goal is to rewrite sets of terms, we need the standard operations from the set theory (namely union, intersection and difference) and a way to efficiently represent such sets. We have seen in the previous section (cf. section 3) that Set Decision Diagrams are well suited to represent sets that are recursively defined and that they allow sharing both data structure and computations. Thus, we have chosen SDD as the basic model to build the ΣDD. We will first define what is a ΣDD, then how to encode a set of terms as ΣDD and finally we will prove that the encoding and decoding morphisms are bijections.

4.1 Term Encoding and Extraction

Definition 4.1 [ΣDD] Let $\Sigma = \langle S, \leq, F\rangle$ be an order-sorted signature and X be a set of S-sorted set of variables and $T_{\Sigma,X}$ be an S-sorted set. The S-sorted set $\mathbb{SIGDD}_\Sigma$ of ΣDD over signature Σ is inductively defined by $t \in \mathbb{SIGDD}_\Sigma$ iff:

- $t = 0$ which represents the empty ΣDD and thus the empty set of terms.
- $t = \langle s, \alpha\rangle$ with $s \in S$ and
  $\alpha : \pi \to \mathbb{SIGDD}_\Sigma$, with $\pi = \{a_1, \ldots a_i, \ldots, a_n\}$ a partition of $\mathbb{SIGDD}_\Sigma \cup F_s \cup X_s$ s.t $\forall a_i, a_j \in \pi$, with $1 \leq i, j \leq n$ and $i \neq j$, $\alpha(a_i) \neq \alpha(a_j)$. Moreover $\forall a_i \in \pi, a_i \in (\mathbb{SIGDD}_\Sigma)_{s'}$ with $s' \leq s \vee a_i \subseteq F_s \cup X_s$.

Example : The following $\Sigma DD : \mathbb{B} \xrightarrow{>} \mathbb{Z} \xrightarrow{\mathbb{N}\xrightarrow{0}1} \mathbb{Z} \xrightarrow{\mathbb{N}\xrightarrow{+}\mathbb{N}\xrightarrow{0}1\mathbb{N}\xrightarrow{s}\mathbb{N}\xrightarrow{0}1} 1$ is the encoding of term $> (0, +(0, s(0)))$ with $F = \{>: \mathbb{Z}, \mathbb{Z} \to \mathbb{B}, + : \mathbb{N}, \mathbb{N} \to \mathbb{N}, s : \mathbb{N} \to \mathbb{N}, 0 :\to \mathbb{N}$, $S = \{\mathbb{B}, \mathbb{N}, \mathbb{Z}\}$ and $\leq = \{\mathbb{N} < \mathbb{Z}\}$.

Set operations and particularly union is an important operation to ensure the canonicity of the representation. This canonicity is very important to enable constant time equality and efficient caching. ΣDD union is extended from the SDD union as defined in [12]. The main difference is the support of order-sorting and hence the notion of compatible variable.

Definition 4.2 [Compatible ΣDD & Union] As for SDD we have the notion of compatible ΣDD :

- $s \xrightarrow{\{x\}} 1$ and 1 are incompatible.
- s and s' are compatible iff $s \leq s'$ or $s' \leq s$.
- $s \xrightarrow{\{x\}} \Sigma DD$ and $s' \xrightarrow{\{x'\}} \Sigma DD'$ are compatible, iff s and s' are compatible and ΣDD and $\Sigma DD'$ are compatible.

This extended compatibility enables union between ΣDD with two sorts that are in the same hierarchy, the union always returns the less specific one:
$s \xrightarrow{\{x\}} 1 \cup s' \xrightarrow{\{x'\}} 1 = s' \xrightarrow{\{x \cup x'\}} 1$ if $s \leq s'$ and $s \xrightarrow{\{x \cup x'\}} 1$ otherwise.

Example: With $\mathbb{N} \subseteq \mathbb{Z}$, following union $\mathbb{N} \xrightarrow{\{1\}} 1 \cup \mathbb{Z} \xrightarrow{\{-1\}} 1$ returns $\mathbb{Z} \xrightarrow{\{-1,1\}} 1$. The rest of the union operation as well as the other set operations remain identical

to the SDD set operations as defined in [12].

We will now analyze how to encode a set of terms as a ΣDD. Since we extended the notion of compatible Decision Diagrams, it enables us to handle order sorting.

Definition 4.3 [ΣDD encoding homomorphism] Let $\Sigma = \langle S, \leq, F\rangle$ be a signature and X be a set of S-sorted set of variables. Let also T be a set of terms over $\langle \Sigma, X\rangle$ noted $T_{\Sigma,X}$. The ΣDD of $T \subseteq \mathcal{P}(T_{\Sigma,X})$ noted ΣDD_T is inductively defined by the encoding morphism $en : \mathcal{P}(T_{\Sigma,X}) \rightarrow \mathbb{SIGDD}_\Sigma$ such that :

- $en(\emptyset) = 0$
- $en(\{t\}) = s \xrightarrow{\{x\}} 1$, if $t = x$ with $x \in X_s$
- $en(\{t\}) = s \xrightarrow{\{k\}} 1$, if $t = k$ with $k \in F_{\epsilon,s}$
- $en(\{t\}) = s \xrightarrow{\{f\}} s_1 \xrightarrow{en(t_1)} \ldots s_n \xrightarrow{en(t_n)} 1$, if $t = f(t_1, \ldots, t_n)$ with $f : s_1, \ldots, s_n \rightarrow s$
- $en(\{t\} \cup T) = en(\{t\}) \cup en(T)$ where the union between ΣDD is defined in definition 4.2.

Sets are first class citizens in this definition. This enables easy encoding of a set of terms which is a very important feature is number of applications.

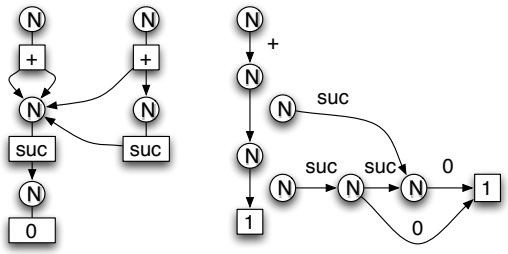


Fig. 3. ΣDD : Term Graph vs ΣDD

Fig. 3 shows the difference in sharing between regular term graph encoding (left side) and ΣDD encoding (right side). Both graphs represents the terms $\{+(s(0), s(s(0))), +(s(0), s(0))\}$. In the ΣDD approach, the second operand is treated as one sub graph and thus both terms can be rewritten in one sequence.

Of course, we need to be able to extract the set of terms that are encoded in a ΣDD. The extracting morphism takes a ΣDD and returns the associated set of terms.

Definition 4.4 [ΣDD extracting morphism] Let $\Sigma = \langle S, \leq, F\rangle$ be a signature and X be a set of S-sorted set of variables. Let also T be a set of terms over $\langle \Sigma, X\rangle$ noted $T_{\Sigma,X}$. The extracting morphism $ex : \mathbb{SIGDD}_\Sigma \rightarrow \mathcal{P}(T_{\Sigma,X})$ is inductively defined as follow:

- $ex(0) = \emptyset$, the empty set of terms.
- $ex(\Sigma_T) = T_s$, if $\Sigma_T = s \xrightarrow{T_s} 1$ with $T_s \subseteq X_s \cup F_{\epsilon,s}$
- $ex(\Sigma_T) = \bigcup_{f \in T_s} \bigcup_{<st_1,\ldots,st_n> \in \prod_{i=1}^{n} ex(\Sigma_{ST_i})} f(st_1, \ldots, st_n)$, if $\Sigma_t = s \xrightarrow{T_s} s_1 \xrightarrow{\Sigma_{ST_1}} \ldots s_n \xrightarrow{\Sigma_{ST_n}} 1$ with $T_s \subseteq F_{w,s}$, $w = (s_1, s_2, ..., s_n) \neq \epsilon$ and $\prod_{i=1}^{n} ex(\Sigma_{ST_i}) = \cup_{st_1 \in ex(\Sigma_{ST_1})} \ldots \cup_{st_n \in ex(\Sigma_{ST_n})} < st_1, \ldots, st_n >$

Lemma 4.5 (ex is an homomorphism)

$ex(\Sigma_t \cup \Sigma_{T'}) = ex(\Sigma_t) \cup ex(\Sigma_{T'})$ *Proof by induction on the structure of the* ΣDD.

In order for encoded/extracted terms to remain consistent, we must prove that whenever we extract a previously encoded set of terms, it remains identical.

Lemma 4.6 ($ex(en(T_s)) = T_s$) *By structural induction we show that* $\forall T_s \subseteq T_{\Sigma,X}$, $ex(en(T_s)) = T_s$.

Let's establish the base case :

- *if* $T_s \subseteq X_s \cup F_{\epsilon,s}$, $en(T_s) = s \xrightarrow{T_s} 1$ *and* $ex(s \xrightarrow{T_s} 1) = T_s$ *by Def. 4.3 and Def. 4.4*

Now for the inductive step, consider the set of terms $T_s = \{t_s\} \cup T'_s$ *with* $t_s = f(t_1 \dots t_n)$ *where* $f \in F_{w,s}$ *and* $t_1 \dots t_n \in T_{\Sigma.X}$*:*

- $ex(en(t_s)) = t_s$ *by Def. 4.3 and Def. 4.4 and*
- $ex(en(T_s)) = ex(en(\{t_s\} \cup T'_s))$
- $ex(en(\{t_s\} \cup T'_s)) = ex(en(\{t_s\})) \cup ex(en(T'_s))$ *because en and ex are homomorphisms.*

In order to prove the bijection, we first prove that $en \circ ex$ is the identity morphism and thus that en and ex are isomophisms.

Corollary 4.7 (Identity morphism on $\mathcal{P}(T_{\Sigma,X}) \times \mathbb{SIGDD}_\Sigma$**)** *Based on the previous lemma, we deduce that* $ex \circ en$ *is the identity morphism on* $\mathcal{P}(T_{\Sigma,X}) \times \mathbb{SIGDD}_\Sigma$.

Lemma 4.8 ($en(ex(\Sigma_t)) = \Sigma_t$) *As for lemma 4.6, by structural induction, we show that* $en(ex(\Sigma_t)) = \Sigma_t$.

Corollary 4.9 (identity morphism on $(\mathbb{SIGDD}_\Sigma \times \mathcal{P}(T_{\Sigma,X}))$**)** *Based on the previous lemma, we deduce that* $en \circ ex$ *is the identity morphism on* $(\mathbb{SIGDD}_\Sigma \times \mathcal{P}(T_{\Sigma,X}))$.

Lemma 4.10 (en **and** ex **are isomorphisms)** $f : X \to Y$ *is called an isomorphism if there exists a morphism* $g : Y \to X$ *such that* $f \circ g = id_Y$ *and* $g \circ f = id_X$. *Based on previous lemmas we say that given* $en : \mathcal{P}(T_{\Sigma,X}) \to \mathbb{SIGDD}_\Sigma$ *and* $ex : \mathbb{SIGDD}_\Sigma \to \mathcal{P}(T_{\Sigma,X})$ *we have :*

- $id_{T_{\Sigma,X}} = en \circ ex$
- $id_{\Sigma DD_{T_{\Sigma,X}}} = ex \circ en$

And thus en(resp. ex) is an isomorphism of $T_{\Sigma,X}$ *(resp.* $\mathbb{SIGDD}_\Sigma$*) to* $\mathbb{SIGDD}_\Sigma$ *(resp.* $T_{\Sigma,X}$*).*

Corollary 4.11 (Canonicity) *Since en and ex are isomorphisms and consequently bijections they guarantee the canonicity.*

- $\forall t_1, t_2 \in T_{\Sigma,X}$, $en(t_1) = en(t_2) \Leftrightarrow t_1 = t_2$.
- $\forall \Sigma DD_{t_1}, \Sigma DD_{t_2} \in \mathbb{SIGDD}_\Sigma$, $ex(\Sigma DD_{t_1}) = ex(\Sigma DD_{t_2}) \Leftrightarrow \Sigma DD_{t_1} = \Sigma DD_{t_2}$.

Corollary 4.12 *Since en is a bijection,* $en(t_1 \circ t_2) = en(t_1) \circ en(t_2)$ *with* $t_1, t_2 \subseteq T_{\Sigma,X}$ *and* $\circ \in \{\cup, \cap, \setminus\}$

Corollary 4.13 *Since ex is a bijection,* $ex(\Sigma_{t_1} \circ \Sigma_{t_2}) = ex(\Sigma_{t_1}) \circ ex(\Sigma_{t_2})$ *with* $\Sigma_{t_1}, \Sigma_{t_2} \in \mathbb{SIGDD}_\Sigma$ *and* $\circ \in \{\cup, \cap, \setminus\}$

4.2 Rewriting operations on ΣDD

To reach a normal form, one should choose a rewriting strategy. We have implemented both innermost and outermost rewriting. In the sequel, we will only present innermost rewriting. The reader should be able to easily adapt the definition for outermost rewriting. Please note that a ΣDD is said to be in a canonical form (normal form) as soon as the application of the $\phi_{InrMstRw}$ homomorphism reaches a fixpoint. Alternatively, since we do not support axioms between generators, if it is by construction only built upon generators.

Definition 4.14 [InnerMost Rewriting Homomorphism]
Let $t \in \mathbb{SIGDD}$ be a ΣDD (set of terms) to rewrite and $\phi_{InrMstRw}$ the homomorphism that rewrites a ΣDD until it is reduced to a canonical form. Please note that $\phi^*_{InrMstRw}$ stands for the fixpoint application of the $\phi_{InrMstRw}$ homomorphism as described in Def. 3.3.

$$\phi_{InrMstRw}(s \xrightarrow{x} d) = \begin{cases} s \xrightarrow{\phi^*_{InrMstRw}(x)} \phi^*_{InrMstRw}(d) & \text{, if } x \text{ and } d \in \mathbb{SIGDD} \\ \phi_{Apply}(s \xrightarrow{x} \phi^*_{InrMstRw}(d)) & \text{, if } d \in \mathbb{SIGDD} \text{ and } x \subseteq F \end{cases}$$

$$\phi_{InrMstRw}(1) = 1$$

$\phi_{InrMstRw}$ first checks whether the current value on the arc is a ΣDD or a set of operators. In the first case, it reduces the ΣDD to a canonical form and does the same for the sub-graph. In the second case, it applies the ϕ_{Apply} homomorphism to the whole graph after having reduced the sub-graph. The ϕ_{Apply} homomorphism applies a reduction rule on the given ΣDD and thus performs a rewriting step.

4.2.1 Rewriting Step

Given a rewriting rule $Rule = l \rightsquigarrow r$, performing graph rewriting is usually expressed using the following equation : $G_{T'} = (G_T \setminus G_l) \cup G_r$. In which $G_{T'}$ represents the result of the transformation, G_T is the host graph that is the graph on which the transformation is applied, G_l (resp. G_r) the left(resp. right) graph namely the graph that matches the left (resp. right) term of the rewriting rule. Obviously since G_T is, in our case, a ΣDD, rewriting operations are applied on a set of terms.

The ϕ_{Apply} homomorphism, applies a rewriting step as explained in Def. 2.7. By extending $\sigma^\#$ to ΣDD we have $G_l = \sigma^\#(en(l))$, $G_r = \sigma^\#(en(r))$.

Definition 4.15 [Rule Application Homomorphism]
Let $op \in F$ be an operation symbol and let $\phi_{pm_{op}}$ the pattern matcher homomorphism that unifies a ΣDD with the left-hand side of rewriting rules starting with operating symbol *op* and returns the pair $\langle G_l, G_r \rangle$. The rule application homomorphism is defined by :

- $\phi_{Apply}(s \xrightarrow{x} d) = \begin{cases} \bullet\ s \xrightarrow{x} 1 \text{ if } d = 1 \text{ and } x \subseteq \mathcal{P}(F_{\epsilon,s} \cup X_s) \\ \bullet\ \bigcup_{op \in x}(G_T \setminus G_l) \cup G_r \text{ with } G_T = s \xrightarrow{x} d \text{ and} \\ \ \mathrm{l} \xrightarrow{G_l} \mathrm{r} \xrightarrow{G_r} 1 = \phi_{pm_{op}}(G_T) \text{ otherwise.} \end{cases}$
- $\phi_{Apply}(1) = 1$

To apply a rule we must first check whether a given DD fulfills a given pattern and thus does qualify for a given rewriting rule. Moreover, the pattern matcher needs to create a substitution that can later be used to check the conditions of application of the rule and build the right-hand side. The pattern matchers are built from the axioms. If several axioms share the same left-hand side, they must have different conditions of application.

An inductive homomorphism can only return a ΣDD and, therefore, we have to wrap up the left-hand side and the right and side in a single ΣDD.

Definition 4.16 [Pattern Matcher Homomorphism] Let $Spec = \langle \Sigma, X, E \rangle$ be an ordered algebraic specification, let Rew_{Spec} be the set of rewriting rules built upon $Spec$, $rule = \langle l, r, cond \rangle \in Rew_{Spec}$ and its encoding $rule_{\Sigma DD} = \langle en(l), en(r), en(cond) \rangle = \langle \Sigma DD_l, \Sigma DD_r, \Sigma DD_{cond} \rangle$ be a rewriting rule with $\Sigma DD_l = s \xrightarrow{op} d'$. Let $rule'_{\Sigma DD} = \langle d', \Sigma DD_r, \Sigma DD_{cond} \rangle$ be the sub-term to match. The rule-sorted set of pattern matcher homomorphism $(\phi_{pm})_{rule \in Rew_{Spec}}$ is defined by :

$$\phi_{pm_{rule_{\Sigma DD},\sigma}}(s \xrightarrow{op} d) = \begin{cases} \bullet\ \phi_{pm_{rule'_{\Sigma DD},\sigma \cup \langle x, op \rangle}}(d), \text{ if } \Sigma DD_l = s \xrightarrow{x} d', \text{ and } x \in Xs \\ \bullet\ \phi_{pm_{rule'_{\Sigma DD},\sigma}}(d), \text{ if } \Sigma DD_l = s \xrightarrow{f} d', \text{ and } f \in F\epsilon, s \\ \bullet\ \phi_{pm_{rule''_{\Sigma DD},\sigma}}(op) \circ \phi_{pm_{rule'_{\Sigma DD},\sigma}}(d), \text{ if } \Sigma DD_l = s \xrightarrow{op'} d' \\ \ \text{where } op' \in \mathbb{SIGDD} \text{ and } rule''_{\Sigma DD} = \langle op', 0, 0 \rangle \\ \bullet\ 0, \text{ otherwise} \end{cases}$$

$\sigma \cup \langle x, op \rangle$ stands for adding a pair $\langle variable, value \rangle$ to the current substitution σ.

If the inductive homomorphism gets to the terminal node (1), at least, one of the ΣDD fulfills the l pattern. In that case, the homomorphism has to check whether the conditions of application has also been fulfilled. If so, it returns both l and r. If r is empty (0), it means we are currently in a subterm matcher (not at the top level). In this case, $\phi_{pm_{rule_{\Sigma DD},\sigma}}$ simply returns 1 because the conditions can only be checked when the complete term has been scanned.

$$\phi_{pm_{rule_{\Sigma DD},\sigma}}(1) = \begin{cases} \bullet\ 1 \text{ if } \Sigma DD_r = 0 \\ \bullet\ l \xrightarrow{\phi_{subst,\sigma}(\Sigma DD_l)} r \xrightarrow{\phi_{subst,\sigma}(\Sigma DD_r)} 1 \\ \ \text{if } \phi^*_{InrMstRw}(\phi_{subst,\sigma}(\Sigma DD_{cond_l})) = \phi^*_{InrMstRw}(\phi_{subst,\sigma}(\Sigma DD_{cond_r})) \\ \bullet\ 0, \text{ otherwise} \end{cases}$$

The last homomorphism substitutes values to variables. Namely, it walks through the graph and each time it crosses a variable it tries to replace it. The extending substitution $\sigma^{\#}$ (see Def. 2.5) is embeded as a parameter.

Definition 4.17 [Substitution Homomorphism] Let $Spec = \langle \Sigma, X, E \rangle$ be an ordered algebraic specification and σ a substitution. The substitution homomorphism ϕ_{subst_σ} is defined by :

$$\phi_{subst_\sigma}(s \xrightarrow{op} d) = \begin{cases} s \xrightarrow{x/\sigma} \phi_{subst_\sigma}(d') \text{ if } x \in Xs \\ s \xrightarrow{f} \phi_{subst_\sigma}(d') \text{ if } f \in F \\ \phi_{subst_\sigma}(op) \circ \phi_{subst_\sigma}(d) \text{ if } op \in \mathbb{SIGDD} \\ 0, \text{ otherwise} \end{cases}$$

$$\phi_{subst_\sigma}(1) = 1$$

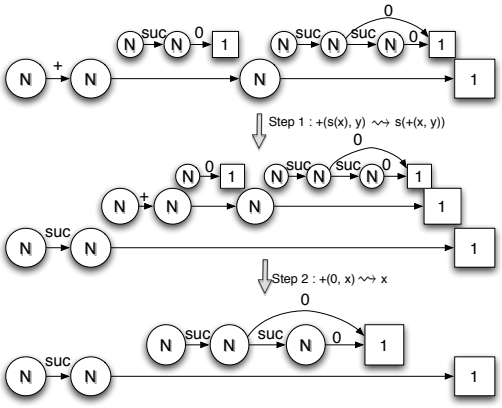


Fig. 4. ΣDD : 2 Rewriting steps

Fig. 4 shows how two terms encoded as a ΣDD are rewritten. Those terms : $+(suc(0), suc(0))$ and $+(suc(suc(0)), s(0))$ are rewritten using the following rewriting rules:

- +(suc(x), y) ⇝ suc(+(x, y))
- +(0, x) ⇝ x

Please note that in this example we rewrite two terms in a single rewriting sequence (2 rewriting steps) using the innermost strategy. For the sake of simplicity some part of the graph (i.e. $\mathbb{N} \xrightarrow{s} \mathbb{N} \xrightarrow{0} 1$) are drawn serveral times however they are shared in the implementation thanks to the canonicity.

Theorem 4.18 ($\phi_{InrMstRw}$ preserves the termination and confluence) *Under termination and confluence hypothesis of the rewrite system, we provide by rewriting on ΣDD values, a valid and complete calculus s.t : $\forall t, t' \in T_\Sigma$; $Rew^*(t) = Rew^*(t') \Leftrightarrow \phi^*_{InrMstRw}(en(t)) = \phi^*_{InrMstRw}(en(t'))$.*
Proof : By induction on the $\phi^*_{InrMstRw}$ homomorphism.

5 Implementation & Benchmarks

5.1 Implementation

The ΣDD library is implemented in Java and requires at least version 5 since it heavily uses the new language's features such as generics, variable arguments

size or boxing. The library is built on top of the *JDD* [13] library that provides support for the *Data Decision Diagrams* and the *Set Decision Diagrams* in Java. Both libraries are freely available under the GNU license at http://smv.unige.ch/tiki-list_file_gallery.php?galleryId=59. Even if Java provides an efficient object creation, performant garbage collection and a very good tooling, the performances of the implementation suffer from the lack of tail recursion. This issue may be solved in a future version of Java. The ΣDD library provides a user friendly API to describe an AADT, build terms and perform rewriting.

Both libraries (*JDD* & ΣDD) have been successfully used in our model checker called *AlPiNA* [11] that enables reachability analysis on *Algebraic Petri Net*.

5.2 Benchmarks

Although we have performed benchmarks[3] on our implementation, they are not exhaustive and must be improved. We present here some results to compare ΣDD's performances to the well-known rewriter *Maude* [14]. We used a new feature of the version 2.4 namely Built-in support for sets. The following figures are given as an indication in order to illustrate our approach. We will perform more extensive (Maude and ATerm) benchmarks in the near future. The benchmarks are based on the specification of naturals because it is well known and easy to understand.

The first benchmark rewrites a term and indicates that Maude is 35 times faster that ΣDD when rewriting a single term with poor or no sub-term sharing. In this case, both rewriters used the same number of steps (10002). This indicates that the cost of one rewriting step is 35 times higher using ΣDD. One of the reasons is the management of sets even in the case of single term rewriting. The second one[4] illustrates the impact of the cache in a single term rewriting, thanks to this, the number of rewritings is much smaller in the ΣDD case. The third example uses a conditional rewriting rule : $-(suc(x), suc(y)) = -(x, y)$ if $y < x$. Thanks to the cache, ΣDD is better (10x) than Maude due to the inductive definition of this axioms, and thus the high caching. The fourth example proves a property on a set of terms, namely that for all i s.t $0 \leq i \leq 80$, $\Sigma_{x=0}^{i} x \leq i^2$. Sharing and caching help to reduce the difference (2x). When performing model checking on *Algebraic Petri Nets*, we must often rewrite sets of very similar terms. This is a very favorable case for ΣDD because each rewriting step is applied to the whole set. In the last case, ΣDD is 42x faster than Maude.

Terms			ΣDD		Maude	
	$Rew_{M/\Sigma}$	$T_{M/\Sigma}$	$\#Rew_{\Sigma}$	Time(s)	$\#Rew_{M}$	Time(s)
$(1000 + 5000) + 5000$	1	0.028	10002	0.53	10002	0.015
Single term (sub-term sharing)	6.99	0.031	5513	0.54	38521	0.017
$(1000 + 5000) - 5000$ (conditions)	834	10.74	15002	4.66	12512502	50.05
$\forall i$ s.t $0 \leq i \leq 80$ $\Sigma_{x=0}^{i} x \leq i^2$	39.54	0.47	177365	26.29	7012853	12.40
$\cup_{i=1}^{100}(((1000 + i) + 5000) + 5000)$	100	4.33	10002	0.40	1000200	1.73
$\cup_{i=1}^{1000}(((1000 + i) + 5000) + 5000)$	1000	41.67	10002	0.43	10002000	17.92

ΣDD performs most of the time less rewriting steps thanks to sharing and

[3] Performed on a Mac Book Pro with 1 Intel Core 2 Duo at 2.5 Ghz and 2GB of RAM.

[4] $(5000 + 5000) + (2000 + ((500 + (500 + 500)) + 5))$

caching, but the cost of single rewriting is much higher. Our implementation is far from having the maturity of Maude and thus we are confident that optimizing the cost of a single rewriting will bring much better performances. The other important point is that for practical reasons we used the so called "iter theory" in the Maude examples: $suc(suc(suc(0))) = suc^3(0)$, this is not implemented so far in ΣDD and will improve both the memory footprint and processing time. The benchmarks, our implementation, as well as the specifications for Maude can be found under http://smv.unige.ch/tiki-list_file_gallery.php?galleryId=59.

As mentioned before, we primarily developed this technology to tackle the state space explosion problem that occurs when performing model checking on an Algebraic Petri Net model. We use the Decision Diagrams framework to represent the states in a symbolic way. In an Algebraic Petri Net, a state is represented as a vector of places that contain multisets of terms. Since we use DD for representing set of vector of places and multisets, we have naturally extended them to support Terms. This helps to share the common subterms between the states and thus common rewriting steps. Using this technology, we are able to handle much bigger models than the competition (300 philosophers for AlPiNA vs. 15 for Maria [15]) as detailed in [11]. The performances (both memory footprint and processing time) are also much better.

6 RelatedWork

Several approaches leverage either on graph rewriting or caching like the ATerm library [16] to provide efficient term encoding and term rewriting. However, the novelty of our approach is that the structure is optimized not only for large terms but also for large sets of large terms. Since our goal is to primarily encode large sets of states containing multisets of terms, we didn't use ATerm, in order to maximize the sharing between the states by encoding everything in a DD-like structure. Like ATerm, this approach does not suffer from the "rewrite in place" approach of Term Graph Rewriting. Namely, some examples of non-confluent term rewriting in GTR [17] are confluent with this approach without loosing the sharing.

7 Conclusion and Future Work

We have presented the ΣDD which is a powerful data structure inspired by the Decision Diagrams to encode and manipulate set of terms. The novelty of this approach resides in the strong leverage of the shared parts in set of terms to optimize both the memory footprint and the computation time. Although not as efficient as other approaches on a per rewriting basis, the huge sharing induced much less rewriting step to get to a normal form when working on large sets. We plan to extend our work in the following ways:

- Support for axioms between generators.
- Optimize rewriting step.
- Extensive benchmarking with Maude/ATerm.

- Axioms between generators and Associative/Commutative rewriting.
- Add support for the so-called iter theory, namely a compressed representation of stacked operators : $suc(suc(0)) = suc^2(0)$.

We successfully used the ΣDD in our model checker called $AlPiNA$, which is, along with the others libraries, freely available on http://smv.unige.ch/tiki-list_file_gallery.php?galleryId=59

8 Acknowledgments

The authors wish to express special thanks to the members of the MoVe laboratory of University Paris VI for their valuable help and suggestions, especially to Yann Thierry-Mieg and Alban Linard. We are also grateful to the anonymous referees for their useful comments and to the Hasler Foundation for supporting this work.

References

[1] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[2] Wolfgang Reisig. Petri nets and algebraic specifications. In *Theoretical Computer Science*, volume 80, pages 1–34. Elsevier, 1991.

[3] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. CO-OPN/2: A concurrent object-oriented formalism. In *Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS), Canterbury, UK, July 21-23 1997*, pages 57–72. Chapman and Hall, Lo, 1997.

[4] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–94, New York, NY, USA, 1990. ACM.

[5] J.M. Couvreur and Y. Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In *FORTE*, pages 443–457, 2005.

[6] J.-M. Couvreur, E.Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P. Wacrenier. Data decision diagram for petri nets analysis. In *23rd international conference on application and theory of Petri Nets (ATPN 2002), jun 2002, Australia.*, volume LNCS vol 2360, 2002.

[7] R. Bryant. Graph-based algorithms for boolean function manipulation. In *Transactions on Computers, C-35*, pages 677–691. IEEE, 1986.

[8] Enno Ohlebusch. *Advanced topics in term rewriting*. Springer-Verlag, London, UK, 2002.

[9] A. J. J. Dick and P. Watson. Order-sorted term rewriting. *Comput. J.*, 34(1):16–19, 1991.

[10] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1 : Equations and Initial Semantics*, volume 6 of *EATC Monographs*. Springer-Verlag, 1985.

[11] Didier Buchs and Steve Hostettler. Toward efficient state space generation of algebraic petri net. Technical report, CUI, Université de Genève, January 2009. Available as http://smv.unige.ch/tiki-download_file.php?fileId=1151.

[12] Yann Thierry-Mieg. *Techniques for Model-Checking of high-level specifications.* PhD thesis, University of Paris Marie Curie, 2004.

[13] Steve Hostettler. Java decisions diagrams library. Technical report, CUI, Université de Genève, June 2008. Available as http://smv.unige.ch/tiki-download_file.php?fileId=1003.

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[15] Modular reachability analyzer. http://www.tcs.hut.fi/Software/maria/.

[16] M. G. T. Van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30(3):259–291, 2000.

[17] D. Plump. Term graph rewriting. *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*, pages 3–61, 1999.

Recursive Functions with Pattern Matching in Interaction Nets

Maribel Fernández Ian Mackie Shinya Sato Matthew Walker

King's College London, Department of Computer Science, Strand, London WC2R 2LS, U.K.

LIX, École Polytechnique, 91128 Palaiseau Cedex, France

Faculty of Econoinformatics, Himeji Dokkyo University, 5-7-1 Kamiohno, Himeji-shi, Hyogo 670-8524, Japan

Abstract

We compile functional languages with pattern-matching features into interaction nets, extending the well-known efficient evaluation strategies developed for the pure λ-calculus. We give direct translations of recursion and pattern matching for languages with a strict matching semantics, implementing an evaluation strategy that is natural in interaction nets and has a high degree of sharing.

Keywords: pattern matching, recursion, interaction nets

1 Introduction

Evaluation strategies and compilation schemes for the λ-calculus are well studied. In particular, several interaction net evaluators are now available, including versions that implement optimal reduction [11,2] and other efficient evaluation strategies [17,18].

Interaction nets [14] are graph rewrite systems in which *all* the computation steps are explicit and expressed in the same formalism (there is no external machinery). This facilitates the analysis of cost of computation and the comparison between different evaluation strategies implemented as interaction nets. Also, since reduction in interaction nets is local and strongly confluent, reductions can take place in any order, even in parallel (see [21]), which makes this formalism well-suited for the implementation of programming languages and rewriting systems [8,7].

In this paper, we describe an interaction net compiler for a small functional language that can be seen as an extension of the λ-calculus with data constructors, a case construct to define functions by pattern matching on constructors, and a fixpoint operator to define recursive functions.

Traditionally, the λ-calculus is considered to be the abstract computation model underlying the functional programming paradigm, and graph-based implementa-

Layout based on the macro package of the Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

tions or environment machines are used to describe evaluation strategies (see for instance [23]) and to derive efficient interpreters or compilers. However, the λ-calculus does not provide direct support for important features of modern functional programming languages, such as pattern matching. Pattern calculi [20,19,3,5,6,12] have been put forward as a semantic model for functional programming languages with pattern matching. The rewriting calculus (or ρ-calculus) introduced by Cirstea and Kirchner [5] provides support not only for pattern matching as found in modern functional languages, but also for features such as non-determinism, advanced matching theories, object-orientation and imperative traits. Recently, interaction net evaluators for the rewriting calculus have been developed [10], which provide direct compilations of pattern matching. The advantage of a direct compilation of pattern-matching (over pre-processing, which would translate pattern-matching definitions to pure λ-terms) is that we obtain new, more efficient strategies of reduction. In particular, the direct translation of ρ-calculus pattern matching into interaction nets brings to light the implicit parallelism that exists in this calculus. The same technique was used in [4] to derive a compilation scheme for case constructs. In this paper, we refine the technique and provide also a direct encoding for recursion.

Together with pattern-matching, recursion is an essential feature in functional programming. It is widely acknowledged that a direct translation of recursion is better in practice than translating a recursive definition in terms of fixpoint combinators in the pure λ-calculus (see, for instance, [20]). We provide a new compilation scheme for recursive definitions, which is based on the use of recursion agents instead of the standard compilation based on cyclic graphs [20].

To define an interaction net compilation of a functional programming language with pattern matching, in this paper we extend [18], which is one of the most efficient interaction net λ-evaluators currently available. The extension is modular. It is inspired by the interaction net implementation of matching in the ρ-calculus, combined with a new technique to deal with recursive definitions.

Summarising, the main contributions of this paper are:

- a new implementation technique for recursive functions using interaction nets;
- a modular compilation scheme for pattern matching;
- the smooth integration of these techniques, extending the λ-evaluator defined in [18].

The compiler has been implemented in Java (see [26]), and is available from `http://www.dcs.kcl.ac.uk/pg/walkerm`.

This paper is organised as follows: after recalling the main notions of interaction nets (Section 2), in Section 3 we define a minimalistic functional language with a case construct and recursion. The compilation into interaction nets is given in Section 4. Finally, we conclude in Section 5.

2 Background: Interaction nets

We recall the main notions from interaction nets that will be needed in the rest of the paper; for more details and examples we refer to [14].

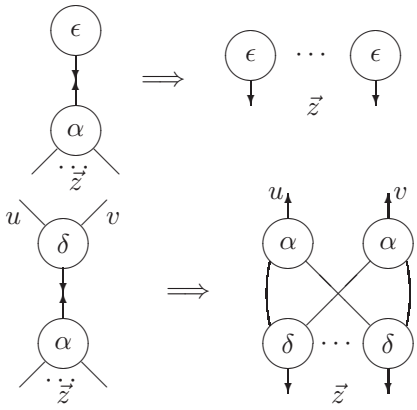
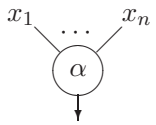


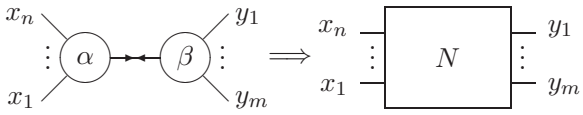
Fig. 1. Erasing and Copying

A system of interaction nets is specified by a set Σ of symbols with fixed arities, and a set $\mathcal{R}$ of interaction rules. An occurrence of a symbol $\alpha \in \Sigma$ is called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a *principal port* depicted by an arrow, and n *auxiliary ports*. Such an agent will be drawn in the following way:



Intuitively, a net N is a graph (not necessarily connected) with agents at the vertices and each edge connecting at most two ports. The ports that are not connected are *free*. There are two special instances of a net: a wiring (no agents) and the empty net; the extremes of wirings are also called free ports. The interface of a net is its set of free ports.

An interaction rule $((\alpha, \beta) \Longrightarrow N) \in \mathcal{R}$ replaces a pair of agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected together on their principal ports (an *active pair* or *redex*) by a net N with the same interface. Reduction is local, and there may be at most one rule for each pair of agents. The following diagram shows the format of interaction rules (N can be any net built from Σ).



We show as an example the interaction rules for ϵ (the *erasing* agent), of arity 0, which deletes everything it interacts with, and δ, the *duplicator*, of arity 2, which copies everything. These are given in Figure 1, where α is any node.

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 it is in normal form. The key property of interaction nets, besides locality of reduction, is strong confluence.

There are several implementations of interaction nets; e.g., [15,22], the latter can take advantage of additional processors, giving a parallel implementation.

3 A simple functional language

We consider a simple functional language with terms built from variables $x, y, \ldots$, functional abstraction, application, data constructors C (each with a fixed arity), a case construct to define functions by pattern matching on constructors, and a fixpoint operator to define recursive functions. We abbreviate $t_1, \ldots, t_n$ as $\vec{t}$. Patterns are defined by the following grammar:

$$p ::= x \mid C(\vec{p})$$

with the usual linearity constraint (each variable may occur at most once in a pattern). The syntax of terms is given by the grammar:

$$t, u \ ::= \ x \mid \texttt{fn}\ x.t \mid t\ u \mid C(\vec{t}) \mid \ \texttt{case}\ t\ \texttt{of}\ (p_i \rightsquigarrow u_i)_{i \in I} \mid \ \texttt{fix}\ f.t$$

In the syntax above, `fn`, `case`, and `fix` are binders. In the case of $\texttt{fn}\ x.t$, the variable x is bound in t, whereas in $\texttt{fix}\ f.t$, the variable f is bound. In a `case` construct, a branch of the form $(p_i \rightsquigarrow \cdot)$ acts as a binder: $\mathsf{fv}(p_i \rightsquigarrow u_i) = \mathsf{fv}(u_i) \backslash \mathsf{fv}(p_i)$ where $\mathsf{fv}(u_i)$ denotes the set of free variables of u_i. Terms are defined modulo α-equivalence, as usual.

