European Joint Conferences on Theory and Practice of Software

26 March – 03 April, 2011

Saarbrücken, Germany

TERMGRAPH 2011

6th International Workshop on Computing with Terms and Graphs

Rachid Echahed (Ed.)

http://www.etaps.org

http://etaps2011.cs.uni-saarland.de

Universität des Saarlandes

EPTCS 48

Proceedings of the

6th International Workshop on
Computing with Terms and Graphs

Saarbrücken, Germany, 2nd April 2011

Edited by: Rachid Echahed

Published: 11th February 2011
DOI: 10.4204/EPTCS.48
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Table of Contents i

Preface ii
Rachid Echahed

Invited Presentation: On the Observable Behavior of Graph Transformation Systems 1
Reiko Heckel

Invited Presentation: From Infinitary Term Rewriting to Cyclic Term Graph Rewriting and back . . 2
Patrick Bahr

Term Graph Rewriting and Parallel Term Rewriting 3
Andrea Corradini and Frank Drewes

Invited Presentation: Nominal Graphs 19
Maribel Fernandez

Rule-based transformations for geometric modelling 20
Thomas Bellet, Agnès Arnould and Pascale Le Gall

Dependently-Typed Formalisation of Typed Term Graphs 38
Wolfram Kahl

PORGY: Strategy-Driven Interactive Transformation of Graphs 54
Oana Andrei, Maribel Fernandez, Hélène Kirchner, Guy Melançon, Olivier Namet and Bruno Pinaud

A new graphical calculus of proofs 69
Sandra Alves, Maribel Fernández and Ian Mackie

Repetitive Reduction Patterns in Lambda Calculus with letrec 85
Jan Rochel and Clemens Grabmayer

Preface

This volume contains the proceedings of the Sixth International Workshop on Computing with Terms and Graphs (TERMGRAPH 2011). The workshop took place in Saarbrcken, Germany, on April 2nd, 2011, as part of the fourteenth edition of the European Joint Conferences on Theory and Practice of Software (ETAPS 2011).

Research in term and graph rewriting ranges from theoretical questions to practical issues. Computing with graphs handles the sharing of common subexpressions in a natural and seamless way, and improves the efficiency of computations in space and time. Sharing is ubiquitous in several research areas, for instance : the modelling of first- and higher-order term rewriting by (acyclic or cyclic) graph rewriting, the modelling of biological or chemical abstract machines, the implementation techniques of programming languages: many implementations of functional, logic, object-oriented, concurrent and mobile calculi are based on term graphs. Term graphs are also used in automated theorem proving and symbolic computation systems working on shared structures. The aim of this workshop is to bring together researchers working in different domains on term and graph transformation 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.

Previous editions of TERMGRAPH series were held in Barcelona (2002), in Rome (2004), in Vienna (2006), in Braga (2007) and in York (2009).

These proceedings contain six accepted papers and the abstracts of three invited talks. All submissions were subject to careful refereeing. The topics of accepted papers range over a wide spectrum, including theoretical aspects of term graph rewriting, proof methods, semantics as well as application issues of term graph transformation.

We would like to thank all who contributed to the success of TERMGRAPH 2011, especially the Program Committee for their valuable contributions to the selection process as well as the contributing authors. We would like also to express our gratitude to all members of ETAPS 2011 organizing committee for their help in organizing TERMGRAPH 2011 at Saarbrcken, Germany.

February, 2011

Rachid Echahed
Program chair of TERMGRAPH2011

Rachid Echahed (Ed.): 6th International Workshop
on Computing with Terms and Graphs (TERMGRAPH 2011)
EPTCS 48, 2011, pp. ii–iii, doi:10.4204/EPTCS.48.0

© R. Echahed
This work is licensed under the
Creative Commons Attribution License.

Program committee of TERMGRAPH 2011

Paolo Baldan	University of Padova, Italy
Andrea Corradini	University of Pisa, Italy
Frank Drewes	Umea University, Sweden
Rachid Echahed	CNRS, LIG Lab., Grenoble, France (chair)
Tetsuo Ida	University of Tsukuba, Japan
Wolfram Kahl	McMaster University, Canada
Ian Mackie	Ecole Polytechnique, France
Detlef Plump	University of York, UK

On the Observable Behavior of Graph Transformation Systems

Joint GT-VMT and TERMGRAPH Invited Talk

Reiko Heckel
Department of Computer Science
University of Leicester, UK

While modelling or testing component-based or service-oriented applications we often complement the external perspective, describing the system in terms of its interactions, by the internal one specifying its implementation. To formalise this view, a graph transformation system is seen as an encapsulated component: The implementation, described by type graph and rules, is equipped with an interface (a signature of rules with parameters) defining possible observations. We use this model to study the conditions under which observations of interactions denote faithfully the internal concurrent processes in the system.

The approach is based on observing implementation-level dependencies and conflicts in terms of critical pairs, and requiring that labels preserve these dependencies. Besides establishing the fundamental relationship between transformations and observations, we investigate possible applications in testing and software evolution.

This is a joint work with Tamim Khan (University of Leicester, UK) and Rodrigo Machado (Universidade Federal do Rio Grande do Sul, Porto Allegre, Brazil).

Rachid Echahed (Ed.): 6th International Workshop
on Computing with Terms and Graphs (TERMGRAPH 2011)
EPTCS 48, 2011, pp. 1–1, doi:10.4204/EPTCS.48.1

© R. Heckel
This work is licensed under the
Creative Commons Attribution License.

From Infinitary Term Rewriting to Cyclic Term Graph Rewriting and back

Invited Talk

Patrick Bahr
Department of Computer Science
University of Copenhagen
Universitetsparken 1, 2100 Copenhagen, Denmark
paba@diku.dk

Cyclic term graph rewriting has been shown to be adequate for simulating certain forms of infinitary term rewriting. These forms are, however, quite restrictive and it would be beneficial to lift these restriction at least for a limited class of rewriting systems. In order to better understand the correspondences between infinite reduction sequences over terms and finite reductions over cyclic term graphs, we explore different variants of infinitary term graph rewriting calculi.

To this end, we study different modes of convergence for term graph rewriting that generalise the modes of convergence usually considered in infinitary term rewriting. After discussing several different alternatives, we identify a complete semilattice on term graphs and derive from it a complete metric space on term graphs. Equipped with these structures, we can – analogously to the term rewriting case – define both a metric and a partial order model of infinitary term graph rewriting. The resulting calculi of infinitary term graph rewriting reveal properties similar to the corresponding infinitary term rewriting calculi.

Rachid Echahed (Ed.): 6th International Workshop
on Computing with Terms and Graphs (TERMGRAPH 2011)
EPTCS 48, 2011, pp. 2–2, doi:10.4204/EPTCS.48.2

© P. Bahr
This work is licensed under the
Creative Commons Attribution License.

Term Graph Rewriting and Parallel Term Rewriting

Andrea Corradini
Università di Pisa
Dipartimento di Informatica
andrea@di.unipi.it

Frank Drewes
Department of Computing Science
Umeå University
drewes@cs.unu.se

The relationship between Term Graph Rewriting and Term Rewriting is well understood: a single term graph reduction may correspond to several term reductions, due to sharing. It is also known that if term graphs are allowed to contain cycles, then one term graph reduction may correspond to infinitely many term reductions. We stress that this fact can be interpreted in two ways. According to the *sequential interpretation*, a term graph reduction corresponds to an infinite sequence of term reductions, as formalized by Kennaway et. al. using strongly converging derivations over the complete metric space of infinite terms. Instead according to the *parallel interpretation* a term graph reduction corresponds to the parallel reduction of an infinite set of redexes in a rational term. We formalize the latter notion by exploiting the complete partial order of infinite and possibly partial terms, and we stress that this interpretation allows to explain the result of reducing circular redexes in several approaches to term graph rewriting.

1 Introduction

The theory of *Term Graph Rewriting* (TGR) studies the issue of representing finite terms with directed, acyclic graphs, and of modeling term rewriting via graph rewriting. This field has a long history in the realm of theoretical computer science, its origin dating back to the seventies of the last century, when dags were proposed in [25] as an efficient implementation of recursive program schemes. Among the many contributions to the foundations of this field, we mention [26, 5, 22, 20, 10, 1, 8]. The various approaches may differ for the way term graphs are represented or for the precise definition of the graph rewriting mechanism, but they all present equivalent results for what concerns the speed-up of term rewriting due to the explicit sharing.