We assume the language is typed. For simplicity, we consider a simply-typed system where each constructor is associated to a datatype. We will base this discussion on the following form of a datatype declaration, which introduces a datatype DT with constructors $C_1, \ldots, C_n$, taking arguments of types $\vec{\alpha_i}$.

$$DT = C_1(\vec{\alpha_1}) \mid \cdots \mid C_n(\vec{\alpha_n})$$

Example 3.1 We will use the following datatypes for numbers and lists with elements of type α, respectively:

$$Int = Z \mid S(Int)$$
$$List\ \alpha = Nil \mid Cons(\alpha, List\ \alpha)$$

As usual, the type system ensures that in a case construct $\texttt{case}\ t\ \texttt{of}\ (p_i \rightsquigarrow u_i)_{i \in I}$ all the branches have the same type and t has the same type as the patterns p_i (for all $i \in I$), that is, some datatype DT. We do not assume that the cases are exhaustive, but we do assume they are non-overlapping; i.e., at most one pattern can match a term at a given position [1]. We omit the typing rules, which are standard.

The following reduction rules give the dynamics of the language. Reduction is denoted by $\rightarrow_f$ or simply $\rightarrow$. The first rule corresponds to the familiar β rule of the λ-calculus, where $\{x := u\}$ denotes the usual capture avoiding notion of substitution of x by u, the second rule deals with case constructs, and the last one is used to evaluate recursive functions via fixpoint operators, as in PCF [24].

[1] This restriction can be easily overcome by specifying, for instance, a priority on the selection of branches in a case.

$$(\texttt{fn}\ x.t)\ u \to t\{x := u\}$$
$$\texttt{case}\ t\ \texttt{of}\ (p_i \rightsquigarrow u_i)_{i\in I} \to u_k\,\sigma \qquad (\text{if } t \text{ matches } p_k \text{ with substitution } \sigma)$$
$$(\texttt{fix}\ f.t)\ u \to t\{f := \texttt{fix}\ f.t\}\ u$$

We will not impose a strategy of evaluation yet, but note that since the rewrite rules are left-linear and non-overlapping (that is, they define an orthogonal system [13]), the language is confluent. It is easy to see that it is not terminating, due to the presence of recursion. We assume a strict matching semantics, as in ML (i.e., an application of a function to an argument that is not covered by the case definition will produce a runtime error).

Programs in this language are well-typed, closed terms (i.e., terms with no free variables). We give now some simple examples.

Example 3.2 (i) Assuming that Nil with arity 0, and $Cons$ with arity 2, are used to define the datatype $List$ as in Example 3.1, and that $True$ and $False$ are the boolean constants, we can define the boolean function `null` by pattern matching as follows:
$\texttt{null} \triangleq \texttt{fn}\ l.\texttt{case}\ l\ \texttt{of}\ (Nil \rightsquigarrow True, Cons(x,y) \rightsquigarrow False)$

(ii) Assuming that Z with arity 0, and S with arity 1 are used to define the datatype Int as in Example 3.1, the recursive function `length` can be defined by pattern matching as follows:
$\texttt{length} \triangleq \texttt{fix}\ len.\texttt{fn}\ l.\texttt{case}\ l\ \texttt{of}\ (Nil \rightsquigarrow Z, Cons(x,y) \rightsquigarrow S(len\ y))$

Notice that we have not included a conditional in the syntax of the language, but it can be easily encoded with a `case`. Also, we do not have named functions and `letrec` but these can be easily encoded using `fix`:

$$\texttt{let}\ \ x = t\ \ \texttt{in}\ \ u \triangleq (\texttt{fn}\ x.u)t$$
$$\texttt{letrec}\ \ f = t\ \ \texttt{in}\ \ u \triangleq \texttt{let}\ \ f = \texttt{fix}\ f.t\ \ \texttt{in}\ \ u$$

We can also define mutually recursive definitions by an encoding as follows:

$$\begin{array}{l}\texttt{letrec}\ \ f = u\ \ \texttt{and}\ \ g = v\ \ \texttt{in}\ \ w \triangleq \\ \qquad \texttt{letrec}\ \ h = \texttt{fn}\ g.(\texttt{let}\ \ f = h\ g\ \ \texttt{in}\ \ u)\ \ \texttt{in} \\ \qquad\qquad \texttt{letrec}\ \ g = (\texttt{let}\ \ f = h\ g\ \ \texttt{in}\ \ v)\ \ \texttt{in} \\ \qquad\qquad\qquad \texttt{let}\ \ f = h\ g\ \ \texttt{in}\ \ w\end{array}$$

In the remainder of the paper we define the compilation of the functional language into interaction nets.

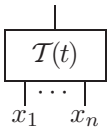


Fig. 2. Translation of a term t with $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$.

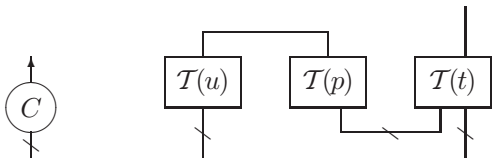


Fig. 3. Translation of constants (left) and matching constraints (right).

4 Implementing the language via interaction nets

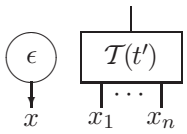
In this section, we describe the encoding of programs in the simple functional language into interaction nets and give the interaction rules that will be used to evaluate them. For functional abstraction and application, we use the encoding of [18] but any other interaction net λ-evaluator could be used. The rewriting calculus (or ρ-calculus) introduced by Cirstea and Kirchner [5] motivates the use of the case construct as it permits abstraction on patterns as well as variables. The encoding of matching is inspired by the ρ-calculus encoding described in [10].

A term with free variables $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$ will be translated to a net $\mathcal{T}(t)$ with the root edge at the top, and n free edges corresponding to the free variables, as shown in Figure 2. We now define by induction the function $\mathcal{T}(\cdot)$.

Variable: If t is a variable then $\mathcal{T}(t)$ is just a wire.

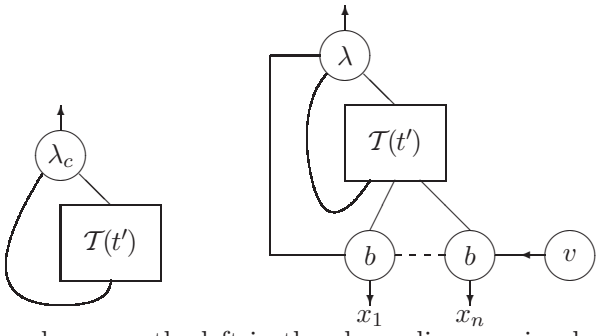
Constructor: For each constructor C we introduce an agent as shown in Figure 3 (left) [2] with the arity of the constructor matching the arity of the agent.

Abstraction: As mentioned above, we use the encoding of abstraction in the λ-calculus from [18]. If t is an abstraction, say $\texttt{fn}\ x.t'$, then we first require that $x \in \mathsf{fv}(t')$. If this condition is not satisfied, then we can add the following agent to the translation of the body:



Having assured this condition, there are two alternative translations of the abstraction, which are both given in the following diagram:

[2] A dashed edge represents a bunch of edges (a *bus*).

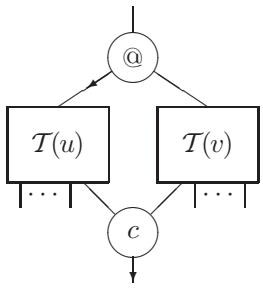


The first case, shown on the left in the above diagram, is when $\mathsf{fv}(\lambda x.t') = \emptyset$. Here we use one agent λ_c to represent a *closed abstraction*. Note that we explicitly connect the occurrence of the variable to the binding λ.

The second case, shown on the right, is when $\mathsf{fv}(\lambda x.t') = \{x_1, \ldots, x_n\}$. Here we introduce three different kinds of agent: λ of arity 3, for abstraction, and two kinds of agent representing a list of free variables. An agent b is used for each free variable, and we end the list with an agent v. The idea is that there is a pointer to the free variables of an abstraction; the body of the abstraction is encapsulated in a box structure. Multiple occurrences of the same variable in $\mathcal{T}(t')$ are grouped using c (contraction) agents (see the encoding of application below). We assume that the (unique) occurrence of the variable x is in the leftmost position of $\mathcal{T}(t')$.

We remark that a closed term will never become open during reduction (although of course open terms may become closed, and indeed there are interaction rules which will create a λ_c agent from a λ agent when needed). The use of the λ_c agent identifies the case where there are no free variables, and plays a crucial role in the efficient dynamics of this system.

Application: To encode uv, we introduce an agent @ with its principal port oriented towards the left subterm so that interaction with an abstraction is possible. If a variable occurs in both u and v, we group both occurrences with a contraction agent (c).



We postpone discussion of case structures and recursion until the end of this section.

4.1 Implementing term reduction

We define an interaction rule between abstraction and application as in the λ-calculus, as well as rules dealing with the bookkeeping related to box structures. A summary is given in Figure 4; we refer the reader to [18] for more details.

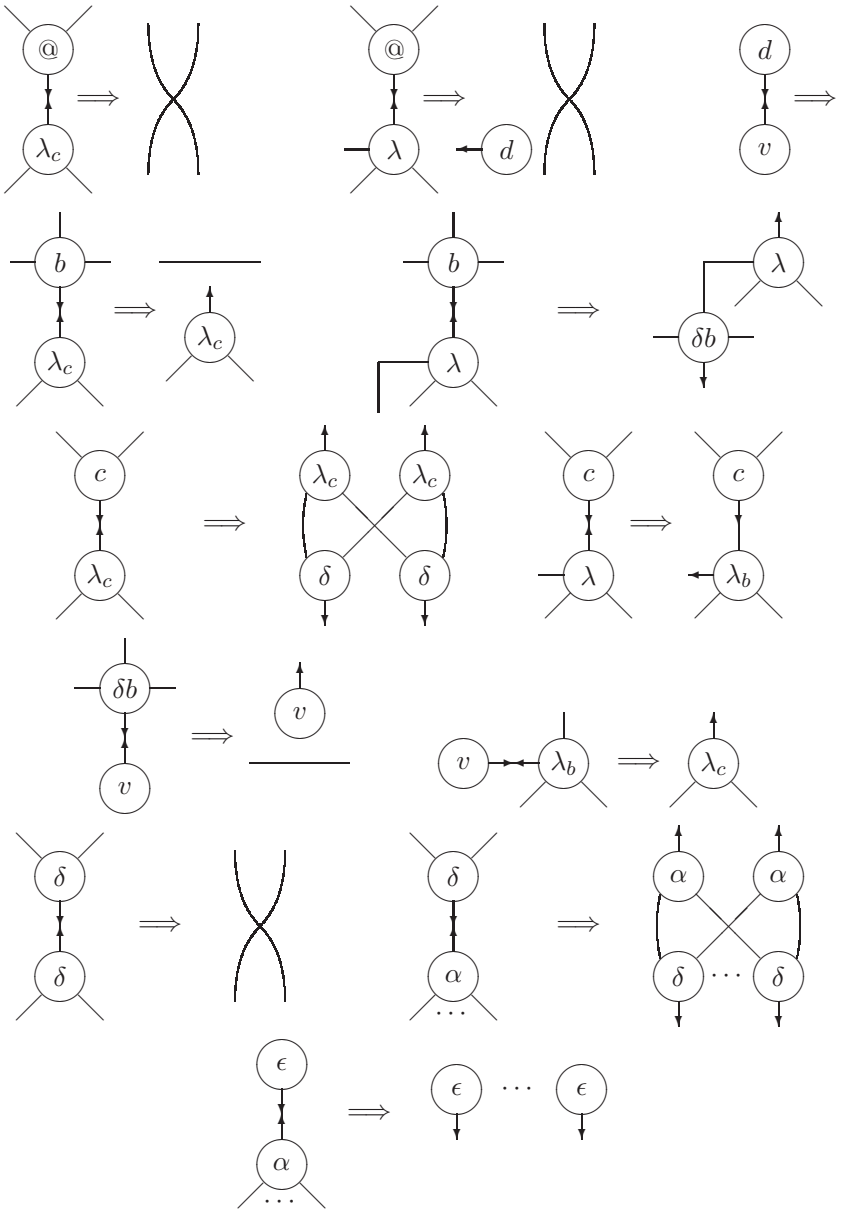


Fig. 4. Lambda calculus rules

4.2 Pattern Matching

The matching rules are inspired by the "simple" encoding of [10]. Assume we have just one matching constraint to solve; i.e., given a pattern p and a term t, we need to find a substitution σ such that $p\sigma = t$, if there is one (the generalisation to case structures with multiple branches will be given below). The matching algorithm is initiated by connecting the root of the pattern p with the term t (see Figure 3, right). Thus, matching against a variable is realised for free, as in the λ-calculus. Two identical constants cancel each other and the matching continues in the arguments

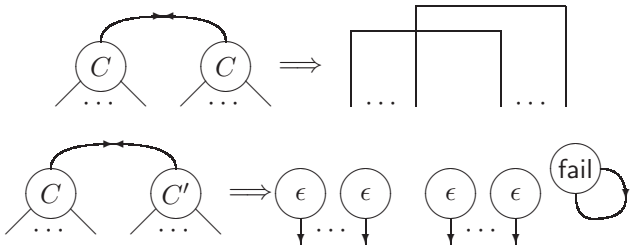


Fig. 5. Matching of constructors (success and failure where C and C' are distinct)

(or results in the empty net if the constant has arity zero), as indicated in Figure 5 (upper). If the agents are not the same, then we introduce an agent fail, which represents a failure in the matching algorithm, as indicated in Figure 5 (lower). We interpret a net containing an agent fail as an overall failure, thus implementing the strict matching semantics. We do not need interaction rules for a constructor and an abstraction because the language is typed.

We refer to [10] for a detailed description and correctness proofs for matching constraints. In particular, in [10] it is shown that with this encoding we can only implement a strict matching semantics, but, on the positive side, it allows us to obtain a strategy of evaluation with a good potential for parallelism. This is because matching interactions can take place in parallel with traditional β reductions, without introducing any 'administrative' agents (i.e., no overheads). We use this feature in the encoding of case structures below, to derive an evaluation strategy with the same potential for parallelism.

4.3 Case structures

We now describe the encoding of case structures

$$\texttt{case}\ t\ \texttt{of}\ (p_1 \rightsquigarrow u_1, \ldots, p_n \rightsquigarrow u_n)$$

and the respective reduction rules. This is one of the main contributions of this paper. Our goal is to avoid making multiple copies of t and to permit matching to proceed in parallel with functional computation, whenever possible. For these reasons, for each case structure occurring in a program we will introduce a bespoke `case` agent as explained below (see Figure 6), where we build a net that minimises the number of selections necessary (this differs from [4]).

The role of a `case` agent is to determine which of the patterns p_i should be chosen to commence pattern matching with t. The top auxiliary port of `case` represents the output. When $\mathcal{T}(t)$ and `case` interact the former is connected to the appropriate pattern using a collection of rules determined during compilation. The output is rewired to the output of the corresponding u_i. The diagram in Figure 6 depicts the case where the top-level constructor of $\mathcal{T}(t)$ matches that of p_n and all other branches of the structure are garbage collected with the use of ϵ-agents (note that this only accounts for patterns with different root symbols; patterns with the same top-level constructors are dealt with later).

Two further modifications are needed to the encoding of structures: Firstly the free variables of all u_i need to be 'boxed'. A chain of $\texttt{cb}_n$ agents is introduced,

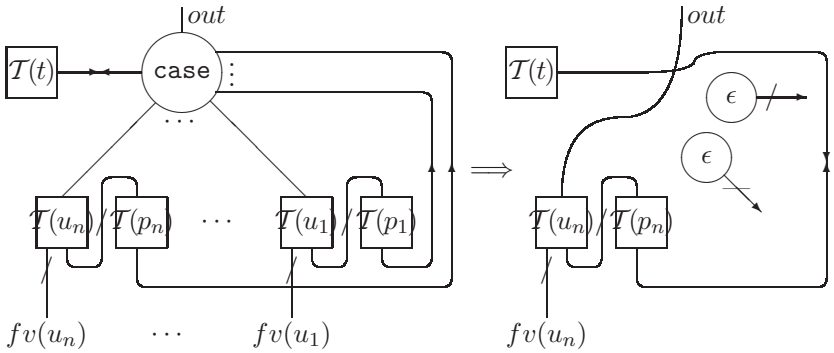


Fig. 6. A reduction where `case` allows matching to continue with p_n.

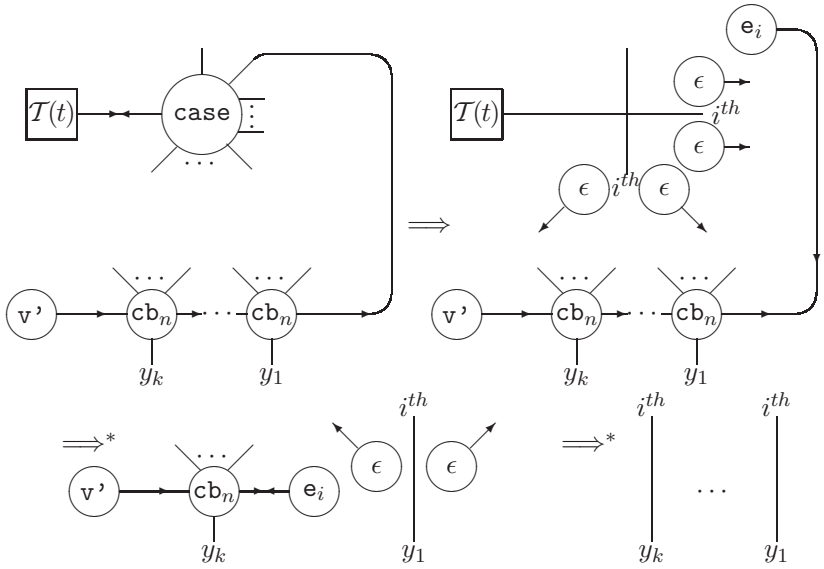


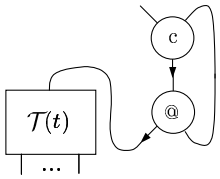
Fig. 7. Chaining of free variables within the structure with $k = max\{|\bigcup_i fv(u_i)|\}$. Note that if $y_m \notin fv(u_i)$ then it is connected to an ϵ-agent.

terminated at one end by a `v'` agent and at the other by an additional auxiliary port of `case`. Figure 7 depicts a simplified reduction sequence for `case` allowing pattern matching with $\mathcal{T}(t)$ by p_i; the agent $\mathtt{e}_i$ traverses the chain linking every free variable, y_1 to y_k, with u_i, and garbage collects everywhere else:

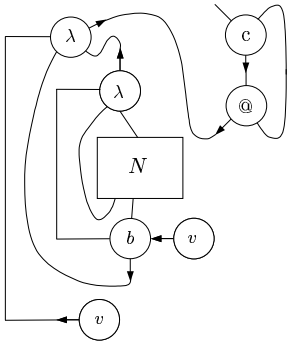
Finally, the encoding of structures should take into account possible patterns with a common prefix. In this instance a net of the mutual pattern interacts with $\mathcal{T}(t)$ until the unique constructor identifying a branch is isolated, interaction with the `case`-agent then proceeds as normal. For a mutual pattern with constructors of binary or greater arity the `case`-agent will have to anchor the variables to be linked with the patterns in the case structure. Additionally all patterns $\mathcal{T}(p_i)$ in the structure will be compiled to $\mathcal{T}(p_i) - \mathcal{T}(t_{mp})$ where t_{mp} is the common prefix.

As an example, we give the compilation of the function `length` after describing the encoding of recursion in the next section.

(1) Translation of Yt:



(2) Translation of the recursive function `length`:



where N is the net $\mathcal{T}(\texttt{case}\ l\ \texttt{of}\ (Nil \rightsquigarrow Z, Cons(x,y) \rightsquigarrow S(len\ y)))$.

Fig. 8. Recursion using cycles

Proposition 4.1 *If t matches p_i with substitution σ then*

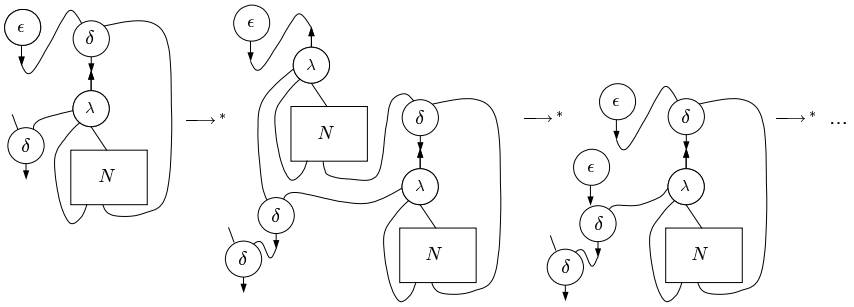
$$\mathcal{T}(\texttt{case}\ t\ \texttt{of}\ (p_1 \rightsquigarrow u_1, \ldots, p_n \rightsquigarrow u_n)) \Longrightarrow^* \mathcal{T}(u_i\sigma)$$

Proof. The interaction rules for the case agent corresponding to this particular case construct ensure that the principal port at the root of the net $\mathcal{T}(t)$ gets connected to the principal port at the root of $\mathcal{T}(p_i)$, as depicted in Figure 6. Since the matching algorithm we are using is correct [10], the interactions in this subnet will generate the matching substitution $\mathcal{T}(\sigma)$. Finally, the interactions between the boxing agents $\texttt{cb}_n$ and $\texttt{e}_i$ connect the $\mathcal{T}(\sigma)$ to $\mathcal{T}(u_i)$. □

4.4 Recursion

There is a standard way to encode recursion in interaction nets for the λ-calculus, which consists of building a cyclic structure which explicitly "ties the knot". The idea corresponds exactly to an encoding of recursion in graph reduction [20] and was adapted to interaction nets in [16]. For example, the translation of Yt where t is a λ-term t and Y is a fixpoint combinator, is the net $\mathcal{T}(Yt)$ shown in Figure 8(1). According to this translation, the recursive function `length` can be compiled as shown in Figure 8(2).

Note that, with this encoding, the reduction of a recursive function generates an infinite reduction sequence, even if the function terminates. Generally speaking, recursive functions consist of a base part and an induction part which should be discarded when the base case is reached (in the case of `length`, the part of $Cons(x,y) \rightsquigarrow S(len\ y)$ has to be eliminated when l is Nil). However, with interaction nets, non-terminating nets cannot be erased, so in the case of the function `length` we are left with an infinite reduction sequence:

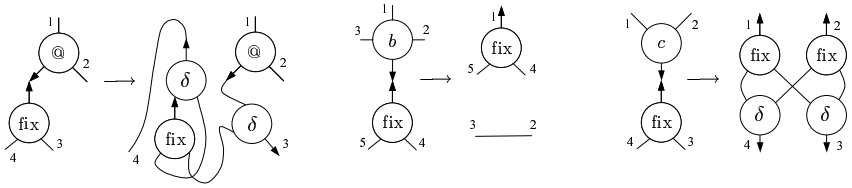


The standard solution to this problem relies on the introduction of a reduction strategy, called *connected reduction* or reduction to interface normal form (INF for short, see [9]), which restricts reductions to active pairs connected to the interface of the net (in this way, non-terminating reduction sequences on disconnected nets are prevented). Another solution is described in [1] using a token-passing style of compilation, where an evaluation token controls the creation of active pairs.

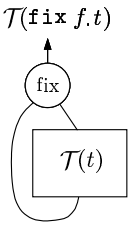
Neither of these solutions is modular; they impose restrictions on the λ-evaluator that would not be necessary otherwise. More precisely, a global strategy such as reduction to INF cannot be imposed just on the translation of recursion, and similarly, it is not possible to use a token-passing style just for recursion. In this paper, we propose a compilation of recursion which is inspired by [25] where two agents are used to control the creation of copies of recursive functions. This encoding uses neither cyclic nets nor global reduction strategies such as INF, and works in the traditional interaction net style (that is, each β-redex in the program is compiled as an active pair in the net, unlike the token-passing translation, and all active pairs present in the net can be reduced in any order, even in parallel).

First, recall that recursive functions in the functional language are defined using the syntax $\texttt{fix}\, f.t$, where we can assume $\texttt{fv}(t) = \{f\}$ in the case of programs (i.e., closed terms). We have the following reduction: $(\texttt{fix}\, f.t)u \to^* (t\{f := \texttt{fix}\, f.t\})u$, which we implement by introducing the following binary agent `fix`:

and the following interaction rules:

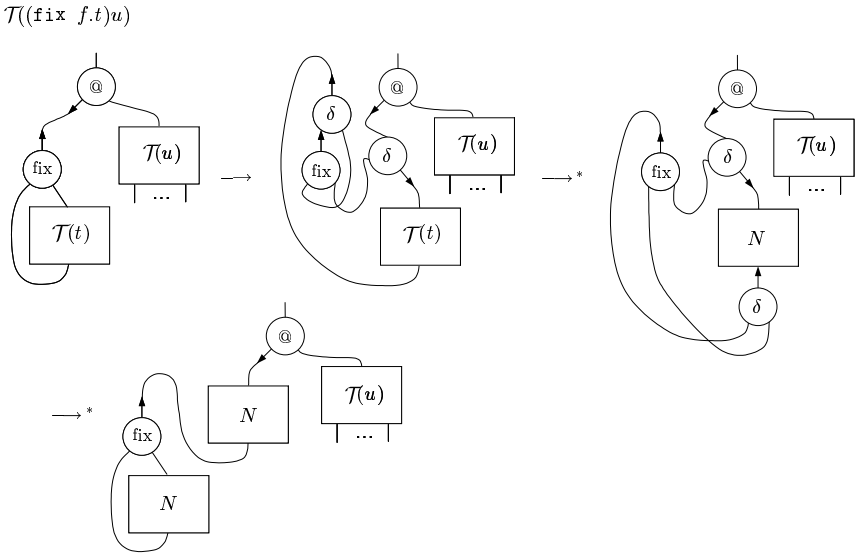


The translation of $\texttt{fix}\, f.t$, $\mathcal{T}(\texttt{fix}\, f.t)$, is shown below.



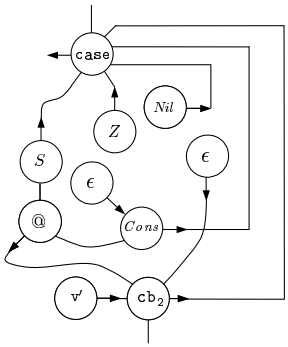
Proposition 4.2 *If $\mathcal{T}(t)$ has a normal form that contains no cycles of principal ports, then $\mathcal{T}((\texttt{fix}\ f.t)u)$ and $\mathcal{T}((t\{f := \texttt{fix}\ f.t\})u)$ have a common reduct.*

Proof. We assume $\mathcal{T}(t)$ has a normal form N which contains no cycles of principal ports. Then, from $\mathcal{T}((\texttt{fix}\ f.t)u)$ we can perform the following reduction:



The resulting net can be obtained from $\mathcal{T}((t\{f := \texttt{fix}\ f.t\})u)$ by reducing $\mathcal{T}(t)$ to a normal form N. □

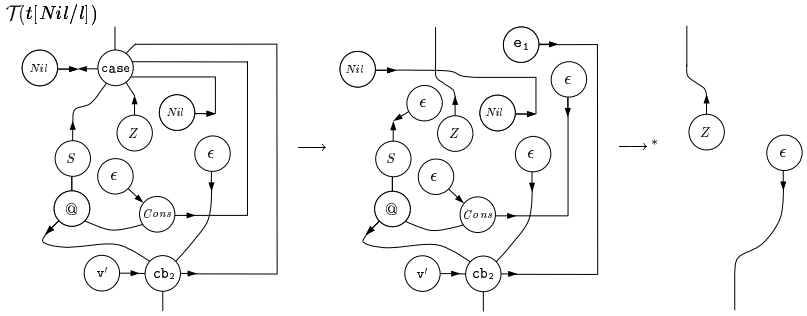
As an example, below we show the compilation of the recursive function `length` ($\triangleq$ `fix` len.`fn` l.`case` l `of` $(Nil \rightsquigarrow Z, Cons(x,y) \rightsquigarrow S(len\ y))$) given in the Example 3.2. Let $t =$ `case` l `of` $(Nil \rightsquigarrow Z, Cons(x,y) \rightsquigarrow S(len\ y))$, then $\mathcal{T}(t)$ is:



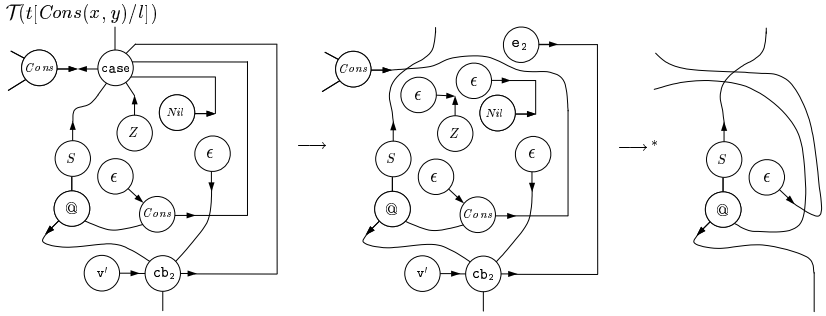
For $\mathcal{T}(t\{l := Nil\})$ and $\mathcal{T}(t\{l := Cons(x,y)\})$, we can perform the following reduc-

tions:

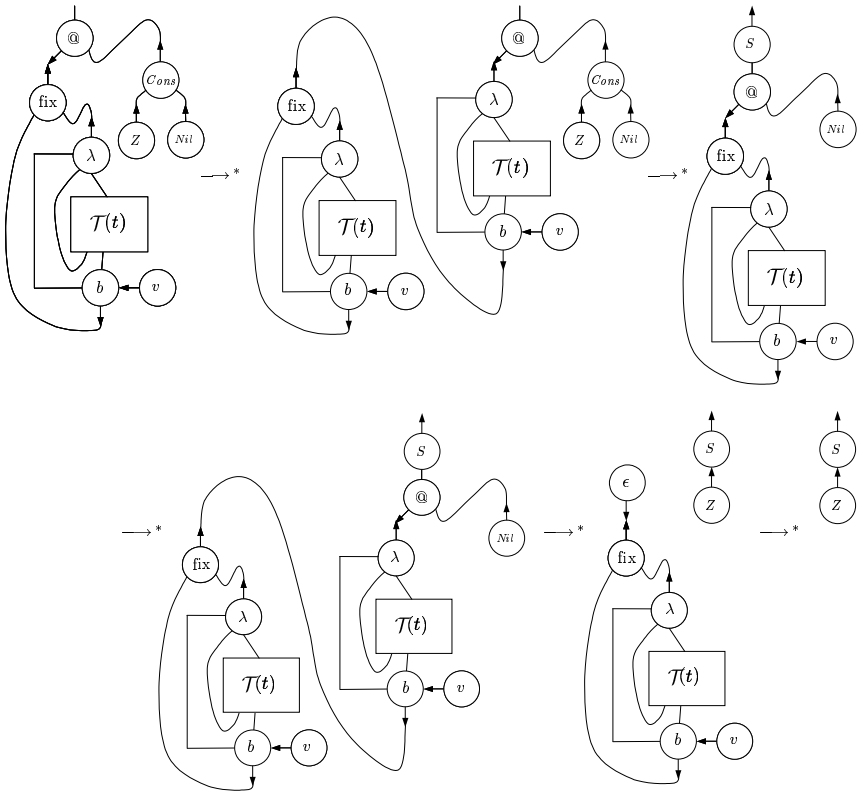
$\mathcal{T}(t[Nil/l])$



$\mathcal{T}(t[Cons(x,y)/l])$



We get the result of `length` $Cons(Z, Nil)$ as follows:



5 Conclusion

This paper shows how to extend interaction net λ-evaluators to richer rewriting formalisms, such as the rewriting calculus and simple functional programming languages. The next step is to investigate the use of non-strict matching semantics, and to compare with other implementations. For non-strict matching, we forsee the use of linking agents as in the compilation of the non-strict ρ-calculus presented in [10]. Bigraphical nets, which generalise interaction nets by defining a location graph in addition to the usual linking graph, might offer a better framework for the compilation of languages with non-strict matching.

References

[1] J. B. Almeida, J. S. Pinto, and M. Vilaça. Token-passing nets for functional languages. *Electr. Notes Theor. Comput. Sci.*, 204:181–198, 2008.

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

[3] V. Breazu-Tannen, D. Kesner, and L. Puel. A typed pattern calculus. In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science (LICS'93), Montreal, Canada*, 1993.

[4] H. Cirstea, G. Faure, M. Fernández, I. Mackie, and F.-R. Sinot. From functional programs to interaction nets via the rewriting calculus. *Electronic Notes in Theoretical Computer Science*, 174(10):39–56, 2007.

[5] H. Cirstea and C. Kirchner. The rewriting calculus part I and II. *Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, 2001.

[6] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting calculus with(out) types. In F. Gadducci and U. Montanari, editors, *Proceedings of the fourth workshop on rewriting logic and applications*, Pisa (Italy), Sept. 2002. Electronic Notes in Theoretical Computer Science.

[7] M. Fernández and L. Khalil. Interaction nets with McCarthy's amb: Properties and applications. *Nordic Journal of Computing*, 10(2), 2003.

[8] M. Fernández and I. Mackie. Interaction nets and term rewriting systems. *Theoretical Computer Science*, 190(1):3–39, January 1998.

[9] M. Fernández and I. Mackie. A calculus for interaction nets. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 170–187. Springer-Verlag, September 1999.

[10] M. Fernández, I. Mackie, and F.-R. Sinot. Interaction nets vs. the rho-calculus: Introducing bigraphical nets. In *Proceedings of EXPRESS'05, satellite workshop of Concur, San Francisco, USA, 2005*, Electronic Notes in Computer Science. Elsevier, 2005.

[11] G. Gonthier, 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)*, pages 15–26. ACM Press, Jan. 1992.

[12] C. B. Jay and D. Kesner. Pure pattern calculus. In *Proceedings of the European Symposium on Programming (ESOP) LNCS 3924*, 2006.

[13] J.-W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems, introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

[14] Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, Jan. 1990.

[15] S. Lippi. in^2 : A graphical interpreter for interaction nets. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 380–386. Springer, 2002.

[16] I. Mackie. *The Geometry of Implementation*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, September 1994.

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

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

[19] V. Oostrom. Lambda calculus with patterns. Technical Report IR 228, Vrije Universiteit, Amsterdam, November 1990.

[20] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, 1987.

[21] J. S. Pinto. Sequential and concurrent abstract machines for interaction nets. In J. Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, volume 1784 of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, 2000.

[22] J. S. Pinto. Parallel evaluation of interaction nets with mpine. In A. Middeldorp, editor, *RTA*, volume 2051 of *Lecture Notes in Computer Science*, pages 353–356. Springer, 2001.

[23] R. Plasmeijer and M. V. Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[24] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[25] S. Sato and T. Sugimoto. An implementation of recursive operations in the lambda-evaluator yale on interaction nets, 2004. Technical report of the University of Munster.

[26] M. Walker. An implementation of the rewriting calculus in interaction nets, 2007. Available from http://www.dcs.kcl.ac.uk/pg/walkerm.

Graph Grammars for Local Bigraphs

Davide Grohmann[a,1] Marino Miculan[a,2]

[a] *Department of Mathematics and Computer Science, University of Udine, Italy*

Abstract

We give a formal connection between the graph grammars of *Synchronized Hyperedge Replacement* and *Architectural Design Rewriting*, and *local bigraphs*. First, we define a category of SHR agents, which extend SHR graphs allowing for open systems (i.e., systems with "holes") and multiple locations. Then, we show that these agents correspond precisely to local bigraphs with atomic controls. Standard SHRs can be identified as the link graphs underlying these local bigraphs. Hence, SHR agents can be used as a language for describing local bigraphs with atomic controls.
Finally, we extend these results to ADR-like agents, where nesting of graphs within nodes is allowed. We show that these ADR agents correspond precisely to local bigraphs. Therefore, ADR agents can be used as a language for describing general local bigraphs, alternative to the (more complex) bigraphical algebra.

Keywords: Local bigraphs; Synchronized Hyperedge Replacement; Concurrency Models.

1 Introduction

Bigraphical Reactive Systems (BRSs) [11] have been proposed as a promising meta-model for ubiquitous, mobile systems. The key feature of BRSs is that their states are *bigraphs*, semi-structured data which can represent at once both the (physical, logical) location and the connections of the components of a system. The dynamics of the system is represented by a set of rewrite rules on this semi-structured data.

Bigraphs and BRSs are very flexible: they have been successfully used for representing many domain-specific calculi and models, from traditional programming languages, to process calculi for concurrency and mobility, from context-aware systems to web-service orchestration languages [7,10,1,6,3].

However, a general foundational theory like this should be compared with other theories proposed for similar purposes. In this paper, we compare bigraphs and the graph grammars used in *Synchronized Hyperedge Replacement* (SHR) and *Architectural Design Rewriting* (ADR) frameworks [4,5,2].

More precisely, we show that (a slight generalization of) SHR graphs correspond precisely to *local bigraphs* (a variant of bigraphs which can deal with localized names,

[1] Email: grohmann@dimi.uniud.it

[2] Email: miculan@dimi.uniud.it

Layout based on the macro package of the
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

see [7] and Section 3) with atomic controls. Moreover, a similar generalization of ADR graphs correspond to local bigraphs.

To this end, we have to solve some technical issues. In particular, SHR and ADR graphs do not have a notion of "sub-graph (hierarchical) composition", and hence they do not yield immediately a category. We tackle this problem by extending these graph grammar with graph variables and substitutions, thus obtaining a symmetric monoidal category of *SHR (ADR) agents* (Section 2). An agent is a collection of SHR graphs, each representing a system on its own location, but possibly connected by hyperlinks; graph variables represent open parts of the agents. Then, the encoding is a functor from this category of agents into a corresponding category of local bigraphs (Section 4.1). The functor is monoidal, that is, it respects the parallel compositions of agents. Moreover, it has an inverse functor: any local bigraph with only atomic controls can be represented as an SHR agent (Section 4.2).

It is interesting to give also a characterization of the standard SHR graphs. In Section 5 we show that if we drop the place graph structure from local bigraphs (over atomic controls), we obtain a class of link graphs which correspond exactly to (possibly open) SHR graphs (not agents). Thus, given a local bigraph, its underlying link graph correspond to an SHR graph, while its place graph is simply an extra structure needed for enforcing name scoping.

In Section 6 we consider an extension of SHR agents in the spirit of ADR graphs, where a notion of nesting among edges/graphs is allowed. We generalize the previous results by showing that ADR agents correspond precisely to local bigraphs.

In our opinion, these results are important for several reasons. First, they show once more that bigraphs are a quite expressive general framework of ubiquitous systems. But more importantly, the correspondence points out that SHR and ADR graphs can be used as a language for local bigraphs, alternative to the bigraph algebra [7]. Finally, these constructions paves the way for the application of the rich theory of (local) bigraphs to the SHR and ADR frameworks, as suggested in the concluding Section 7.

2 Synchronized Hyperedge Replacement

Synchronized Hyperedge Replacement (SHR) is a hypergraph framework that allows graph transformations by means of local productions replacing a single hyperedge by a generic hypergraph, possibly with constraints given by the surrounding nodes. The global rewriting is obtained by combining different local production whose conditions are compatible (w.r.t. some synchronization model). For more details see [4,5]. In this paper, we consider a graph grammar which extends the standard SHR grammar by adding graph variables. In this way we can define a composition mechanism among SHR graphs. To ensure that composition is always well-defined we introduce a type system for SHR graphs, which allows to replace variables with graphs having the same type.

A hyperedge is a labelled element from a ranked alphabet $\mathcal{L} = \{l_n\}_{n\in\mathbb{N}}$, each of which has many ordered tentacles as the rank of its label. A hypergraph is formed by a set of nodes and a set of edges, whose tentacles are connected to nodes. Moreover, a hypergraph has a set of external nodes, i.e., its interface with the environment.

Definition 2.1 Let $\mathcal{N}$ be a fixed infinite set of names, $\mathcal{V}$ be a fixed infinite set of variables, and $\mathcal{L}$ be a ranked finite alphabet of labels. A syntactic judgement is of the form $\Gamma \vdash G : \tau$ where:

(i) G is a term generated by the following grammar:

$$G ::= \mathbf{0} \mid l(\vec{x}) \mid X \mid G|G \mid \nu y.G \mid G[w \mapsto z] \tag{1}$$

where $\vec{x} \subseteq \mathcal{N}$, $l \in \mathcal{L}$ with $rank(l) = |\vec{x}|$, $X \in \mathcal{V}$, and $y, w, z \in \mathcal{N}$.

(ii) τ is a type, i.e., a finite set of names.

(iii) Γ is a list of typed variables, that is it is of the form $\Gamma = X_1 : \tau_1, \ldots, X_n : \tau_n$. $\langle\rangle$ is the empty list.

Intuitively, terms are defined inductively: the elementary graphs are: the empty graph ($\mathbf{0}$), a single hyperedge l, whose tentacles are linked to the names in $\vec{x}$ and a variable (X); complex graph are derived by parallel composing of two subgraphs ($G \mid G$), by restricting the scope of a name ($\nu y.G$) in a subgraph and by substituting a name with another ($G[w \mapsto z]$) in a subgraph.

Types (τ) describe the visible names of a graph from a context. Instead environments (Γ) describe the variables' names.

We define the function $v(t)$, that gives the set of all variables in t. We use the notation $\Gamma, X : \tau$ to denote the append of $X : \tau$ to Γ when $X \notin v(\Gamma)$. Analogously, we write Γ_1, Γ_2 to mean the concatenation of Γ_1 and Γ_2 when $v(\Gamma_1) \cap v(\Gamma_2) = \emptyset$. We defined the functions fn, bn and n w.r.t. an environment Γ, as follows:

$$fn_\Gamma(\mathbf{0}) = \emptyset \qquad fn_\Gamma(l(\vec{x})) = \{x_1, \ldots, x_{|\vec{x}|}\} \qquad fn_\Gamma(X) = \tau \quad (\text{if } X : \tau \in \Gamma)$$
$$fn_\Gamma(G_1|G_2) = fn_\Gamma(G_1) \cup fn_\Gamma(G_2) \qquad fn_\Gamma(\nu y.G) = fn_\Gamma(G) \setminus \{y\}$$
$$fn_\Gamma(G[w \mapsto z]) = (fn_\Gamma(G) \setminus \{w\}) \cup \{z\}$$
$$bn_\Gamma(\mathbf{0}) = bn_\Gamma(l(\vec{x})) = bn_\Gamma(X) = \emptyset \qquad bn_\Gamma(G_1|G_2) = bn_\Gamma(G_1) \cup bn_\Gamma(G_2)$$
$$bn_\Gamma(\nu y.G) = bn_\Gamma(G) \cup \{y\} \qquad bn_\Gamma(G[w \mapsto z]) = bn_\Gamma(G)$$

Finally, $n_\Gamma(G) = fn_\Gamma(G) \cup bn_\Gamma(G)$. Notice that in the case $G[w \mapsto z]$ if $w \notin fn(G)$ $G[w \mapsto z]$ has z as free name anyway. The idea is that in a such a case the operator $[w \mapsto z]$ can "create" unused (or idle) free names, i.e., names linked to nothing. [3]

The judgments are taken up-to a structural congruence, that captures graph isomorphisms up-to free nodes, the full list of axioms is shown in Fig. 1.

The type inference rules for judgments on terms are listed below.

$$\frac{}{\langle\rangle \vdash \mathbf{0} : \emptyset} \qquad \frac{}{\langle\rangle \vdash l(\vec{x}) : n(\vec{x})} \qquad \frac{}{X : \tau \vdash X : \tau} \qquad \frac{\Gamma \vdash G : \tau \quad \Gamma' \vdash G' : \tau'}{\Gamma, \Gamma' \vdash G \mid G' : \tau \cup \tau'}$$
$$\frac{\Gamma \vdash G : \tau}{\Gamma \vdash \nu x.G : \tau \setminus \{x\}} \qquad \frac{\Gamma \vdash G : \tau}{\Gamma \vdash G[x \mapsto y] : (\tau \setminus \{x\}) \cup \{y\}}$$

Intuitively, an empty graphs has no names. A hyperedge whose tentacles are linked to the names $\vec{x}$ exposes those names to a context. If a variable is typed by the set

[3] Idle closed names can be safely removed from the graph, see structural congruence in Fig. 1.

$$
\begin{aligned}
\Gamma \vdash G \mid \mathbf{0} &\equiv G \\
\Gamma \vdash G_1 \mid G_2 &\equiv G_2 \mid G_1 \\
\Gamma \vdash (G_1 \mid G_2) \mid G_3 &\equiv G_1 \mid (G_2 \mid G_3) \\
\Gamma \vdash \nu x.\mathbf{0} &\equiv \mathbf{0} \\
\Gamma \vdash \nu x.\nu y.G &\equiv \nu y.\nu x.G \\
\Gamma \vdash \nu x.(G_1 \mid G_2) &\equiv \nu x.G_1 \mid G_2 \text{ if } x \notin fn_\Gamma(G_2) \\
\Gamma \vdash \nu x.G &\equiv \nu y.(G\{y/x\}) \text{ if } y \notin fn_\Gamma(G) \\
\Gamma \vdash G[x \mapsto y] &\equiv (G\{z/x\})[z \mapsto y] \text{ if } z \notin fn_\Gamma(G) \\
\Gamma \vdash l(\vec{x})[y \mapsto z] &\equiv l(\vec{x})\{z/y\} \text{ if } \{y, z\} \cap \vec{y} \neq \emptyset \\
\Gamma \vdash G[x \mapsto y] &\equiv G \text{ if } x \notin fn_\Gamma(G) \text{ and } y \in fn_\Gamma(G) \\
\Gamma \vdash (G_1 \mid G_2)[x \mapsto y] &\equiv G_1[x \mapsto y] \mid G_2 \text{ if } x \notin fn_\Gamma(G_2) \\
\Gamma \vdash (G_1 \mid G_2)[x \mapsto y] &\equiv G_1[x \mapsto y] \mid G_2[x \mapsto y] \text{ if } x \in fn_\Gamma(G_1) \cap fn_\Gamma(G_2) \\
\Gamma \vdash G[x \mapsto y][w \mapsto z] &\equiv G[w \mapsto z][x \mapsto y] \text{ if } x \neq z, y \neq w \\
\Gamma \vdash \nu y.(G[x \mapsto y]) &\equiv \nu x.G \text{ if } y \notin fn_\Gamma(G) \\
\Gamma \vdash \nu z.(G[x \mapsto y]) &\equiv (\nu z.G)[x \mapsto y] \text{ if } z \notin \{x, y\}
\end{aligned}
$$

Fig. 1. Structural congruence for term judgments.

of names τ by an environment Γ, then it exposes such names. The names exposed by a composition of two subgraphs are the union of the names exposed by the two subgraphs. The restriction delete a name from the set of exposed names. Finally, the substitution $[w \mapsto z]$ (possibly) deletes the name w from the set of exposed names and adds z to this set.

Notice that in the last rule if $w \notin fn(G)$ then it effectively adds a new name z to G and to its type τ. Moreover, substitutions are important combined with variables: they should be used to rename variables' names.

Proposition 2.2 *Let $\Gamma \vdash G \equiv G'$, then $\Gamma \vdash G : \tau$ if and only if $\Gamma \vdash G' : \tau$.*

Now, we can introduce a notion of composition among term graphs.

Lemma 2.3 (substitution lemma) *The following rule is admissible.*

$$
\frac{\Gamma_1 \vdash G_1 : \tau_1 \ \ldots \ \Gamma_n \vdash G_n : \tau_n \quad X_1 : \tau_1, \ldots, X_n : \tau_n \vdash G : \tau}{\Gamma_1, \ldots, \Gamma_n \vdash G\{G_1/X_1, \ldots, G_n/X_n\} : \tau}
$$

In order to define a category for judgments we need a notion of tensor product, i.e., a way "to put n graphs side by side". Indeed, as shown in Lemma 2.3, if we want to substitute n variables in a graph G, we need a morphism representing G with "n holes" and another one that provides n graphs and it is composable with the former. To this end, we introduce a new operator ($\|$) and we extend the grammar (1) adding the definition of *agent-graphs* (A):

$$
\begin{aligned}
A &::= \epsilon \mid G \mid A \parallel A \mid \nu y.A \mid A[w \mapsto z] \\
G &::= \mathbf{0} \mid l(\vec{x}) \mid X \mid G|G
\end{aligned}
\tag{2}
$$

$$
\begin{aligned}
\Gamma \vdash G \mid \mathbf{0} &\equiv G \\
\Gamma \vdash G_1 \mid G_2 &\equiv G_2 \mid G_1 \\
\Gamma \vdash (G_1 \mid G_2) \mid G_3 &\equiv G_1 \mid (G_2 \mid G_3) \\
\Gamma \vdash \nu x.\mathbf{0} &\equiv \mathbf{0} \\
\Gamma \vdash (G_1 \mid G_2)[x \mapsto y] &\equiv G_1[x \mapsto y] \mid G_2 \text{ if } x \notin fn_\Gamma(G_2) \\
\Gamma \vdash (G_1 \mid G_2)[x \mapsto y] &\equiv G_1[x \mapsto y] \mid G_2[x \mapsto y] \text{ if } x \in fn_\Gamma(G_1) \cap fn_\Gamma(G_2) \\
\Gamma \vdash l(\vec{x})[y \mapsto z] &\equiv l(\vec{x})\{z/y\} \text{ if } \{y, z\} \cap \vec{x} \neq \emptyset \\
\Gamma \vdash \Gamma \vdash A \parallel \epsilon &\equiv A \\
\Gamma \vdash \epsilon \parallel A &\equiv A \\
\Gamma \vdash (A_1 \parallel A_2) \parallel A_3 &\equiv A_1 \parallel (A_2 \parallel A_3) \\
\Gamma \vdash \nu x.\epsilon &\equiv \epsilon \\
\Gamma \vdash \nu x.\nu y.A &\equiv \nu y.\nu x.A \\
\Gamma \vdash \nu x.(A_1 \parallel A_2) &\equiv \nu x.A_1 \parallel A_2 \text{ if } x \notin fn_\Gamma(A_2) \\
\Gamma \vdash \nu x.A &\equiv \nu y.(A\{y/x\}) \text{ if } y \notin fn_\Gamma(A) \\
\Gamma \vdash A[x \mapsto y] &\equiv (A\{z/x\})[z \mapsto y] \text{ if } z \notin fn_\Gamma(A) \\
\Gamma \vdash A[x \mapsto y][w \mapsto z] &\equiv A[w \mapsto z][x \mapsto y] \text{ if } x \neq z, y \neq w \\
\Gamma \vdash A[x \mapsto y] &\equiv A \text{ if } x \notin fn_\Gamma(A) \text{ and } y \in fn_\Gamma(A) \\
\Gamma \vdash A[x \mapsto x] &\equiv A \text{ if } x \in fn_\Gamma(A) \\
\Gamma \vdash (A_1 \parallel A_2)[x \mapsto y] &\equiv A_1[x \mapsto y] \parallel A_2 \text{ if } x \notin fn_\Gamma(A_2) \\
\Gamma \vdash (A_1 \parallel A_2)[x \mapsto y] &\equiv A_1 \parallel A_2[x \mapsto y] \text{ if } x \notin fn_\Gamma(A_1) \\
\Gamma \vdash (A_1 \parallel A_2)[x \mapsto y] &\equiv A_1[x \mapsto y] \parallel A_2[x \mapsto y] \text{ if } x \in fn_\Gamma(A_1) \cap fn_\Gamma(A_2) \\
\Gamma \vdash \nu y.(A[x \mapsto y]) &\equiv \nu x.A \text{ if } y \notin fn_\Gamma(A) \\
\Gamma \vdash \nu z.(A[x \mapsto y]) &\equiv (\nu z.A)[x \mapsto y] \text{ if } z \notin \{x, y\}
\end{aligned}
$$

Fig. 2. Structural congruence for agent-graph judgment.

where ϵ represents the empty agent-graph. All the functions defined on terms $(v, fn, \dots)$ can be smoothly lifted to agent-graphs. Similarly to the case of terms, we consider agent-graphs up-to a structural congruence capturing graph isomorphisms (all the axioms are shown in Fig. 2).

Now, we extend the definition of graph types over agent-graphs:

$$\sigma ::= \varepsilon \mid \tau \mid \sigma\sigma$$

where juxtaposition denotes concatenation. In practice the new types are sequences of name sets. We extend the operations $\cup$ and $\setminus$ over sequences as follows:

$$(S \text{ is a set}) \qquad \tau_1 \dots \tau_n \cup S \stackrel{\triangle}{=} (\tau_1 \cup S) \dots (\tau_n \cup S) \qquad \tau_1 \dots \tau_n \setminus S \stackrel{\triangle}{=} (\tau_1 \setminus S) \dots (\tau_n \setminus S)$$

We extend the previous type inference rules for judgment on agent-graphs as

shown below.

$$\frac{}{\langle\rangle \vdash \mathbf{0} : \emptyset} \qquad \frac{}{\langle\rangle \vdash l(\vec{x}) : n(\vec{x})} \qquad \frac{}{X : \tau \vdash X : \tau} \qquad \frac{\Gamma \vdash G : \tau \quad \Gamma' \vdash G' : \tau'}{\Gamma, \Gamma' \vdash G \mid G' : \tau \cup \tau'}$$

$$\frac{\Gamma \vdash A : \sigma}{\Gamma \vdash \nu x.A : \sigma \setminus \{x\}} \qquad \frac{\Gamma \vdash A : \sigma}{\Gamma \vdash A[x \mapsto y] : (\sigma \setminus \{x\}) \cup \{y\}}$$

$$\frac{}{\langle\rangle \vdash \epsilon : \varepsilon} \qquad \frac{\Gamma \vdash A : \sigma \quad \Gamma' \vdash A' : \sigma'}{\Gamma, \Gamma' \vdash A \parallel A' : \sigma\sigma'} \qquad \frac{\Gamma \vdash A : \sigma \quad \pi \text{ permutation}}{\pi(\Gamma) \vdash A : \sigma}$$

Intuitively, the empty agent-graph represent the empty agent and it is typed by the empty list. Notice that there is a difference between $\mathbf{0}$ and ϵ: $\mathbf{0}$ is the null process, and it is a non-empty agent. The rule for $\parallel$ are quite similar to the one for $|$, but in this case the two graphs lives into two difference locations, and hence the names can be treated in a different way, so the new type is defined by juxtaposing the two subtypes. Finally, last rule describes the ability to reorder the variables in the environment, it will be important in the definition of a category for agent-graphs.

Proposition 2.4 *Let $\Gamma \vdash A \equiv A'$, $\Gamma \vdash A : \sigma$ if and only if $\Gamma \vdash A' : \sigma$.*

Now, we are able to define a category for typed SHR graphs.

Definition 2.5 The category $\mathbf{H}$ of SHR graphs has graph types (σ) as objects, and judgments on agent-graphs as morphisms, that is, if $X_1 : \tau_1, \ldots, X_n : \tau_n \vdash A : \sigma$ then $(X_1, \ldots, X_n, A) : \tau_1 \ldots \tau_n \to \sigma$ is a morphism. Composition is defined in virtue of Lemma 2.3:

$$\frac{\Gamma \vdash G_1 \parallel \ldots \parallel G_n : \tau_1 \ldots \tau_n \quad X_1 : \tau_1, \ldots, X_n : \tau_n \vdash A : \sigma}{\Gamma \vdash A\{G_1/X_1, \ldots, G_n/X_n\} : \sigma}$$

Proposition 2.6 $(\mathbf{H}, \parallel, \epsilon)$ *is a strict symmetric monoidal category.*

3 Local Bigraphs

In this section we recall Milner's *local bigraphs*, a subclass of binding bigraphs.

Intuitively, a (local) bigraph represents an open system, so it has an inner and an outer interface to "interact" with subsystems and the surrounding environment. An example of a local bigraph is shown in Fig. 3. The *ordinals* of the interfaces describe the *roots* in the outer interface (that is, the various locations where the nodes live) and the *sites* in the inner interface (that is, the gray holes where other bigraphs can be inserted). On the other hand, the *names* in the interfaces describe the free links, that is end points where links from the outside world can be pasted, creating new links among nodes. We refer the reader to [7,8,12,13] for more details.

Let $\mathcal{K}$ be a *binding signature* of controls, and $ar\colon \mathcal{K} \to \mathbb{N} \times \mathbb{N}$ be the arity function. The arity pair (h, k) (written $h \to k$) consists of the *binding arity* h and the *free arity* k, indexing respectively the binding and the free ports of a control.

Definition 3.1 A *local interface* is a list $(X_0, \ldots, X_{n-1})$, where n is a finite ordinal (called *width*) and X_is are sets of names. X_i represents the names located at i.

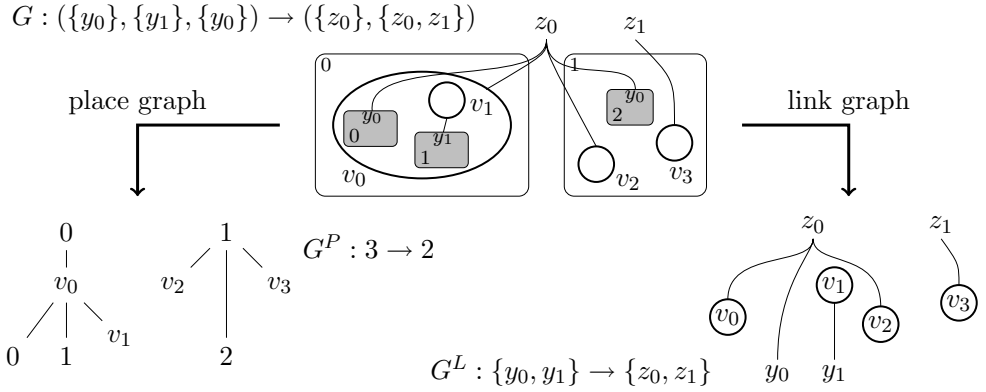


Fig. 3. An example of a local bigraph.

Notice that it is not necessary that the X_is are disjoint. We say that a name has multiple locations in I if it belongs to more than one name set of I.

Definition 3.2 A *local bigraph* $G\colon (\vec{X}) \to (\vec{Y})$ is defined as a (pure) bigraph $G^u\colon \langle |\vec{X}|, \bigcup \vec{X} \rangle \to \langle |\vec{Y}|, \bigcup \vec{Y} \rangle$ satisfying certain locality conditions.

G^u is defined by composing a *place graph* G^P, describing the nesting of nodes, and a *link graph* G^L, describing the (hyper)links among nodes.

$$
\begin{aligned}
G^u &= (G^P, G^L)\colon \langle m, X \rangle \to \langle n, Y \rangle && \text{(pure bigraph)} \\
G^P &= (V, ctrl, prnt)\colon m \to n && \text{(place graph)} \\
G^L &= (V, E, ctrl, link)\colon X \to Y && \text{(link graph)}
\end{aligned}
$$

where V, E are the sets of nodes and edges respectively, $ctrl\colon V \to \mathcal{K}$ is the *control map,* which assigns a control to each node, $prnt\colon m \uplus V \to V \uplus n$ is the (acyclic) *parent map,* $P = \sum_{v \in V} \pi_1(ar(ctrl(v)))$ is the set of ports and $B = \sum_{v \in V} \pi_2(ar(ctrl(v)))$ is the set of bindings (associated to all nodes), and $link\colon X \uplus P \to E \uplus B \uplus Y$ is the *link map.*

The locality conditions are the following:

(i) if a link is bound, then its inner names and ports must lie within the node that binds it;

(ii) if a link is free, with outer name x, then x must be located in every region that contains any inner name or port of the link.

Definition 3.3 The category of local bigraphs over a signature $\mathcal{K}$ ($\mathbf{Lbg}(\mathcal{K})$) has local interfaces as objects, and local bigraphs as morphisms.

Given two local bigraphs $G\colon (\vec{X}) \to (\vec{Y})$, $H\colon (\vec{Y}) \to (\vec{Z})$, the composition $H \circ G\colon (\vec{X}) \to (\vec{Z})$ is defined by composing their place and link graphs:

(i) the composition of $G^P\colon |\vec{X}| \to |\vec{Y}|$ and $H^P\colon |\vec{Y}| \to |\vec{Y}|$ is defined as

$$H^P \circ G^P = (V_G \uplus V_H, ctrl_G \uplus ctrl_H, (id_{V_G} \uplus prnt_H) \circ (prnt_G \uplus id_{V_H}))\colon |\vec{X}| \to |\vec{Y}|;$$

(ii) the composition of $G^L\colon \bigcup \vec{X} \to \bigcup \vec{Y}$ and $H^L\colon \bigcup \vec{Y} \to \bigcup \vec{Z}$ is defined as

$$H^L \circ G^L = \\ (V_G \uplus V_H, E_G \uplus E_H, ctrl_G \uplus ctrl_H, (id_{E_G} \uplus link_H) \circ (link_G \uplus id_{P_H}))\colon \bigcup \vec{X} \to \bigcup \vec{Z}.$$

Plg and **Lnk** will denote the categories of place and link graphs, respectively.

Intuitively, composition is performed in two steps. First, the place graph are composed by putting the roots of the "lower" bigraph inside the sites (i.e., holes) of the "upper" one, respecting the order given by the ordinal in the interface; then, the links are connected by sticking together the end parts of connections in the two link graphs, labelled with the same name in the common interface.

An important operation about (bi)graphs, is the *tensor product*. Intuitively, the tensor product of two bigraphs $G\colon (\vec{X}) \to (\vec{Y})$ and $H\colon (\vec{X'}) \to (\vec{Y'})$, is a bigraph $G \otimes H\colon (\vec{X}\vec{X'}) \to (\vec{Y}\vec{Y'})$ is defined when $X \cap X' = Y \cap Y' = \emptyset$ and it obtained by putting "side by side" G and H, without merging any root nor any name in the interfaces. Two useful variant of tensor product can be defined using tensor and composition: the *parallel product* $\|$, which merges shared names between two bigraphs or link graphs, and the *prime product* $|$, that moreover merges all roots in a single one. Due to lack of space, we cannot give a formal definition of these operations; we refer the reader to [13].

It is easy to check that composition and tensor preserve the locality conditions.

4 From SHR graphs to Local Bigraphs, and back

4.1 Encoding SHR graphs into Local Bigraphs

In this section we introduce an encoding functor from the SHR graph category **H** to the local bigraph category **Lbg**.

The first step is notice that the SHR ranked alphabet of labels ($\mathcal{L}$) and the bigraphical signature ($\mathcal{K}$) are quite similar. So, the idea is to choose as bigraphical signature exactly the alphabet of labels. In other words, our encoding translates the SHR hyperedges into nodes, and nodes (or names) into links (i.e., outer names and edges). Finally, note that every $l \in \mathcal{L}$ has only "exiting tentacles", hence the corresponding l in the bigraphical signature has no binding ports. Formally:

$$\mathcal{K}_{\mathcal{L}} \stackrel{\triangle}{=} \{l : 0 \to n \mid l_n \in \mathcal{L}\}.$$

Now we can define our encoding functor $[\![-]\!] : \mathbf{H}(\mathcal{L}) \to \mathbf{Lbg}(\mathcal{K}_{\mathcal{L}})$ as follows.

Objects:

$$[\![\varepsilon]\!] = ()$$
$$[\![\tau_1 \dots \tau_n]\!] = (\tau_1, \dots, \tau_n)$$

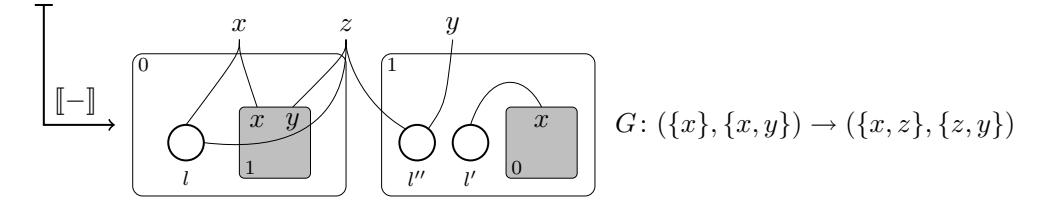


Fig. 4. An example of encoding an agent-graph into a local bigraph.

Morphisms:

$$
\begin{aligned}
[\![\langle\rangle \vdash \epsilon : \varepsilon]\!] &= id_{()} \\
[\![\Gamma, \Gamma' \vdash A \parallel A' : \sigma\sigma']\!] &= [\![\Gamma \vdash A : \sigma]\!] \parallel [\![\Gamma' \vdash A' : \sigma']\!] \\
[\![\pi(\Gamma) \vdash A : \sigma]\!] &= [\![\Gamma \vdash A : \sigma]\!] \circ \pi \\
[\![\Gamma \vdash \nu x.A : \sigma \setminus \{x\}]\!] &= /(x) \circ ([\![\Gamma \vdash A : \sigma]\!] \parallel \{x\}) \\
[\![\Gamma \vdash A[x \mapsto y] : (\sigma \setminus \{x\}) \cup \{y\}]\!] &= (y)/(x) \circ ([\![\Gamma \vdash A : \sigma]\!] \parallel \{x\}) \\
[\![\langle\rangle \vdash \mathbf{0} : \emptyset]\!] &= 1 \\
[\![\langle\rangle \vdash l(\vec{x}) : n(\vec{x})]\!] &= l_{\vec{x}} \\
[\![X : \tau \vdash X : \tau]\!] &= id_{(\tau)} \\
[\![\Gamma, \Gamma' \vdash G \mid G' : \tau \cup \tau']\!] &= [\![\Gamma \vdash G : \tau]\!] \mid [\![\Gamma' \vdash G' : \tau']\!]
\end{aligned}
$$

An example of encoding is given in Fig. 4. Basically, each variable of type τ is encoded as a site having τ local names; therefore, permutation of variables is permutation of sites. Restricted names are represented by bigraph edges, not accessible from the context. The empty graph $\mathbf{0}$ is represented by the empty root 1.

We can now prove the following result about the adequacy of the encoding.

Proposition 4.1 *Let A, A' be agent-graphs; then, for every Γ: $\Gamma \vdash A \equiv A'$ if and only if $[\![\Gamma \vdash A : \sigma]\!] = [\![\Gamma \vdash A' : \sigma]\!]$.*

4.2 Encoding Local Bigraphs into SHR graphs

In this section we provide a translation of local bigraphs into SHR graphs, supposing that every control in $\mathcal{K}$ is atomic (since SHR graphs do not have allow nestings). We take as alphabet of ranked labels the bigraphical signature. Formally:

$$\mathcal{L}_{\mathcal{K}} \stackrel{\triangle}{=} \{k_n \mid k : 0 \to n \in \mathcal{K}\}.$$

In order to simplify the translation procedure, we suppose that all local bigraphs are provided in a normal form, i.e., the *connected normal form* [8, Proposition 10.3]. The idea of this normal form is pushing all wirings inwards as far as we can: by using $\parallel$ and $\mid$ in place of $\otimes$, and also pushing closure inwards wherever possible.

Definition 4.2 Each local bigraph G, prime P and molecule M can be expressed

by an equation of the following form, named *connected normal form*:

$$\begin{aligned} G &= (/(Z) \parallel id_{(\vec{Y})}) \circ (P_0 \parallel \cdots \parallel P_{n-1}) \circ \pi \\ P &= (/(Z) \mid id_{(X)}) \circ (\delta \mid M_0 \mid \cdots \mid M_{k-1}) \circ \pi \\ M &= (/(Z) \mid id_{(W)}) \circ c \end{aligned}$$

where δ is a local substitution, π is a local permutation, and c is an atom.

Now we can define our encoding functor $(\!|-|\!) : \mathbf{Lbg}(\mathcal{K}) \to \mathbf{H}(\mathcal{L}_{\mathcal{K}})$ as follows.

Objects:

$$\begin{aligned} (\!|()|\!) &= \varepsilon \\ (\!|(X_1, \ldots, X_n)|\!) &= X_1 \ldots X_n \end{aligned}$$

Morphisms: for $G = (/(Z) \parallel id_{(\vec{Y})}) \circ (P_0 \parallel \cdots \parallel P_{n-1}) \circ \pi$, where $Z = \{z_1, \ldots, z_n\}$:

$$\begin{aligned} (\!|G : (\vec{X}) \to (\vec{Y})|\!) &= \vec{W} : (\!|\vec{X}|\!) \vdash \nu z_1 \ldots \nu z_n.((\!|p_0|\!) \parallel \cdots \parallel (\!|p_{|\vec{Y}|-1}|\!)) : (\!|\vec{Y}|\!) \\ \text{where } p_0 \parallel \ldots \parallel p_{|\vec{Y}|-1} &= (P_0 \parallel \cdots \parallel P_{|\vec{Y}|-1}) \circ \pi \circ (v^0_{X_0} \parallel \cdots \parallel v^{|\vec{X}|-1}_{X_{|\vec{X}|-1}}) \\ (\!|id_{()}|\!) &= \epsilon \\ (\!|(w)/(\{w_1, \ldots, w_n\}) \mid \delta|\!) &= [w_1 \mapsto w] \ldots [w_m \mapsto w](\!|\delta|\!) \\ (\!|p_i|\!) &= \nu z_1 \ldots \nu z_n.((\!|m_0|\!) \mid \cdots \mid (\!|m_{k_i}|\!))(\!|\delta|\!)) \quad 0 \le i < |\vec{Y}| \\ (\!|m_j|\!) &= \nu z_1 \ldots \nu z_n.(\!|c|\!) \quad 0 \le j \le k_i \\ (\!|v^i_{X_i}|\!) &= W_i \quad 0 \le i < |\vec{X}| \\ (\!|K_{\vec{x}}|\!) &= K(\vec{x}) \end{aligned}$$

where the nodes $v^0, \ldots, v^{|\vec{X}|-1}$ have special controls not present in $\mathcal{K}$, and they are used only to simplify the translation procedure. In practice, these variables give a "name" to each hole of the bigraphs, i.e., the node v^i represents the i-hole of the bigraphs. Notice that, the hole sequence may not follow necessarily the numeration of holes, as shown in the bigraph of the example encoding in Fig. 5.

Now we can state and prove the following result on the adequacy of the encoding.

Proposition 4.3 *Let G, G' be local bigraphs over an atomic signature; then, $G = G'$ if and only if there exists Γ such that $\Gamma \vdash (\!|G|\!) \equiv (\!|G'|\!)$.*

Moreover, we can establish nice connections between the two categories.

Proposition 4.4 (i) *Let G be a local bigraph, then $[\![(\!|G|\!)]\!] = G$.*

(ii) *Let $\Gamma \vdash A : \sigma$ be a graph judgment, then $\Gamma \vdash (\!|[\![\Gamma \vdash A : \sigma]\!]|\!) \equiv A$.*

5 SHR graphs as link graphs

In this section, we give a characterization of (possibly open) standard SHR graphs, as the link graphs underlying the local bigraphs over atomic signatures. This cat-

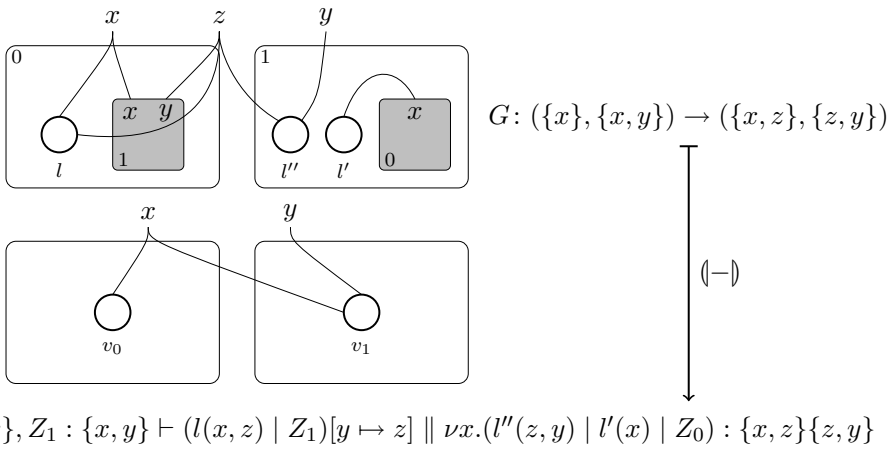


Fig. 5. An example of encoding a local bigraph into an agent-graph.

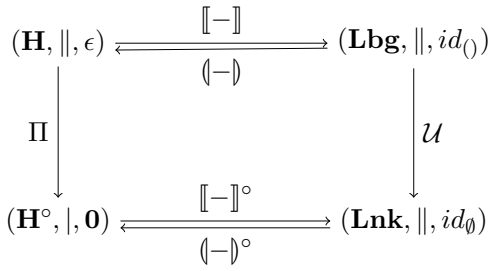


Fig. 6. Functors among the categories under consideration.

egory of link graphs can be characterized by the forgetful functor from bigraphs to link graphs. A summary diagram of the functors we are dealing with is in Fig. 6.

Definition 5.1 The forgetful functor $\mathcal{U} : \mathbf{Lbg}(\mathcal{K}) \to \mathbf{Lnk}(\mathcal{K})$ is defined as follows:

Objects: $\mathcal{U}((X_0, \ldots, X_{n-1})) = \bigcup_{i=0}^{n-1} X_i$

Morphisms: $\mathcal{U}((V, E, ctrl, prnt, link)) = (V, E, ctrl, link)$.

Following the encoding idea of the functor $(\!|-|\!)$ from bigraphs to SHR, we can map the link graphs inside $\mathbf{H}$. The idea is to consider a link graph as one-locality bigraphs, i.e., local bigraphs having only one root and zero or one holes. Clearly, the vice versa does not hold. We can recover the (almost) one to one correspondence by defining a sub-category of $\mathbf{H}$.

Definition 5.2 The category $\mathbf{H}^\circ$ has types (τ) as objects, and one variable judgments on terms as morphisms, i.e., if $X : \tau \vdash G : \tau'$ then $(X, G) : \tau \to \tau'$ is a morphism. Composition is defined as follows:

$$\frac{X : \tau \vdash G : \tau' \quad Y : \tau' \vdash G' : \tau''}{X : \tau \vdash G'\{G/Y\} : \tau''}$$

Proposition 5.3 $(\mathbf{H}^\circ, |, \mathbf{0})$ *is a strict symmetric monoidal category.*

For completeness we list below the definition of the two new functors. Their definition is a "simplification" of the more general functor seen before.

$[\![-]\!]^\circ : \mathbf{Lnk}(\mathcal{K}) \to \mathbf{H}^\circ(\mathcal{K})$:

Objects: $[\![\tau]\!]^\circ = \tau$

Morphisms:

$$\begin{aligned}
[\![\langle\rangle \vdash \mathbf{0} : \emptyset]\!]^\circ &= id_\emptyset \\
[\![\Gamma \vdash \nu x.A : \sigma \setminus \{x\}]\!]^\circ &= /(x) \circ ([\![\Gamma \vdash A : \sigma]\!]^\circ \parallel \{x\}) \\
[\![\Gamma \vdash A[x \mapsto y] : (\sigma \setminus \{x\}) \cup \{y\}]\!]^\circ &= (y)/(x) \circ ([\![\Gamma \vdash A : \sigma]\!]^\circ \parallel \{x\}) \\
[\![\langle\rangle \vdash l(\vec{x}) : n(\vec{x})]\!]^\circ &= l_{\vec{x}} \\
[\![X : \tau \vdash X : \tau]\!]^\circ &= id_\tau \\
[\![\Gamma, \Gamma' \vdash G \mid G' : \tau \cup \tau']\!]^\circ &= [\![\Gamma \vdash G : \tau]\!]^\circ \mid [\![\Gamma' \vdash G' : \tau']\!]^\circ
\end{aligned}$$

$(\!|-|\!)^\circ : \mathbf{H}^\circ(\mathcal{L} \to \mathbf{Lnk}(\mathcal{L}))$:

Objects: $(\!|X|\!)^\circ = X$

Morphisms:

$$\begin{aligned}
(\!|G : X \to Y|\!)^\circ &= Z : X \vdash (\!|/(Z)|\!)^\circ (\!|P \circ \pi \circ v_X|\!)^\circ : Y \\
(\!|id_\emptyset|\!)^\circ &= \mathbf{0} \\
(\!|/(\{z_1, \ldots, z_n\})|\!)^\circ &= \nu z_1. \cdots \nu z_n. \\
(\!|(w)/(\{w_1, \ldots, w_n\}) \mid \delta|\!)^\circ &= [w_1 \mapsto w] \ldots [w_m \mapsto w] (\!|\delta|\!)^\circ \\
(\!|p|\!)^\circ &= (\!|/(Z)|\!)((\!|m_0|\!)^\circ \mid \cdots \mid (\!|m_k|\!)^\circ)(\!|\delta|\!)^\circ) \\
(\!|m_j|\!)^\circ &= (\!|/(Z)|\!)^\circ (\!|c|\!)^\circ \quad j \in \{0, \ldots, k\} \\
(\!|v_X|\!)^\circ &= Z \quad i \in \{0, \ldots, |\vec{X}| - 1\} \\
(\!|K_{\vec{x}}|\!)^\circ &= K(\vec{x})
\end{aligned}$$

All previous functors suggest a forgetful functor from the SHR graph category to the SHR graph sub-category, $\Pi : \mathbf{H} \to \mathbf{H}^\circ$, defined as follows:

Objects: $\Pi(\tau_1 \ldots \tau_n) = \bigcup_{i=1}^n \tau_i$

Morphisms:

$$\begin{aligned}
\Pi(\langle\rangle \vdash \epsilon : \varepsilon) &= id_\emptyset \\
\Pi(\Gamma, \Gamma' \vdash A \parallel A' : \sigma\sigma') &= \Pi(\Gamma \vdash A : \sigma) \mid \Pi(\Gamma' \vdash A' : \sigma') \\
\Pi(\pi(\Gamma) \vdash A : \sigma) &= \Pi(\Gamma \vdash A : \sigma) \\
\Pi(\Gamma \vdash \nu x.A : \sigma \setminus \{x\}) &= /(x) \circ (\Pi(\Gamma \vdash A : \sigma) \mid \{x\}) \\
\Pi(\Gamma \vdash A[x \mapsto y] : (\sigma \setminus \{x\}) \cup \{y\}) &= (y)/(x) \circ (\Pi(\Gamma \vdash A : \sigma) \mid \{x\}) \\
\Pi(\langle\rangle \vdash \mathbf{0} : \emptyset) &= id_\emptyset \\
\Pi(\langle\rangle \vdash l(\vec{x}) : n(\vec{x})) &= l_{\vec{x}} \\
\Pi(X : \tau \vdash X : \tau) &= id_\tau \\
\Pi(\Gamma, \Gamma' \vdash G \mid G' : \tau \cup \tau) &= \Pi(\Gamma \vdash G : \tau) \mid \Pi(\Gamma' \vdash G' : \tau')
\end{aligned}$$

In practice the above functor merges all the separate graph-agents graphs into a unique bigger graphs. In other words, it translates a $\parallel$ operator with the $\mid$ one.