In fact, the main advantage of using graphs is that when applying a rewrite rule, the subterm matched by a variable x of the left-hand side does not need to be copied if x appears more than once in the right-hand side, because the sharing of subterms can be represented explicitly. Therefore the rewriting process is speeded up, because the rewriting steps do not have to be repeated for each copy of a subterm.

For example, suppose that rule $s(x) \to k(x, r(x))$ is applied to term $s(f(a))$, obtaining term $t = k(f(a), r(f(a)))$. Now rule $R_f : f(x) \to g(x)$ can be applied twice to term t, yielding in two steps term $t' = k(g(a), r(g(a)))$. If instead t is represented as a graph where the two identical subterms are shared

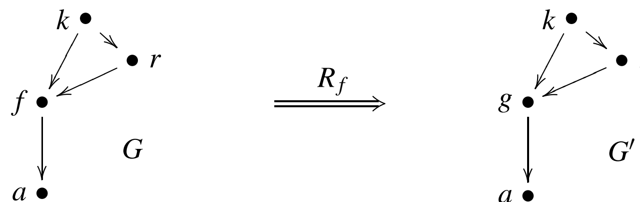


Figure 1: An example of term graph rewriting

Rachid Echahed (Ed.): 6th International Workshop
on Computing with Terms and Graphs (TERMGRAPH 2011)
EPTCS 48, 2011, pp. 3–18, doi:10.4204/EPTCS.48.3

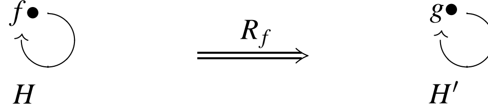


Figure 2: An example of cyclic term graph rewriting

(as in graph G of Figure 1), then a single application of the rule is sufficient to reduce it to graph G' of Figure 1, which clearly represents term t'. Thus a single graph reduction may correspond to n term reductions, where n is the "degree of sharing" of the reduced subterm.

Often this fact is spelled out by observing that a single TGR reduction corresponds to *a sequence* of n term reductions [5, 20], but one may equivalently think that the n term reductions are performed *in parallel*, in a single step. The idea of reducing families of redexes in parallel (*family reductions*) was proposed already in [28, 6] as an alternative to the use of dags for representing the sharing of subterms.

Some authors considered the extension of term graph rewriting to the cyclic case, allowing (finite, directed) cyclic graphs as well. Actually, already in [5] the definition did not forbid cycles, and the relationship with term rewriting in this case was analyzed in depth in [23], as discussed below. It was observed that allowing term graphs with cycles one could represent certain structures that arise when dealing with recursive definitions (as for the implementation of the fixed point operator Y proposed in [27]). Interestingly, even using acyclic rules, cyclic graphs can be produced by rewriting in presence of suitable sharing strategies [17, 18]. Cyclic term graph rewriting was also defined abstractly in a categorical setting in [9], showing the equivalence with an operational definition; and it was discussed in the framework of Equational Term Graph Rewriting in [1].

A renewed interest in term graph rewriting with cycles is witnessed by some recent publications. In [3, 4] the authors propose an extension of the ρ-calculus where the sharing of subterms can be modeled explicitly and cyclic definitions are allowed. In [15] cyclic term graphs are used to represent data structures, and rewriting models the transformation of such structures with both local and global redirection of pointers. In [14] they are used for the definition of the type system of an object oriented language.

Considering the relationship with term rewriting, the first consequence of the extension to cyclic term graphs is that infinite terms (or, more precisely, *rational* terms, i.e., infinite terms with a finite number of distinct subterms) can be represented as well. The second effect is that a single graph reduction may now correspond to some infinite term rewriting. Consider for example rule R_f above: by applying it to graph H of Figure 2 one obtains the graph H'. Clearly, H represents the infinite term $f^\omega = f(f(f(\ldots)))$, while H' represents term g^ω. As for the finite case, there are two possible ways of interpreting the rewriting of term f^ω to term g^ω via infinitely many applications of rule R_f:

[sequential interpretation] g^ω is the limit of an infinite sequence of applications of R_f, i.e., $f^\omega \to_{R_f} g(f^\omega) \to_{R_f} g(g(f^\omega)) \to_{R_f} \cdots \leadsto_\omega g^\omega$.

[parallel interpretation] g^ω is the result of the simultaneous application of R_f to an infinite number of redexes in f^ω: in a single step all the occurrences of f in f^ω are replaced by g.

The first interpretation has been thoroughly formalized in [12, 13, 17, 18, 23], where the authors elaborated a theory of (rational) *transfinite term rewriting*, showing that finite, cyclic graph rewriting is an adequate implementation for it. In essence, a finite graph derivation sequence has the "same effect" of a *strongly convergent* infinite term rewriting sequence. As far as the notion of convergence is concerned, the well-known topological structure of (possibly infinite) terms is used, which, equipped with a suitable notion of distance, form a complete ultra-metric space [2].

In this paper we provide instead a formalization of the *parallel interpretation* above. To this aim we exploit the theory of *infinite parallel term rewriting* that has been proposed by the first author in [7] by exploiting the complete partially ordered structure of CT_Σ, the collection of possibly infinite, possibly partial terms over signature Σ.

Interestingly, unlike the acyclic case for which the sequential and the parallel interpretations of term graph rewriting are completely equivalent, in the cyclic case there are cases where the two interpretations lead to different results. This happens when *collapsing* rules are considered, i.e., rules having a variable as right-hand side. The canonical collapsing rule is the rule for identity, $R_I : I(x) \to x$, and the pathological case (considered already by many authors) is the application of R_I to I^ω. Using the sequential interpretation, we have that $I^\omega \to_{R_I} I^\omega \to_{R_I} \ldots$, and thus the limit of the sequence is I^ω itself. Instead according to the parallel interpretation all the occurrences of I in I^ω are deleted in a single step, and thus we should obtain as result a term that does not contain any function symbol: we will show indeed that we get the completely undefined term $\perp$, the bottom element of the complete partial ordering CT_Σ.

Both the sequential and the parallel interpretations turn out to be meaningful from the point of view of cyclic term graph rewriting, because depending on the chosen rewriting approach one can get different results when applying the collapsing rule R_I to the *circular-I*, i.e., to the graph having a single node labelled by I and a loop, which clearly unravels to I^ω. In fact, if one uses the operational definition of term graph rewriting proposed in [5] (as done in [23, 17, 18]) then the circular-I reduces via R_I to itself, and this is consistent with the sequential interpretation.

Instead for several other definitions of term graph rewriting (including the double-pushout [20, 10], the single-pushout [22], the equational [1], an the categorical [9] approaches) the circular-I rewrites via R_I to a graph consisting of a single, unlabeled node, which can be regarded as the graphical representation of the undefined term $\perp$, and is therefore consistent with the parallel interpretation.

The paper is organized as follows. In Section 2 we summarize the basic definitions about infinite terms [19], orthogonal term rewriting [21], and parallel term rewriting [7]. In Section 3 we introduce (possibly cyclic) term graphs, and we make precise their relationship with (sets of) rational terms, via the unraveling function. Algebraic term graph rewriting is the topic of Section 4, where we recall the basics of the double-pushout approach [16], apply it to the category of term graphs, and provide an encoding of term rewrite rules as graph rules. The main results of the paper are in Section 5. We first prove that a single reduction of a graph induces on the unraveled term a possibly infinite parallel reduction. Next this fact is used as a main lemma in the proof that the unraveling function is an adequate mapping from any orthogonal TGRS system to the orthogonal TRS obtained by unraveling its rules, provided that only rational terms and rational parallel reduction sequences are considered. Finally in Section 6 we summarize our contribution.

2 Infinite terms and parallel term rewriting

In this background section we first introduce the algebra CT_Σ of possibly partial, possibly infinite terms. The collection of such terms forms a complete partial ordering which has been studied in depth in [19]. Next we introduce the basic definitions related to (orthogonal) term rewriting, which apply to infinite terms as well. Finally, we introduce the definition of infinite parallel rewriting, which exploits the CPO structure of terms according to [7].

2.1 Infinite Terms

Most of the following definitions are borrowed from [19].

Let ω^* be the set of all finite strings of positive natural numbers. Elements of ω^* are called **occurrences**. The empty string is denoted by λ, and $u \leq w$ indicates that u is a prefix of w. Occurrences u, w are called **disjoint** (written $u|w$) if neither $u \leq w$ nor $w \leq u$.