As a consequence of the definitions of the functors $[\![-]\!]$, $\mathcal{U}$ and $(\!|-|\!)^\circ$, we can prove the following result.

Proposition 5.4 *Let $\Gamma \vdash A : \sigma$ be a graph judgment; then, $\Gamma \vdash (\!|\mathcal{U}([\![\Gamma \vdash A : \sigma]\!])|\!)^\circ \equiv \Pi(\Gamma \vdash A : \sigma)$.*

Proposition 5.5 (i) *Let H be a link graph, then $[\![(\!|H|\!)^\circ]\!]^\circ = H$.*
(ii) *Let $X : \tau \vdash G : \tau'$ be a term judgment, then $X : \tau \vdash (\!|[\![X : \tau \vdash G : \tau']\!]^\circ|\!)^\circ \equiv G$.*

6 From ADR graphs to local bigraphs, and back

In this section we generalize further the syntax of SHR agents to get a language capable to express any local bigraph, not only those on atomic controls. To this end, we have to allow nesting of graphs within edge labels. Such kind of graph grammars have been already analyzed in the case of *Architectural Design Rewriting* (ADR) graphs [2], where a notion of nesting among edges/graphs is considered.

The idea is to add to $\mathcal{L}$ a new set of egde labels, which can contain graphs. We call the old labels *atomic* (written l_n, where n is the exit-rank of l), and the new ones *non-atomic* (written $L_n(m)$, where n is the exit-rank and m is the in-rank of L). Then, the syntax in (2) can be extended as follows to define layered-graphs:

$$A ::= \epsilon \mid G \mid A \parallel A \mid \nu y.A \mid A[w \mapsto z]$$
$$G ::= \mathbf{0} \mid l(\vec{x}) \mid X \mid G|G \mid L(\vec{y})[G\backslash\vec{x}]$$

where the graph $L(\vec{y})[G\backslash\vec{x}]$ describes a non-atomic edge L having exiting tentacles linked to the names in $\vec{y}$, and also has a subgraph G inside itself. Finally, $\backslash\vec{x}$ means that the names $\vec{x}$ coming from the graph G are stopped by the edge itself by linking them to "incoming" tentacles of L.

All the functions defined on terms and agent-graphs ($v, fn, \dots$) can be smoothly lifted to the layered-graphs. Similarly to the case of terms and agent-graphs, we consider those graphs up-to a structural congruence capturing graph isomorphisms; to this end, we extend the rules of Figure 2 with the followings:

$$\Gamma \vdash L(\vec{y})[G\backslash\vec{x}] \equiv L(\vec{y})[(G\{z/x\})\backslash(\vec{x}\{z/x\})] \text{ if } x \in \vec{x} \text{ and } z \notin \vec{x} \cup fn_\Gamma(G)$$
$$\Gamma \vdash L(\vec{y})[G\backslash\vec{x}][y \mapsto z] \equiv L(\vec{y}\{z/y\})[G[y \mapsto z]\backslash\vec{x}] \text{ if } \{y, z\} \cap \vec{y} \neq \emptyset \text{ and } y, z \notin \vec{x}$$
$$\Gamma \vdash \nu z.L(\vec{y})[G\backslash\vec{x}] \equiv L(\vec{y})[(\nu z.G)\backslash\vec{x}] \text{ if } z \notin \vec{y} \cup \vec{x}$$

The set of types considered here is the same of agent-graphs, i.e., types are sequences of name sets. Now we must extend the set of type inference rules defined for agent-graphs to be used for layered-graphs by adding the following new rule:

$$\frac{\Gamma \vdash G : \tau \quad n(\vec{x}) \subseteq \tau}{\Gamma \vdash L(\vec{y})[G\backslash\vec{x}] : (\tau \setminus n(\vec{x})) \cup n(\vec{y})}$$

This new rule allows to insert a graphs inside a non-atomic edge; the unique requirement is that the graph must correspond to a graph term, i.e., its type is a single set of names. The new type is constructed by deleting the names $\vec{x}$ of G stopped by L and by adding the names $\vec{y}$ defining the exiting tentacles of L.

Notice that, composition remains unchanged as well as the definition of the category. We call this new category *layered-graph category* (**L**).

Proposition 6.1 $(\mathbf{L}, \|, \epsilon)$ *is a strict symmetric monoidal category.*

The translations from ADR graphs to local bigraphs and vice versa is quite similar to the one shown in sections 4.1 and 4.2, more precisely they are a straightforward extension of them.

From ADR graphs to local bigraphs. The signature $\mathcal{K}^{\Box}_{\mathcal{L}}$ for the category of local bigraphs is defined from the ranked set of labels $\mathcal{L}$ as follows:

$$\mathcal{K}^{\Box}_{\mathcal{L}} \triangleq \{l : 0 \to n \mid l_n \in \mathcal{L}\} \cup \{L : m \to n \mid L_n(m) \in \mathcal{L}\}$$

where the ls are atomics and the Ls are non-atomic[4].

We define our encoding functor $[\![-]\!]^{\Box} : \mathbf{L}(\mathcal{L}) \to \mathbf{Lbg}(\mathcal{K}^{\Box}_{\mathcal{L}})$ as the functor $[\![-]\!]$ by adding the following case.

$$[\![\Gamma \vdash L(\vec{y})[G\backslash\vec{x}] : (\tau \setminus n(\vec{x})) \cup n(\vec{y})]\!]^{\Box} = L_{\vec{y}(\vec{x})} \circ [\![\Gamma \vdash G : \tau]\!]^{\Box}$$

From local bigraphs to ADR graphs. The ranked set of labels $\mathcal{L}^{\Box}_{\mathcal{K}}$ for the layered graphs is defined from the signature $\mathcal{K}$ of local bigraphs as follows:

$$\mathcal{L}^{\Box}_{\mathcal{K}} \triangleq \{k_n \mid k : 0 \to n \in \mathcal{K}\} \cup \{K_n(m) \mid K : m \to n \in \mathcal{K}\}$$

where the ks are atomics and the Ks are non-atomic.

We define our encoding functor $(\!|-|\!)^{\Box} : \mathbf{Lbg}(\mathcal{K}) \to \mathbf{L}(\mathcal{L}^{\Box}_{\mathcal{K}})$ as the functor $(\!|-|\!)$ by adding the following case.

$$(\!|K_{\vec{y}(\vec{x})} \circ p|\!)^{\Box} = K(\vec{y})[(\!|p|\!)^{\Box}\backslash\vec{x}]$$

Now we can state and prove the following result on the encodings.

Proposition 6.2 *Let G, G' be local bigraphs; then, $G = G'$ if and only if there exists Γ such that $\Gamma \vdash (\!|G|\!)^{\Box} \equiv (\!|G'|\!)^{\Box}$.*

Proposition 6.3 (i) *Let G be a local bigraph, then $[\![(\!|G|\!)^{\Box}]\!]^{\Box} = G$.*

(ii) *Let $\Gamma \vdash A : \sigma$ be a graph judgment, then $\Gamma \vdash (\!|[\![\Gamma \vdash A : \sigma]\!]^{\Box}|\!)^{\Box} \equiv A$.*

7 Conclusions

In this paper we have first investigated the connection between *Synchronized Hyperedge Replacement* graphs and *(local) bigraphs*, two different graphical frameworks for concurrency. We have given a mild generalization of SHR graph grammar, allowing for "holes" and localities, which turns out to correspond exactly to local bigraphs over atomic controls. Moreover, the link graphs underlying these local bigraphs correspond to usual SHR graphs. Then, we have extended this approach to *Architectural Design Rewriting* graphs, which turn out to correspond to general local bigraphs. Therefore, SHR graph grammar is a language for local bigraphs over

[4] We can further distinguish the non-atomic controls in *active* of *passive*, but in the present case it is not necessary, since we do not deal with the bigraph semantics.

atomic signatures, and ADR graph grammar is a language for local bigraphs; these languages can be used in place of the more complex bigraph algebra.

A possible future work is to take advantage of the rich theory provided by bigraphical reactive systems [7], in order to obtain interesting results about SHR and ADR. In particular, local bigraphs allow to synthesise labelled transition systems out of rewriting rules, via the so-called *idem-pushout* construction [9]; it is important to notice that the bisimilarity induced by this labelled transitions system (LTS) is always a congruence. Therefore, given a reactive system over SHR (ADR) graphs, we can derive the labelled transition system in local bigraphs, and remap it on SHR (ADR) graphs. Then, the inductive definition of SHR (ADR) agents can be useful for defining an SOS-like presentation of the LTS derived in this way.

Acknowledgments. The authors thank Emilio Tuosto and Ivan Lanese for useful discussions about SHR.

References

[1] Lars Birkedal, Søren Debois, Ebbe Elsborg, Thomas Hildebrandt, and Henning Niss. Bigraphical models of context-aware systems. In Luca Aceto and Anna Ingólfsdóttir, editors, *Proc. FoSSaCS*, volume 3921 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2006.

[2] Roberto Bruni, Alberto Lluch-Lafuente, Ugo Montanari, and Emilio Tuosto. Service oriented architectural design. In Gilles Barthe and Cédric Fournet, editors, *Proc. TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 186–203. Springer, 2007.

[3] Mikkel Bundgaard, Arne J. Glenstrup, Thomas T. Hildebrandt, Espen Højsgaard, and Henning Niss. Formalizing higher-order mobile embedded business processes with binding bigraphs. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 83–99. Springer, 2008.

[4] Pierpaolo Degano and Ugo Montanari. A model for distributed systems based on graph rewriting. *J. ACM*, 34(2):411–449, 1987.

[5] Gian Luigi Ferrari, Ugo Montanari, and Emilio Tuosto. A lts semantics of ambients via graph synchronization with mobility. In Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi, editors, *Proc. ICTCS*, volume 2202 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2001.

[6] Davide Grohmann and Marino Miculan. Reactive systems over directed bigraphs. In L. Caires and V.T. Vasconcelos, editors, *Proc. CONCUR 2007*, volume 4703 of *Lecture Notes in Computer Science*, pages 380–394. Springer-Verlag, 2007.

[7] Ole Høgh Jensen and Robin Milner. Bigraphs and transitions. In *Proc. POPL*, pages 38–49, 2003.

[8] Ole Høgh Jensen and Robin Milner. Bigraphs and mobile processes (revised). Technical report UCAM-CL-TR-580, Computer Laboratory, University of Cambridge, 2004.

[9] James J. Leifer and Robin Milner. Deriving bisimulation congruences for reactive systems. In Catuscia Palamidessi, editor, *Proc. CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2000.

[10] James J. Leifer and Robin Milner. Transition systems, link graphs and petri nets. *Mathematical Structures in Computer Science*, 16(6):989–1047, 2006.

[11] Robin Milner. Bigraphical reactive systems. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *Proc. 12th CONCUR*, volume 2154 of *Lecture Notes in Computer Science*, pages 16–35. Springer, 2001.

[12] Robin Milner. Bigraphs whose names have multiple locality. Technical Report 603, University of Cambridge, CL, September 2004.

[13] Robin Milner. Local bigraphs and confluence: Two conjectures. In *Proc. EXPRESS 2006*, volume 175(3) of *Electronic Notes in Theoretical Computer Science*, pages 65–73. Elsevier, 2007.

Compilation of Interaction Nets

Abubakar Hassan Ian Mackie Shinya Sato

Department of Computer Science, University of Sussex, Falmer, Brighton, U.K.

LIX, CNRS UMR 7161, École Polytechnique, 91128 Palaiseau Cedex, France

Faculty of Econoinformatics, Himeji Dokkyo University, 5-7-1 Kamiohno, Himeji-shi, Hyogo 670-8524, Japan

Abstract

This paper is about a new implementation technique for interaction nets—a visual programming language based on graph rewriting. We compile interaction nets to C, which offers a robust and efficient implementation, in addition to portability. In the presentation of this work we extend the interaction net programming paradigm to introduce a number of features which make it a practical programming language.

Keywords: Interaction Nets; Compilation.

1 Introduction

Interaction nets [6] are a graph rewriting formalism where programs are represented as graphs and computation is based on graph rewriting. They enjoy good properties such as strong confluence, Turing completeness and locality of reduction. For these reasons, optimal [3,7] and efficient [9] λ-calculus evaluators based on interaction nets have evolved. Indeed, interaction nets have proved to be very fruitful in the study of the dynamics of computation. However, they are currently only useful for theoretical investigations.

In this paper we take a step towards developing a practical language for interaction nets. In the same way that functional languages are based on the λ-calculus, logic languages are based on Horn clauses, or the pict [10] language is based on the π-calculus, here we present a language based on this graph rewriting formalism and give a compilation into C.

There are several implementations of interaction nets [11,8,4,5], but they all suffer for one or more drawbacks: execution speed, lack of modern language constructs such as built-in types, input/output etc. The main goal of this paper is to address these issues so that we can shift the use of interaction nets from theoretical investigations to a practical programing paradigm. Firstly, we develop a textual syntax for interaction nets with higher level constructs that provide programing comfort. We then show how this language can be compiled down to native codes via the C

Layout based on the macro package of the
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

language [12]. C is a machine independent low-level language that is well suited as a portable target language for the implementation of programming languages. Over the years, C compilers have gone through many improvements to generate optimised machine code. By compiling to C, we also benefit in the improvements of C code generation. In addition, we gain instant portability because C is implemented on a variety of platforms. Many languages [14,1,13] have benefited from this line of compilation.

To summarise, the main contributions of this paper are as follows:

- We extend the definition of interaction nets to allow: built in data types and conditional rewrite rules; states and state transformers.
- We define a compiler from interaction nets to native codes via the C language.

The extensions will break some of the main theoretical properties of interaction nets, but our computations stay deterministic since we fix a particular strategy. Interaction nets are one-step confluent, which means that all reduction sequences to normal form are the same length: by picking one at random we are not affecting the efficiency of the system.

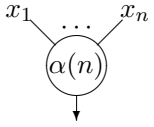
In our previous work [4] we defined a textual language for interaction nets and described how it is transformed into an intermediate language. We then defined an abstract machine that executes Inets instructions. This paper is concerned with: developing a richer language for interaction nets, extending the language to cater for the introduced source language constructs and compiling into C code.

In the next section we present some background material on interaction nets. We discuss our source language in Section 3. In Sections 4 and 5 we define the compilation schemes from our source language to C. We conclude the paper in Section 6.

2 Interaction nets

Here we review the basic notions of interaction nets. We refer the reader to [6] for a more detailed presentation. Interaction nets are specified by the following data:

- A set Σ of *symbols*. Elements of Σ serve as *agent* (node) labels. Each symbol has an associated arity *ar* that determines the number of its *auxiliary ports*. If $ar(\alpha) = n$ for $\alpha \in \Sigma$, then α has $n+1$ *ports*: n auxiliary ports and a distinguished one called the *principal port*. Each agent may have attributes. In this paper, we will restrict attributes to just base types: integers and booleans, and we write the attribute in brackets after the name.



  We can represent this agent textually as $x_0 \sim \alpha(n)[x_1, \ldots, x_n]$, where x_0 is the principal port.
- A *net* built on Σ is an undirected graph with agents at the vertices. The edges of the net connect agents together at the ports such that there is only one edge

at every port. A port which is not connected is called a *free port*. A set of free ports is called an *interface*.

- Two agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected via their principal ports form an *active pair* (analogous to a redex). An interaction rule $((\alpha, \beta) \Longrightarrow N) \in \mathcal{R}$ replaces the pair (α, β) by the net N. All the free ports are preserved during reduction, and there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.

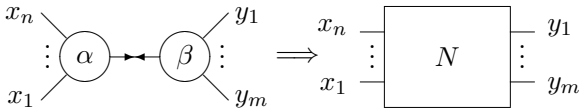


Figure 1 gives a simple example of an interaction net system that encodes the addition operation. We represent numbers using agents `S` and `Z`, corresponding to the usual constructors. Figure 2 gives an example reduction sequence that shows how a net representing $1 + 1$ is reduced to 2.

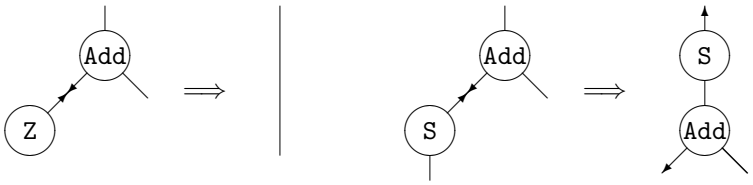


Fig. 1. Rules for addition

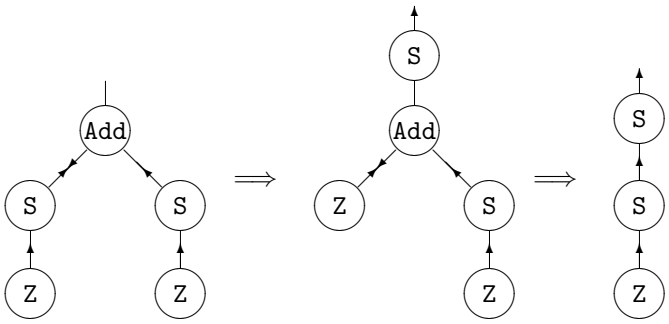


Fig. 2. Example reduction sequence

3 The source language - Inets

Following [2], an interaction net system can be described as a configuration $c = (\Sigma, \Delta, \mathcal{R})$, where Σ is a set of symbols, Δ is a multiset of active pairs, and $\mathcal{R}$ is a set of rules. A language for interaction nets needs to capture each component of the configuration, and provide ways to structure and organise the components. Starting from a calculus for interaction nets we build a core language. A core language can be seen both as a programming language and as a target language where we can compile high-level constructs. Drawing an analogy with functional programming, we can write programs in the pure λ-calculus and can also use it as a target language

to map high-level constructs. In this way, complex high-level languages can be obtained which by their definition automatically get a formal semantics based on the core language.

We write nets textually as a comma separated list of agents. This just corresponds to a flattening of the net, and there are many different (equivalent) ways to do this depending on the order the agents are enumerated. As an example, we write the initial net in Figure 2 as:

```
a~Add[x,y], a~S[b], b~Z, y~S[v], v~Z.
```

This can be simplified by replacing equals for equals:

```
S[Z]~Add[x,S[Z]].
```

In this notation the general form of an active pair is $\alpha[\ldots] \sim \beta[\ldots]$. All variable names occur at most twice. If a name occurs once, then it corresponds to a free port of the net (x is free in the above). If a name occurs twice, then it represents an edge between two ports, in which case we say that the variable is *bound*.

We represent rules by writing $l \Longrightarrow r$, where l is an active pair on the left of the rule, and r is the resulting net. In particular, we note that l will always consist of two agents connected at their principal ports. We also note that all rules can be written in a form $\alpha(..)[..] \sim \beta(..)[..] \Longrightarrow N$, and as such we replace the '$\sim$' by '`><`' so that we can distinguish an occurrence of a rule from an occurrence of an active pair. For example, the rules for the addition operation in the Figure 1 can be represented using the syntax:

```
Add[x,y] >< Z =>  x~y
Add[x,y] >< S[a] =>  x~S[b], a~Add[b,y]
```

or in a more compact way:

```
Add[x,y] ><
 Z =>  x~y
 S[a] =>  x~S[b], a~Add[b,y]
```

The names of the bound variables in the two nets must be disjoint, and the free variables must coincide, which corresponds to the condition that the free variables must be preserved under reduction.

The introduction of agents with values provide us with an efficient representation of data types in interaction nets. For example, we can represent numbers in a way that is directly supported by hardware rather than using `S` and `Z` agents. We also introduce a set of operations on the built-in data types: booleans, integers and characters. The example rule below shows how we can encode the addition operation in a way that is directly supported by hardware.

```
Add(int x)[res] ><  Num(int y)[] => res~Num(x+y);
```

3.1 Conditional rewrite rules

We allow rules to contain multiple right-hand side (rhs) nets. If either (or both) of the interacting agents are holding a value, then we can use these values to give different rhs of the rule by specifying a condition. The conditions must be all disjoint

```
Fact[result] ><
 {
   Num(int x) =>
    if(x < 0)
      result~Error;
    else  if(x == 0)
       result~Num(1);
     else
      Fact[Mult(x)[result]]~Num(x-1) ;
 }
Mult(int x)[res] ><
 {
    Num(int y) =>
      res~Num(x*y);
 }
main(){
  Fact[result]~Num(6);
}
```

Fig. 3. Example program: factorial

(there can not be two rhs nets that can be applied). During reduction, we evaluate the conditional expression to determine which net will be applied to an active pair. Figure 3 gives an example program that computes the factorial of a number. The rule between the agents `Fact` and `Num` will rewrite depending on the boolean value.

3.2 State transformers

We allow our programs to be decorated with global and local states. A local variable is only visible within the scope of its definition that is in a rule or net. State transformers defined in a rule will be executed either before or after the rule is applied. The structure of a rule is given by:

$$\begin{array}{l} \alpha(..)[..] \;\; >< \\ \quad stmt1 \\ \beta(..)[..] \implies N \\ \quad stmt2 \end{array}$$

The sequence of statements $stm1$ will be executed before the application of the rule α, β and $stmt2$ will be executed after the rewrite. The example in Figure 3 can be written with state decorations:

```
int counter;
Fact[result] ><
 {
    Num(int x) =>
      counter = counter + 1;
     if(x < 0)
       result~Error;
     else  if(x == 0)
```

```
        result~Num(1);
      else
        Fact[Mult(x)[result]]~Num(x-1) ;
  }
 Mult(int x)[res] ><
  {
     Num(int y) =>
       counter = counter + 1;
       res~Num(x*y);
 }
 main(){
   counter = 0;
   Fact[result]~Num(6);
   print "total number of interactions ",counter;
 }
```

In this example a global variable `counter` is declared that counts the number of interactions. When each rule is applied, the counter is incremented. The net named `main` is the entry point to the program. We initialise the global variable `counter` followed by an active pair definition and a print statement.

4 Representing interaction nets in C

Definition 4.1 Let $Loc \subseteq \mathbb{N}$ be a set of memory locations. We define a set of agent nodes $Agent = \{(Id \times k \times P \times V)\}$ where Id is an identifier that represents the name of the agent, k is an integer that indicates the number of values an agent holds, V is the set of values that an agent holds, and P is a set of ports. Each port $p \in P$ is a pair (l_a, n) where $l_a \in Loc$ is a pointer to another agent node, n is the port that the other node connects to this. The heap $H : Loc \rightarrow Agent$ returns a node $a \in Agent$ given some location $l \in Loc$.

We represent an agent node graphically using:

Id	k	p_0	p_1	$\cdots$	p_{ar}	v_1	$\cdots$	v_k

where ar is the arity of the agent. The port p_0 represents the principal port. It is straightforward to represent this memory model in the language C. We use the following C structure to represent *agent* nodes:

```
typedef struct Agent{
   unsigned int Id;
   unsigned int k
   struct Port *port;
   union V *value;
}Agent;

typedef struct Port{
   unsigned int portNum;
   unsigned int agent;
```

```
}Port;

typedef union V{
   char char_value;
   int int_value;
   float float_value;
}V;
```

The C union `V` contains the primitive types in our source program. The type `Net` is translated into an integer in C. We represent the heap as an array of *agent* nodes. This representation of nets does not use variable nodes. Consequently, the model cannot represent nets that are formed from *wires* only (for example `x~y` and `x~x`).

We use a function `mkAgent` : $Id \times \mathbb{N} \times \mathbb{N} \rightarrow Loc$ that given an Id, the arity ar and the value k will construct an agent node in the heap and return its location $l \in dom(H)$. In the rest of this document, we omit the last argument k to the function `mkAgent` when it is 0.

We define two functions that manipulate agent nodes:

(i) `connect` : $Loc \times \mathbb{N} \times Loc \times \mathbb{N} \rightarrow Void$ that connects two agent ports. For example, if we have the agents:

$$H(l_\alpha) = \boxed{\alpha \mid 0 \mid p_0 \mid p_1 \mid \cdots \mid p_n}$$

$$H(l_\beta) = \boxed{\beta \mid 0 \mid p_0 \mid p_1 \mid \cdots \mid p_m}$$

then `connect`$(l_\alpha, 1, l_\beta, 0)$ will transform the structure of the nodes to:

$$H(l_\alpha) = \boxed{\alpha \mid 0 \mid p_0 \mid (l_\beta, 0) \mid \cdots \mid p_n}$$

$$H(l_\beta) = \boxed{\beta \mid 0 \mid (l_\alpha, 1) \mid p_1 \mid \cdots \mid p_m}$$

where $ar(\alpha) = n$, $ar(\beta) = m$. The updated nodes above represent the net $\alpha[\beta[s_1, \ldots, s_m], t_2, \ldots, t_n]$. The function `connect` updates the connection information in the ports of two agent nodes. We can represent this function using the following C macro definition:

```
#define connect(a1,p1,a2,p2)          \
   heap[a1].port[p1].agent   = a2;    \
   heap[a1].port[p1].portNum = p2;    \
   heap[a2].port[p2].agent   = a1;    \
   heap[a2].port[p2].portNum = p1;    \
   if(p1 == 0 && p2 == 0) pushActive(a1,a2)
```

The conditional statement checks if we are connecting principal ports and subsequently pushes the two (interacting) agents into a stack S of active pairs.

We can build nets using the instructions `mkAgent` and `connect` defined above. Below we give an example sequence of instructions that will construct in memory the net `p~Add[S[Z],y]`.

l_{Add} `= mkAgent (`Add`,2)`

l_S = mkAgent (S,1)
l_Z = mkAgent (Z,0)
connect(l_{Add},1,l_S,0)
connect(l_S,1,l_Z,0)

Note that the ports which have no connections represent the interface of the net. The ports p and y are the free ports in the example net above. As another example, the cyclic net x∼B[x] can be represented using:
l_B = mkAgent (B,1)
connect($l_B, 0, l_B, 1$)

(ii) The function getPort : $Loc \times \mathbb{N} \rightarrow Loc \times \mathbb{N}$ returns the connection information stored in a port of some agent. getPort$(l_a, n) = (l_b, m)$ where $l_a, n \in Loc \times \mathbb{N}$ and $l_b, m \in Loc \times \mathbb{N}$. As a consequence of the connect function, if getPort$(l_a, n) = (l_b, m)$ then getPort$(l_b, m) = (l_a, n)$. We can represent the function getPort using the following C macro definition:
#define getPort(agent,port) (heap[agent].port[port])

To represent a rule, we construct the rhs net of the rule and connect it to the auxiliary agents of the active pair. The rewiring is accomplished with the help of the function getPort which is used to *fetch* the auxiliary agents that will connect to the rhs net. As an example, the sequence of instructions below will represent the rule:

Add[x,y] >< S[a] => x∼S[b],a∼Add[b,y]

(l_x, p_x) = getPort($l_{a_{Add}}$,1)
l_s = mkAgent(S,1)
connect(l_s,0,l_x, p_x)
(l_a, p_a) = getPort(l_{a_S},1)
l_{Add} = mkAgent(Add,2)
connect(l_{Add},0,l_a, p_a)
connect(l_{Add},1,l_s,1)
(l_y, p_y) = getPort($l_{a_{Add}}$,2)
connect(l_{Add},2,l_y, p_y)

we assume $l_{a_{Add}}$ and l_{a_S} to be the locations of the active pair agents Add and S respectively.

5 Compilation

In this section we define the compilation schemes from our source language to C source code. We use existing C compilers to translate the generated C source files to native codes. When executed, the generated codes will build the corresponding net in memory and reduce it to full normal form.

The basic model is that we compile each rule and each net to a C function. The functions generated for rules take a pair of (active) agents as parameters. They contain code that will build the rhs net of a rule and wire it to the agents that are connected to the auxiliary ports of the active pair.

5.1 Runtime Environment

The compiled C files need to be linked with a run-time library. That library contains the internal Inets primitives, runtime error reporting routines and definition of the runtime data areas. The generated files contain a main method that calls for the initialisation of the three runtime data areas:

- the heap H. All agents are allocated in the heap.
- the evaluation stack S, contains pointers to active pairs. Evaluation pops a pointer to an active pair, evaluates the pair and pushes any newly created pairs into S.
- the *rule table* R maps pointers to functions generated for the rules. The evaluation function examines this table to select the appropriate function to reduce a given pair.

We define R in C using:

```
typedef void(*RuleFun)();
RuleFun R[MAX_RULES];
```

For simplicity, we assume a pre-defined constant `MAX_RULES` that gives the maximal number of rules in a given program. For each function that a rule generates, we create an entry of the function name in the table `R`. We shall see later that function names are formed from an ordered concatenation of active pair names. Since there can only be one interaction rule for any pair of agents, all function names that are generated for a rule are unique. The injective function *hash* takes a string (function name) and returns a unique integer i, $0 \leq i \leq$ `MAX_RULES`. The *hash* function ensures that each entry in `R` has a unique index.

The evaluation function *eval* reduces a given net to normal form. It pops an active pair (α, β) from S, examines the rule table R (by computing *hash* of the ordered concatenation of the active pair names (α, β)) to determine the appropriate function to invoke. The function is invoked with the arguments (α, β). Evaluation stops once S is empty.

5.2 Compilation schemes

Here, we present the compilation function $\mathcal{T}$ that will translate our source language into the intermediate language C. The environment Γ maps identifiers to memory locations $l \in Loc$. Intuitively, Γ binds agents to their memory locations. We write $[]$ for the empty map and $\Gamma(x) = \bot$ when there is no entry for x in Γ. We use the following notation:

$$\Gamma[x \mapsto s](z) = \begin{cases} s & \text{if } z = x \\ \Gamma(z) & otherwise \end{cases}$$

The function *fresh* returns a unique string. It is used to generate fresh variables for our target language. The notation $\{x\}$ will replace the variable x with its actual value. For example, if $x =$ "*abc*", then "123$\{x\}$456" = "123*abc*456". The given value of x may be an integer. We write $\{p_1\} +$ "*str*" to concatenate the value of the string variable p_1 to the sequence of characters *str*.

The compilation of Inets into C is governed by the schemes: $\mathcal{T}_{inet}$ compiles a program, $\mathcal{T}_{exp}$ compiles expressions, $\mathcal{T}_{ns}$ compiles nets and $\mathcal{T}_{rs}$ compiles rules. We will now look at each of these schemes in turn.

The scheme $\mathcal{T}_{inet}$ calls for the compilation of an Inets program composed of a set of rules, nets and state declarations.

$$\mathcal{T}_{inet}(\Sigma, \langle e_1, \ldots, e_k \rangle, \langle n_1, \ldots, n_n \rangle, \mathcal{R}) = \begin{cases} \mathcal{T}_{exps}(e_1, \ldots e_k,);\ \mathcal{T}_{ns}(n_1, \ldots, n_n); \\ \mathcal{T}_{rs}(r_1); \ldots; \mathcal{T}_{rs}(r_n);\ \texttt{initRules} \end{cases}$$

where Σ is a set of symbols, $r_1, ..., r_n = \mathcal{R}$ are instances of rules, each e_i is a state declarations and each n_i is a net definition. The function `initRules` generates code that will fill the table R with function pointers.

$$\texttt{initRules} = \begin{cases} \text{let } prog_1 = map(\alpha_1, \beta_1); \ldots; prog_n = map(\alpha_n, \beta_n) \\ \text{in } \text{“}\texttt{void initRules() \{}\text{”} + \{prog_1\} + \ldots + \{prog_n\} + \text{“}\}\text{”} \\ \text{end} \end{cases}$$

$$map(\alpha, \beta) = \text{“}\texttt{R[}\text{”} + hash(\{\alpha\} + \{\beta\}) + \text{“}\texttt{] = ``}\{\alpha\} + \{\beta\}\text{”}\texttt{;}\text{”}$$

where the active pair agents (α_i, β_i) are the interacting agents for each rule $r_i \in \mathcal{R}$. The string $\{\alpha\} + \{\beta\}$ is an ordered concatenation of the active pair names.

The compilation scheme $\mathcal{T}_{ns}(n_1, \ldots, n_n)$ compiles a sequence of net definitions:

$$\mathcal{T}_{ns}(n_1, \ldots, n_n) = \mathcal{T}_n(n_1); \ldots; \mathcal{T}_n(n_n);$$

A net named `myNet` with formal parameters $x_1, \ldots, x_n$ and body P translates to a C function named `myNet`:

$$\mathcal{T}_n(\texttt{myNet}(x_1 : T_1, \ldots, x_n : T_n)\{P\}) = \text{let } prog_1 = \mathcal{T}_b(P) \text{ in } \text{“}\texttt{void myNet(}T_1\ x_1, \ldots, T_n\ x_n\texttt{)\{}\text{”} + \{prog_1\} + \text{“}\}\text{”} \text{ end}$$

where the parameter x_i is of type T_i. The scheme $\mathcal{T}_b(P)$ translates the body of a net definition. From the grammar of the language, the body of a net may contain a list of expressions and/or a list of active pairs. A C function named `main` will be generated: “`int main() {...}`”, which is the entry point for the execution of our generated program and allow the codes to run as stand alone programs. Our main function simply calls for the execution of the codes generated for the main net.

The scheme $\mathcal{T}_t$ emits codes that will construct a term in memory. Given a term T which is not a variable, the scheme $\mathcal{T}_t$ generates code that will: create the root agent, construct the sub-terms t_i that connect to the auxiliary port of the root agent, and finally connect the sub-terms to the appropriate auxiliary port. If t_i is

an agent $\alpha \in \Sigma$, it will connect to its *parent* agent via its principal port (port 0).

$$\mathcal{T}_t(\alpha(e_1 : T_1, ..., e_m : T_m)[t_1, \ldots, t_n], l, p, \Gamma) =$$

$$\begin{cases} \text{let } alpha = fresh \\ \quad prog_0 = \text{ “}\texttt{int } \{alpha\}\texttt{= mkAgent(}\{\alpha\}, n, m\texttt{)}\text{”} \\ \quad prog_v = \text{“}\{alpha_1\}\texttt{.value[1].}T_1\texttt{=}\{e_1\}\texttt{;”} + \cdots + \\ \quad \text{“}\{alpha_1\}\texttt{.value[}m\texttt{].}T_m\texttt{=}\{e_m\}\texttt{;”} + (prog_1, (l_1, p_1), \Gamma_1) = \mathcal{T}_t(t_1, alpha, 1, \Gamma), \\ \quad \cdots, (prog_n, (l_n, p_n), \Gamma_n) = \mathcal{T}_t(t_n, alpha, n, \Gamma_{n-1}) \\ \text{in} \\ \quad (\{prog_0\} + \{prog_1\} + \texttt{connect(}\{alpha\}, 1, l_1, p_1\texttt{)} + \cdots + \\ \quad \{prog_n\} + \texttt{connect(}\{alpha\}, n, l_n, p_n\texttt{)}, \{alpha\}, 0, \Gamma_n) \\ \text{end} \end{cases}$$

where the expression e_i is of type T_i.