Let Σ be a (one-sorted) signature, i.e., a ranked alphabet of operator symbols $\Sigma = \cup_{n \in \mathbb{N}} \Sigma_n$ and let X be a set of variables. A **term** over (Σ, X) is a partial function $t : \omega^* \to \Sigma \cup X$, such that the domain of definition of t, $\mathscr{O}(t)$, satisfies the following (where $w \in \omega^*$ and all $i \in \omega$):

- $wi \in \mathscr{O}(t) \Rightarrow w \in \mathscr{O}(t)$
- $wi \in \mathscr{O}(t) \Rightarrow t(w) \in \Sigma_n$ for some $n \geq i$.

$\mathscr{O}(t)$ is called the **set of occurrences** of t. We denote by $\perp$ (called **bottom**) the empty term, i.e. the only term such that $\mathscr{O}(\perp) = \emptyset$.

Given an occurrence $w \in \omega^*$ and a term t, the **subterm** of t at (occurrence) w is the term t/w defined as $t/w(u) = t(wu)$ for all $u \in \omega^*$. A term t is **finite** if $\mathscr{O}(t)$ is finite; it is **total** if $t(w) \in \Sigma_n \Rightarrow wi \in \mathscr{O}(t)$ for all $0 < i \leq n$; it is **linear** if no variable occurs more than once in it; and it is **rational** if it has a finite number of different subterms. Given terms t, s and an occurrence $w \in \omega^*$, the **replacement** of s in t at (occurrence) w, denoted $t[w \leftarrow s]$, is the term defined as $t[w \leftarrow s](u) = t(u)$ if $w \not\leq u$ or $t/w = \perp$, and $t[w \leftarrow s](wu) = s(u)$ otherwise.

The set of terms over (Σ, X) is denoted by $CT_\Sigma(X)$ (CT_Σ stays for $CT_\Sigma(\emptyset)$). Throughout the paper we will often use (for finite terms) the equivalent and more usual representation of terms as operators applied to other terms. Partial terms are made total in this representation by exploiting the empty term $\perp$. Thus, for example, if $x \in X$, $t = f(\perp, g(x))$ is the term such that $\mathscr{O}(t) = \{\lambda, 2, 2 \cdot 1\}$, $t(\lambda) = f \in \Sigma_2$, $t(2) = g \in \Sigma_1$, and $t(2 \cdot 1) = x \in X$.

It is well known that $CT_\Sigma(X)$ forms a **complete partial order** with respect to the "approximation" relation. We say that t **approximates** t' (written $t \leq t'$) iff t is less defined than t' as partial function. The least element of $CT_\Sigma(X)$ with respect to $\leq$ is clearly $\perp$. An **ω-chain** $\{t_i\}_{i<\omega}$ is an infinite sequence of terms $t_0 \leq t_1 \leq \ldots$. Every ω-chain $\{t_i\}_{i<\omega}$ in $CT_\Sigma(X)$ has a **least upper bound** (lub) $\bigcup_{i<\omega}\{t_i\}$ characterized as follows:

$$t = \bigcup_{i<\omega}\{t_i\} \quad \Leftrightarrow \quad \forall w \in \omega^* . \exists i < \omega . \forall j \geq i . t_j(w) = t(w)$$

Moreover, every pair of terms has a greatest lower bound. All this amounts to say that $CT_\Sigma(X)$ is an **ω-complete lower semilattice**.

2.2 Term Rewriting

We recall here the basic definitions of (orthogonal) term rewriting [21], which apply to infinite terms as well.

Let X and Y be two sets of variables. A **substitution** (from X to Y) is a function $\sigma : X \to CT_\Sigma(Y)$ (used in postfix notation). Such a substitution σ can be extended in a unique way to a continuous (i.e., monotonic and lub-preserving) function $\sigma : CT_\Sigma(X) \to CT_\Sigma(Y)$, which extends σ as follows

- $\perp\sigma = \perp$,
- $f(t_1, ..., t_n)\sigma = f(t_1\sigma, ..., t_n\sigma)$,
- $(\bigcup_{i<\omega}\{t_i\})\sigma = \bigcup_{i<\omega}\{t_i\sigma\}$.

A **rewrite rule** $R = (l, r)$ is a pair of total terms of $CT_\Sigma(X)$, where $var(r) \subseteq var(l)$, l is finite and it is not a variable.[1] Terms l and r are called the **left-** and the **right-hand side** of R, respectively. A rule is called **left-linear** if l is linear, and it is **collapsing** if r is a variable. A **term rewriting system** (shortly **TRS**) $\mathscr{R}$ is a finite set of rewrite rules, $\mathscr{R} = \{R_i\}_{i \in I}$.

Given a term rewriting system $\mathscr{R}$, a **redex** (for REDucible EXpression) Δ of a term t is a pair $\Delta = (w, R)$ where $R : l \to r \in \mathscr{R}$ is a rule, and w is an occurrence of t, such that there exists a substitution σ which **realizes** Δ, i.e., such that $t/w = l\sigma$. In this case we say that t **reduces** (via Δ) to the term $t' = t[w \leftarrow r\sigma]$, and we write $t \to_\Delta t'$ or simply $t \to t'$. A **reduction sequence** $t_1 \to t_2 \to \ldots t_n$ is a finite sequence of reductions.

A TRS $\mathscr{R}$ is **orthogonal** (shortly, it is an **OTRS**) if all the rules in $\mathscr{R}$ are left-linear and it is **non-overlapping**; that is, the left-hand side of each rule does not unify with a non-variable subterm of any other rule in $\mathscr{R}$, or with a proper, non-variable subterm of itself. In this paper we will be concerned with orthogonal TRS's only, because the confluence of such systems is a key property needed in the next section to define parallel term rewriting.

2.3 Parallel Term Rewriting

As discussed in the introduction, in the parallel interpretation of term graph rewriting the reduction of a cyclic graph corresponds to the parallel reduction of a possibly infinite set of redexes in the corresponding term. The definitions below summarize, in a simplified way, those in [7]. Intuitively, *finite* parallel rewriting can be defined easily by exploiting the confluence of orthogonal term rewriting. In fact, the parallel reduction of a finite number of redexes is defined simply as any *complete development* of them: Any such development ends with the same term, so the result is well defined. Let us recall the relevant definitions.

Given two redexes of a term, the reduction of one of them can transform the other in various ways. The second redex can be destroyed, it can be left intact, or it can be copied a number of times. This is captured by the definition of residuals. We assume here that the rules belong to an OTRS, thus two redexes in a term are either the same or do not overlap.

Definition 1 (residuals)**.** Let $\Delta = (w, R)$ and $\Delta' = (w', R' : l' \to r')$ be two redexes in a term t. The **set of residual of** Δ **by** Δ' is denoted by $\Delta \backslash \Delta'$, and it is defined as

$$\Delta \backslash \Delta' = \begin{cases} \emptyset & \text{if } \Delta = \Delta' \\ \{\Delta\} & \text{if } w \not\geq w' \\ \{(w' w_x u, R) \mid r'/w_x = l'/v_x\} & \text{if } w = w' v_x u \text{ and } l'/v_x \text{ is a variable} \end{cases}$$

If Φ is a finite set of redexes of t and Δ is a redex of t, then the set of residuals of Φ by Δ, denoted $\Phi \backslash \Delta$, is defined as the union of $\Delta' \backslash \Delta$ for all $\Delta' \in \Phi$.

If Φ is a set of redexes of t and $s = (t \to_{\Delta_1} t_1 \ldots \to_{\Delta_n} t_n)$ is a reduction sequence, then $\Phi \backslash s$ is defined as Φ if $n = 0$, and as $(\Phi \backslash \Delta_1) \backslash s'$, where $s' = (t_1 \to_{\Delta_2} t_2 \ldots \to_{\Delta_n} t_n)$, otherwise.

In the last definition, if $t \to_\Delta t'$ and Δ' is a redex of t, the orthogonality of the system ensures that every member of $\Delta' \backslash \Delta$ is a redex of t'.

Definition 2 (complete development)**.** Let Φ be a finite set of redexes of t. A **development of** Φ is a reduction sequence such that after each initial segment s, the next reduced redex is an element of $\Phi \backslash s$. A **complete development of** Φ is a development s such that $\Phi \backslash s = \emptyset$.

[1] The restriction to left-finite rules can be motivated intuitively, as in [23], by the requirement of checking in finite time the applicability of a rule to a term. On a more technical ground, in [7] it is shown that point 3 of Theorem 6 below does not hold for left-infinite rules.