The compilation of a variable x does not generate any code. We consider two cases to compile variable nodes: 1) when $\Gamma(x) = \bot$, we create an entry $\Gamma[x \mapsto (l, p)]$ that maps the variable name x to the location and port number of the (*parent*) agent node that the variable wants to connect to. 2) when $\Gamma(x) = (l_2, p_2)$, we use the pair (l_2, p_2) to connect to the parent node of the variable. This scheme provides a mechanism to connect the ports of agents in a direct way other than through variable nodes.

$$\mathcal{T}_t(x, l, p, \Gamma) = \begin{cases} \text{if } (\Gamma(x) = \bot \vee x \notin dom(\Gamma)) \text{ then}(-, l, p, \Gamma[x \mapsto (l, p)]) \\ \text{else let } (l_x, p_x) = \Gamma(x) \text{ in } (-, l_x, p_x, \Gamma[x \mapsto \bot])\text{end} \end{cases}$$

We use the scheme $\mathcal{T}_{exp}$ to translate a list of expressions.

$$\mathcal{T}_{exps}(e_1, \ldots, e_n) = \mathcal{T}_{exp}(e_1); \ldots; \mathcal{T}_{exp}(e_n);$$

$$\mathcal{T}_{exp}(e1 \ op \ e2) = \begin{cases} \text{let } r_1 = \mathcal{T}_{exp}(e1)\texttt{;}, \quad r_2 = \mathcal{T}_{exp}(e2)\texttt{;} \\ \text{in } \{r_1\} + op + \{r_2\} \\ \text{where } op \in \{+, -, /, *, =, <, >, <=, >=, \%, !=\} \\ \text{end} \end{cases}$$

$$\mathcal{T}_{exp}(\texttt{print } e_1, \ldots, e_n) =$$

$$\begin{cases} \text{let } r_1 = \mathcal{T}_{exp}(e_1)\texttt{;}, \ldots, \quad r_n = \mathcal{T}_{exp}(e_n)\texttt{;} \\ \text{in “}\texttt{printf(}F\text{”} + r_1 + \text{“}\texttt{);”} + \ldots + \text{“}\texttt{printf(}F\text{”} + r_n + \text{“}\texttt{);”} \\ \quad \text{where } F = \text{“}\%d\text{” if r}_\text{i} \text{ is an integer} \\ \qquad F = \text{“}\%c\text{” if r}_\text{i} \text{ is a character} \\ \text{end} \end{cases}$$

$$\mathcal{T}_{exp}(n) = \text{“}\{n\}\text{”};$$
where $n \in \mathbb{Z}$ or n is a variable name

The scheme $\mathcal{T}_{eqs}$ translates a list of active pairs. We use $\mathcal{T}_{eq}$ to generate the code for each active pair.

$$\mathcal{T}_{eqs}(u_1 \sim v_1, \ldots u_n \sim v_n, \Gamma) = \begin{cases} \text{let } (prog_1, l, p, \Gamma_1) = \mathcal{T}_{eq}(u_1 \sim v_1, -, -, \Gamma), \ldots, \\ \quad (prog_n, l, p, \Gamma_n) = \mathcal{T}_{eq}(u_n \sim v_n, -, -, \Gamma_{n-1}); \\ \text{in } (\{prog_1\} + \{prog_2\} + \ldots + \{prog_n\}, \Gamma_n) \\ \text{end} \end{cases}$$

For each active pair, we use the scheme $\mathcal{T}_{eq}$ to generate the code for each interacting agent.

$$\mathcal{T}_{eq}(\alpha(V_\alpha)[u_1, \ldots, u_n] \sim \beta(V_\beta)[v_1, \ldots, v_y], l, p, \Gamma) = \begin{cases} \text{let } (prog_1, (l_1, p_1), \Gamma_1) = \mathcal{T}_t(\alpha(V_\alpha)[u_1, \ldots, u_n], l, p, \Gamma), \\ \quad (prog_2, (l_2, p_2), \Gamma_2) = \mathcal{T}_t(\beta(V_\beta)[v_1, \ldots, v_y], l, p, \Gamma_1) \\ \text{in } (\{prog_1\} + \{prog_2\} + \texttt{connect(}l_1, p_1, l_2, p_2\texttt{)+ eval()}, l, p, \Gamma_2) \\ \text{end} \end{cases}$$

$$\mathcal{T}_{eq}(x \sim \alpha(V_\alpha)[u_1, \ldots, u_n], l, p, \Gamma) = \begin{cases} \text{let } (prog_1, (l_1, p_1), \Gamma_1) = \mathcal{T}_t(\alpha(V_\alpha)[u_1, \ldots, u_n], l, p, \Gamma), \\ \quad (prog_2, (l_2, p_2), \Gamma_2) = \mathcal{T}_t(x, l_1, p_1, \Gamma_1) \\ \text{in } (\{prog_1\} + \{prog_2\} + \texttt{connect(}l_1, p_1, l_2, p_2\texttt{)}, l, p, \Gamma_2) \\ \text{end} \end{cases}$$

The compilation scheme $\mathcal{T}_{rs}$ compiles a rule definition. We use the scheme $\mathcal{T}_r$ to generate a C function for each binary rule. This function contains code that will

build the rhs net and wire it to the corresponding auxiliary ports of the active pair.

$$\mathcal{T}_{rs}(\alpha(x_1:T_1,\ldots,x_j:T_j)[t_1,\ldots,t_n] \texttt{ >< } e_1\ldots,e_m\ \{r_1,\ldots,r_k\}) = \begin{cases} \mathcal{T}_r(r_1,\alpha(x_1:T_1,\ldots,x_j:T_j)[t_1,\ldots,t_n],e_1,\ldots,e_n); \\ \quad\vdots \\ \mathcal{T}_r(r_k,\alpha(x_1:T_1,\ldots,x_j:T_j)[t_1,\ldots,t_n],e_1,\ldots,e_n); \end{cases}$$

where each r_i is either of the form

$$\beta(y_1:T_{y_1},\ldots,y_n:T_{y_n})[s_1,\ldots,s_n]\Rightarrow e_1,\ldots,e_k,eqs,If$$

or

$$\{e_1,\ldots,e_m,\beta(y_1:T_{y_1},\ldots,y_n:T_{y_n})[s_1,\ldots,s_n]\Rightarrow e_{m+1},\ldots,e_n,eqs,If\},$$

$eqs = u_1 \sim v_1,\ldots,u_n \sim v_n,$

$If = \texttt{If}\ (b)\ eqs_1\ \texttt{else}\ If, eqs_2.$

The remainder of this section gathers together the schemes that will generate codes for a rule.

$$\mathcal{T}_r(\beta(y_1:T_{y_1},\ldots,y_k:T_{y_k})[u_1,\ldots,u_n]\Rightarrow e_1,\ldots,e_k,\ eqs,\ If,\ \alpha(x_1:T_1,\ldots,x_l:T_l)[t_1,\ldots,t_m],e_{k+1},\ldots,e_l) = \begin{cases} \text{let } alpha = fresh;\ beta = fresh; \\ \quad prog_0 = \text{“}\{T_{y_1}\}\ \{y_1\}\texttt{;”} + \ldots + \text{“}\{T_{y_k}\}\ \{y_k\}\texttt{;”}+ \\ \quad \text{“}\{T_1\}\ \{x_1\}\texttt{;”} + \ldots + \text{“}\{T_l\}\ \{x_l\}\texttt{;”} \\ \quad prog_1 = \mathcal{T}_{exps}(e_l,\ldots,e_k,\ldots,e_1); \\ \quad prog_2 = rewrite(\beta[u_1,\ldots,u_n],\alpha[t_1,\ldots,t_m],\ eqs,\ alpha,beta,[]) \\ \quad prog_3 = \mathcal{T}_{if}(If,\beta[u_1,\ldots,u_n],\alpha[t_1,\ldots,t_m],alpha,beta) \\ \text{in} \\ \quad \text{“}\texttt{void}\ \{\alpha\} + \{\beta\}\texttt{(int}\ \{alpha\},\texttt{int}\ \{beta\}\texttt{)\{”}+ \\ \quad \{prog_0\} + \text{“}\{y_1\}\ \texttt{=}\ \{beta\}\texttt{.value[1].}T_{y_1}\texttt{;”} + \ldots + \\ \qquad \text{“}\{y_k\}\ \texttt{=}\ \{beta\}\texttt{.value[}k\texttt{].}T_{y_k}\texttt{;”}+ \\ \quad \text{“}\{x_1\}\ \texttt{=}\ \{alpha\}\texttt{.value[1].}T_1\texttt{;”} + \ldots + \\ \qquad \text{“}\{x_l\}\ \texttt{=}\ \{alpha\}\texttt{.value[}l\texttt{].}T_l\texttt{;”}+ \\ \quad \{prog_1\} + \{prog_2\} + \{prog_3\} + \text{“}\texttt{\}”} \\ \text{where } If = \texttt{if}\ (b)\ eqs_1\ \texttt{else}\ If, eqs_2 \\ eqs = u_1 \sim v_1,\ldots,u_n \sim v_n, \\ \text{end} \end{cases}$$

$$\mathcal{T}_r(\{e_1,\ldots,e_m\ ,r,e_{m+1},\ldots,e_n\}\alpha(x_1:T_1,\ldots,x_n:T_n)[t_1,\ldots,t_m]) =$$
$$\begin{cases} \mathcal{T}_{exps}(e_1,\ldots,e_m) \\ \mathcal{T}_r(r,\alpha(x_1:T_1,\ldots,x_n:T_n)[t_1,\ldots,t_m]) \\ \mathcal{T}_{exps}(e_{m+1},\ldots,e_n) \end{cases}$$

$$\mathcal{T}_{if}(\texttt{if}\ (b)\ eqs_1\ \texttt{else}\ If, eqs_2, \alpha[t_1,\ldots,t_n], \beta[s_1,\ldots,s_n], alpha, beta) =$$
$$\begin{cases} \text{let } label = fresh, \quad next = fresh \\ \quad prog_1 = \mathcal{T}_{exp}(b) \\ \quad prog_2 = rewrite(\alpha[t_1,\ldots,t_n], \beta[s_1,\ldots,s_n], eqs_1, alpha, beta, []) \\ \quad prog_3 = \mathcal{T}_{if}(If) \\ \quad prog_4 = rewrite(\alpha[t_1,\ldots,t_n], \beta[s_1,\ldots,s_n], eqs_2, alpha, beta, []) \\ \text{in } \text{“}\texttt{if (!}\text{”} + \{prog_1\} + \text{“}\texttt{) }\{label\}\texttt{;}\text{”} + \{prog_2\} + \text{“}\texttt{goto }\{next\}\texttt{;}\text{”} \\ \quad \text{“}\{label\}\texttt{:}\text{”} + \{prog_3\} + \{prog_4\} + \text{“}\texttt{goto }\{next\}\texttt{;}\text{”} + \text{“}\{next\}\texttt{:}\text{”} \\ \text{end} \end{cases}$$

$$rewrite(\alpha[x_1,\ldots,x_k], \beta[y_1,\ldots,y_m], t_1 \sim s_1,\ldots,t_n \sim s_n, alpha, beta, \Gamma) =$$
$$\begin{cases} \text{let } \mathcal{N} = \{x_1\ldots,x_n\} \cup \{y_1,\ldots,y_m\} \\ \quad \Gamma[x_1 \mapsto (alpha,1),\ldots,x_n \mapsto (alpha,n), \\ \quad y_1 \mapsto (beta,1),\ldots,y_m \mapsto (beta,m)] \\ \quad (prog_{t_1},(l_{t_1},p_{t_1}),\Gamma_1) = \mathcal{T}_{t_r}(t_1,-,-,\Gamma,\mathcal{N}) \\ \quad (prog_{s_1},(l_{t_n},p_{t_n}),\Gamma_2) = \mathcal{T}_{t_r}(s_1,-,-,\Gamma_1,\mathcal{N}) \\ \qquad \vdots \\ \quad (prog_{t_n},(l_{s_1},p_{s_1}),\Gamma_{j+1}) = \mathcal{T}_{t_r}(t_n,-,-,\Gamma_j,\mathcal{N}), \\ \quad (prog_{s_n},(l_{s_n},p_{s_n}),\Gamma_k) = \mathcal{T}_{t_r}(s_n,-,-,\Gamma_{k-1},\mathcal{N}) \\ in \\ \quad (\{prog_{t_1}\} + \{prog_{s_1}\} + \texttt{connect(}l_{t_1},p_{t_1},l_{s_1},p_{s_2}\texttt{)} + \ldots + \\ \quad \{prog_{t_n}\} + \{prog_{s_n}\} + \texttt{connect(}l_{t_n},p_{t_n},l_{s_n},p_{s_n}\texttt{)},l,p,\Gamma_2) \end{cases}$$

$$\mathcal{T}_{tr}(x, l, p, \Gamma, \mathcal{N}) =$$
$$\left\{\begin{array}{l} \text{let } (l_x, p_x) = \Gamma(x) \\ \quad prog_1 = \text{“}\texttt{Port p = getPort(}l_x, p_x\texttt{);”}, \; l_a = \text{“}\texttt{p.agent”}, \\ \quad\quad p_l = \text{“}\texttt{p.portNum”} \\ in \\ \quad \text{if } (\Gamma(x) = \bot \vee x \notin dom(\Gamma)) \text{ then} \quad (-, l, p, \Gamma[x \mapsto (l, p)]) \\ \quad \text{else} \\ \quad\quad \text{if } (x \in \mathcal{N}) \text{ then} \\ \quad\quad\quad (prog_1, l_a, p_l, \Gamma) \\ \quad\quad \text{else } (-, l_x, p_x, \Gamma[x \mapsto \bot]) \end{array}\right.$$

$$\mathcal{T}_{tr}(\alpha[t_1, \ldots, t_n], l, p, \Gamma, \mathcal{N}) =$$
$$\left\{\begin{array}{l} \text{let } alpha = fresh \\ \quad prog_0 = \text{“}\{alpha\}\texttt{= mkAgent(}\{\alpha\}, n\texttt{);”} \\ \quad (prog_1, (l_1, p_1), \Gamma_1) = \mathcal{T}_{tr}(t_1, alpha, 1, \Gamma, \mathcal{N}), \ldots, \\ \quad (prog_n, (l_n, p_n), \Gamma_n) = \mathcal{T}_{tr}(t_n, alpha, n, \Gamma_{n-1}, \mathcal{N}) \\ in \\ \quad (\{prog_0\} + \{prog_1\} + \texttt{connect(}\{alpha\}, 1, l_1, p_1\texttt{)} + \ldots + \\ \quad \{prog_n\} + \texttt{connect(}\{alpha\}, n, l_n, p_n\texttt{)}, \\ \quad \{alpha\}, 0, \Gamma_n) \end{array}\right.$$

We conclude this section with an example hand-compiled code generated for the main net given in the example program in Section 3.2:

```
void main(){
  counter = 0;
  int Fact = mkAgent("Fact",1);
  int Num = mkAgent("Num",0);
  Num.value[1].int_value = 6;
  connect(Fact,0,Num,0);
  eval();
  printf("total number of interactions");
  printf("%d",counter);
 }
```

6 Conclusions

In this paper we have presented our language for interaction nets, and given a compilation into C. We have implemented this language, which is available from the

web page: http://www.interaction-nets.org/, and is one of the main building blocks for building a programming environment for interaction nets. Current work is focussed on giving a formal operational semantics of this language, and also building a richer set of programming tools.

References

[1] D. Diaz. Wamcc: Compiling prolog to c. In *In 12th International Conference on Logic Programming*, pages 317–331. MIT PRess, 1995.

[2] M. Fernández and I. Mackie. A calculus for interaction nets. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, number 1702 in LNCS, pages 170–187. Springer-Verlag, September 1999.

[3] G. Gonthier, 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)*, pages 15–26. ACM Press, Jan. 1992.

[4] A. Hassan, I. Mackie, and S. Sato. Interaction nets: programming language design and implementation. In *Proceedings of the seventh international workshop on Graph Transformation and Visual Modeling Techniques*, March 2008.

[5] M. J.B. Almeida, J.S.Pinto. A tool for programming with interaction nets. Technical report, University of Minho, 2006.

[6] Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, Jan. 1990.

[7] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 16–30. ACM Press, Jan. 1990.

[8] S. Lippi. in^2 : A graphical interpreter for interaction nets. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 380–386. Springer, 2002.

[9] I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, September 1998.

[10] B. C. Pierce and D. N. Turner. Pict: a programming language based on the pi-calculus. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction*, pages 455–494. The MIT Press, 2000.

[11] J. S. Pinto. Parallel evaluation of interaction nets with mpine. In A. Middeldorp, editor, *RTA*, volume 2051 of *Lecture Notes in Computer Science*, pages 353–356. Springer, 2001.

[12] D. M. Ritchie. The c programming language, 1988.

[13] D. Tarditi, P. Lee, and A. Acharya. No assembly required: Compiling Standard ML to C. Technical report, ACM Letters on Programming Languages and Systems, 1990.

[14] G. Wong. Compiling erlang via C. Technical report, 1998.

Iterators, Recursors and Interaction Nets

Ian Mackie [1]

LIX, École Polytechnique
91128 Palaiseau Cedex, France

Jorge Sousa Pinto, Miguel Vilaça [2]

CCTC / Departamento de Informática
Universidade do Minho
4710-057 Braga, Portugal

Abstract

We propose a method for encoding iterators (and recursion operators in general) using interaction nets. There are two main applications for this: the method can be used to obtain a visual notation for functional programs, in a visual programming system; and it can be used to extend the existing translations of the λ-calculus into interaction nets (that have been proposed as efficient implementation mechanisms) to languages with recursive types. This work can also be seen as a study of the relation between interaction net programming and functional programming.

1 Introduction

Interaction nets have been extensively used to produce new, efficient implementation mechanisms for the λ-calculus [5,7,8]. On the other hand, the use of visual notations for functional programs has long been an active research topic, whose main goal is to have a notation that can be used (i) to define functional programs visually, and (ii) to animate visually the execution of functional programs.

In this paper we propose a graphical system for functional programming, based on token-passing interaction nets. The system offers an adequate solution for classic problems of visual notations, including the treatment of higher-order functions, pattern-matching, and recursion (based on the use of recursion operators). The system implements a call-by-name semantics, with a straightforward correspondence between functional programs and graphical objects.

Most approaches to visual programming simply propose a notation for programs. Program evaluation is animated by representing visually the intermediate programs

[1] Email: mackie@lix.polytechnique.fr

[2] Emails: {jsp,jmvilaca}@di.uminho.pt. The work of the 3rd. author was funded by FCT grant SFRH / BD / 18874 / 2004.

Layout based on the macro package of the
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

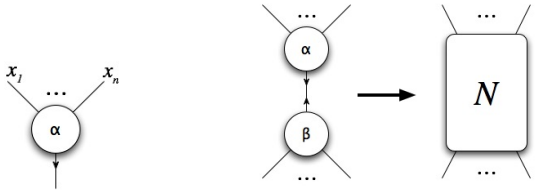


Fig. 1. An agent and an interaction rule

that result from executing reduction steps on the initial program, using the operational semantics of the underlying functional language. Our approach differs from this in that we use a graph-rewriting formalism with its own operational semantics.

Technically the main contribution of the paper is an extension of Sinot's token-passing implementation of the λ-calculus [10] to typed languages with recursive types and recursive function definitions based on recursion operators. We illustrate our ideas using the simply-typed λ-calculus with booleans, natural numbers, lists, and their respective iterators, but in fact the system can be extended smoothly to arbitrary polynomial types. Call-by-name evaluation is used for this language, but call-by-value and call-by-need could easily be obtained by building on previous results by Sinot. The token-passing encoding of the λ-calculus, to be combined with the encoding of recursion patterns proposed here, is just meant to be an illustrative choice for its simplicity and standard strategies.

An interesting feature of the work presented in this paper is that it results in interaction systems that are very similar to the typical examples of ("direct") interaction net programs. In this sense our work justifies semantically a functional subset of interaction nets. Moreover this provides further evidence that our approach is indeed an appropriate and natural way to represent functional programs visually.

The paper is structured as follows: Sections 2 and 3 contain background material on visual programming with interaction nets and on the token-passing encoding of the λ-calculus. Section 4 defines the functional language used in the paper. Sections 5 and 6 contain the translation of functional programs into token-passing nets and examples of its use. Section 7 considers extensions of the language with other recursion operators. We conclude the paper in Section 8.

2 Interaction Nets

Interaction nets [6] are constrained graph rewriting systems that can still encode all the computable functions. Interaction nets provide a model of computation in a graphical setting. Programs are represented as particular kinds of graphs, and computation is expressed as graph transformations. Interaction net systems are user-defined, in the same way as term rewriting systems, by giving a signature Σ (a set of symbols with a given arity) and a set of interaction rules R. An occurrence of a symbol is called an *agent*. An agent with arity n has $n+1$ *ports*: a distinguished one, depicted by an arrow, called the principal port, and n auxiliary ports. Agents are represented graphically as shown in Figure 1, on the left.

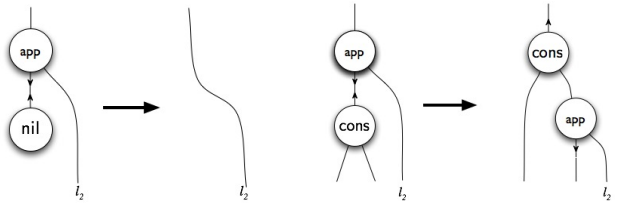
A net N built on a signature Σ is a graph (not necessarily connected) with agents at the vertices. The edges of the net connect agents together at the ports

such that there is only one edge at every port. Edges may connect two different ports of the same agent. The ports of an agent that are not connected to another agent are called the *free ports* of the net.

A pair of agents, say (α, β), connected on their principal ports is called an *active pair*, which is the interaction net analogue of a redex. An interaction rule replaces an occurrence of the active pair (α, β) by a net N. The rule has to satisfy a very strong condition: all the free ports are preserved during reduction, and moreover there is at most one rule for each pair of agents. The diagram shown on the right in Figure 1 illustrates the idea, where N is any net built from the signature.

An interaction net system is therefore fully defined by the pair (Σ, R). We say that a net is in normal form if it does not contain any active pairs. We use the notation $\Longrightarrow$ for one-step reduction and $\Longrightarrow^*$ for its transitive reflexive closure. Additionally, we write $N \Downarrow N'$ if there is a sequence of interaction steps $N \Longrightarrow^* N'$, such that N' is a net in normal form. The strong constraints on the definition of interaction rules imply that reduction commutes (the one-step diamond property holds), and thus confluence is easily obtained. Consequently, any normalizing interaction net is strongly normalizing.

The advantages of using interaction nets for visual programming can be understood by looking at a simple example. The following interaction rules define visually the behaviour of the list concatenation operation.



where the symbol app is used for concatenation agents, and nil and cons are the expected list constructors. The principal port of app is connected to the first list argument, and the result of the operation is obtained in the auxiliary port shown on top. This form of visual programming can be summarized as follows.

- Both programs and data are represented in the same simple graphical formalism.
- Programs can be animated without leaving the interaction formalism: instead of resorting to an external interpreter and then displaying the result of each evaluation step, a program can be animated by simply reducing the net. The reader can try this by connecting two lists of some type to an app agent and then applying the rules given above.
- Pattern-matching for external constructors is in-built.
- Recursive definitions are expressed very naturally as interaction rules involving agents (such as app) that are reintroduced on the right-hand side. Rule application then corresponds to the expansion of a recursive definition.

The above example is functional in nature: app can be written in a straightforward way as a function of two arguments that performs recursion on its first argument. But the interaction net formalism does not offer a satisfactory semantic interpretation for the behaviour of that symbol. Moreover, many interaction net

systems can be defined that do not have this functional reading.

What is missing is a clear correspondence between functional definitions and interaction systems like the one shown. In this paper we establish a correspondence between agents with "obviously functional" interaction rules like those given for app and functions defined with recursion operators.

We remark that the inherent inability of interaction nets to match constructors at a level deeper than one raises no problems: the simple form of pattern-matching available in interaction nets is sufficient to capture the behaviour of many powerful operators, such as recursors and accumulations, as will be shown in Section 7.

3 The Token-passing Encoding of the λ-calculus

A number of different translations of the λ-calculus into interaction nets exist. These have in common some basic principles. Let $\mathcal{T}(\cdot)$ be such a translation:

- Terms are translated into nets of a fixed interaction net system.
- Variables are translated simply as edges in $\mathcal{T}(t)$.
- If t is a closed λ-term then the net $\mathcal{T}(t)$ has one free port, corresponding to the *root* of the term, which will be drawn at the top of the net. If not, and $x_1 \dots x_n$ are free variables in t, then the net $\mathcal{T}(t)$ has n additional free ports (represented at the bottom) corresponding to each of the variables.
- $\mathcal{T}(\lambda x.t)$ is a net constructed structurally from $\mathcal{T}(t)$. This introduces an abstraction symbol λ at the root of the term, with ports linked to the edge representing the bound variable x and to the root of the abstraction body net, $\mathcal{T}(t)$. The special case of $x \notin \mathcal{FV}(t)$ is handled by introducing an *erasing agent* ε.
- $\mathcal{T}(t\,u)$ is a net constructed structurally from $\mathcal{T}(t)$ and $\mathcal{T}(u)$. This introduces an application symbol @ with ports connected to the root ports of $\mathcal{T}(t)$ and $\mathcal{T}(u)$. The special case of a free variable occurring in both terms is handled by introducing a *copying agent* c, with its two auxiliary ports connected to the edges representing the free variable in $\mathcal{T}(t)$ and $\mathcal{T}(u)$, and the edge connected to its principal port represents the variable in $\mathcal{T}(t\,u)$.

The *token-passing* encodings [10] use an interaction system where two different symbols exist for application: one is the syntactic symbol @ introduced by the translation; the corresponding agents have their principal ports facing the root of the term and will be depicted by triangles. A second symbol $\widehat{@}$ exists that will be used for computation; to simplify the figures, the corresponding agents will be depicted by circles equally labelled with @. Their principal ports face the net that represents the applied function, to make possible interaction with λ agents.

The translation $\mathcal{T}_{\text{tp}}(\cdot)$ encodes terms in the system $(\Sigma_{\text{tp}}, R_{\text{tp}})$ where $\Sigma_{\text{tp}} = \{\Downarrow, @, \widehat{@}, \lambda, c, \varepsilon, \delta\}$. The translation is shown in Figure 2 where $\mathcal{T}(.)$ stands for $\mathcal{T}_{\text{tp}}(\cdot)$. It generates nets containing no active pairs, so no reduction can happen. The special symbol $\Downarrow$ is used as an evaluation *token*: an agent $\Downarrow$ traverses the net, transforming occurrences of @ into $\widehat{@}$, thus triggering reductions. The evaluation rules involving $\Downarrow$ can be tailored to a specific evaluation strategy. For call-by-name, R_{tp} consists of the rules in Figure 3 (the arity of each symbol can be inferred from the rules). This

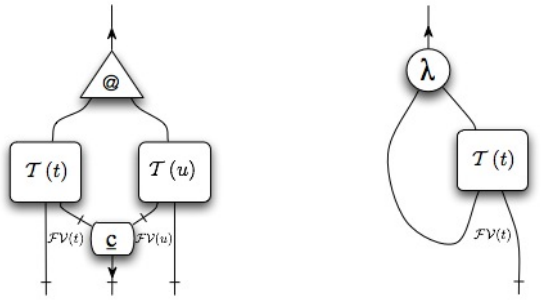


Fig. 2. The token-passing translation of λ-terms: the nets $\mathcal{T}(t\,u)$ and $\mathcal{T}(\lambda x.t)$. c denotes an array of c agents, one for each free variable occurring in both t and u. In $\mathcal{T}(\lambda x.t)$, a special case exists (not depicted) when the bound variable does not occur in the term: an ε agent must be connected to the λ agent instead.

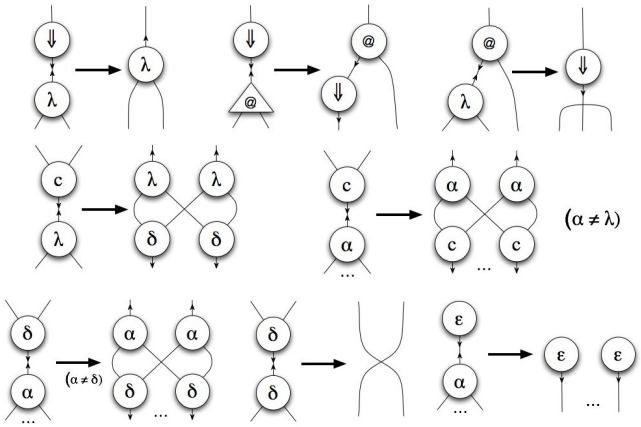


Fig. 3. The token-passing rules R_{tp}. Note the rule *templates* for (c, α), (δ, α), and (ε, α), which generate different rules for each instance of the agent α.

comprises evaluation rules involving ⇓, a computation rule involving @ and λ, and management (copying and erasing) rules. The symbol δ is a mutation of c used for copying abstractions.

To start the reduction (corresponding to normal order evaluation), a ⇓ symbol must be connected to the root port of the term. Let $\Downarrow N$ denote the net obtained by connecting a ⇓ agent to the root port of N; then the following correctness result holds: $t \Downarrow z$ iff $\Downarrow \mathcal{T}_{\text{tp}}(t) \longrightarrow^* \mathcal{T}_{\text{tp}}(z)$, where the evaluation relation $\cdot \Downarrow \cdot$ is defined by the standard evaluation rules for call-by-name:

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.t' \qquad t'[u/x] \Downarrow z}{t\,u \Downarrow z}$$

4 The language BNL

The language used in this paper is the simply-typed λ-calculus extended with natural numbers, booleans, lists, and iterators for these recursive types. BNL is defined by the following syntax for types and terms (x, y range over a set of variables):

$$\begin{array}{rcl} \tau, \sigma & ::= & \mathbf{Bool} \mid \mathbf{Nat} \mid \mathbf{List}(\tau) \mid \tau \to \sigma \\ t, u, v & ::= & x \mid \lambda x.t \mid t\,u \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{iterbool}(t,u,v) \mid 0 \mid \mathsf{suc}(t) \mid \mathsf{iternat}(\lambda x.t, u, v) \\ & \mid & \mathsf{nil} \mid \mathsf{cons}(t,u) \mid \mathsf{iterlist}(\lambda xy.t, u, v) \end{array}$$

and by the typing rules given by:

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau} \qquad \frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash t\,u : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{tt} : \mathbf{Bool}} \quad \frac{}{\Gamma \vdash \mathsf{ff} : \mathbf{Bool}}$$

$$\frac{}{\Gamma \vdash 0 : \mathbf{Nat}} \quad \frac{\Gamma \vdash t : \mathbf{Nat}}{\Gamma \vdash \mathsf{suc}(t) : \mathbf{Nat}} \quad \frac{}{\Gamma \vdash \mathsf{nil} : \mathbf{List}(\tau)}$$

$$\frac{\Gamma \vdash h : \tau \qquad \Gamma \vdash t : \mathbf{List}(\tau)}{\Gamma \vdash \mathsf{cons}(h,t) : \mathbf{List}(\tau)}$$

$$\frac{\Gamma \vdash t : \mathbf{Bool} \qquad \Gamma \vdash V : \tau \qquad \Gamma \vdash F : \tau}{\Gamma \vdash \mathsf{iterbool}(V, F, t) : \tau}$$

$$\frac{\Gamma \vdash t : \mathbf{Nat} \qquad \Gamma \vdash \lambda x.S : \tau \to \tau \qquad \Gamma \vdash Z : \tau}{\Gamma \vdash \mathsf{iternat}(\lambda x.S, Z, t) : \tau}$$

$$\frac{\Gamma \vdash t : \mathbf{List}(\sigma) \quad \Gamma \vdash \lambda xy.C : \sigma \to \tau \to \tau \quad \Gamma \vdash N : \tau}{\Gamma \vdash \mathsf{iterlist}(\lambda xy.C, N, t) : \tau}$$

The call-by-name evaluation semantics is as follows. Note that constructor terms of a given type are taken to be canonical forms.

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.t' \qquad t'[u/x] \Downarrow z}{t\,u \Downarrow z}$$

$$\frac{}{0 \Downarrow 0} \quad \frac{}{\mathsf{suc}(n) \Downarrow \mathsf{suc}(n)} \quad \frac{}{\mathsf{tt} \Downarrow \mathsf{tt}} \quad \frac{}{\mathsf{ff} \Downarrow \mathsf{ff}}$$

$$\frac{t \Downarrow \mathsf{tt} \qquad V \Downarrow z}{\mathsf{iterbool}(V, F, t) \Downarrow z} \qquad \frac{t \Downarrow \mathsf{ff} \qquad F \Downarrow z}{\mathsf{iterbool}(V, F, t) \Downarrow z}$$

$$\frac{t \Downarrow 0 \qquad Z \Downarrow z}{\mathsf{iternat}(\lambda x.S, Z, t) \Downarrow z}$$

$$\frac{t \Downarrow \mathsf{suc}(n) \qquad S[\mathsf{iternat}(\lambda x.S, Z, n)/x] \Downarrow z}{\mathsf{iternat}(\lambda x.S, Z, t) \Downarrow z}$$

$$\frac{}{\mathsf{nil} \Downarrow \mathsf{nil}} \qquad \frac{}{\mathsf{cons}(u,v) \Downarrow \mathsf{cons}(u,v)}$$

$$\frac{t \Downarrow \mathsf{nil} \qquad N \Downarrow z}{\mathsf{iterlist}(\lambda xy.C, N, t) \Downarrow z}$$

$$\frac{t \Downarrow \mathsf{cons}(u,v) \qquad C[u/x, \mathsf{iterlist}(\lambda xy.C, N, v)/y] \Downarrow z}{\mathsf{iterlist}(\lambda xy.C, N, t) \Downarrow z}$$

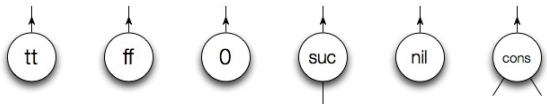
Some variables have been capitalized due to reasons that will become clear later on.

5 A Token-passing Encoding of BNL

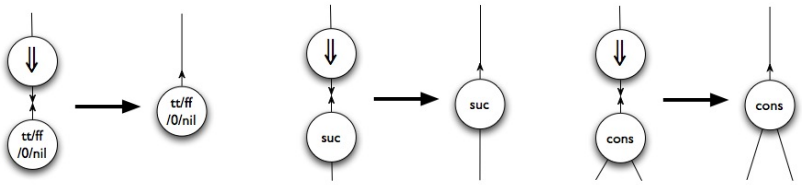
We extend to BNL the token-passing call-by-name translation of the λ-calculus into the interaction system $(\Sigma_{\text{tp}}, R_{\text{tp}})$. We first extend the interaction system and then the translation function. The novelty of this encoding is not the token-passing aspect (which is a natural extension of the encoding of the λ-calculus), but rather the approach to recursion.

We first consider data structures. Terms of inductively defined types can be represented in interaction nets in the natural way, as *trees* where each node corresponds to a constructor, with its principal port facing the parent node. In a (call-by-name) token-passing implementation, there will be an interaction rule between the token agent and each such constructor symbol that will stop evaluation—this corresponds to the fact that constructor terms are canonical forms.

For BNL we define the system $(\Sigma_{\text{BNL}}, R_{\text{BNL}})$ where Σ_{BNL} consists of the symbols tt, ff, 0 and nil with arity 0; suc with arity 1; and cons with arity 2, depicted as



and R_{BNL} consists of the rules given below.



Each recursive program will be encoded in an interaction system specifically generated for it. This is a major novelty of our approach. The interaction system for the λ-calculus will not be extended by introducing a fixed set of symbols; instead a new symbol will be introduced for *each occurrence of a recursion operator*, with an interaction rule for each different constructor of its argument type, so a dedicated interaction system (Σ_t^0, R_t^0) is generated for each term t.

This system is constructed by a recursive function $(\Sigma_t^0, R_t^0) = \mathcal{S}(t)$, defined as follows ($\cup$ is occasionally used to denote pairwise union).

$\mathcal{S}(x) \doteq \mathcal{S}(\mathsf{tt}) \doteq \mathcal{S}(\mathsf{ff}) \doteq \mathcal{S}(0) \doteq \mathcal{S}(\mathsf{nil}) \doteq (\emptyset, \emptyset)$

$\mathcal{S}(\lambda x.t) \doteq \mathcal{S}(\mathsf{suc}(t)) \doteq \mathcal{S}(t)$

$\mathcal{S}(t\,u) \doteq \mathcal{S}(\mathsf{cons}(t,u)) \doteq \mathcal{S}(t) \cup \mathcal{S}(u)$

$\mathcal{S}(\mathsf{iterbool}(V,F,b)) \doteq (\{\mathsf{It}^{\mathbf{Bool}}_{V,F}, \widehat{\mathsf{It}^{\mathbf{Bool}}_{V,F}}\} \cup \Sigma, R_{\mathsf{It}^{\mathbf{Bool}}_{V,F}} \cup R)$,
where $(\Sigma, R) = \mathcal{S}(b) \cup \mathcal{S}(V) \cup \mathcal{S}(F)$, and $R_{\mathsf{It}^{\mathbf{Bool}}_{V,F}}$ consists of the interaction rules included in Figures 4(a) and 4(b).

$\mathcal{S}(\mathsf{iternat}(\lambda x.S,Z,n)) \doteq (\{\mathsf{It}^{\mathbf{Nat}}_{S,Z}, \widehat{\mathsf{It}^{\mathbf{Nat}}_{S,Z}}\} \cup \Sigma, R_{\mathsf{It}^{\mathbf{Nat}}_{S,Z}} \cup R)$
where $(\Sigma, R) = \mathcal{S}(n) \cup \mathcal{S}(S) \cup \mathcal{S}(Z)$ and $R_{\mathsf{It}^{\mathbf{Nat}}_{S,Z}}$ consists of the interaction rules included in Figures 4(a) and 4(c).

$\mathcal{S}(\mathsf{iterlist}(\lambda xy.C,N,l)) \doteq (\{\mathsf{It}^{\mathbf{List}}_{C,N}, \widehat{\mathsf{It}^{\mathbf{List}}_{C,N}}\} \cup \Sigma, R_{\mathsf{It}^{\mathbf{List}}_{C,N}} \cup R)$
where $(\Sigma, R) = \mathcal{S}(l) \cup \mathcal{S}(C) \cup \mathcal{S}(N)$ and $R_{\mathsf{It}^{\mathbf{List}}_{C,N}}$ consists of the interaction rules included in Figures 4(a) and 4(d).

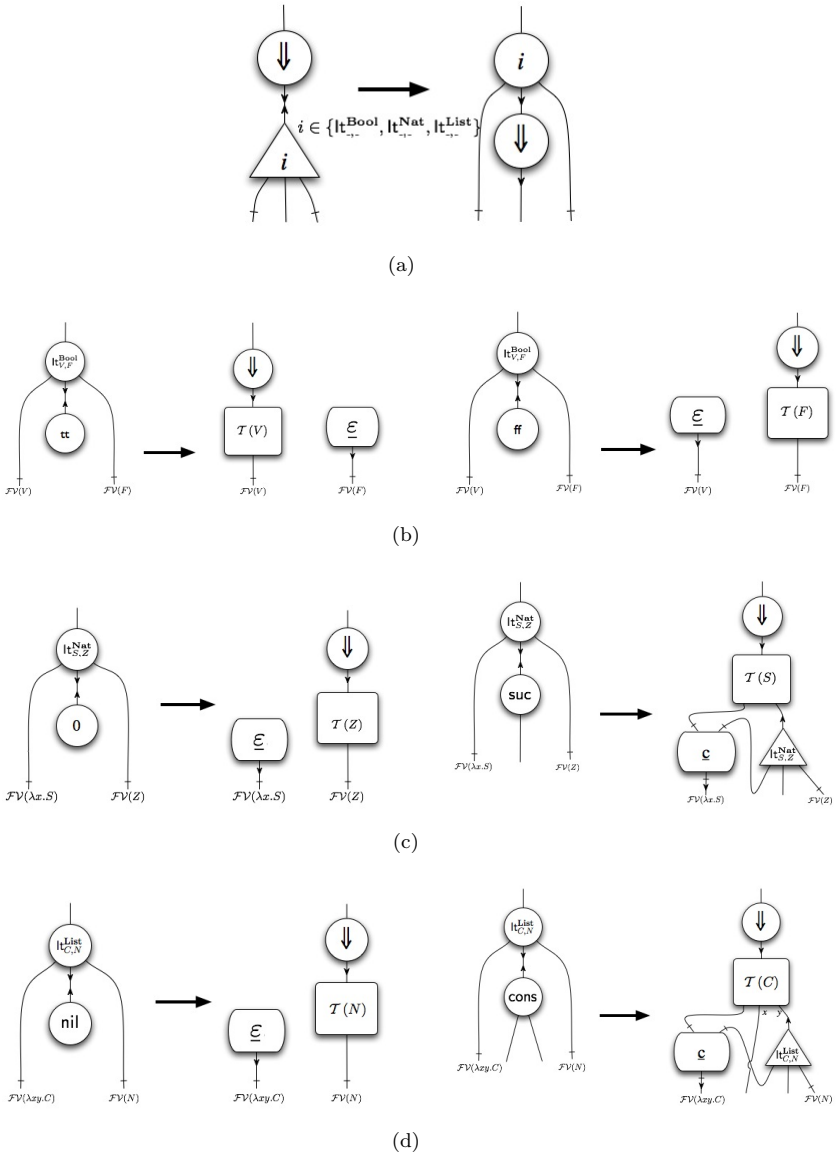


Fig. 4. Interaction rules for iterators

Iterator symbols are introduced in pairs $(\mathsf{lt}^{\cdots}_{\cdots}, \widehat{\mathsf{lt}^{\cdots}_{\cdots}})$ where the first symbol is used for syntactic agents and the second for computation agents (similarly to @, $\widehat{@}$). To simplify the graphical presentation, syntactic agents are depicted by triangles. The arity of each symbol can be inferred from the interaction rules. In Figures 4(b) to 4(d), $\underline{c}$ denotes an array of c agents and $\underline{\varepsilon}$ denotes an array of ε agents. The size of this arrays depends, respectively, on the number of shared and free variables in the corresponding terms.