The following well-known fact [6] is a consequence of the *parallel moves lemma* [11].

Proposition 3. *All complete developments s and s' of a finite set of redexes Φ in a term t are finite, and end with the same term. Moreover, for each redex Δ of t, it holds $\Delta\backslash s = \Delta\backslash s'$. Therefore we can safely denote by $\Delta\backslash\Phi$ the residuals of Δ by any complete development of Φ (and similarly replacing Δ with a set of redexes Φ' of t).*

Exploiting this fact, we define the parallel reduction of a finite set of redexes as any complete development of them.

Definition 4 (finite parallel redex reduction)**.** A **parallel redex** Φ of a term t is a (possibly infinite, necessarily countable) set of distinct redexes in t. Given a *finite* parallel redex Φ of t, we write $t \to_\Phi t'$ and say that there is a **(finite) parallel reduction** from t to t' if there exists a complete development $t \to_{\Delta_1} t_1 \ldots \to_{\Delta_n} t'$ of Φ.

We are now ready to extend the definition of application of parallel redexes to the infinite case. Given an infinite parallel redex Φ (i.e., an infinite set of redexes) of a term t, we consider a chain of approximations of t, $t_0 \leq t_1 \leq t_2 \ldots$, such that their limit is t, and that only a finite subset of Φ applies to each t_i. For each $i < \omega$, let Φ_i be the finite subset of Φ containing all and only those redexes of t which are also redexes of t_i, and call d_i the result of the parallel reduction of Φ_i, i.e., $t_i \to_{\Phi_i} d_i$. Then the crucial fact is that the sequence of terms $d_0, d_1, d_2, \ldots$ defined in this way forms a chain: by definition we say that there is an infinite parallel reduction from t to $d = \bigcup_{i<\omega} d_i$ via Φ, written $t \to_\Phi d$. Here is the formal definition.

Definition 5 (parallel redex reduction)**.** Given an infinite parallel redex Φ of a term t, let $t_0 \leq t_1 \leq \ldots$ $t_n \leq \ldots$ be any chain approximating t (i.e., such that $\bigcup_{i<\omega}\{t_i\} = t$) and such that:

- For each $i < \omega$, every redex $(w, R) \in \Phi$ is either a redex of t_i or $t_i(w) = \bot$. That is, the image of the lhs of every redex in Φ is either all in t_i, or it is outside, but does not "cross the boundary".
- For each $i < \omega$, let $\Phi_i \subseteq \Phi$ be the subset of all redexes in Φ which are also redexes of t_i; then Φ_i must be finite.

For each $i < \omega$, let d_i be the result of the (finite) parallel reduction of t_i via Φ_i (i.e., $t_i \to_{\Phi_i} d_i$). Then we say that there is an **(infinite) parallel reduction** from t to $d \stackrel{def}{=} \bigcup_{i<\omega}\{d_i\}$ via Φ, and we write $t \to_\Phi d$.

Note that in the last definition if the chain approximating t contains finite terms only, then the second condition is automatically satisfied. We consider more general chains, possibly including infinite terms, because they arise naturally in the proof of Theorem 20. The main result of this section states that the last definition is well-given. For a proof we refer to [7] (Theorem 32).

Theorem 6 (parallel redex reduction is well-defined)**.** *Definition 5 is well given, that is:*

1. *given an infinite parallel redex Φ of a term t, there exists a chain $t_0 \leq t_1 \leq \ldots t_n \leq \ldots$ approximating t and satisfying the conditions of Definition 5;*
2. *in this case, $\forall\, 0 \leq i < j < \omega\,.\, d_i \leq d_j$, thus $\{d_i\}_{i<\omega}$ is a chain;*
3. *the result of the infinite parallel reduction of t via Φ does not depend on the choice of the chain approximating t, provided that it satisfies the required conditions.*

As an example, let us consider again the two infinite reductions mentioned in the introduction, according to the parallel interpretation. In the term reduction from f^ω to g^ω (corresponding to the graph reduction of Figure 2), there are infinitely many redexes of rule R_f in the term f^ω, namely at occurrences

$\lambda, 1, 1\cdot 1, \ldots$. Let $\Phi = \{\Delta_n \stackrel{def}{=} (1^n, R_f) \mid n \in \mathbb{N}\}$ be this infinite parallel redex of f^ω. As for the chain of terms approximating $t = f^\omega$, let us choose $t_0 = \bot, t_1 = f(\bot), \ldots, t_n = f^n(\bot)$. Clearly, for each n the set $\Phi_n \subseteq \Phi$ of redexes of t_n contains exactly n redexes. For each n we have $t_n = f^n(\bot) \to_{\Phi_n} g^n(\bot)$, and thus, according to the definition, the result of the parallel reduction of f^ω via Φ is $\bigcup_{i<\omega}\{g^i(\bot)\} = g^\omega$.

In the case of rule R_I and of the circular I, choosing a similar approximating chain we have $\Phi = \{\Delta_n \stackrel{def}{=} (1^n, R_I) \mid n \in \mathbb{N}\}$, $t_n = I^n(\bot)$ for each n, $t_n = I^n(\bot) \to_{\Phi_n} \bot$, and thus I^ω reduces by Φ to $\bigcup_{i<\omega}\{\bot\} = \bot$.

We shall need the following easy result.

Proposition 7 (strong confluence of parallel reduction)**.** *Given an orthogonal TRS $\mathscr{R}$, parallel reduction is strong confluent, i.e., if $t' \;{}_{\Phi'}\!\leftarrow t \to_\Phi t''$, then there exist t''', Ψ, Ψ' such that $t' \to_{\Psi'} t''' \;{}_{\Psi}\!\leftarrow t''$. As a consequence, parallel reduction is confluent.*

3 Term Graphs and Rational Terms

We summarize here the definition of *term graphs* (or simply *graphs*), and their relationship with rational terms, as introduced in [23]. However, since we will apply to those graphs the algebraic approach to graph rewriting, we shall slightly adapt the definition to our framework, emphasizing the categorical structure of the collection of graphs.

Term graphs are obtained from the usual representation of terms with sharing as *dag's* (*directed acyclic graph*), by dropping the acyclicity requirement. In such a way, a finite cyclic graph may represent a possibly infinite, but rational term.

Definition 8 (term graphs)**.** Let Σ be a fixed, one-sorted[2] signature. A **(term) graph** G (over Σ) is a triple $G = (N_G, s_G, l_G)$, where

- N_G is a *finite* set of **nodes**,
- $s_G : N_G \to N_G^*$ is a partial function, called the **successor** function,
- $l_G : N_G \to \Sigma$ is a partial function, called the **labelling** function.

Moreover, it is required that s_G and l_G are defined on the same subset of N_G, and that for each node $n \in N_G$, if $l_G(n)$ is defined and it is an operator of arity k, then $s_G(n)$ has length exactly k.

Definition 9 (morphisms, category of term graphs)**.** A **(graph) morphism** $f : G \to H$ between two graphs G and H is a function $f : N_G \to N_H$, which preserves labelling and successor functions, i.e., for each $n \in N_G$, if $l_G(n)$ is defined, then $l_H(f(n)) = l_G(n)$ and $s_H(f(n)) = f^*(s_G(n))$ (where f^* is the obvious extension of f to lists of nodes).

The composition of graph morphisms is defined in the obvious way, and it is clearly associative; moreover, the identity function on nodes is a morphism, and therefore term graphs (over Σ) and their morphisms form a category that will be denoted by $\mathbf{TGraph}_\Sigma$**.**

Thanks to the conditions imposed on term graphs in Definition 8, a term of CT_Σ can be *extracted* or *unraveled* from every node of a graph.

Definition 10 (from term graphs to terms and backwards)**.** A **path** π in a graph $G = (N_G, s_G, l_G)$ from node n to node n' is a finite sequence $\pi = \langle n_1, j_1, n_2, j_2, \ldots, j_k, n_{k+1}\rangle$, where all j_i are natural numbers,

[2] The generalization to many-sorted signatures is straightforward, by labeling nodes with pairs $\langle$*operator, sort*$\rangle$, with the obvious meaning.

all n_i are nodes, and such that $n_1 = n$, $n_{k+1} = n'$, and for all $1 \leq i \leq k$, $s_G(n_i)|_{j_i} = n_{i+1}$ (here $s|_i$ denotes the i-th element of the sequence s). It follows that there exists exactly one **empty path** '$\langle n \rangle$' from each node n to itself. The **occurrence of a path** $\pi = \langle n_1, j_1, n_2, j_2, \ldots, j_k, n_{k+1} \rangle$ is the list of natural numbers $j_1 \cdot j_2 \cdots j_k$, and it is denoted by $\mathscr{O}(\pi)$. Thus $\mathscr{O}(\pi) = \lambda$ iff π is an empty path. Clearly, for each node n there is at most one path in G having a given occurrence w.