A BNL program t will be translated into a net defined in the system $(\Sigma_t, R_t) = (\Sigma_{\text{tp}} \cup \Sigma_{\text{BNL}} \cup \Sigma_t^0, R_{\text{tp}} \cup R_{\text{BNL}} \cup R_t^0)$ where $(\Sigma_{\text{tp}}, R_{\text{tp}})$ was defined in Section 3.

Definition 5.1 Given a BNL program t, the net $\mathcal{T}(t)$ is given as follows.

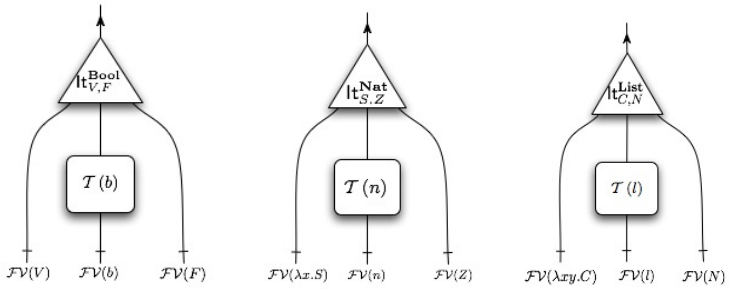


Fig. 5. Translations of iterators. We remark that, if the same variable occurs in more than one of the named sets (say, $\mathcal{FV}(V)$ and $\mathcal{FV}(F)$ for $\mathsf{iterbool}(V, F, b)$), c agents must be used to group the edges, analogously to what happens in the encoding of an application $t\,u$ (see Figure 2).

- If t is an abstraction, variable or application, then $\mathcal{T}(t)$ is defined as in Section 3.
- If t is one of tt, ff, 0, or nil, then $\mathcal{T}(t)$ is an instance of the corresponding symbol.
- If $t = \mathsf{suc}(t')$, then $\mathcal{T}(t)$ is constructed by connecting the auxiliary port of a suc agent to the root port of $\mathcal{T}(t')$.
- If $t = \mathsf{cons}(h, t')$, then $\mathcal{T}(t)$ is constructed by connecting the auxiliary ports of a cons agent to the root ports of $\mathcal{T}(h)$ and $\mathcal{T}(t')$.
- If $t = \mathsf{iterbool}(V, F, b)$ then $\mathcal{T}(t)$ is given by the net in Figure 5(a).
- If $t = \mathsf{iternat}(\lambda x.S, Z, n)$ then $\mathcal{T}(t)$ is given by the net in Figure 5(b).
- If $t = \mathsf{iterlist}(\lambda xy.C, N, l)$ then $\mathcal{T}(t)$ is given by the net in Figure 5(c).

As is characteristic of token-passing implementations, all terms (including iterators) are translated as syntax trees. Syntactic iterator agents i are turned into their computation counterparts $\widehat{i}$ by token agents, in the same way as the @ agents in the encoding of the λ-calculus.

A first key aspect of our approach is that the interaction rules of the (computation) iterator agents internalise the iterator's parameters. For instance the net $\mathcal{T}(\mathsf{iterlist}(\lambda xy.C, N, \mathsf{cons}(h, t)))$ reduces to $\mathcal{T}(C[h/x, \mathsf{iterlist}(\lambda xy.C, N, t)/y])$, with an evaluation token on top to control call-by-name evaluation.

A second key aspect is that each such new symbol will have auxiliary ports in a one-to-one correspondence with the free variables in the iterator term, since iterator terms are not restricted to be closed. The significance of this will become clear from the examples. We end the section with a correctness result. The proofs can be found in a long version of this paper [1].

Lemma 5.2 *Let t be a closed* BNL *term; then: $t \Downarrow z \implies \Downarrow \mathcal{T}(t) \longrightarrow^* \mathcal{T}(z)$.*

Lemma 5.3 *Let t be a closed* BNL *term and z a canonical form, then: $\Downarrow \mathcal{T}(t) \longrightarrow^* \mathcal{T}(z) \implies t \Downarrow z$.*

Proposition 5.4 *(**Correctness**) If t is a closed* BNL *term and z a canonical form, then: $t \Downarrow z \iff \Downarrow \mathcal{T}(t) \longrightarrow^* \mathcal{T}(z)$.*

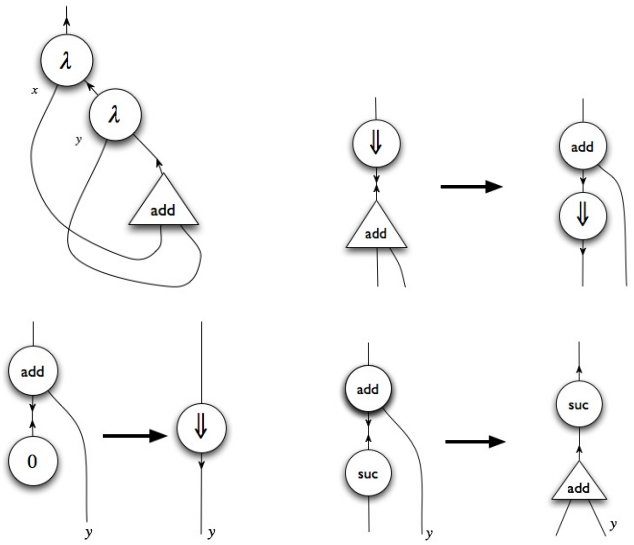


Fig. 6. Encoding of add and corresponding interaction rules

6 Examples

The following examples illustrate the use of the translation with different programs.

Example 6.1 Let $add = \lambda xy.\mathsf{iternat}(\lambda r.\mathsf{suc}(r), y, x)$ of type $\mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Nat}$. The free variable y in the second argument of the iterator creates an auxiliary port in the symbol $\mathsf{It}^{\mathbf{Nat}}_{\mathsf{suc}(r),y}$. The net corresponding to the encoding of the function and the interaction rules generated are given in Figure 6, where add stands for $\mathsf{It}^{\mathbf{Nat}}_{\mathsf{suc}(r),y}$. We remark that the last rule, whose right-hand side contained an active pair, was normalized by reducing that pair. The same will happen in the following examples.

The interaction rules for the computation agent add constitute a highly intuitive visual definition of addition, as should happen in any framework for visual programming. An example evaluation of a program can be found in the appendix.

Example 6.2 The reader is invited to work out the encoding of the append function $app = \lambda l_1 l_2.\mathsf{iterlist}(\lambda hr.\mathsf{cons}(h, r), l_2, l_1)$ with type $\mathbf{List}(\tau) \to \mathbf{List}(\tau) \to \mathbf{List}(\tau)$, and to compare it to the rules given in Section 2 for the agent app as an example of a direct interaction net program.

Example 6.3 Our final example corresponds to a higher-order function, $map = \lambda f l.\mathsf{iterlist}(\lambda hr.\mathsf{cons}(f\, h, r), \mathsf{nil}, l)$ with type $\mathsf{map} : (\tau \to \sigma) \to \mathbf{List}(\tau) \to \mathbf{List}(\sigma)$. This example differs from the previous in that a free variable (f) now occurs in the *first* argument of the iterator. Again this generates an auxiliary port in $\mathsf{It}^{\mathbf{List}}_{\mathsf{cons}(f\,h,r),\mathsf{nil}}$. The function is encoded as the net in Figure 7, where the name map is used for the symbol $\mathsf{It}^{\mathbf{List}}_{\mathsf{cons}(f\,h,r),\mathsf{nil}}$. Its interaction rules are also shown in the figure.

Again the visual representation is intuitive. The role of the copying agent in the second rule is to produce two copies of the encoding of the function: one to be applied to the head of the list, and another to be used in the recursive call.

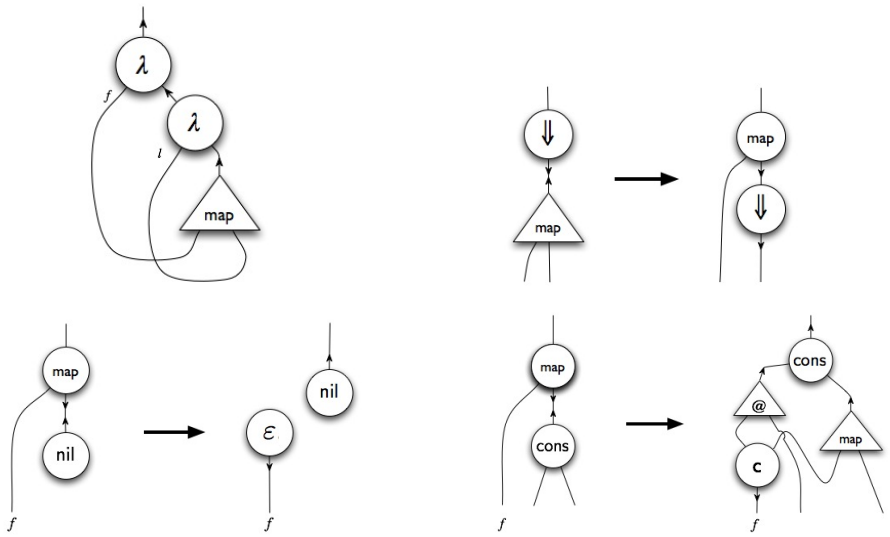


Fig. 7. Encoding of map and corresponding interaction rules

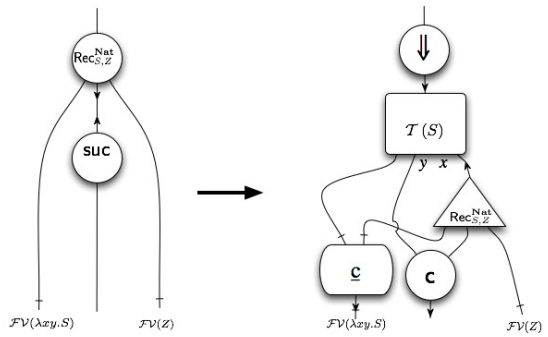


Fig. 8. Interaction rules for natural numbers recursor

7 Other Recursion Operators

A recursor for natural numbers can be added to the language with the following syntax, typing and evaluation rules: $t, u, v ::= \ldots \mid \mathsf{recnat}(\lambda xy.u, v, t)$,

$$\frac{\Gamma \vdash t : \mathbf{Nat} \qquad \Gamma \vdash \lambda xy.S : \tau \to \mathbf{Nat} \to \tau \qquad \Gamma \vdash Z : \tau}{\Gamma \vdash \mathsf{recnat}(\lambda xy.S, Z, t) : \tau}$$

$$\frac{t \Downarrow 0 \qquad Z \Downarrow z}{\mathsf{recnat}(\lambda xy.S, Z, t) \Downarrow z}$$

$$\frac{t \Downarrow \mathsf{suc}(n) \qquad S[\mathsf{recnat}(\lambda xy.S, Z, n)/x, n/y] \Downarrow z}{\mathsf{recnat}(\lambda xy.S, Z, t) \Downarrow z}$$

The computational power of this recursor operator comes from the fact that it has access to its *argument*, in addition to the recursive result on that argument. The factorial function, for instance, can be defined in this way, but not with an iterator. Replacing the iterator with this recursor requires only minor changes in

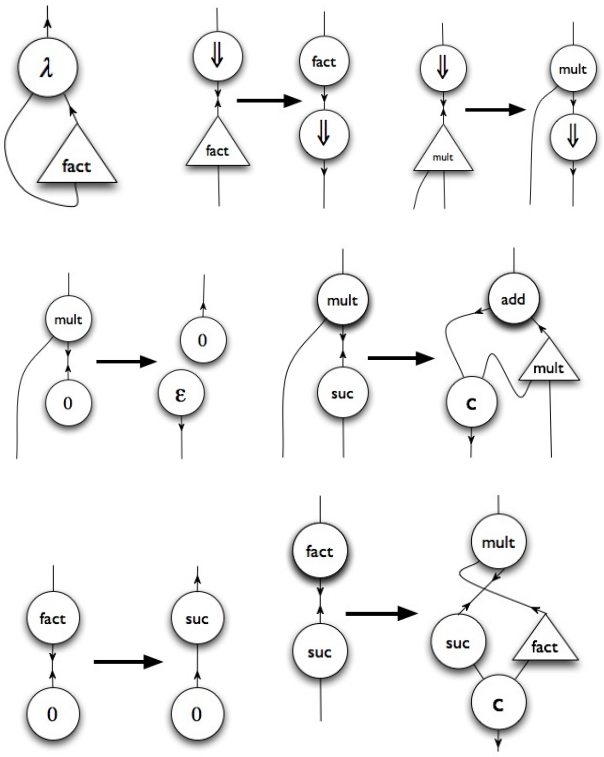


Fig. 9. Encoding of fact and corresponding interaction rules. Also includes rules for mult.

the interaction system: an agent $\mathsf{Rec}^{\mathbf{Nat}}_{S,Z}$ must be used in the translation of the expression $\mathsf{recnat}(\lambda xy.S, Z, t)$ instead of $\mathsf{It}^{\mathbf{Nat}}_{S,Z}$. Its interaction with the successor symbol is given by the rule shown in Figure 8, where we note that for an argument $\mathsf{suc}(n)$, the net representing n must now be duplicated.

For instance, the translation of $\mathit{fact} = \lambda n.\mathsf{recnat}(\lambda xy.\mathit{mult}\,\mathsf{suc}(y)\,x, \mathsf{suc}(0), n)$ with multiplication defined as $\mathit{mult} = \lambda xy.\mathsf{iternat}(\lambda r.\mathit{add}\,y\,r, 0, x)$, is given in Figure 9, where the symbols *fact* and *mult* stand respectively for $\mathsf{Rec}^{\mathbf{Nat}}_{\mathit{mult}\,\mathsf{suc}(y)\,x,\mathsf{suc}(0)}$ and $\mathsf{It}^{\mathbf{Nat}}_{\mathit{add}\,y\,r,0}$. Notice the RHS of the rules are fully or partially reduced (optimized).

In a language where recursion is only available through the use of recursion operators, it is important to have a number of different such operators, each of which may be more convenient for writing certain families of programs. We take as example the Haskell `foldl` (left folding) list operator, which stores intermediate results in an accumulator argument, returned at the end of the list. Even though every program written with it can also be written with the more common `foldr` (right folding operator), it is still convenient to have it in the language. For instance, a linear time, tail-recursive function for reversing lists can be written in the two following ways:

```
revt l = foldr (\h r a -> r(h:a)) id l []

revt l = foldl (\r h -> h:r) [] l
```

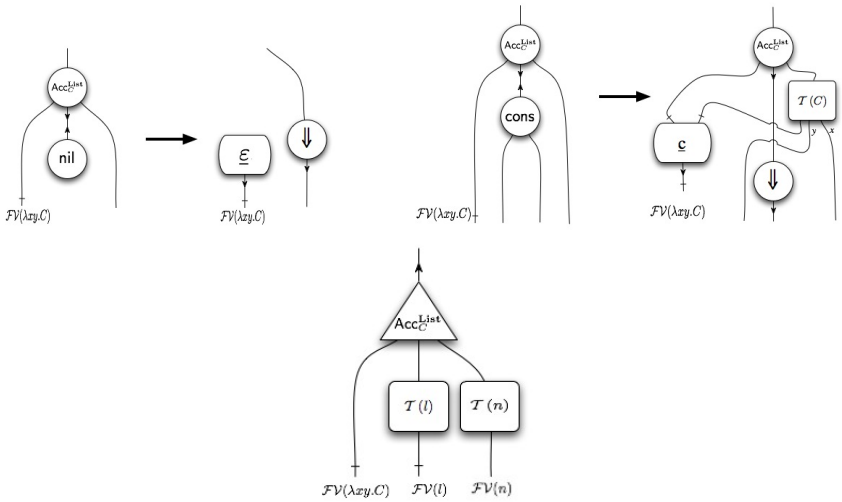


Fig. 10. Interaction rules for accumulations

The first version can be written in BNL. Applying the encoding of Section 5 results in a new agent $\mathsf{It}^{\mathbf{List}}_{(\lambda a.y\,\mathsf{cons}(x,a)),(\lambda x.x)}$. Naturally, the interaction rules for this agent introduce encodings of abstractions in their right-hand sides, which results in a quite complicated definition. To accommodate the second, clearly simpler definition, we now consider the extension of BNL with an *accumulation* operator similar to `foldl`, with the following typing and evaluation rules.

$$t, u, v ::= \ldots \mid \mathsf{acclist}(\lambda xy.t, u, v)$$

$$\frac{\Gamma \vdash t : \mathbf{List}(\tau) \qquad \Gamma \vdash \lambda xy.C : \sigma \to \tau \to \sigma \qquad \Gamma \vdash n : \sigma}{\Gamma \vdash \mathsf{acclist}(\lambda xy.C, n, t) : \sigma}$$

$$\frac{t \Downarrow \mathsf{nil} \qquad n \Downarrow z}{\mathsf{acclist}(\lambda xy.C, n, t) \Downarrow z}$$

$$\frac{t \Downarrow \mathsf{cons}(h, u) \qquad \mathsf{acclist}(\lambda xy.C, C[n/x, h/y], u) \Downarrow z}{\mathsf{acclist}(\lambda xy.C, n, t) \Downarrow z}$$

The function $\mathcal{S}(\cdot)$ that constructs the interaction system is extended as follows.

$$\mathcal{S}(\mathsf{acclist}(\lambda xy.C, n, l)) = (\{\mathsf{Acc}^{\mathbf{List}}_C, \widehat{\mathsf{Acc}^{\mathbf{List}}_C}\} \cup \Sigma, R_{\mathsf{Acc}^{\mathbf{List}}_C} \cup R)$$
$$\text{where } (\Sigma, R) = \mathcal{S}(l) \cup \mathcal{S}(C) \cup \mathcal{S}(n)$$

where $R_{\mathsf{Acc}^{\mathbf{List}}_C}$ consists of the rules of Figure 10, top (together with the obvious evaluation token rule). $\widehat{\mathcal{T}}(\mathsf{acclist}(\lambda xy.C, n, l))$ is then defined as the net shown in Figure 10, bottom. We remark that in the reduction rules for $\mathsf{acclist}(\lambda xy.C, n, l)$ the second argument n is not fixed throughout iteration; as such it cannot be internalized as part of the definition of the agent $\mathsf{Acc}^{\mathbf{List}}_C$. Instead the corresponding net is connected to an auxiliary port in that agent.

The list reversion function can now be written $\mathsf{revt} = \lambda l.\mathsf{acclist}(\lambda xy.\mathsf{cons}(y, x), \mathsf{nil}, l)$. The net $\widehat{\mathcal{T}}(\mathsf{revt})$ and the rules required are shown in Figure 11. Note that the symbol revt is used instead of $\mathsf{Acc}^{\mathbf{List}}_{\mathsf{cons}(y,x)}$.

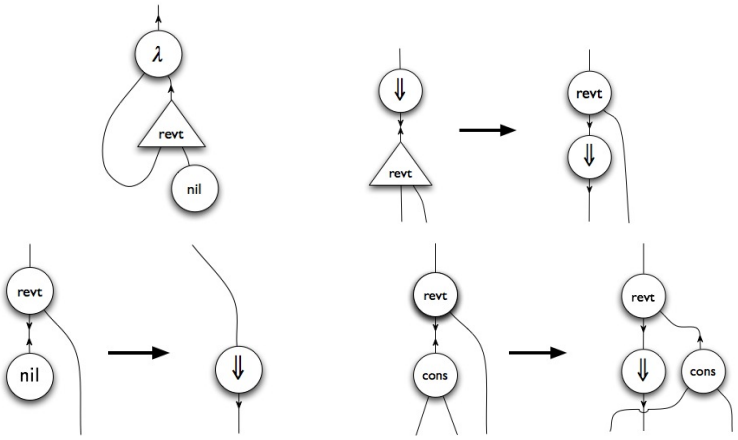


Fig. 11. Net and rules for list reversion

8 Conclusions and Future Work

We have presented an approach to encoding in interaction nets functional programs defined with recursion operators, and given the full details of the application of this approach to the token-passing implementation of a call-by-name language, which results in a very convenient visual notation for this language. The approach can be easily extended to richer sets of recursive types and other recursion operators, and also to new strategies. The novel characteristics of the encoding are (i) the fact that the interaction system is generated dynamically from the program, and (ii) the internalisation of some of the parameters of the recursion operator in the interaction rules of the symbol that encodes the operator's behaviour.

We have left types mostly out of our discussion. A net can be typed by assigning a type to every port. In our context, the types are those defined for the functional language BNL, except that they may occur either positively (in ports corresponding to data structures) or negatively (in ports corresponding to function or constructor arguments). In a correctly-typed net every edge connects two ports typed with $+A$ and $-A$ for some type A. So typing extends smoothly to the visual setting.

A prototype system for visual functional programming has been developed, integrated in the tool INblobs [2,11] for interaction net programming. The tool consists of an evaluator for interaction nets together with a visual editor and a compiler module that translates programs into nets. The latter module allows users to type in a functional program, visualize it, and then follow its evaluation visually step by step. The current compiler module is restricted to the iterators for $\mathbf{Bool}, \mathbf{Nat}$ and $\mathbf{List}(\tau)$, and automatically generates call-by-name (presented in this paper) or call-by-value systems. Additionally, a visual editing mode is available that allows users to construct nets corresponding to functional programs.

A topic that has been left out of the discussion in the paper is to give a direct (i.e. not resulting from a translation) characterization of the class of nets corresponding to recursive programs. This characterization could be used by the tool to restrict nets constructed visually to such a subclass of interaction nets. Also, the current implementation does not automatically normalize the RHS of the generated rules,

and moreover there is no way to convert visual programs back to textual ones.

A different line of work is inspired by work of the datatype-generic programming community and the school of program calculation [3]. This prompts the investigation of visual fusion laws for instance. Fusion laws simplify compositional functional programs before their application to arguments: before calculating $f(g(x))$ one may in certain conditions, by eliminating intermediate data structures, obtain a more efficient function h equivalent to $f \cdot g$, and calculate instead $h(x)$. A classic case is when g is an iterator. We conjecture that these laws can be proved in the interaction net setting by using notions of contextual equivalence [4]. Extending the visual programming tool with fusion capabilities would make possible to perform program transformations at the visual level. In [9] we investigated some preliminary ideas in this direction.

The token-passing translation of the λ-calculus has the advantage of implementing a simple evaluation order and maintaining a structure in the nets that is always immediately recognizable and understandable in terms of the evaluation semantics. As such it is totally appropriate for our goal of providing a visual representation for functional programs. Interaction nets have however also been extensively studied as an implementation mechanism for the λ-calculus. The main motivation for this approach is that it results in highly efficient evaluation strategies, made possible by the close control kept on the erasing and duplication of terms. The token-passing translation is not representative of most work in this area, which has concentrated on designing *efficient* translations. These translations are not controlled by an evaluation token (they produce nets already containing active pairs) and impose reduction strategies that cannot be defined using term-based abstract machines.

There are a number of interaction net encodings of the λ-calculus, which follow different strategies. To give just a sample, Gonthier, Abadi and Lévy [5] presented an implementation of optimal β-reduction. Mackie [7,8] has proposed several systems, each corresponding to a different strategy for reduction in the λ-calculus.

Let $\mathcal{T}(\cdot)$ be one such translation. Typically $\mathcal{T}(t\,u)$ is constructed from $\mathcal{T}(t)$ and $\mathcal{T}(u)$ by introducing an application symbol @ with its principal port connected to the root port of $\mathcal{T}(t)$. Our treatment of iterators can be adapted to this setting by removing the evaluator tokens and introducing the iterator agents with the principal port immediately facing the argument. When the iterated function is a closed term, a correctness result can be easily established: Let $\lambda x.S$ be a closed term, then

(i) $\mathcal{T}(\mathsf{iternat}(\lambda x.S, Z, 0)) \longrightarrow \mathcal{T}(Z)$

(ii) $\mathcal{T}(\mathsf{iternat}(\lambda x.S, Z, \mathsf{suc}(n))) \longrightarrow \mathcal{T}(S[\mathsf{iternat}(\lambda x.S, Z, n)/x])$

We remark that it is always possible to work with iterators with closed functions– thus this result applies to all programs.

For general terms, a correctness result has to be established for each translation, and it still has to be studied if, and in what way, the reduction strategy imposed by the translation for the λ-calculus is modified by this treatment of recursion.

References

[1] J. B. Almeida, I. Mackie, J. S. Pinto, and M. Vilaça. Encoding iterators in interaction nets. Available from `http://www.di.uminho.pt/~jmvilaca`.

[2] J. B. Almeida, J. S. Pinto, and M. Vilaça. A Tool for Programming with Interaction Nets. In *Proceedings of the The Eighth International Workshop on Rule-Based Programming (RULE'07)*, 2007. To appear in Elsevier ENTCS.

[3] R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[4] M. Fernández and I. Mackie. Operational equivalence for interaction nets. *Theoretical Computer Science*, 297(1–3):157–181, February 2003.

[5] G. Gonthier, 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)*, pages 15–26. ACM Press, Jan. 1992.

[6] Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, Jan. 1990.

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

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

[9] I. Mackie, J. S. Pinto, and M. Vilaça. Functional Programming and Program Transformation with Interaction Nets. In P. M. Hill, editor, *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR)*, 2005. Informal Proceedings.

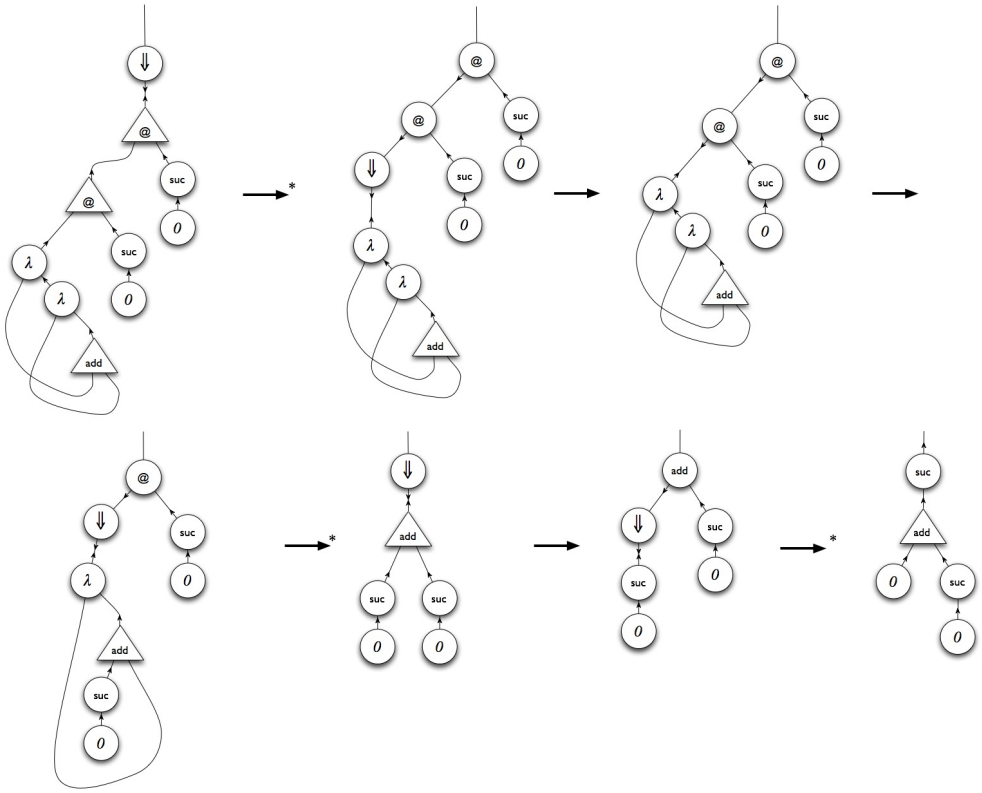
[10] F.-R. Sinot. Call-by-name and call-by-value as token-passing interaction nets. In P. Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2005.

[11] M. Vilaça. Inblobs webpage. http://haskell.di.uminho.pt/jmvilaca/INblobs/.

A Example Evaluation

The following represents some snapshots of the evaluation of the program

$$(\lambda xy.\mathsf{iternat}(\lambda r.\mathsf{suc}(r), y, x))\,(\mathsf{suc}(0))\,(\mathsf{suc}(0))$$



Strong Joinability Analysis for Graph Transformation Systems in CHR

Frank Raiser[1] Thom Frühwirth[2]

Institute for Software Engineering and Compiler Construction
Ulm University

Abstract

The notion of confluence is prevalent in graph transformation systems (GTS) as well as constraint handling rules (CHR). This work presents a generalized embedding of GTS in CHR that allows to consider strong derivations in confluence analyses. Confluence of a terminating CHR program is decidable, but confluence of a terminating GTS is undecidable. We show that observable confluence in CHR is a sufficient criterion for confluence of the embedded GTS. For this purpose the automatic confluence check for CHR can be reused.

Keywords: Graph Transformation Systems; Constraint Handling Rules; Confluence.

1 Introduction

Constraint handling rules (CHR) [6] allow for rapid prototyping and efficient implementation of constraint-based algorithms. Besides constraint reasoning, CHR have been used for various tasks including theorem proving, parsing, and multiset rewriting [6].

Graph transformation systems (GTS) are used to describe complex structures and systems in a concise, readable, and easily understandable way. They have applications ranging from implementations of programming languages over model transformations to graph-based models of computation [5,3].

In this work we present an embedding of graph transformation systems in CHR allowing us to perform strong derivations on partially defined graphs. This behavior provides the basis for the analysis of strong joinability of critical pairs presented in this work. In [9] we provided a similar embedding and used it to analyze non-strong joinability of critical pairs. The generalized embedding presented in this work, together with the recently introduced notion of observable confluence [4], allows us to improve upon this result. Deciding strong joinability of critical pairs comes

[1] Email: frank.raiser@uni-ulm.de
[2] Email: thom.fruehwirth@uni-ulm.de

Layout based on the macro package of the
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

for free as a result of the observable confluence check of the corresponding CHR program containing the embedded GTS.

We begin with the introduction of the necessary notions of graph transformation systems and CHR in Sect. 2. Section 3 then presents our proposed encoding of a GTS in CHR, for which Sect. 4 proves soundness and completeness. Finally, Sect. 5 introduces observable confluence and its application as a sufficient criterion for confluence of an embedded GTS, before we conclude in Sect. 6.

2 Preliminaries

In this section we introduce the required formalisms for graph transformation systems and constraint handling rules.

2.1 Graph Transformation System (GTS)

The following definitions for graphs and graph transformation systems have been adapted from [5].

A *graph* $G = (V, E, \mathrm{src}, \mathrm{tgt})$ consists of a set V of nodes, a set E of edges and two morphisms $\mathrm{src}, \mathrm{tgt} : E \to V$ specifying source and target of an edge, respectively. A *type graph* TG is a graph with unique labels for all nodes and edges.

For the purpose of simplicity, we avoid an additional label morphism in favor of identifying variable names with labels. For multiple graphs we refer to the node set V of a graph G as V_G and analogously for edge sets and the $\mathrm{src}, \mathrm{tgt}$ morphisms. We further define the degree of a node as $\deg : V \to \mathbb{N}, v \mapsto \#\{e \in E \mid \mathrm{src}(e) = v\} + \#\{e \in E \mid \mathrm{tgt}(e) = v\}$. As there are often multiple graphs containing the same node due to inclusion morphisms we use $\deg_G(v)$ to specify the degree of a node v with respect to the graph G. When the context graph is clear the subscript is omitted.

A *typed graph* G is a tuple $(V, E, \mathrm{src}, \mathrm{tgt}, \mathrm{type}, TG)$ where $(V, E, \mathrm{src}, \mathrm{tgt})$ is a graph, TG a type graph, and type a morphism with $\mathrm{type} = (\mathrm{type}_V, \mathrm{type}_E)$ and $\mathrm{type}_V : V \to TG_V, \mathrm{type}_E : E \to TG_E$. The type morphism is a graph morphism, therefore, it has to satisfy the following condition: $\forall e \in E : \mathrm{type}_V(\mathrm{src}(e)) = \mathrm{src}_{TG}(\mathrm{type}_E(e)) \wedge \mathrm{type}_V(\mathrm{tgt}(e)) = \mathrm{tgt}_{TG}(\mathrm{type}_E(e))$

A *Graph Transformation System* (GTS) is a tuple consisting of a type graph and a set of graph production rules. A *graph production rule* – also simply called *rule* if the context is clear – is a tuple $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ of typed graphs L, K, and R with inclusion morphisms $l : K \to L$ and $r : K \to R$.

We distinguish two kinds of typed graphs: *rule graphs* and *host graphs*. Rule graphs are the graphs L, K, R of a graph production rule p and host graphs are graphs to which the graph production rules can be applied. We, furthermore, make use of graph transformations based on the double-pushout approach (DPO) as defined in [5]. Most notably, we require a so-called match morphism $m : L \to G$ to apply a rule p to a typed host graph G. The transformation yielding the typed graph H is written as $G \overset{p,m}{\Longrightarrow} H$. H is given mathematically by constructing D as shown in Figure 1, such that (1) and (2) are pushouts. Intuitively, the graph L on the left-hand side is matched as a subgraph of G and its occurrence in G is then

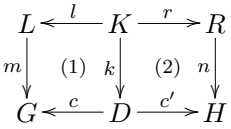


Fig. 1. Double-pushout approach

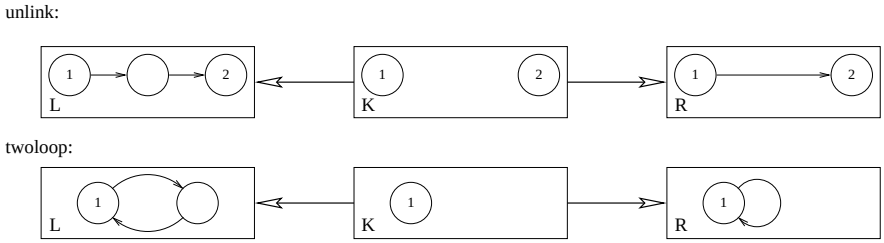


Fig. 2. Graph transformation system for recognizing cyclic lists

replaced by the right-hand side graph R. The intermediate graph K is the context graph containing those items in L that are preserved by the rule.

A graph production rule p can only be applied to a host graph G if the following gluing condition is satisfied. The *gluing condition* [5] is based on the set of gluing points $GP = l(K)$, the set of identification points $IP = \{v \in V_L \mid \exists w \in V_L, w \neq v : m(v) = m(w)\} \cup \{e \in E_L \mid \exists f \in E_L, e \neq f : m(e) = m(f)\}$, and the set of dangling points $DP = \{v \in V_L \mid \exists e \in E_G \setminus m(E_L) : \mathrm{src}_G(e) = m(v) \vee \mathrm{tgt}_G(e) = m(v)\}$ and it is defined as $IP \cup DP \subseteq GP$.

Example 2.1 Figure 2 shows two graph production rules which make up a graph transformation system for detecting cyclic lists. The basic idea of the *unlink* rule is to remove intermediate nodes of the list, while the *twoloop* rule replaces the cyclic list consisting of two nodes by a single node with a loop. To detect if a host graph is a cyclic list the GTS is applied to the host graph until exhaustion. The host graph then is a cyclic list if and only if the final graph consists of a single node with a loop [3].

Note that the example makes use of the type graph consisting only of a single node with a loop. Furthermore, we use a shorthand notation that only shows the morphisms l and r implicitly by the labels of the nodes which are mapped onto each other. Nodes and edges which are removed or added in the graphs L or R are not labeled, as there is no node or edge in K which is mapped to them.

In general, the DPO approach allows for the match morphism m to be non-injective. For injective match morphisms the set IP of identification points is guaranteed to be $\emptyset$. For the remainder of this work we only consider injective match morphisms, as non-injective ones can be simulated as follows: given a rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ and a non-injective match morphism m it holds $\forall v, w \in V_L, v \neq w$ with $m(v) = m(w)$ that the rule is only applicable, if $v, w \in l(V_K)$, i.e. only nodes which are not removed by the rule application are allowed to be matched non-injectively – otherwise $IP \not\subseteq GP$. Therefore, it is possible to add another rule p' which is derived from p by merging the nodes v and w into a node v_w in all three graphs of the rule. Thus, the non-injective matching with $m(v) = m(w)$ can be simulated by injectively matching v_w to $m(v_w)$ where $m(v_w)$ is the same node

in G as $m(v)$. The same argumentation holds for edges, analogously. Therefore, we can restrict ourselves to injective match morphisms by extending the set of rules with new rules for all possible merges of nodes and edges in the graph K. This simplifies the generic gluing condition to $DP \subseteq GP$.

In Sect. 5 we also require the following definition of a track morphism. Intuitively, the track morphism is defined for a node or edge, if it is not removed by the rule application.

Definition 2.2 [Track Morphism] Given $G \Longrightarrow H$ the *track morphism* $\mathrm{tr}_{G \Rightarrow H} : G \to H$ is the partial graph morphism defined by

$$\mathrm{tr}_{G \Rightarrow H}(x) = \begin{cases} c'(c^{-1}(x)) & \text{if } x \in c(D), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Here $c : D \to G$ and $c' : D \to H$ are the morphisms in the lower row of the pushout (1) in Fig. 1 and $c^{-1} : c(D) \to D$ maps each item $c(x)$ to x.

The track morphism of a derivation $\Delta : G_0 \Rightarrow^* G_n$ is defined by $\mathrm{tr}_\Delta = \mathrm{id}_{G_0}$ if $n = 0$ and $\mathrm{tr}_\Delta = \mathrm{tr}_{G_1 \Rightarrow^* G_n} \circ \mathrm{tr}_{G_0 \Rightarrow G_1}$ otherwise, where id_{G_0} is the identity morphism on G_0.

2.2 Constraint Handling Rules (CHR)

This section presents the syntax and operational semantics of constraint handling rules [6]. Constraints are first-order predicates which we separate into *built-in constraints* and *user-defined constraints*. Built-in constraints are provided by the constraint solver while user-defined constraints are defined by a CHR program. In this work we consider a subset of CHR where *Simplification* rules are of the form

$$\mathit{Rulename} \ @ \ H_1, \ldots, H_i \Leftrightarrow B_1, \ldots, B_k$$

where *Rulename* is an optional unique identifier of a rule, the head $H = H_1, \ldots, H_i$ is a non-empty conjunction of user-defined constraints, and the body $B = B_1, \ldots, B_k$ is a conjunction of built-in and user-defined constraints. Note that we make sloppy use of the terms conjunction, sequence, and multiset with respect to $H_1, \ldots, H_i$ and $B_1, \ldots, B_k$.