A graph G is **acyclic** if there are no non-empty paths from one node to itself; graph G is a **tree** with root $\underline{n} \in N_G$ iff there exists exactly one path from $\underline{n}$ to any other node of G.

Let $G = (N_G, s_G, l_G)$ be a term graph. The set $var(G) \subseteq N_G$ of **variable nodes** or **empty nodes** of G is the set of nodes on which the labeling function (and thus also the successor function) is undefined. For each node $n \in N_G$, $\mathscr{U}_G[n]$, the **unraveling of G at** n is the term defined as

$$\mathscr{U}_G[n](w) = \begin{cases} n' & \text{if there is a path } \pi \text{ from } n \text{ to } n' \text{ with } \mathscr{O}(\pi) = w \text{ and } n' \in var(G) \\ l_G(n') & \text{if there is a path } \pi \text{ from } n \text{ to } n' \text{ with } \mathscr{O}(\pi) = w \text{ and } n' \notin var(G) \\ \perp & \text{otherwise.} \end{cases}$$

for all $w \in \omega^*$. It follows immediately from these definitions that for each $n \in N_G$ and for each occurrence w, $\mathscr{U}_G[n]/w = \mathscr{U}_G[n']$ iff there exists a path π from n to n' with $\mathscr{O}(\pi) = w$. By $\mathscr{U}[G]$ we denote the set $\mathscr{U}[G] = \{\mathscr{U}_G[n] \mid n \in N_G\}$.

Conversely, let T be a finite set of rational terms, and let us denote by $\overline{T}$ its closure under the subterm relation (i.e., $\overline{T} = \{t \mid t \text{ is a subterm of some term in } T\}$). Then the **term graph representation of** T, denoted $\mathscr{G}[T]$, is the graph $\mathscr{G}[T] = (N_{\mathscr{G}[T]}, s_{\mathscr{G}[T]}, l_{\mathscr{G}[T]})$ defined as follows:

1. $N_{\mathscr{G}[T]} = \overline{T}$;
2. $s_{\mathscr{G}[T]}(t) = \langle t_1, \ldots, t_k \rangle$ if $t = f(t_1, \ldots, t_k)$, and undefined if t is a variable;
3. $l_{\mathscr{G}[T]}(t) = f$ if $t = f(t_1, \ldots, t_k)$, and undefined if t is a variable.

It is quite easily seen that, for every term graph G and node $n \in N_G$, $\mathscr{U}_G[n]$ is a well-defined and total term containing variables in $var(G)$ (thus $\mathscr{U}_G[n] \in CT_\Sigma(var(G))$). Such term can be infinite (because in a cyclic graph there can be infinitely many paths starting from one node), but since G is finite by definition, $\mathscr{U}_G[n]$ is necessarily a rational term, because the number of distinct subterms is bounded by the cardinality of N_G. It is worth noting also that if G is a tree with root $\underline{n}$, then $\mathscr{U}_G[\underline{n}]$ is a finite and linear term.

The above definitions made clear the relationship between objects of category $\mathbf{TGraph}_\Sigma$ and terms in CT_Σ. Such relationship can be extended to arrows of $\mathbf{TGraph}_\Sigma$ and to term substitutions as follows.

Proposition 11 (from morphism to substitutions and backwards)**.** *Let $f : G \to H$ be a term graph morphism. The* **substitution induced by** *f is the substitution $\sigma_f : var(G) \to CT_\Sigma(var(H))$, defined as $x\sigma_f = \mathscr{U}_H[f(x)]$ for all $x \in var(G)$. Moreover the following hold:*

1. *If $f : G \to H$ is a morphism, then $\sigma_f \circ \mathscr{U}_G = \mathscr{U}_H \circ f$.*
2. *Let G be a tree with root $\underline{n}$ and let $\mathscr{U}_G[\underline{n}] = t$. Then for every graph H and every node $n' \in N_H$ with $t' = \mathscr{U}_H[n']$, there is a substitution σ such that $t\sigma = t'$* if and only if *there is a morphism $f : G \to H$ where $f(\underline{n}) = n'$. Moreover in this case we have $\sigma = \sigma_f$.*

4 Algebraic term graph rewriting

We introduce now term graph rewriting according to the algebraic, double-pushout approach [16]. Let **Graph** be a fixed category of graphs (below we will apply the general definitions to the category of

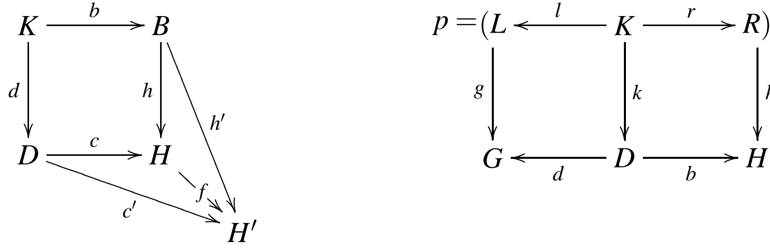


Figure 3: **(a)** Pushout diagram **(b)** Direct derivation as double-pushout construction

term graphs defined in Section 3). The basic categorical construction in the algebraic definition of graph rewriting is that of pushout.

Definition 12 (pushout [24] and pushout complement [16])**.** Given a category C and two arrows $b : K \to B$, $d : K \to D$ of C, a triple $\langle H, h : B \to H, c : D \to H \rangle$ as in Figure 3 (a) is called a **pushout** of $\langle b, d \rangle$ if *[Commutativity]* $h \circ b = c \circ d$, and *[Universal Property]* for all objects H' and arrows $h' : B \to H'$ and $c' : D \to H'$, with $h' \circ b = c' \circ d$, there exists a unique arrow $f : H \to H'$ such that $f \circ h = h'$ and $f \circ c = c'$.

In this situation, H is called a **pushout object** of $\langle b, d \rangle$. Moreover, given arrows $b : K \to B$ and $h : B \to H$, a **pushout complement** of $\langle b, h \rangle$ is a triple $\langle D, d : K \to D, c : D \to H \rangle$ such that $\langle H, h, c \rangle$ is a pushout of b and d. In this case D is called a **pushout complement object** of $\langle b, h \rangle$.

Definition 13 (graph grammars, direct derivations [16])**.** A **(graph) production** $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\to} R)$ is a pair of injective graph morphisms $l : K \to L$ and $r : K \to R$. The graphs L, K, and R are called the **left-hand side**, the **interface**, and the **right-hand side** of p, respectively. A **graph transformation system** $\mathscr{G} = \{p_i\}_{i \in I}$ is a set of graph productions.

Given a graph G, a graph production $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\to} R)$, and an **occurrence** (i.e., a graph morphism) $g : L \to G$, a **direct derivation** α **from** G **to** H **using** p **(based on** g**)** exists if and only if the diagram in Figure 3 (b) can be constructed, where both squares are required to be pushouts in **Graph**. In this case, D is called the **context** graph, and we write $\alpha : G \Rightarrow_{p,g} H$, or simply $\alpha : G \Rightarrow_p H$.

In a graph-theoretical setting, the pushout object H of Figure 3 (a) can be understood as the gluing of graphs B and D, obtained by identifying the images of K along b and d. Therefore the double-pushout construction can be interpreted as follows. In order to apply the production p to G, we first need to find an occurrence of its left-hand side L in G, i.e., a graph morphism $g : L \to G$. Next, to model the deletion of that occurrence from G, we have to find a graph D and morphisms k and d such that the resulting square is a pushout: The context graph D is characterized categorically as the pushout complement object of $\langle l, g \rangle$. Finally, we have to embed the right-hand side R into D: This embedding is expressed by the right pushout.

The conditions for the existence of pushouts and of pushout complements depend on the category **Graphs** for which the above definitions are introduced. Since we are interested just in the graph productions that represent term rewrite rules, in the rest of the paper we shall consider such definitions in the category $\mathbf{TGraph}_\Sigma$, and we will present conditions for the existence of pushouts and pushout complements (see Proposition 17) only for a specific format of productions, called *evaluation rules* (according to the name in [20], where the acyclic case is considered). Such evaluation rules are term graph productions satisfying some additional requirements that make them suitable to represent term rules.