The operational semantics is based on an underlying *constraint theory* $\mathcal{CT}$ for the built-in constraints and a *state*, which is a tuple $\langle G, C, \mathcal{V} \rangle$ where G is a goal store, i.e. a multiset of user-defined constraints, C is a conjunction of built-in constraints, and $\mathcal{V}$ is the set of global variables-of-interest [6].

A simplification rule of the form $r \ @ \ H \Leftrightarrow B$ is *applicable* to a state $\langle E \wedge G, C, \mathcal{V} \rangle$ if $\mathcal{CT} \models \forall (C \to \exists \overline{x}(H = E))$ where $\overline{x}$ are the variables in H and $=$ is syntactic equality. We then define the following state transition for its application: $\langle E \wedge G, C, \mathcal{V} \rangle \mapsto^r \langle B_u \wedge G, (H = E) \wedge C \wedge B_b, \mathcal{V} \rangle$ where $B = B_u \cup B_b$ with B_u being user-defined and B_b being built-in constraints. We use $\mapsto$ when the applied rule is not of interest, and as usual, $\mapsto^*$ denotes the reflexive-transitive closure of the $\mapsto$ relation.

Given a simplification rule $p \ @ \ H \Leftrightarrow B$ and a state $S = \langle E \cup G, C, \mathcal{V} \rangle$ such that p is applicable to S we define for the involved match $\eta(p, S) = (E, C \wedge (H = E))$.

When comparing different states for confluence we make use of an equivalence relation $\equiv$ on CHR states [6]. This equivalence accounts for different syntactical representations, including renaming of local variables, equality substitutions, and logically equivalent built-in stores.

Example 2.3 The following two rules are part of a CHR handler for the boolean `and` constraint. The `and` constraint is ternary here with the meaning that $\texttt{and}(X, Y, Z)$ holds iff $X \wedge Y = Z$.

r_1 @ $\texttt{and}(X, X, Z) \quad \Leftrightarrow \quad Z = X$
r_2 @ $\texttt{and}(X, Y, 1) \quad \Leftrightarrow \quad X = 1, Y = 1$

For a CHR program consisting of these two rules we can consider an initial state $\langle \texttt{and}(0, 0, N) \cup \texttt{and}(A, B, C), C = 1, \{N, A, B, C\}\rangle$ as input, resulting in the following computation. The underlined constraints are matched to one of the rule heads and removed by the rule application.

$\langle \underline{\texttt{and}(0, 0, N)} \cup \texttt{and}(A, B, C), C = 1, \{N, A, B, C\}\rangle$
$\mapsto^{r_1} \langle \underline{\texttt{and}(A, B, C)}, C = 1 \wedge (X = 0 \wedge Z = N \wedge Z = X), \{N, A, B, C\}\rangle$
$\mapsto^{r_2} \langle \emptyset, C = 1 \wedge (X = 0 \wedge Z = N \wedge Z = X) \wedge (X' = A \wedge Y' = B \wedge C = 1 \wedge X' = 1 \wedge Y' = 1), \{N, A, B, C\}\rangle$

As this example shows the built-in store can include redundant information when the above transition definition is applied directly. CHR implementations simplify the built-in store with respect to the variables of interest using the built-in solver for the constraint theory $\mathcal{CT}$. This yields the following simplification of the final state above: $\langle \emptyset, N = 0 \wedge A = 1 \wedge B = 1 \wedge C = 1, \{N, A, B, C\}\rangle$. This state is equivalent to the final state above, i.e. the two states are contained in the $\equiv$ relation.

Example 2.4 An important property of the equivalence relation $\equiv$ between CHR states is equivalence modulo renaming of local variables. In this work we make use of this property to deal with graph isomorphism in CHR. Without going into details on the encoding of graphs in CHR yet, consider the following states σ_1, σ_2, and σ_3:
$\sigma_1 = \langle \text{node}(N, 1) \cup \text{node}(M, 1) \cup \text{edge}(E, D, N, M), \top, \{N\}\rangle$
$\sigma_2 = \langle \text{node}(N, 1) \cup \text{node}(M', 1) \cup \text{edge}(E', D', N, M'), \top, \{N\}\rangle$
$\sigma_3 = \langle \text{node}(N', 1) \cup \text{node}(N, 1) \cup \text{edge}(\hat{E}, \hat{D}, N', N), \top, \{N\}\rangle$

The variable N is a global variable in all these states and the remaining variables are local. Therefore, $\sigma_1 \equiv \sigma_2$ as they differ only by renaming of local variables. This is similar to considering isomorphism between two graphs, each consisting of two nodes connected by an edge. However, in CHR we can also consider these graphs in a different way, as it holds that $\sigma_3 \not\equiv \sigma_1$ although the graph described by σ_3 is an isomorphic graph. This is due to the global variable N occurring as a source of the edge in σ_1, but as a target in σ_3. This distinction is the basis of our strong joinability analysis.

3 Representation of Graphs in CHR

In order to embed a GTS in CHR, we have to encode its graph production rules as CHR rules and provide a conjunction of goal constraints corresponding to the host graph. To this end, we provide a correspondence between graphs and their representation by CHR constraints given by the constructions in Sect. 3.1. Sec-

tion 3.2 presents the encoding of the rules of the GTS for recognizing cyclic lists and a complete example derivation.

3.1 CHR Encoding of a GTS

For encoding a GTS in CHR we first determine the constraints needed for encoding the rules and host graph. At this point we require the GTS to be typed, so we can directly infer the necessary constraints from the corresponding type graph as explained in Def. 3.1. Note that this is not a restriction though, as every untyped graph can be typed over the type graph consisting of a single node with a loop.

Definition 3.1 [Type Graph Encoding] For a type graph TG we define the set $\mathcal{C}$ of required constraints to encode graphs typed over TG as the minimal set including $v/2 \in \mathcal{C}$ for $v \in V_{TG}$ and $e/4 \in \mathcal{C}$ for $e \in E_{TG}$.

We assume all nodes and edges of the type graph TG to be uniquely labeled such that the introduced constraints have unique names as well. Note that when annotating host graphs with these labels they can occur multiple times, i.e. their uniqueness is restricted to the type graph only.

Definition 3.2 [Typed Graph Encoding] For a typed graph G based on a type graph TG the set of constraints encoding G is defined differently for host and rule graphs. We define the following mappings for the encoding for an infinite set of variables VARS:

- $[\text{type}_G(x)]$ denotes the corresponding constraint name for encoding a node or edge of the given type.
- $\text{var} : G \to \text{VARS}, x \mapsto X_x$ such that X_x is a unique variable associated to x, i.e. var is injective for the set of all graph nodes and edges.
- $\text{dvar} : G \to \text{VARS}, x \mapsto X_x$ such that X_x is a unique variable associated to x, i.e. dvar is injective for the set of all graph nodes and edges.

Using these mappings we define the following encoding of graphs:

$$\texttt{chr}_G(\text{host}, x) = \begin{cases} [\text{type}_G(x)](\text{var}(x), \deg_G(x)) & \text{if } x \in V_G \\ [\text{type}_G(x)](\text{var}(x), \texttt{del}, \text{var}(\text{src}(x)), \text{var}(\text{tgt}(x))) & \text{if } x \in E_G \end{cases}$$

$$\texttt{chr}_G(\text{keep}, x) = \begin{cases} [\text{type}_G(x)](\text{var}(x), \text{dvar}(x)) & \text{if } x \in V_G \\ [\text{type}_G(x)](\text{var}(x), \text{dvar}(x), \text{var}(\text{src}(x)), \text{var}(\text{tgt}(x))) & \text{if } x \in E_G \end{cases}$$

We use the notations $\texttt{chr}(\text{host}, G) = \{\texttt{chr}_G(\text{host}, x) \mid x \in G\}$ and $\texttt{chr}(\text{keep}, G) = \{\texttt{chr}_G(\text{keep}, x) \mid x \in G\}$. Furthermore, we omit the index G if the context is clear. Edges e encoded with $\texttt{chr}(\text{host}, e)$, such that the second argument of the constraint is `del` are called *deletion edges*. If the encoding of the edge as $\texttt{chr}(\text{keep}, e)$ uses $\text{dvar}(e)$ instead, we call $\text{dvar}(e)$ the *deletion variable*. Similarly, $\text{dvar}(v)$ for a node v is called the *degree variable*.

Section 4 discusses the importance of deletion and degree variables with respect to the encoded GTS. Intuitively, nodes and edges using these cannot be removed by

a rule application. These nodes and edges prove to be vital for the strong joinability analysis presented in Sect. 5.

Example 3.3 [cont] For our example of the GTS for recognizing cyclic lists every node in the typed graph has the same type, just like every edge has the same type. Based on this we need the following constraints: node /2, edge /4

The host graph G that contains a cyclic list consisting of exactly two nodes is encoded in $\texttt{chr}(\text{host}, G)$ as follows:

$$\text{node}(N_1, 2), \text{node}(N_2, 2), \text{edge}(E_1, \texttt{del}, N_1, N_2), \text{edge}(E_2, \texttt{del}, N_2, N_1)$$

The same graph G occurring as a rule graph is encoded in $\texttt{chr}(\text{keep}, G)$ as follows:

$$\text{node}(N_1, D_1), \text{node}(N_2, D_2), \text{edge}(E_1, F_1, N_1, N_2), \text{edge}(E_2, F_2, N_2, N_1).$$

We can now encode a complete graph production rule based on these definitions:

Definition 3.4 [GTS Rule in CHR] For a graph production rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ from a GTS we define $\rho(p) = (C_L, C_R)$ with

- $C_L = \{\texttt{chr}(\text{keep}, x) \mid x \in K\} \cup \{\texttt{chr}(\text{host}, x) \mid x \in L \setminus K\}$
- $C_R = \{\texttt{chr}(\text{host}, x) \mid x \in R \setminus K\} \cup \{\texttt{chr}(\text{keep}, e) \mid e \in E_K\}$
  $\cup \{\texttt{chr}(\text{keep}, v'), \text{var}(v) = \text{var}(v'), \text{dvar}(v') = \text{dvar}(v) - \deg_L(v) + \deg_R(v) \mid v \in V_K\}$

The rule p is then encoded in CHR using $\rho(p) = (C_L, C_R)$ and in abuse of notation we use $\rho(p)$ for the CHR rule $p \mathbin{@} C_L \Leftrightarrow C_R$ as well as for the tuple (C_L, C_R).

Example 3.5 [cont.]As an example, consider the second rule from our example GTS, which reduces two cyclic nodes to a single node with a loop. Its encoding as a CHR simplification rule is given below:

$$\text{twoloop} \mathbin{@} \text{node}(N_1, D_1), \text{node}(N_2, 2), \text{edge}(E_1, \texttt{del}, N_1, N_2), \text{edge}(E_2, \texttt{del}, N_2, N_1) \Leftrightarrow \text{node}(N_1', D_1'), N_1' = N_1, D_1' = D_1 - 2 + 2, \text{edge}(E_3, \texttt{del}, N_1, N_1)$$

It is also possible to simplify such rules resulting in the following rule:

$$\text{twoloop} \mathbin{@} \text{node}(N_1, D_1), \text{node}(N_2, 2), \text{edge}(E_1, \texttt{del}, N_1, N_2), \text{edge}(E_2, \texttt{del}, N_2, N_1) \Leftrightarrow \text{node}(N_1, D_1), \text{edge}(E_3, \texttt{del}, N_1, N_1)$$

3.2 Example Computation

Soundness and completeness of the above encoding is shown in Sect. 4, however, to ease the understanding, we present a complete computation here for our cyclic list example. The following two rules are the CHR encoding of the rules from Fig. 2:

$$\begin{aligned}
\textit{unlink} \ @ \quad & \mathrm{node}(N_1, D_1), \mathrm{node}(N, 2), \mathrm{node}(N_2, D_2), \\
& \mathrm{edge}(E_1, \mathtt{del}, N_1, N), \mathrm{edge}(E_2, \mathtt{del}, N, N_2) \\
& \Leftrightarrow \\
& \mathrm{node}(N_1', D_1'), N_1' = N_1, D_1' = D_1{+}1{-}1, \\
& \mathrm{node}(N_2', D_2'), N_2' = N_2, D_2' = D_2{+}1{-}1, \mathrm{edge}(E, \mathtt{del}, N_1, N_2)
\end{aligned}$$

$$\begin{aligned}
\textit{twoloop} \ @ \quad & \mathrm{node}(N_1, D_1), \mathrm{node}(N, 2), \mathrm{edge}(E_1, \mathtt{del}, N_1, N), \mathrm{edge}(E_2, \mathtt{del}, N, N_1) \\
& \Leftrightarrow \\
& \mathrm{node}(N_1', D_1'), N_1' = N_1, D_1' = D_1{+}2{-}2, \mathrm{edge}(E, \mathtt{del}, N_1, N_1)
\end{aligned}$$

The following state S is the encoding of a simple cycle consisting of three nodes. To demonstrate strong computations the degree of the third node is left uninstantiated:

$S = \langle \mathrm{node}(N_1, 2) \cup \mathrm{node}(N_2, 2) \cup \mathrm{node}(N_3, D_3) \cup \mathrm{edge}(E_1, \mathtt{del}, N_1, N_2)$
$\cup\, \mathrm{edge}(E_2, \mathtt{del}, N_2, N_3) \cup \mathrm{edge}(E_3, \mathtt{del}, N_3, N_1), \top, \{N_1, N_2, N_3, E_1, E_2, E_3, D_3\}\rangle$

Rule *unlink* can then be applied to the state S resulting in the following state S':

$S' = \langle \mathrm{node}(N_1, 2) \cup \mathrm{node}(N_3, D_3') \cup \mathrm{edge}(E, \mathtt{del}, N_1, N_3) \cup \mathrm{edge}(E_3, \mathtt{del}, N_3, N_1),$
$D_3' = D_3{+}1{-}1, \{N_1, N_2, N_3, E_1, E_2, E_3, D_3\}\rangle$

Finally, rule *twoloop* can be applied to S' to remove node N_1, resulting in the following final state S'':

$S'' = \langle \mathrm{node}(N_3, D_3'') \cup \mathrm{edge}(E', \mathtt{del}, N_3, N_3), D_3' = D_3{+}1{-}1$
$\wedge\, D_3'' = D_3'{+}2{-}2, \{N_1, N_2, N_3, E_1, E_2, E_3, D_3\}\rangle$

As can be seen from the state S'' the built-in store contains a chain of degree adjustments for nodes with initially uninstantiated degree and the node N_3 remains throughout the whole computation. These properties are investigated more thoroughly in Sect. 4.

4 Soundness and Completeness

In this section we show soundness and completeness of our encoding. Whereas in [9] we showed soundness and completeness only for an encoding corresponding to $\mathtt{chr}(\mathrm{host}, G)$ we generalize these results in this work for an encoding based on $\mathtt{chr}(\mathrm{keep}, G)$. The following definitions specify these strictly more generic host graph encodings, as well as some properties of our encoding used throughout the remainder of this section.

We then discuss in Sect. 4.1 that CHR rule application respects the gluing condition, before Sect. 4.2 shows that rule applicability of GTS and CHR coincide. Finally, Sect. 4.3 combines these results to prove soundness and completeness.

In Sect. 3.2 the example shows that during the CHR computations we may encounter states which are not a direct encoding of a host graph. Nevertheless, these states represent a graph G without explicitly specifying node degrees or `del` constants. In order to uniformly argue on all of these states we introduce an invariant on states which, intuitively, is satisfied when a state is an encoding of a graph.

Definition 4.1 [Invariant]An *invariant* $\mathcal{I}(S)$ is a property such that for all S_0 and S_1, we have that if $S_0 \rightarrow S_1$ (or $S_0 \equiv S_1$) and $\mathcal{I}(S_0)$ holds then $\mathcal{I}(S_1)$ holds.

Definition 4.2 [Graph Invariant]The *graph invariant* $\mathcal{G}(S)$ with $S = \langle E, C, \mathcal{V}\rangle$

holds if there exist a graph G and a conjunction of equality constraints C', such that $\langle E, C \wedge C', \emptyset\rangle \equiv \langle \texttt{chr}(\text{host}, G), \top, \emptyset\rangle$. For a state S for which $\mathcal{G}(S)$ holds with a graph G we say S is a *$\mathcal{G}$-state based on G*.

The fact, that $\mathcal{G}$ is an invariant is shown in Cor. 4.12 using other results from this section which only make use of the definition of $\mathcal{G}$, but do not require it to be an invariant. The following definition allows us to argue directly on those nodes and edges of a $\mathcal{G}$-state based on G for which the state has uninstantiated degree or deletion variables:

Definition 4.3 [Strong Nodes and Edges] For a CHR state $S = \langle \texttt{chr}(\text{keep}, G), C, \mathcal{V}\rangle$ which is a $\mathcal{G}$-state based on G we define the set of *strong nodes and edges* as: $\mathcal{S}(S) = \{v \in V_G \mid \text{dvar}(v) = \deg_G(v) \notin C\} \cup \{e \in E_G \mid \text{dvar}(e) = \texttt{del} \notin C\}$

A consequence of the degree of a node not being specified as a constant is that such a strong node cannot be deleted by any rule, just like strong edges cannot be deleted either. This feature is used in Sect. 5 where overlaps of rules are investigated and strong nodes and edges are responsible for enforcing strong joinability.

Next we show how a matching in one formalism can be transferred to the other formalism:

Definition 4.4 [GTS Match Implies CHR Match] Let G be a host graph, $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ a GTS rule, and m a match morphism such that $G \overset{p,m}{\Longrightarrow} G'$. Furthermore, let $S = \langle \texttt{chr}(\text{keep}, G), C, \mathcal{V}\rangle$ be a $\mathcal{G}$-state based on G and $\rho(p) = (C_L, C_R)$.

Then m implies the CHR match $\eta(\rho(p), S) = (\tilde{G}, Eq)$ with

$\tilde{G} = \{\texttt{chr}(\text{keep}, x) \mid x \in m(L)\}$

$Eq = C \wedge \{\text{var}(v) = \text{var}(m(v)) \mid v \in V_L\} \wedge \{\text{var}(e) = \text{var}(m(e)) \mid e \in E_L\}$
$\wedge \{\text{dvar}(v) = \text{dvar}(m(v)) \mid v \in V_K\} \wedge \{\text{dvar}(e) = \text{dvar}(m(e)) \mid e \in E_K\}$

Definition 4.5 [CHR Match Implies GTS Match] Let $S = \langle \texttt{chr}(\text{keep}, G), C, \mathcal{V}\rangle$ be a $\mathcal{G}$-state based on G, $\rho(p)$ be the CHR rule for $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, and $S \mapsto S'$ using rule $\rho(p)$ with match $\eta(p, S) = (\tilde{G}, Eq)$.

Then $\eta(p, S)$ implies the injective GTS match morphism $m : L \to G$ with

$v \mapsto v'$ with $\text{var}(v) = \text{var}(v') \in Eq \wedge [\text{type}_L(v)](\text{var}(v'), \_) \in \tilde{G}$

$e \mapsto e'$ with $\text{var}(e) = \text{var}(e') \in Eq \wedge [\text{type}_L(e)](\text{var}(e'), \_, \_, \_) \in \tilde{G}$

Note that the implied CHR match from Def 4.4 matches all constraints in the head of the corresponding CHR rule and the implied match m from Def. 4.5 always corresponds to an injective total graph morphism.

4.1 Gluing Condition

As applicability of GTS rules is tied to satisfaction of the gluing condition we first ensure that our encoding given in Sect. 3 adheres to this restriction as well. It follows from the definition of a dangling edge, that one exists if and only if $DP \not\subseteq GP$.

Lemma 4.6 (Dangling Edges) *If the application of rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ to G using match m violates the gluing condition, such that $DP \not\subseteq GP$, then the*

corresponding CHR rule $\rho(p) = (C_L, C_R)$ is not applicable to a $\mathcal{G}$-state based on G using the match implied by Def. 4.4.

4.2 Applicability

Next we show that applicability of GTS rules and the corresponding rules encoded in CHR coincides. The following two lemmata show that the implied matchings are sufficient for the corresponding rule applicability:

Lemma 4.7 (GTS Rule Applicability) *Let $\rho(p) = (C_L, C_R)$ be applicable to a $\mathcal{G}$-state based on G then $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ is applicable to G using the implied match morphism m from Def. 4.5.*

Lemma 4.8 (Graph Rule Applicability) *Let $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, $G \overset{p,m}{\Longrightarrow} G'$, and let $S = \langle \texttt{chr}(\text{keep}, G), C, \mathcal{V}\rangle$ be a $\mathcal{G}$-state based on G.*

If $\forall x \in L \setminus K : m(x) \notin \mathcal{S}(S)$, then $\rho(p) = (C_L, C_R)$ is applicable to S using the implied match $\eta(p, S) = (\tilde{G}, Eq)$ from Def. 4.4.

With the above lemmata it can be shown that applicability directly coincides with respect to states that fully encode a host graph:

Theorem 4.9 (Applicability For Host Graphs) *A graph production rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ is applicable to a typed host graph G if and only if $\rho(p)$ is applicable to $S = \langle \texttt{chr}(\text{host}, G), \top, \mathcal{V}\rangle$.*

Proof. *As $\mathcal{G}(S)$ holds S is a variant of a $\mathcal{G}$-state based on G. The proof is then immediate from the combination of Lemma 4.7 and Lemma 4.8. Note that for Lemma 4.8 the additional demand on $\mathcal{S}(S)$ is satisfied, as using $\texttt{chr}(\text{host}, G)$ implies $\mathcal{S}(S) = \emptyset$.* □

4.3 Soundness and Completeness

In order to argue on the relationship between computations in CHR and the corresponding GTS derivations w.r.t. a defined track morphism we define strong derivations:

Definition 4.10 [Strong Derivation] A GTS derivation $G \overset{p,m}{\Longrightarrow} G'$ using $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ is *strong with respect to* $S \subset (V_G \cup E_G)$ if $\forall s \in S : s \in m(K) \vee s \notin m(L)$.

Def. 4.10 implies that the track morphism is defined $\forall x \in m(S)$. Together with the soundness result below this allows us to consider strong derivations. The basic notion behind these is that the initial state S contains only partial instantiations of deletion and degree variables. Then all rule applications correspond to strong derivations with respect to $\mathcal{S}(S)$, and hence, the track morphism is defined $\forall x \in \mathcal{S}(S)$ over all the involved rule applications, because the final state still contains all constraints corresponding to nodes and edges in $\mathcal{S}(S)$.

Theorem 4.11 (Soundness) *Let $\rho(p) = (C_L, C_R)$ be applicable to $S = \langle \texttt{chr}(\text{keep}, G), C, \mathcal{V}\rangle$ where $\mathcal{G}(S)$ holds with match $\eta(p, S) = (\tilde{G}, Eq)$, such that $S \mapsto S'$.*

Then $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *is applicable to* G *using the implied match morphism* m *from Def. 4.5 such that* $G \overset{p,m}{\Longrightarrow} G'$ *is strong w.r.t.* $\mathcal{S}(S)$*. Furthermore,* $S' \equiv \langle \texttt{chr}(\text{keep}, G'), C', \mathcal{V} \rangle$ *and* $\mathcal{G}(S')$ *holds.*

From this soundness result it follows directly that $\mathcal{G}$ is indeed an invariant:

Corollary 4.12 ($\mathcal{G}$ is an Invariant) *For CHR programs consisting of rules encoding a GTS the graph invariant* $\mathcal{G}$ *is an invariant according to Definition 4.1.*

Proof. *This is a direct consequence of Thm. 4.11.* □

As uninstantiated degree and deletion variables inhibit the application of rules that remove the corresponding nodes or edges we can only have completeness if the removed elements are not among the set of strong nodes and edges. When considering a $\texttt{chr}(\text{host}, G)$ encoding completeness is given as the following strongness condition is always satisfied.

Theorem 4.13 (Completeness) *Let* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *,* $G \overset{p,m}{\Longrightarrow} G'$*, and let* $S = \langle \texttt{chr}(\text{keep}, G), C, \mathcal{V} \rangle$ *be a* $\mathcal{G}$*-state based on* G*.*

If $\forall x \in L \setminus K : m(x) \notin \mathcal{S}(S)$*, then* $\rho(p) = (C_L, C_R)$ *is applicable to* S *using the implied match* $\eta(p, S)$ *from Def. 4.4. Furthermore, for* $S \mapsto S'$ *using this match* $S' \equiv \langle \texttt{chr}(\text{keep}, G'), C', \mathcal{V} \rangle$ *and* $\mathcal{G}(S')$ *holds.*

Analogously to before, when working on a $\texttt{chr}(\text{host}, G)$-based encoding, i.e. an encoding without variable degrees or deletion variables, Thm. 4.11 and Thm. 4.13 yield full soundness and completeness as $\mathcal{S}(S) = \emptyset$ for such a state S.

5 Confluence

Both graph transformation systems and constraint handling rules provide the notion of a confluence property. This property guarantees that any derivation made for an initial state results in the same final state no matter which applicable rules are applied. This section introduces the necessary definitions used for GTS and CHR confluence before comparing the two notions. It is shown how automatic observable confluence checking in CHR can be reused to yield a decidable sufficient criterion for confluence of a GTS encoded in CHR.

Note that for the remainder of this section a CHR program always assumes a program consisting only of rules encoding a GTS as explained above. Furthermore, all CHR programs, and therefore graph transformation systems, are assumed to be terminating.

5.1 Preliminaries

Definition 5.1 [GTS Confluence]A GTS is called *confluent* if, for all typed graph transformations $G \overset{*}{\Longrightarrow} H_1$ and $G \overset{*}{\Longrightarrow} H_2$, there is a typed graph X together with typed graph transformations $H_1 \overset{*}{\Longrightarrow} X$ and $H_2 \overset{*}{\Longrightarrow} X$. *Local confluence* means that this property holds for all pairs of direct typed graph transformations $G \Rightarrow H_1$ and $G \Rightarrow H_2$ [5].

Newman's general result for rewriting systems implies that it is sufficient to consider local confluence for terminating graph transformation systems. To verify local confluence we particularly need to study critical pairs and their joinability, according to the following definition based on [5,8].

Definition 5.2 [Joinability of Critical GTS Pair] Let $r_1 = (L_1 \overset{l}{\leftarrow} K_1 \overset{r}{\rightarrow} R_1), r_2 = (L_2 \overset{l}{\leftarrow} K_2 \overset{r}{\rightarrow} R_2)$ be two GTS rules. A pair $P_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} P_2$ of direct typed graph transformations is called a *critical GTS pair* if it is parallel dependent, and minimal in the sense that the pair (m_1, m_2) of matches $m_1 : L_1 \to G$ and $m_2 : L_2 \to G$ is jointly surjective.

A pair $P_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} P_2$ of direct typed graph transformations is called *parallel independent* if $m_1(L_1) \cap m_2(L_2) \subseteq m_1(K_1) \cap m_2(K_2)$, otherwise it is called *parallel dependent*.

A critical GTS pair $P_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} P_2$ is called *joinable* if there exists a typed graph X together with typed graph transformations $P_1 \overset{*}{\Longrightarrow} X_1 \simeq X_2 \overset{*}{\Longleftarrow} P_2$. It is *strongly joinable* if there is an isomorphism $f : X_1 \to X_2$ such that for each node v, for which $\mathrm{tr}_{G \Rightarrow P_1}(v)$ and $\mathrm{tr}_{G \Rightarrow P_2}(v)$ are defined, the following holds:

(i) $\mathrm{tr}_{G \Rightarrow P_1 \Rightarrow X_1}(v)$ and $\mathrm{tr}_{G \Rightarrow P_2 \Rightarrow X_2}(v)$ are defined and

(ii) $f_V(\mathrm{tr}_{G \Rightarrow P_1 \Rightarrow X_1}(v)) = \mathrm{tr}_{G \Rightarrow P_2 \Rightarrow X_2}(v)$

A similar notion of confluence has been developed for CHR [6]:

Definition 5.3 [CHR Confluence] A CHR program is called *confluent* if for all states S, S_1, and S_2: If $S \mapsto^* S_1$ and $S \mapsto^* S_2$, then S_1 and S_2 are joinable. Two states S_1 and S_2 are called *joinable* if there exist states $T_1 \equiv T_2$ such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$.

Analogous to a GTS, the confluence property for terminating CHR programs is determined by local confluence which can be checked through critical pairs:

Definition 5.4 [Joinability of Critical CHR Pair] Let r_1 be a simplification rule and r_2 be a (not necessarily different) rule whose variables have been renamed apart. Let $H_i \uplus A_i$ be the head, G_i be the guard, and B_i be the body of rule $r_i (i = 1, 2)$, then an *overlap* $\sigma_{\mathcal{CP}}$ of r_1 and r_2 is $\sigma_{\mathcal{CP}} = \langle H_1 \cup A_1 \cup H_2, (A_1 = A_2) \wedge G_1 \wedge G_2, \mathcal{V} \rangle$, provided A_1 and A_2 are non-empty conjunctions, $\mathcal{V} = \mathrm{vars}(H_1 \cup A_1 \cup H_2 \cup A_2 \cup G_1 \cup G_2)$ and $CT \models \exists((A_1 = A_2) \wedge G_1 \wedge G_2)$.

Let $S_1 = \langle B_1 \cup H_2, (A_1 = A_2) \wedge G_1 \wedge G_2, \mathcal{V} \rangle$ and $S_2 = \langle B_2 \cup H_1, (A_1 = A_2) \wedge G_1 \wedge G_2, \mathcal{V} \rangle$. Then the tuple $\mathcal{CP} = (S_1, S_2)$ is a *critical CHR pair* of r_1 and r_2. A critical CHR pair (S_1, S_2) is *joinable* if S_1 and S_2 are joinable.

5.2 Critical Pair Properties

After defining the different notions of confluence we now further investigate the difference between critical GTS pairs and critical CHR pairs for CHR programs encoding a GTS. The following lemma shows that there exists a corresponding CHR overlap for each critical GTS pair. Therefore, by examining the overlaps and using the previous soundness result we can transfer joinability results to the critical GTS pair.

Lemma 5.5 (Overlap for Critical GTS Pair) *If* $P_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} P_2$ *is a critical GTS pair, then there exists an overlap* $\sigma_{\mathcal{CP}}$ *of* $\rho(r_1) = (C_L^1, C_R^1)$ *and* $\rho(r_2) = (C_L^2, C_R^2)$ *which is a* $\mathcal{G}$*-state based on* G *and a critical CHR pair* (S_1, S_2) *such that* S_1 *is a* $\mathcal{G}$*-state based on* P_1 *and* S_2 *is a* $\mathcal{G}$*-state based on* P_2*.*

If we try to directly transfer the confluence property of a GTS to the corresponding CHR program, we cannot succeed however, as in general there are too many critical CHR pairs that could cause the CHR program to be non-confluent. The following example provides a rule, which only has one critical GTS pair, but for which the corresponding CHR rule has three critical CHR pairs.

Example 5.6 Consider a graph production rule for removing a loop from a node and its corresponding CHR rule:

$R@\,\mathrm{node}(N, D), \mathrm{edge}(E, \mathtt{del}, N, N) \Leftrightarrow \mathrm{node}(N, D'), D' = D - 2$

For investigating confluence one must overlap this rule with itself which yields the following three CHR overlap states:

(i) $\langle \mathrm{node}(N, D) \cup \mathrm{edge}(E, \mathtt{del}, N, N) \cup \mathrm{edge}(E', \mathtt{del}, N', N'), N = N', \mathcal{V} \rangle$

(ii) $\langle \mathrm{node}(N, D) \cup \mathrm{node}(N', D') \cup \mathrm{edge}(E, \mathtt{del}, N, N), N = N', \mathcal{V} \rangle$

(iii) $\langle \mathrm{node}(N, D) \cup \mathrm{edge}(E, \mathtt{del}, N, N), \top, \mathcal{V} \rangle$

State 1 is not critical, because the corresponding pair of graph transformations is parallel independent, and hence, directly joinable by applying the rule again. State 2 is an invalid state as it has multiple encodings of the same node and state 3 is the encoding of the corresponding critical pair for the graph production rule.

As we want to rule out invalid states, we use the following notion of observable confluence presented in [4]. It is based on restricting confluence investigations to states that satisfy an invariant. Based on these invariants, observable confluence (or $\mathcal{I}$-confluence) is defined as follows:

Definition 5.7 [Observable Confluence] A CHR program P is $\mathcal{I}$-confluent with respect to invariant $\mathcal{I}$ if the following holds for all states S_0, S_1, and S_2 where $\mathcal{I}(S_0)$ holds: If $S_0 \rightarrow^* S_1$ and $S_0 \rightarrow^* S_2$ then S_1 and S_2 are joinable.

In order to use the graph invariant $\mathcal{G}$ for the notion of observable confluence, we have to investigate the properties of this invariant. We introduce the following definitions from [4]. As overlap states themselves may not satisfy the invariant we have to examine all possible extensions that satisfy it [4].

Definition 5.8 [Extension, Valid Extension] A state $\sigma = \langle G, B, \mathcal{V} \rangle$ can be *extended* by another state $\sigma_e = \langle G_e, B_e, \mathcal{V}_e \rangle$ as follows $\sigma \oplus \sigma_e = \langle G \uplus G_e, B \wedge B_e, \mathcal{V}_e \rangle$. We say that σ_e is an *extension* of σ. A *valid extension* $\sigma_e = \langle G_e, B_e, \mathcal{V}_e \rangle$ of a state $\sigma = \langle G, B, \mathcal{V} \rangle$ is an extension such that $v \in vars(G \cup B) \wedge v \notin \mathcal{V} \Rightarrow v \notin vars(G_e \cup B_e \cup \mathcal{V}_e)$.

To minimize the number of extensions that have to be investigated only minimal extensions w.r.t. a partial order $\prec_\sigma$ on extensions [4] are considered. $\mathcal{M}_e^{\mathcal{I}}(\sigma)$ denotes the set of these minimal extensions of a state σ and is used in the following decision criterion of $\mathcal{I}$-local-confluence.

Note that for any extension $\sigma_e = \langle G_e, B_e, \mathcal{V}_e \rangle$ of a state $\sigma = \langle G, B, \mathcal{V} \rangle$ there

exists a valid extension $\sigma_\emptyset = \langle \emptyset, \top, \mathcal{V} \rangle$ that is smaller than σ_e w.r.t. the partial order on extensions. This results in $\mathcal{M}_e^{\mathcal{I}}(\sigma) = \{\sigma_\emptyset\}$ iff $\mathcal{I}(\sigma)$ holds.

Lemma 5.9 (Deciding $\mathcal{I}$-Local-Confluence [4]) *Given that $\prec_{\sigma_{\mathcal{CP}}}$ is well-founded for all overlaps $\mathcal{CP}$, then: P is $\mathcal{I}$-local-confluent iff for all critical pairs $\mathcal{CP} = (\sigma_1, \sigma_2)$ with overlap $\sigma_{\mathcal{CP}}$, and for all $\sigma_e \in \mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$, we have that $(\sigma_1 \oplus \sigma_e, \sigma_2 \oplus \sigma_e)$ is joinable.*

Although, in our programs built-in constraints + and − occur, we can consider $\prec_{\sigma_{\mathcal{CP}}}$ well-founded, as $\sigma_\emptyset$ is always smaller than any other extension. The following discussion shows that either $\mathcal{M}_e^{\mathcal{G}}(\sigma_{\mathcal{CP}}) = \{\sigma_\emptyset\}$ or $\Sigma_e^{\mathcal{G}}(\sigma_{\mathcal{CP}}) = \mathcal{M}_e^{\mathcal{G}}(\sigma_{\mathcal{CP}}) = \emptyset$. This means, that for all elements $\sigma_e \in \Sigma_e^{\mathcal{G}}(\sigma_{\mathcal{CP}})$ we have $\sigma_\emptyset \preceq_{\sigma_{\mathcal{CP}}} \sigma_e$, and hence, $\prec_{\sigma_{\mathcal{CP}}}$ is well-founded. Whether $\sigma_\emptyset$ is the minimal element depends solely on $\mathcal{G}(\sigma_{\mathcal{CP}})$ holding as the following lemma shows.

Lemma 5.10 (No Minimal Elements) *If $\mathcal{G}(\sigma_{\mathcal{CP}})$ is violated for an overlap $\sigma_{\mathcal{CP}}$ then no extension σ_e exists such that $\mathcal{G}(\sigma_{\mathcal{CP}} \oplus \sigma_e)$ is satisfied, i.e. $\Sigma_e^{\mathcal{G}}(\sigma_{\mathcal{CP}}) = \mathcal{M}_e^{\mathcal{G}}(\sigma_{\mathcal{CP}}) = \emptyset$.*

Combining these two results yields the criterion in Cor. 5.11 for deciding $\mathcal{G}$-local-confluence. Note that this decision criterion is essentially the same criterion as used for traditional local confluence, except that the invariant $\mathcal{G}$ restricts the set of investigated overlaps.

Corollary 5.11 (Deciding $\mathcal{G}$-Local-Confluence) *P is $\mathcal{G}$-local-confluent if and only if for all critical pairs $\mathcal{CP} = (\sigma_1, \sigma_2)$ with overlap $\sigma_{\mathcal{CP}}$, for which $\mathcal{G}(\sigma_{\mathcal{CP}})$ holds, $\mathcal{CP}$ is joinable.*

Proof. *This follows from the combination of Lemma 5.9, Lemma 5.10 and the insight that $\sigma_\emptyset$ is the minimal extension in the case of $\mathcal{G}(\sigma_{\mathcal{CP}})$ holding.* □

Next we transfer the joinability of critical CHR pairs to strong joinability in GTS:

Lemma 5.12 ($\mathcal{G}$-Confluence Implies Strong Joinability) *If the CHR program for a terminating GTS is $\mathcal{G}$-confluent, then all critical GTS pairs are strongly joinable.*

Proof. *Let $P_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} P_2$ be a critical GTS pair. Let $r_i = (L_i \leftarrow K_i \rightarrow R_i)$ and $\rho(r_i) = (C_L^i, C_R^i)$ for $i = 1, 2$.*

By Lemma 5.5 there exists an overlap $\sigma_{\mathcal{CP}}$ which is a $\mathcal{G}$-state based on G. As the critical pair (S_1, S_2) created by the overlap $\sigma_{\mathcal{CP}}$ is joinable we have the computations $\sigma_{\mathcal{CP}} \mapsto S_1 \mapsto^* T_1$ and $\sigma_{\mathcal{CP}} \mapsto S_2 \mapsto^* T_2$ with $T_1 \equiv T_2$. From Thm. 4.11 we know that there exist corresponding GTS transformations $G \overset{r_1,m_1}{\Longrightarrow} P_1 \Longrightarrow^* X_1 \simeq X_2 {}^* \Longleftarrow P_2 \overset{r_2,m_2}{\Longleftarrow} G$. The isomorphism between X_1 and X_2 follows from $T_1 \equiv T_2$. Hence, the critical GTS pair is joinable.