Definition 14 (evaluation rules)**.** An **evaluation rule** is a term graph production $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\to} R)$ such that

1. L is a tree and it is not a single empty node. Let $\underline{n}$ be the root of L.

2. $K = (N_L, s_L \downarrow (N_L \backslash \{\underline{n}\}), l_L \downarrow (N_L \backslash \{\underline{n}\}))$, that is, K is obtained from L by making the successor and labeling functions undefined on the root. Morphism $l : K \to L$ is the inclusion; notice that l is an isomorphism on nodes and that $var(K) = var(L) \cup \{\underline{n}\}$.

3. If restricted to $var(L)$ ($\subset var(K)$), morphism $r : K \to R$ is an isomorphism between $var(L)$ and $var(R)$.

Let us make explicit the relationship between evaluation rules and rewrite rules. From every evaluation rule p one can easily unravel a term rewrite rule $\mathscr{U}[p]$; furthermore, for each rewrite rule R we propose a suitable representation as evaluation rule, $\mathscr{G}[R]$.

Definition 15 (from evaluation to rewrite rules and backwards)**.** Let $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ be an evaluation rule. The **unraveling** of p is the term rewrite rule $\mathscr{U}[p] : t \to s$ defined as follows.

1. $t = \mathscr{U}_L[\underline{n}]$ (where $\underline{n}$ is the root of L, as usual);

2. $s = \mathscr{U}_R[r(\underline{n})]\sigma$, where $\sigma : var(R) \to var(L)$ is the substitution defined as $\sigma(x) = y \quad \text{if } r(y) = x$.

We shall say that an evaluation rule p is **non-self-overlapping** if so is the rewrite rule $\mathscr{U}[p]$. A **term graph rewriting system** (shortly **TGRS**) $\mathscr{P}$ is a finite set of evaluation rules, $\mathscr{P} = \{p_i\}_{i \in I}$; $\mathscr{P}$ is called **orthogonal** if so is the term rewriting system $\mathscr{U}[\mathscr{P}] \stackrel{def}{=} \{\mathscr{U}[p] \mid p \in \mathscr{P}\}$.

The other way around, let $R : t \to s$ be a rational, left-finite and left-linear term rewrite rule, and let $t = f(t_1, \ldots, t_k)$ (t must have this form because it cannot be a variable). Then its **graph representation** $\mathscr{G}[R]$ is the production $\mathscr{G}[R] = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$, where

1. $L = \mathscr{G}[\overline{\{t\}}]$ (by $\overline{T}$ we denote the closure of T with respect to the subterm relation).

2. K and $l : K \to L$ are defined according to point 2 of Definition 14.

3. $R = \mathscr{G}[\overline{\{s, t_1, \ldots, t_k\}}]$.

4. $r : K \to R$ is defined as $r(\underline{n}) = s$, and $r(n) = \mathscr{U}_K[n]$ if $n \in N_K \backslash \{\underline{n}\}$ (this is well-defined because the nodes of R are subterms of $\{s, t_1, \ldots, t_k\}$, according to Definition 10).

By the properties of the unraveling function and of evaluation rules (Definitions 10 and 14) it is routine to check that for each evaluation rule p the term rewrite rule $\mathscr{U}[p]$ is well-defined, rational, total, left-finite, and left-linear (the last two because the left-hand side of an evaluation rule is a tree). The substitution σ applied to $\mathscr{U}_R[r(\underline{n})]$ in the above definition is needed to ensure that $var(s) \subseteq var(t)$. Similarly, it follows directly from the definitions that $\mathscr{G}[R]$ is a well-defined evaluation rule for each term rewrite rule R satisfying the required conditions.

Example 16 (evaluation rules)**.** Figure 4 shows the evaluation rules $\mathscr{G}[R_f]$ and $\mathscr{G}[R_{cdr}]$, which are the graph representations of the rewrite rules $R_f : f(x) \to g(x)$ and $R_{cdr} : cdr(cons(x,y)) \to y$. The left morphisms of the rules are the obvious inclusions, while the right morphisms are determined by the mapping of nodes that is depicted with dotted arrows. Rule R_{cdr} is a collapsing rule which describes the behaviour of the *cdr* operator on LISP-like lists built with the pairing operator *cons*.

The next proposition ensures that if we consider a non-self-overlapping evaluation rule, then the existence of an occurrence morphism from its left-hand side to a term graph is a sufficient condition for the applicability of the rule, i.e., the pushout complement and the pushout of Figure 3 (b) can always be constructed.

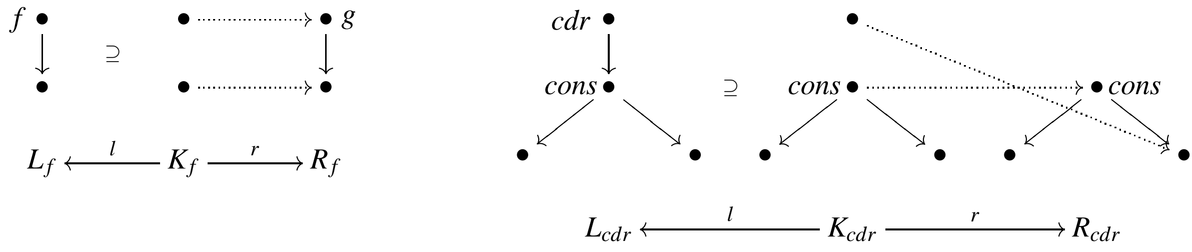


Figure 4: The evaluation rules $\mathscr{G}[R_f]$ and $\mathscr{G}[R_{cdr}]$

Proposition 17 (existence of pushout complements and pushouts)**.** *Let $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ be an evaluation which is not self-overlapping,*[3] *and let $g : L \rightarrow G$ be an occurrence morphism. Then in category* **TGraph** *there exists a pushout complement $\langle D, k : K \rightarrow D, d : D \rightarrow G \rangle$ of $\langle l, g \rangle$, where D is obtained from graph G by making the labeling and the successor function undefined on the image of the root of L.*

Moreover, a pushout $\langle H, h : R \rightarrow H, b : D \rightarrow H \rangle$ of the resulting arrow $k : K \rightarrow D$ and r always exists, and therefore in the above hypotheses there exists a direct derivation $G \Rightarrow_{p,g} H$.

The proof of the last proposition is reported (for the equivalent category **Jungle**) in [10], where also general conditions for the existence of pushouts are presented.[4] It is worth stressing that in the hypotheses of the last proposition, the pushout complement object D has the same nodes of G; thus the nodes of G can be "traced" after the rewriting. More formally, there is a total function, called the **track function** [20] $tr : N_G \rightarrow N_H$, defined as $tr(n) = b(n)$ for all $n \in N_G$ $(= N_D)$.

5 Adequacy of algebraic term graph rewriting for rational parallel term rewriting

The relationship between term and term graph rewriting has been nicely formalized in [23] with the notion of *adequate mapping* between rewriting systems. We recall here the definition, referring to that paper for the precise motivations. A *rewriting system* is defined in this context as a triple (A, R, S), where A is a set of states (in our case terms or graphs), R is a set of rules, and S is a set of reduction sequences, closed under certain operations.

Definition 18 (adequate mapping between rewriting systems)**.** Let (A_1, R_1, S_1) and (A_2, R_2, S_2) be two rewriting systems. A mapping $\mathscr{U} : A_1 \rightarrow A_2$ is **adequate** if:

[*Surjectivity*] $\mathscr{U}$ is surjective;

[*Preservation of normal forms*] $a \in A_1$ is in normal form iff $\mathscr{U}[a] \in A_2$ is in normal form.[5]

[*Preservation of reductions*] If $a \rightarrow^* a'$ with a reduction sequence in S_1, then $\mathscr{U}[a] \rightarrow^* \mathscr{U}[a']$ with a reduction sequence in S_2.[6]

[3] Without this condition the statement would not be true, because the *identification condition* [16] may not be satisfied.

[4] An interesting fact, which is not relevant for this paper, is that the pushout of two arrows exists in **TGraph**$_\Sigma$ iff the associated substitutions unify, and in this case the pushout is a most general unifier.

[5] A state is in normal form if there is no reduction sequence starting from it.

[6] We consider only reduction sequences of finite length.

[*Cofinality*] For $a \in A_1$ and $b \in A_2$, if $\mathscr{U}[a] \to^* b$ in S_2, then there is an a' in A_1 such that $a \to^* a'$ in S_1 and $b \to^* \mathscr{U}[a']$ in S_2.

We show below that the unraveling function $\mathscr{U}$ introduced in the previous sections is an adequate mapping from a given orthogonal TGRS $\mathscr{P}$ to its unraveled orthogonal TRS $\mathscr{U}[\mathscr{P}]$, restricting the allowed parallel reduction sequences to the rational ones. Intuitively, the restriction to *rational* parallel reductions is justified by the fact that an occurrence of an evaluation rule p in a term graph G induces a possibly infinite, but certainly rational parallel redex in the term obtained by unraveling G.

Definition 19 (rational parallel reduction sequences)**.** Let $\mathscr{R}$ be an orthogonal TGR. A parallel reduction $t \to_\Phi t'$ is **rational** if the term obtained by marking in t all the occurrences of redexes of Φ is rational. A **rational parallel reduction sequence** is a parallel reduction sequence where each step is rational. A **rational** TRS is a triple $(CT_\Sigma^{rat}, \mathscr{R}, \mathscr{S}_{rat}(\mathscr{R}))$, where CT_Σ^{rat} is the set of all rational terms in CT_Σ, $\mathscr{R}$ is an orthogonal TRS where all right-hand sides are rational terms, and $\mathscr{S}_{rat}(\mathscr{R})$ is the set of all rational parallel reduction sequences using rules in $\mathscr{R}$.

It is worth stressing that strong confluence and confluence (Proposition 7) also hold for rational parallel reductions.

In order to prove the adequacy of the unraveling function, we need the following fundamental result, which formalizes the "parallel interpretation" discussed in the introduction: A single (possibly cyclic) term graph reduction can be interpreted as a parallel (possibly infinite) term reduction.

Theorem 20 (from graph reductions to parallel term reductions)**.** *Let $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ be a non-self-overlapping evaluation rule, let G be a term graph, and let $g : L \to G$ be an occurrence morphism. By Proposition 17 we know that $G \Rightarrow_{p,g} H$, with the corresponding track function $tr : N_G \to N_H$. Then for each node $n \in N_G$, we have*

$$\mathscr{U}_G[n] \to_\Phi \mathscr{U}_H[tr(n)]\sigma$$

where Φ is the (possibly infinite) parallel redex $\Phi = \{(\mathscr{O}(\pi), \mathscr{U}[p]) \mid \pi \text{ is a path in } G \text{ from } n \text{ to } g(\underline{n})\}$ ($\underline{n}$ is the root of L), and where substitution $\sigma : var(H) \to var(G)$ is defined as $\sigma(x) = y$ if $y \in var(G) \wedge tr(y) = x$, and $\sigma(x) = \perp$ if $\nexists y \in var(G) \,.\, tr(y) = x$.

Proof. For the sake of simplicity, let us assume that $var(G) = \emptyset$, which implies that substitution σ becomes $x\sigma = \perp$ for all $x \in var(H)$.[7]

Let $R : l_p \to r_p$ be the rewrite rule $\mathscr{U}[p]$. We first have to show that Φ is a parallel redex of $\mathscr{U}_G[n]$, i.e., that for each $(\mathscr{O}(\pi), R) \in \Phi$, there is a substitution τ such that $\mathscr{U}_G[n]/\mathscr{O}(\pi) = l_p\tau$. In fact we have

$$\begin{array}{ll} \mathscr{U}_G[n]/\mathscr{O}(\pi) & = \text{[because } \pi \text{ is a path from } n \text{ to } g(\underline{n}) \text{ (see Definition 10)]} \\ \mathscr{U}_G[g(\underline{n})] & = \text{[by point 1 of Proposition 11 and by Definition 15]} \\ \mathscr{U}_L[\underline{n}]\sigma_g = l_p\sigma_g & \end{array}$$

Thus all redexes in Φ are realized by the same substitution σ_g. Let W be the set of all occurrences of paths from n to $g(\underline{n})$ in G (thus W is the set of all occurrences of redexes in Φ), and let $\langle w_1, w_2, \ldots \rangle$ be an arbitrary but fixed enumeration of W such that if $w_i < w_j$, then $i < j$ (in words, no occurrence can be followed by one of its prefixes). For all $i < \omega$, define

$$t_i(u) = \begin{cases} \mathscr{U}_G[n](u) & \text{if } \forall j > i \,.\, u \not\geq w_j \\ \perp & \text{otherwise.} \end{cases}$$

[7] The general case needs an additional technical lemma showing that the trace function maps variables to variables in an injective way (thus σ is well-defined); this can be proved by a careful inspection of the double-pushout diagram.

Obviously, $\{t_i\}_{i<\omega}$ is a chain and $\bigcup_{i<\omega}\{t_i\} = \mathscr{U}_G[n]$. Furthermore, chain $\{t_i\}_{i<\omega}$ satisfies the conditions of Definition 5: by orthogonality, (w_j, R) is a redex of t_i if $j \leq i$, while $t_i(w_j) = \bot$ if $j > i$; and the subset Φ_i of Φ including all redexes of t_i is finite (more precisely, $\Phi_i = \{(w_1, R), \ldots, (w_i, R)\}$, and in particular $\Phi_0 = \emptyset$). Thus by Definition 5 we have $\mathscr{U}_G[n] \to_\Phi \bigcup_{i<\omega}\{s_i\}$, where, for each i, $t_i \to_{\Phi_i} s_i$.

It remains to prove that $\bigcup_{i<\omega}\{s_i\} = \mathscr{U}_H[tr(n)]\sigma$. Let us first show by induction that $\forall i < \omega \,.\, s_i \leq \mathscr{U}_H[tr(n)]\sigma$.[8]

[Base Case] Since $\Phi_0 = \emptyset$, we have that $s_0 = t_0$. Let $u \in \mathscr{O}(s_0)$. By definition $u \not\geq w$ for all $w \in W$. Thus we have

$$\begin{array}{rcl}
s_0(u) = \mathscr{U}_G[n](u) & = & \text{[assuming that } \pi \text{ is the only path from } n \text{ to } n' \text{ in } G \\
 & & \text{with occurrence } u\text{, and since } var(G) = \emptyset] \\
l_G(n') & = & \text{[by the explicit definition of } D \text{ (Proposition 17),} \\
 & & \text{since } n' \neq g(\underline{n})] \\
l_D(n') & = & \text{[by properties of morphisms and definition of } tr] \\
l_H(b(n')) = l_H(tr(n')) & = & \text{[because the track function preserves all paths (like} \\
 & & \pi) \text{ not containing } g(\underline{n})] \\
\mathscr{U}_H[tr(n)](u) & = & [\sigma \text{ does not affect occurrences of operators, like } u] \\
\mathscr{U}_H[tr(n)]\sigma(u) & &
\end{array}$$

[Inductive Case] We must show that $s_i \leq \mathscr{U}_H[tr(n)]\sigma \Rightarrow s_{i+1} \leq \mathscr{U}_H[tr(n)]\sigma$. By the above definitions, the only redex of t_{i+1} which is not of t_i is $\Delta_{i+1} = (w_{i+1}, R)$. If $t_{i+1} \to_{\Phi_i} t'$, let $\Delta_{i+1} \backslash \Phi_i$ be the residual, and let $V = \{v_1, \ldots, v_k\}$ its set of occurrences. Then, if τ is the substitution such that $l_p\tau = t_{i+1}/v$ for all $v \in V$,[9] a careful inspection reveals that s_{i+i} can be defined in term of s_i as

$$s_{i+1}(u) = \begin{cases} s_i(u) & \text{if } u \in \mathscr{O}(s_i) \\ r_p\tau(w) & \text{if there is a } v \in V \text{ such that } u = vw\text{, and } w \in \mathscr{O}(r_p\tau) \\ \bot & \text{otherwise.} \end{cases}$$

Exploiting the induction hypothesis, it remains to show that $s_{i+1}(vw) = \mathscr{U}_H[tr(n)]\sigma(vw)$ only for $v \in V$ and $w \in \mathscr{O}(r_p\tau)$. This can be done by combining the techniques used for the acyclic case in [20, 10], and those used for the Base Case above, because the definition of t_{i+1} ensures that no redex of Φ appears in substitution τ (although τ may substitute an infinite term for a variable).