To see that it is strongly joinable consider the set $\mathcal{S}(\sigma_{\mathcal{CP}})$. Every node v for which $\mathrm{tr}_{G \Rightarrow P_1}(v)$ and $\mathrm{tr}_{G \Rightarrow P_2}(v)$ are defined is a node which is not deleted by either r_1 or r_2. As m_1 and m_2 are jointly surjective w.l.o.g. there exists a node $v' \in V_{L_1}$ of rule r_1 with $m(v') = v$. As the node is not removed we know $v' \in V_{K_1}$, and

therefore, $[\text{type}_{K_1}(v')](\text{var}(v'), \text{dvar}(v')) \in C_L^1$. *Either the node is not part of the overlap, or if it is overlapped with a node* $v'' \in V_{L_2}$ *such that* $m(v') = m(v'')$, *then we also know that* $v'' \in V_{K_2}$ *due to the defined track morphism. Therefore, we always have the node constraint* $[\text{type}_{K_1}(v')](\text{var}(v), \text{dvar}(v)) \in \sigma_{\mathcal{CP}}$ *and* $v \in \mathcal{S}(\sigma_{\mathcal{CP}})$. *As this node cannot be removed during the transformation a variant of this constraint with adjusted degree is also present in* T_1 *and* T_2. *These two variant constraints are uniquely determined, as* $\text{var}(v) \in \mathcal{V}$, *and hence, they both have to use* $\text{var}(v)$ *for the node identifier variable. This means we still have to show for such a node* v *that the two conditions from Def. 5.2 are satisfied:*

(i) $\text{tr}_{G \Rightarrow P_1 \Rightarrow X_1}(v)$ *and* $\text{tr}_{G \Rightarrow P_2 \Rightarrow X_2}(v)$ *are defined:*
By Thm. 4.11 we know that the GTS transformations are strong w.r.t. $\mathcal{S}(\sigma_{\mathcal{CP}})$. *As* $v \in \mathcal{S}(\sigma_{\mathcal{CP}})$ *this implies* $v \in m(K) \vee v \notin m(L)$ *for each of the applied rules, i.e. the node remains during the transformation and hence the track morphisms are defined as in Def. 2.2.*

(ii) $f_V(\text{tr}_{G \Rightarrow P_1 \Rightarrow X_1}(v)) = \text{tr}_{G \Rightarrow P_2 \Rightarrow X_2}(v)$*:*
An isomorphism f' *between* T_1 *and* T_2 *exists, because* $T_1 \equiv T_2$. *Consider the constraints in* T_1 *and* T_2 *which are the encoding of node* v *in* $\sigma_{\mathcal{CP}}$ *and let them use the degree variables* $\text{dvar}(v_1)$ *and* $\text{dvar}(v_2)$ *(with the corresponding chain of constraints* $\text{dvar}(v_i) = \text{dvar}(v_i') - n' + m' = \ldots = \text{dvar}(v) - n + m$ *for* $i = 1, 2$ *that have been accumulated during the computation). Then there exist corresponding nodes* $\text{tr}_{G \Rightarrow P_1 \Rightarrow X_1}(v) = v_1 \in V_{X_1}$ *and* $\text{tr}_{G \Rightarrow P_2 \Rightarrow X_2}(v) = v_2 \in V_{X_2}$ *and the isomorphism* f' *between* T_1 *and* T_2, *which equalizes* $\text{dvar}(v_1)$ *and* $\text{dvar}(v_2)$, *implies an isomorphism* f *with* $f_V(v_1) = v_2$.

□

Finally, this gives the following connection of confluence between both systems:

Theorem 5.13 ($\mathcal{G}$-Confluence Implies GTS Confluence) *A terminating GTS is confluent if the corresponding CHR program is* $\mathcal{G}$*-confluent.*

Proof. *By Lemma 5.12 all critical GTS pairs are strongly joinable. Hence, the GTS is locally confluent and as it is terminating it is also confluent [5].* □

In practical terms Theorem 5.13 effectively means that the automatic confluence check for terminating CHR programs [2,6] can be reused to prove confluence of a terminating GTS encoded as a CHR program. Due to the earlier results presented in this section we can apply the standard confluence checker only to those overlaps satisfying the invariant $\mathcal{G}$. The possible causes for an overlap to not satisfy $\mathcal{G}$ are duplicate node constraints or inconsistent degrees which can easily be checked. If all critical CHR pairs stemming from these overlaps are joinable we know by Cor. 5.11 that the CHR program is $\mathcal{G}$-confluent, and hence by Thm. 5.13, that the GTS is confluent. As no modification is needed for the confluence checker itself this means that by a simple restriction of inputs to the confluence checker we can decide $\mathcal{G}$-confluence and in turn get a sufficient criterion for GTS confluence for free.

6 Conclusion

In [9] we have shown that constraint handling rules (CHR) provide an elegant way for embedding graph transformation systems (GTS). The resulting rules are concise and directly related to the corresponding graph production rules. We presented a generalization of this encoding. It allows to model strong derivations that are used to analyze strong joinability.

The combination of our work with the research on observable confluence [4] resulted in a direct application of the CHR confluence check to decide $\mathcal{G}$-confluence. Invalid overlaps introduced by the CHR encoding of a GTS can elegantly be handled by considering $\mathcal{G}$-confluence which reduces the confluence analysis to the essential overlaps that yield strong joinability of critical GTS pairs.

The connection between CHR and GTS provides room for further research. This work only considers typed graphs, but could be extendend to support typed *attributed* graphs as well. As our generalized encoding allows computations on partially defined graphs this allows considering derivations as being applicable to the corresponding set of fully defined graphs.

Furthermore, it seems possible to transfer other results from CHR to GTS and vice versa. The approaches used for termination analysis of CHR [7] and GTS [5] seem to be distinct, such that both may profit from applying the approaches from the other formalism. Similarly, CHR provides a strong result on operational equivalence [1] that may provide a decidable criterion for equivalence of embedded graph transformation systems.

References

[1] Abdennadher, S. and T. Frühwirth, *Operational equivalence of CHR programs and constraints*, in: J. Jaffar, editor, *Principles and Practice of Constraint Programming, CP 1999*, Lecture Notes in Computer Science **1713** (1999), pp. 43–57.

[2] Abdennadher, S., T. Frühwirth and H. Meuss, *Confluence and semantics of constraint simplification rules*, Constraints **4** (1999), pp. 133–165.

[3] Bakewell, A., D. Plump and C. Runciman, *Specifying pointer structures by graph reduction.*, in: J. L. Pfaltz, M. Nagl and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Revised Selected and Invited Papers*, Lecture Notes in Computer Science **3062** (2003), pp. 30–44.

[4] Duck, G. J., P. J. Stuckey and M. Sulzmann, *Observable confluence for constraint handling rules*, in: V. Dahl and I. Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007*, Lecture Notes in Computer Science **4670** (2007), pp. 224–239.

[5] Ehrig, H., K. Ehrig, U. Prange and G. Taentzer, "Fundamentals of Algebraic Graph Transformation," Springer-Verlag, 2006.

[6] Frühwirth, T., "Constraint Handling Rules," Cambridge University Press, 2009, to appear.

[7] Pilozzi, P. and D. De Schreye, *Termination analysis of CHR revisited*, in: T. Schrijvers, F. Raiser and T. Frühwirth, editors, *Constraint Handling Rules, 5th Workshop, CHR 2008*, Hagenberg, Austria, 2008, pp. 35–50.

[8] Plump, D., *Confluence of graph transformation revisited*, in: A. Middeldorp, V. van Oostrom, F. van Raamsdonk and R. C. de Vrijer, editors, *Processes, Terms and Cycles*, Lecture Notes in Computer Science **3838** (2005), pp. 280–308.

[9] Raiser, F., *Graph Transformation Systems in CHR*, in: V. Dahl and I. Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007*, Lecture Notes in Computer Science **4670** (2007), pp. 240–254.

Lazy Constraint Imposing for Improving the Path Constraint

Ruben Duarte Viegas[1,2] Francisco Azevedo[3]

CENTRIA - Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Caparica, Portugal

Abstract

In this paper we propose a lazy constraint imposing mechanism for improving the path constraint in *GRASPER*, a state-of-the-art graph constraint solver, having obtained very promising results in terms of both time and space in solving an interesting problem in the Biochemistry subject area, in comparison with *CP(Graph)*, the state-of-the-art solver.

Keywords: Constraint Programming; Graph Constraint Propagation; Path Constraint.

1 Introduction

Constraint Programming (CP) [14,6,17] has been extensively used for solving combinatorial [16], scheduling [11], allocation [13] problems, among others, in various domains. After the appearance of sets [10] in CP, graph domain variables and corresponding operations were defined [9,8,22,23] allowing users to directly create and manipulate these variables in order to model their actual problem in a much more higher level than before.

One of the definitions proposed for graph domain variables is the one specified in [22,23], which is implemented in *GRASPER (GRaph ConstrAint Satisfaction Problem solvER)*, available in the *CaSPER* [4] platform [4]. As the name indicates, *GRASPER* is a graph constraint satisfaction problem solver. It is directly based upon a finite set solver, *Cardinal* [2] and it provides the means for creating directed and undirected graph variables, a nucleus of basic constraints upon which more complex and useful constraints can be provided ranging from constraints to impose

[1] Thanks to the Fundação para a Ciência e Tecnologia for the financial support (SFRH/BD/41362/2007) and a special thanks to Marco Correia for his invaluable help and guidance

[2] Email:ruben.viegas@di.fct.unl.pt

[3] Email:fa@di.fct.unl.pt

[4] Available at http://proteina.di.fct.unl.pt/casper/

Layout based on the macro package of the
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

graph properties (order, size, degree, reachability, connectedness, path, ...) and to impose graph relationships (underlying and oriented, reverse, complementary graph relationships). As demonstrated in [22,23], *GRASPER* appeared as an alternative to *CP(Graph)*, the state-of-the-art graph constraint solver [9,8] in the comparison made between the two for the *Metabolic Pathways Problem* [1,21,15], a problem which can be viewed as a path discovery problem in biochemical networks.

Among all the constraints available on a typical graph solver, one of the most important ones is the *path* constraint. By definition, a *path* between two vertices is a sequence of unique vertices contained in the vertex-set of a graph, starting at an initial vertex and finishing in a terminal vertex and such that for every pair of successive vertices there is an edge linking them in the edge-set E of G.

In this paper we explain how we can improve, not only in time but also in space, the *path* constraint by employing a lazy constraint imposing mechanism. In section 2 we introduce *GRASPER* and define the *path* constraint and in section 3 we explain how the *path* constraint, as defined in [22,23], was implemented upon *GRASPER*. Subsequently, in section 4 we explain how, using lazy constraint imposing, we can implement a much more efficient *path* constraint and we use these two implementations for solving the *Metabolic Pathways Problem*, presenting results and comparing both implementations, against the state-of-the-art solver, in section 5. Finally, we end with our closing remarks and future work, in section 6.

2 *GRASPER* and the *path* constraint

A graph [3,24,7], is composed by a set of vertices and by a set of edges, where each edge connects a pair of the graph's vertices. Therefore a graph variable can be seen as a pair (V, E) where both V and E are finite set variables. In a directed graph variable each edge is represented by a pair (X, Y) specifying a directed arc from X towards Y.

As for finite integer domains, where variables have a lower bound and an upper bound delimiting the set of possible values that the variable can be instantiated to, we have for finite set and graph domains the same concept.

In finite sets, the domain of each variable is represented by two sets: the *greatest lower bound (glb)* set and the *least upper bound (lub)* set, ordered by set inclusion, which define, respectively, the smallest and the biggest sets to which the variable can be instantiated. In finite graphs, the graph's *glb* is defined as the composition of its vertex-set and edge-set *glb*s and, similarly, the graph's *lub* is defined as the composition of its vertex-set and edge-set *lub*.

We start by defining finite set and finite graph (both directed and undirected) domain variables and then proceed to the description of the functionality we intend to improve.

Definition 2.1 *[Set variable]* A set variable X is represented by $[a_X, b_X]_{c_X}$ where a_X is the set of elements known to belong to X (its greatest lower bound (*glb*)), $b_X$is the set of elements not excluded from X (its least upper bound (*lub*)), and c_X its cardinality (a finite domain variable). We define $p_X = b_X \setminus a_X$ to be the set of elements, not yet excluded from X and that can still be added to a_X (or, to put it

short, *poss*).

Definition 2.2 *[Directed Graph variable]* A directed graph variable X is represented by $dirgraph(V_X, E_X)$ where V_X is a finite set variable representing the vertices of X and E_X another finite set variable representing the edges of X.

Definition 2.3 *[Undirected Graph variable]* An undirected graph variable X is represented by $undirgraph(V_X, E_X)$ where V_X is a finite set variable representing the vertices of X and E_X another finite set variable representing the edges of X.

CaSPER, the framework where *GRASPER* is built upon provides a very useful structure for use in propagators: *delta domain* variables. A *delta domain* represents the set of updates on a variable domain between two consecutive executions of some propagator. In the following, let $X \ominus Y = \langle a_X \setminus a_Y, b_X \setminus b_Y \rangle$ be the standard (bounds) difference between two set variables X and Y:

Definition 2.4 *[Delta domain]* Let $D_I(X)$ and $D_F(X)$ denote respectively the initial domain of X (i.e. before any propagator is executed), and final domain of X (i.e. after fixpoint is reached). The delta domain of variable X is $\Delta(X) = D_I(X) \ominus D_F(X)$. Let $D_{\pi_i}(X)$ be the domain of variable X right after the i'th execution of propagator π. The delta domain of variable X with respect to propagator π_i is $\Delta_{\pi_i}(X) = D_{\pi_{i-1}}(X) \ominus D_{\pi_i}(X)$.

Maintaining delta domains is a complex task. Delta domains must be collected, stored and made available later during a fixpoint operation. Moreover, each propagator has its own (possibly distinct) set of deltas which must be updated independently.

The basic idea is to store $\Delta(X) = \{\delta_1 \dots \delta_n\}$ in each set variable X as the sequence (a singly-linked list is used) of every atomic operation δ_i applied on its domain since the last fixpoint. In this context, δ_i is either a removal or insertion of a range of contiguous elements respectively from the set *lub* or in the set *glb*. A delta domain with respect to some propagator execution $\Delta_{\pi_i}(X)$ is then just a subsequence from the current $\Delta(X)$. Although the full details of this task are out of the scope for this paper, we note that domains may be maintained almost for free on constraint solvers with a smart garbage collection mechanism.

In order to create and manipulate graph domain variables we provide two constructors (one for directed and one for undirected graph variables) which provide the core constraints of the graph constraint solver.

All the basic operations for accessing and modifying the vertices and edges are supported by finite sets primitives, so no additional functionality is needed. Therefore, it is possible to create and manipulate graph variables for use in constraint problems just by providing two simple constraints for graph variable creation and delegating to a set solver the underlying core operations on sets.

These core constraints allow basic manipulation of graph variables, but we also define some other, more complex, constraints based on the core ones thus providing a more powerful, intuitive and declarative set of functions for graph variable manipulation. One of these constraints is the path constraint.

As specified in [22,23] the *path* constraint can be specified as (we only specify the rule for directed graph variables, being the one for undirected ones very similar):

$$path(G_D, v_0, v_f) \equiv quasipath(G_D, v_0, v_f) \wedge weakly\_connected(G_D)$$

which basically says that for ensuring the *path* constraint, one can just ensure a *quasipath* constraint and a *weakly_connected* constraint. *weakly_connected* is a constraint imposing that any two vertices of a graph are reachable from one another, disregarding the orientation of the edges (please consult [23] for details). In turn, the *quasipath* constraint, is a degree constraint, imposing that every vertex of a graph has exactly one predecessor and one successor in the graph. The *quasipath* constraint, for directed graph variables, is specified as:

$$quasipath(G_D, v_0, v_f) \equiv \begin{array}{c} \forall v \in V(G_D) \\ predecessors(G_D, v, P) \wedge \\ successors(G_D, v, S) \wedge \end{array} \begin{cases} \#P = 0 \wedge \\ \#S = 1 & , if\ v = v_0 \\ \\ \#P = 1 \wedge \\ \#S = 0 & , if\ v = v_f \\ \\ \#P = 1 \wedge \\ \#S = 1 & , otherwise \end{cases}$$

which basically dictates that every vertex that belongs to the graph has to have exactly one predecessor and one successor, exceptions being the initial vertex which has no predecessor and the final vertex which has no successor. $predecessors(G_D, v, P)$ and $successors(G_D, v, S)$ represent the constraints for imposing the predecessors and successors of a vertex in a graph.

The $predecessors(G_D, v, P)$ constraint can be expressed as:

$$predecessors(G_D, v, P) \equiv P \subseteq V(G_D) \wedge \forall v' \in V(G_D) : (v' \in P \equiv (v', v) \in E(G_D))$$

Similarly, the $successors(G_D, v, S)$ constraint can be expressed as:

$$successors(G_D, v, S) \equiv S \subseteq V(G_D) \wedge \forall v' \in V(G_D) : (v' \in S \equiv (v, v') \in E(G_D))$$

In the next section we explain how the *path* constraint was implemented in *GRASPER* and also explain, in general terms, how *CP(Graph)* imposed this constraint, analyzing both solutions.

3 Imposing the *path* constraint

Regarding *GRASPER*'s initial implementation, on imposing the *quasipath* constraint the first task was to iterate over all vertices in the graph variable's *glb* and to impose that their predecessor and successor sets had a cardinality of 1 (exceptions being the initial and final vertices as explained previously), thus ensuring that every vertex imposed *a priori* to be part of the solution respects the *quasipath* constraint.

The next task was to iterate over all vertices in the graph variable's *poss*. Since they are in the graph variable's *poss* we can not just impose the cardinality of their predecessor and successor sets to have a cardinality of 1 because some of

these vertices may not become part of the solution and hence they must have the cardinality of their predecessor and successor sets set to 0. A strategy is needed to enforce the *quasipath* constraint on these vertices, such that when one of them is imposed to be part of the solution, the cardinality of its predecessor and successor sets is set to 1, and such that when one of them is imposed not to be part of the solution, the cardinality of its predecessor and successor sets is set to 0.

In order to tackle this problem the following strategy was adopted: we obtained the predecessor and successor sets for each of these vertices and stored them for future access. After this storage, we could reason upon these sets in the following way:

- If at any time, a vertex is removed from the graph then its predecessor and successor sets cardinality is set to 0
- If at any time, a vertex is added to the graph then its predecessor and successor sets cardinality is set to 1
- If at any time, one vertex has the cardinality of one of its predecessor or successors sets instantiated to 0, then the vertex is removed from the graph
- If at any time, one vertex has the cardinality of one of its predecessor or successors sets instantiated to 1, then the vertex is added to the graph

This implementation is indeed very declarative and intuitive since it is basically a direct translation into constraints of the actual problem. We used this implementation and developed a solution for the *Metabolic Pathways Problem*, whose results we were able to compare against the ones obtained with *CP(Graph)*'s solution and even though not as efficient for the best heuristic we concluded that the results were acceptable and that *GRASPER* was nonetheless an alternative to using *CP(Graph)*.

There were, however, some problems with this implementation regarding both space and time. The problem with space is that basically we are obtaining and storing the predecessor and successor sets of each vertex in the graph variable's *poss* even though we do not know whether a given vertex will become part of the actual solution or not. In a worst case scenario, if we have N vertices and the graph is complete (every vertex is adjacent to every other), then we will have $O(N^2)$ spatial complexity just to store the predecessor and successor sets, which is clearly very expensive.

Regarding time, this solution had several problems. First of all, and applying the same reasoning as before, we were wasting time obtaining the predecessor and successor sets of each vertex in the graph variable's *poss* not knowing if they would ever become useful. Not only did we wasted time obtaining these sets but we also wasted time in maintaining them, since for each of these sets we had to maintain their consistency with the graph variable. Considering for instance the successor set of a vertex v, we had to maintain consistency in the following way:

- If a vertex s is removed from the successor set, the corresponding edge (v, s) must be removed from the graph
- If a vertex s is added to the successor set, the corresponding edge (v, s) must be added to the graph
- If an edge (v, s) is removed from the graph, the vertex s is removed from v's

successor set

- If an edge (v, s) is added to the graph, the vertex s is added to v's successor set

A cost linear in the number of vertices and edges of the graph is required, in a worst case scenario, every time one of these operations were performed. Given that each of these operations is performed for every predecessor or successor sets, and that we have a predecessor and successor sets for each vertex in the graph variable's *poss*, many of which may not belong to the solution, it is easy to conclude we were wasting a considerable amount of time.

CP(Graph), in turn, uses a different method for imposing this constraint. It defines a view over the graph variable's domain, more suitable for this problem than a vertex-set and edge-set representation. This view provides an adjacency representation, i.e., it maintains for each vertex a list of its adjacent vertices and it requires some form of consistency maintenance, ensuring that any change in the raw domain representation is reflected into the view and vice-versa.

Upon this view, *CP(Graph)* enforces directly the constraint by enforcing each vertex (except for the initial and final one) to:

- Having exactly one predecessor iff the vertex is in the graph's *glb*
- Having exactly one successor iff the vertex is in the graph's *glb*

which is basically a direct translation of the problem into a network of constraints. The major difference between both methods is the underlying structure that is used to impose these constraints. In the case of *GRASPER* we fetched *a priori* all the predecessor and successor sets for each vertex, which as explained previously, is very time and space consuming, whereas *CP(Graph)* opted for developing a view over the graph raw domain structure which could provide very efficient access to the vertices adjacency sets.

In *GRASPER* maintaining consistency for the *path* constraint implies sweeping the graph raw domain entirely, for each predecessor and each successor sets of each vertex, whereas in *CP(Graph)* consistency maintenance requires only sweeping the domain once and updating the view.

However, *CP(Graph)*'s method still has some of the undesirable properties mentioned previously for the *GRASPER* implementation. First of all, albeit in a much smaller scale, there is still some overhead in maintaining consistency between the graph raw domain and the view since a change in the graph raw domain will require an entire sweep over it, in order to update the view. Secondly, using this method implies a duplication of memory usage, since the view is actually another data structure that stores the graph information but in a different way. Last, but not least, the problem of imposing constraints over vertices that may not be part of the solution remains and hence, the feeling of wasting resources needlessly persists.

In the next section we explain how we can use a lazy mechanism for imposing constraints that will both save considerable space and, most importantly, considerable time on imposing the *path* constraint and that can solve the mentioned problems of the methods used by *GRASPER* and *CP(Graph)*.

4 Lazy constraint imposing for the *path* constraint

As explained in the previous section, considerable space was required, in *GRASPER* to store the predecessor and successor sets of all vertices belonging the the graph variable's *poss* as well as considerable time for obtaining those sets and maintaining their consistency.

While one is able to accept consuming space and time for storing and maintaining those sets for vertices that may become part of the solution one is not, however, able to accept consuming those resources for vertices that present no guarantee of becoming part of the solution.

What we are seeking is basically, a lazy mechanism for delaying, as much as possible, obtaining the predecessor and successor sets (and maintaining their consistency) of a given vertex until it is actually considered part of the solution.

This is easily achieved in the following way. First, and as done in the original implementation, the predecessor and successor sets for all the vertices already in the graph variable's *glb* are obtained and their cardinality is instantiated to 1 (except for the initial and final vertices, as explained previously), by the same reasons we mentioned in the previous section.

Secondly, all vertices in the graph variable's *poss* are iterated upon and an associative table is built, as before, but this time the vertices predecessor and successor sets will not be stored there. This time, only two integer variables are stored: one integer variable for the number of edges having the vertex as out-vertex, i.e., the number of predecessors of the vertex; and another integer variable for the number of edges having the vertex as in-vertex, i.e., the number of successors of the vertex.

This far, a considerable amount of memory has been spared since only two integer variables are stored for each of the vertices in the graph variable's *poss*.

After this initialization phase, one can reason upon the information present in this table in order to perceive when to impose the actual degree constraint over the vertices. Consistency between the graph raw domain and the degree associative table is done whenever there is a change in the graph's domain (removal of vertex, removal of edge, addition of vertex, addition of edge), in the following way:

- If a vertex is removed from the graph then it is not being considered to make part of the solution and therefore no degree constraint should be posed upon it and thus the information present in the associative table, for that vertex, may be simply disregarded.
- If an edge is removed from the graph, an update of the table is performed. The successor counter for the in-vertex and the predecessor counter for the out-vertex are decremented. If any of these values reaches 0 then, clearly, the corresponding vertex cannot be part of the solution and, therefore can be removed from the graph.
- If a vertex is added to the graph then it may make part of the solution and therefore we are finally in the situation where one is able to accept consuming resources to obtain the vertex predecessor and successor set, to maintain their consistency and to impose that their cardinality is 1.
- If an edge is added to the graph, we are again in the situation where one is able to

accept consuming the above mentioned resources and therefore the predecessor and successor sets of both end-vertices are obtained (if they have not already been obtained) and their cardinality is instantiated to 1.

This method is clearly very efficient in terms of memory consumption and it also manages not to waste time imposing heavy constraints on vertices that may not ever be part of a solution and thus, with this mechanism, we solve *GRASPER*'s problem of consuming resources for vertices that present no guarantee of becoming part of the solution.

Additionally, we improve, in space, on *CP(Graph)* since we do not need an actual duplication of the graph. Our associative degree table stores a pair of integers for each vertex, whereas *CP(Graph)* maintains a view over the graph raw domain, which is actually, a complete copy of the graph but with a different representation, more suitable for the operations required by the *path* constraint.

Finally, we also improve in time, on *CP(Graph)* since every time a change in the graph domain occurs, we do not need an entire sweep over the domain in order to maintain consistency between the domain and the associative table. Since *GRASPER* has access to delta domain variables and these store information of what changed in a variable's domain we can, in constant time, determine what this change was in our propagators. Hence, whenever a change occurs, we just need to consult the delta domain variable, query what the change was and update the corresponding information in the associative table.

5 Results

In this section results are presented, for both implementations of *GRASPER* and *CP(Graph)*, obtained in solving the *Metabolic Pathways Problem*.

Metabolic networks (see [15,12,21] for a general overview of metabolic networks) are biochemical networks which encode information about molecular compounds and reactions which transform these molecules into substrates and products. A pathway in such a network represents a series of reactions which transform a given molecule into others.

An application for pathway discovery (see [18,20] for more details on pathway discovery) in metabolic networks is the explanation of DNA experiments. An experiment is performed on DNA cells and these mutated cells (called RNA cells) are placed on DNA chips, which contain specific locations for different strands, so when the cells are placed in the chips, the different strands will fit into their specific locations. Once placed, the DNA strands (which encode specific enzymes) are scanned and catalyse a set of reactions. Given this set of reactions the goal is to know which products were active in the cell, given the initial molecule and the final result.

A recurrent problem in metabolic networks pathway finding is that many paths take shortcuts, in the sense that they traverse highly connected molecules (act as substrates or products of many reactions) and therefore cannot be considered as belonging to an actual pathway. However there are some metabolic networks for which some of these highly connected molecules act as main intermediaries.

It is also possible that a path traverses a reaction and its reverse reaction: a

reaction from substrates to products and one from products to substrates. Most of the time these reactions are observed in a single direction so we can introduce *exclusive pairs of reactions* to ignore a reaction from the metabolic network when the reverse reaction is known to occur, so that both do not occur simultaneously.

Additionally, it is possible to have various pathways in a given metabolic experiment and often the interest is not to discover one pathway but to discover a pathway which traverses a given set of intermediate products or substrates, thus introducing the concept of *mandatory molecule*. These *mandatory molecules* are useful, for example, if biologists already know some of the products which are in the pathway but do not know the complete pathway.

The problem of metabolic pathway finding is thus to determine a sequence of reactions that form a path between the starting and finishing molecule, avoiding whenever possible highly connected molecules, ensuring that exclusive pair of reactions cannot appear simultaneously in a solution and that all the mandatory molecules are visited.

Basically, assuming that $G = dirgraph(V, E)$ is the original graph, composed of all the vertices and edges of the problem, that v_0 and v_f are the initial and the final vertices, that $Mand = \{v_1, \ldots, v_n\}$ is the set of *mandatory* vertices, that $Excl = \{(v_{e11}, v_{e12}), \ldots, (v_{em1}, v_{em2})\}$ is the set of *exclusive* pairs of vertices and that W_f is a function mapping each vertex and each edge to its degree, this problem can be easily modeled in *GRASPER* as:

$$
\begin{gathered}
minimise(W): \\
subgraph(dirgraph(SubV, SubE), G) \wedge \\
Mand \subseteq SubV \wedge \\
\forall (v_{ei1}, v_{ei2}) \in Excl : (v_{ei1} \notin SubV \vee v_{ei2} \notin SubV) \wedge \\
path(dirgraph(SubV, SubE), v_0, v_f) \\
weight(dirgraph(SubV, SubE), W_f, W)
\end{gathered}
$$

The minimisation function can be found built-in in almost every constraint programming environment. The subgraph relation is directly mapped to our *subgraph* constraint (consult [23] for details on the *subgraph* constraint) and its objective is to allow the extraction of the actual pathway from the original graph containing every vertex and edge from the original problem. The introduction of the *mandatory vertices* is easily achieved by a mere set inclusion operation. The *exclusive pairs of reactions* demand the implementation of a very simple propagator which basically removes one vertex once it is known that another vertex has been added to the graph and they form an *exclusive pair of reactions*. The weighting of the graph is performed using the *weight* constraint (consult [23] for details on the *weight* constraint). These simple operations sketch the basic modelling for this problem, however it is still necessary to perform search so as to trigger the propagators and determine the set of vertices that belong to the pathway and the edges that connect them.

We use a labeling strategy that consists in iteratively extending a path (initially

formed only by the starting vertex) until reaching the final vertex. At every step, we determine the next vertex which extends the current path to the final vertex minimizing the overall path cost. Having this vertex we obtain the next edge to label by considering the first edge extending the current path until the determined vertex. The choice step consists in including/excluding the edge from the graph variable. If the edge is included the current path is updated and the last vertex of the path is the out-vertex of the included edge, otherwise the path remains unchanged and we try another extension. The search ends as soon as the final vertex is reached and the path is minimal. This heuristic shall be referred as *shortest-path* [19].

Below, we present the results obtained for the problem of solving the shortest metabolic pathways for three metabolic chains (glycosis, heme and lysine) and for increasing graph orders (the order of a graph is the number of vertices that belong to the graph), having one instance per graph order. The instances were obtained from [5] and are the same ones used in [9,8].

We ran both implementations and *CP(Graph)*'s implementation [5] on an Intel Core 2 Duo 2.16 GHz, 4 Mb of L2 Cache, 1.5 Gb of RAM, on graph instances having from 500 to 2000 vertices and for different metabolic networks (glycosis, lysine and heme) and using the *shortest-path* heuristic. Table 1 presents the results, in seconds, where G_{old} denotes the original version, G_{new} the version with the lazy constraint imposing mechanism and *CP(Graph)*, the *CP(Graph)*'s implementation.

Order	Glycosis			Lysine			Heme		
	G_{old}	G_{new}	$CP(Graph)$	G_{old}	G_{new}	$CP(Graph)$	G_{old}	G_{new}	$CP(Graph)$
500	1.13	0.28	0.21	1.37	0.36	0.41	0.73	0.22	0.10
600	1.75	0.38	0.31	1.74	0.48	0.44	1.05	0.28	0.12
700	2.23	0.45	0.35	2.16	0.47	0.75	1.34	0.36	0.16
800	2.86	0.53	0.50	2.65	0.53	1.00	1.67	0.41	0.19
900	3.69	0.64	0.68	3.23	0.57	1.29	2.12	0.51	0.27
1000	4.85	0.77	0.84	3.57	0.60	1.37	2.62	0.62	0.32
1100	6.10	0.91	1.00	4.66	0.73	1.29	2.98	0.65	0.33
1200	6.60	0.96	1.08	5.76	0.86	2.23	3.73	0.80	0.41
1300	7.47	1.03	1.21	6.95	0.99	2.50	5.06	0.94	0.47
1400	9.12	1.23	1.56	7.99	1.12	2.84	5.12	1.11	0.51
1500	10.60	1.40	1.85	8.98	1.25	2.92	5.46	1.14	0.52
1600	12.50	1.67	2.14	9.80	1.30	2.97	6.60	1.35	0.61
1700	14.70	1.93	2.40	10.40	1.41	3.03	7.61	1.57	0.69
1800	16.70	2.11	2.77	12.00	1.53	3.69	8.69	1.72	0.77
1900	18.70	2.27	3.02	13.60	1.75	3.93	9.75	1.96	0.84
2000	19.50	2.40	3.14	15.30	1.96	2.18	10.80	2.18	0.91

Table 1
Metabolic Pathways Problem results between *GRASPER* versions and *CP(Graph)*

Analyzing the results obtained for both implementations of *GRASPER* we conclude that, for every instance of the problem and for all of the metabolic networks, the lazy constraint imposing mechanism has a major impact on the effectiveness of the application, managing to increase efficiency up to 8 times when comparing to the original version.

[5] We used version 1.3.1 of *GECODE* (available at http://ww.gecode.org) which is the last version upon which *CP(Graph)* runs on. *CP(Graph)* has been discontinued on *GECODE*.

We also conclude that *GRASPER* is able to outperform *CP(Graph)* for the *glyco* chain, being able to improve 25% over *CP(Graph)*'s results on the higher instances. Regarding the *lysine* chain, *GRASPER* achieved a speed-up of 2 for some of the larger instances. Conversely, for the *heme* chain, *GRASPER* could not achieve the same results as *CP(Graph)*, taking sometimes twice the time of *CP(Graph)* to solve the problem. The heuristic used can find a solution for the instances of the *heme* chain very efficiently when we directly apply all the constraints, which may explain why the results obtained using the lazy constraint imposing mechanism were not as efficient as the ones obtained with *CP(Graph)*.

6 Conclusions and future work

In this paper, *GRASPER*'s *path* constraint definition was presented, being specified how its first implementation was performed, showing that it used considerable space and used often too much time for imposing its constraints, although showing acceptable results when comparing to a state-of-the-art solver.

We proposed to use a lazy constraint imposing mechanism for a new implementation that could optimize used space and that would only spend time maintaining consistency on variables that would give some evidence of being part of the solution. We implemented a new version of *GRASPER* with such a mechanism and we compared it against the original one and against *CP(Graph)*, the state-of-the-art solver, and in that comparison, *GRASPER*'s new version was able to outperform the old version (by far) and also *CP(Graph)* for a large set of instances, appearing as a serious alternative to it.

Future work includes investigating where could this mechanism be also applied to, in order to further decrease space consumption and also to further improve the solver's efficiency.

We are also applying the same mechanism for undirected graph variables, which allowed us to discover a possible improvement in the graph variable domain representation that we also believe may substantially improve the solver's efficiency.

Additionally, we intend to implement solutions to other path constraining problems and using bigger and more difficult instances in order to determine the limits of our application.

References

[1] Teresa Attwood and Douglas Parry-Smith. *Introduction to Bioinformatics*. Prentice Hall, 1999.

[2] Francisco Azevedo. Cardinal: A Finite Sets Constraint Solver. *Constraints journal*, 12(1):93–129, 2007.

[3] Gary Chartrand. *Introductory Graph Theory*. Dover Publications, 1984.

[4] Marco Correia, Pedro Barahona, and Francisco Azevedo. CaSPER: A Programming Environment for Development and Integration of Constraint Solvers. In F. Azevedo, C. Gervet, and E. Pontelli, editors, *Proceedings of the First International Workshop on Constraint Programming Beyond Finite Integer Domains (BeyondFD'05)*, pages 59 – 73, 2005.

[5] Didier Croes. *Recherche de chemins dans le réeseau métabolique et mesure de la distance métabolique entre enzymes*. PhD thesis, ULB, Belgium, 2005.

[6] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[7] Reinhard Diestel. *Graph Theory*. Springer-Verlag, 2005.

[8] Grégoire Dooms. *The CP(Graph) Computation Domain in Constraint Programming*. PhD thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain, 2006.

[9] Grégoire Dooms, Yves Deville, and Pierre Dupont. CP(Graph): Introducing a Graph Computation Domain in Constraint Programming. In *Eleventh International Conference on Principles and Practice of Constraint Programming*, number 3709 in Lecture Notes in Computer Science, pages 211–225. Springer-Verlag, 2005.

[10] C. Gervet. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints journal*, 1(3):191–244, 1997.

[11] Jeffrey Herrmann. *Handbook of Production Scheduling*. Springer-Verlag, 2006.

[12] Hawoong Jeong, Bálint Tombor, Réka Albert, Zoltán Oltvai, and Albert-László Barabási. The large-scale organization of metabolic networks. *Nature*, 406, 2000.

[13] R. Lyon. Auctions and alternative procedures for allocating pollution rights. *Land Economics*, 58(1):16–32, 1982.

[14] Kim Marriot and Peter Stuckey. *Programming with Constraints: An introduction*. MIT Press, 1998.

[15] Christopher Mathews and Kensal van Holde. *Biochemistry*. Benjamin Cummings, 2 edition, 1996.

[16] J. McMillan. Selling spectrum rights. *Journal of Economic Perspectives*, 8(3):145–162, 1994.

[17] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier Science Inc., 2006.

[18] Christophe Schilling, Stefan Schuster, Bernhard Palsson, and Reinhart Heinrich. Metabolic pathway analysis: basic concepts and scientific applications in the post-genomic era. *Biotechnology Programming*, 15(3), 1999.

[19] Meinolf Sellmann. Cost-based filtering for shorter path constraints. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2833 of *Lecture Notes in Computer Science*, pages 694–708. Springer-Verlag, 2003.

[20] Axel von Kamp Stefan Schuster and Mikhail Pachkov. *Understanding the Roadmap of Metabolism by Pathway Analysis*, volume 358 of *Methods in Molecular Biology*, chapter Biomedical and Life Sciences, pages 199–226. Humana Press, 2008.

[21] Jacques van Helden, Lorenz Wernisch, David Gilbert, and Shoshana Wodak. *Graph-based analysis of metabolic networks*, pages 245–274. Springer-Verlag, 2002.

[22] Ruben Viegas and Francisco Azevedo. GRASPER: A Framework for Graph CSPs. In Jimmy Lee and Peter Stuckey, editors, *Proceedings of the Sixth International Workshop on Constraint Modelling and Reformulation (ModRef'07)*, Providence, Rhode Island, USA, September 2007.

[23] Ruben Duarte Viegas. GRASPER: Constraint Reasoning with graphs. Master's thesis, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, February 2008.

[24] Junming Xu. *Theory and Application of Graphs*, volume 10 of *Network Theory and Applications*. Kluwer Academic Publishers, 2003.

Author Index