Finally it remains to prove that $\mathscr{U}_H[tr(n)]\sigma \leq \bigcup_{i<\omega}\{s_i\}$, and this can be done as follows. If rule R is not collapsing, it can be shown that $\bigcup_{i<\omega}\{s_i\}$ is a total term (and therefore it is a maximal element of the approximation ordering), because the least depth of $\bot$'s in s_i tends to infinity. If instead R is collapsing, then it is easy to check that $s_0 = s_1 = \ldots = \bigcup_{i<\omega}\{s_i\}$, and that $\mathscr{U}_H[tr(n)]\sigma = s_0$ using the Base Case above and the fact that the only variable in H, $tr(g(\underline{n}))$, is substituted for $\bot$ by σ. □

Exploiting this main result, we can prove the adequacy of the unraveling function. One minor problem is due to the fact that the unraveling of a term graph is not a term, but a set of terms. Thus either we consider TRS where a state can be a set of terms (which seems quite unnatural), or we have to consider *pointed* term graphs, i.e., graphs with a distinguished node, and to assume that the $\mathscr{U}$ unravels each graph at that specific node. However, in this last case, unraveling would not preserve normal forms, because a graph may contain a redex in a part that is not reachable from the distinguished node (in the *garbage*), and such redex would not appear in the unraveled term. Full adequacy could be recovered by adding garbage collection at each graph rewriting step (as it is done in [5, 23]) but we prefer to prove a slightly weaker result without modifying the graph rewriting formalism.

[8] It is worth recalling that $t \leq t' \Leftrightarrow \forall u \in \mathscr{O}(t)\,.\,t(u) = t'(u)$.

[9] It can be checked that all redexes in $\Delta_{i+1} \backslash \Phi_i$ are realized by the same substitution in t'.

Theorem 21 (adequacy of TGR for rational parallel TR)**.** *Given an orthogonal TGRS $\mathscr{P}$, let $(\mathscr{G},\mathscr{P},\mathscr{S}(\mathscr{P}))$ be the rewriting system where $\mathscr{G}$ is the set of all pointed term graphs over Σ and $\mathscr{S}(\mathscr{P})$ is the set of all graph reduction sequences using rules in $\mathscr{P}$ and such that the track function preserves the distinguished node.*

Then the unraveling function $\mathscr{U}$ is a mapping from $(\mathscr{G},\mathscr{P},\mathscr{S}(\mathscr{P}))$ to the rational TRS $(CT_{\Sigma}^{rat},\mathscr{U}[\mathscr{P}],\mathscr{S}_{rat}(\mathscr{U}[\mathscr{P}]))$ which satisfies all the conditions of adequate mappings (Definition 18), except of the preservation of normal forms. Nevertheless, it satisfies the following weaker version: [Weak preservation of normal forms] *if $a \in A_1$ is a normal form, then so is $\mathscr{U}[a] \in A_2$.*

Proof. According to Definition 18, we have:

[*Surjectivity*] Immediate by Definition 10, using pointed graphs.

[*Weak preservation of normal forms*] Follows from point 2 of Proposition 11: if there is a redex of rule $\mathscr{U}[p]$ in term $\mathscr{U}[G]$, then there is an occurrence morphism from the left-hand side of p to G.

[*Preservation of reductions*] By repeated applications of Theorem 20.

[*Cofinality*] Let G be a pointed graph, and suppose that $\mathscr{U}[G] \to^* t$ via a rational reduction sequence. We show by induction on the length of the sequence that there is a G' such that $t \to^* \mathscr{U}[G']$, and $G \Rightarrow^* G'$.

[Base Case] If $\mathscr{U}[G] \to_\Phi t$ and Φ is rational, it is possible to show that there exists a finite set M of occurrence morphisms of rules of $\mathscr{P}$ in G, such that their "unraveling" (that we do not define formally) $\mathscr{U}[M]$ is a parallel redex of $\mathscr{U}[G]$ containing Φ. Then let G' be the graph obtained by applying to G all the occurrence morphisms in M in any order (by orthogonality G' is well-defined). Then it can be shown that $t \to_{\mathscr{U}[M]\backslash\Phi} \mathscr{U}[G']$.

$$\begin{array}{ccccc}
\mathscr{U}[G] & \xrightarrow{\;n\;} & t' & \xrightarrow[d]{\;*\;} & \mathscr{U}[G'] & & \\
 & & \downarrow{\scriptstyle\Phi} & & \downarrow{\scriptstyle\Phi\backslash d} & & \\
 & & t & \xrightarrow[d\backslash\Phi]{\;*\;} & t'' & \xrightarrow{\;*\;} & \mathscr{U}[G'']
\end{array}$$

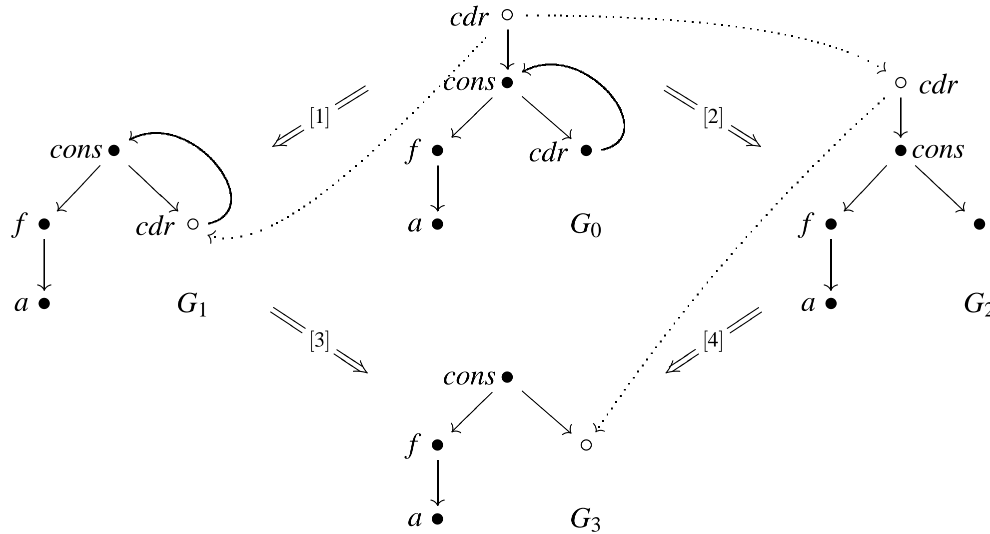
$$G \Longrightarrow^* G' \Longrightarrow^* G''$$

[Inductive Case] Suppose that $\mathscr{U}[G] \to^n t' \to_\Phi t$, and consider the diagram above. By inductive hypothesis there exists G' such that $G \Rightarrow^* G'$ and $t' \to^* \mathscr{U}[G']$. By strong confluence of rational parallel reductions, there exists a t'' such that the square commutes, and the residual $\Phi\backslash d$ is a rational parallel redex. Then applying the Base Case to $\mathscr{U}[G'] \to_{\Phi\backslash d} t''$, we have that there exists a G'' such that $G' \Rightarrow^* G''$ and $t'' \to^* \mathscr{U}[G'']$.

□

Example 22 (Rewriting steps)**.** The diagram below shows a few direct derivations using the evaluation rule $\mathscr{G}[R_{cdr}]$ of Example 16. The corresponding track functions are uniquely determined by the dotted arrows and the fact that the *cons* node is preserved. Unraveling the four graphs at the nodes $\circ$, we get the following terms: $t_0 = \mathscr{U}_{G_0}[\circ] = cdr(cons(f(a), cdr(cons(f(a), cdr(\ldots)))))$, $t_1 = \mathscr{U}_{G_1}[\circ] = t_0$, $t_2 = \mathscr{U}_{G_2}[\circ] = cdr(cons(f(a), \bot))$, and $t_3 = \mathscr{U}_{G_3}[\circ] = \bot$. By Theorem 20, each direct derivation $G_i \overset{[k]}{\Rightarrow} G_j$ corresponds to a rational parallel reduction $\mathscr{U}_{G_i}[\circ] \to_{\Phi_k} \mathscr{U}_{G_j}[\circ]$: it is easy to check that the four rational parallel redexes are $\Phi_1 = \{\lambda\}$, $\Phi_2 = \{12(12)^*\}$, $\Phi_3 = \{(12)^*\}$ and $\Phi_4 = \{\lambda\}$. To conclude, note that

if we consider also the evaluation rule $\mathscr{G}[R_f]$ of Example 16, then term graph G_3 is not a normal form, while $t_3 = \mathscr{U}_{G_3}[\circ] = \perp$ is, showing that unraveling does not reflect normal forms in general.



6 Conclusions

We showed that by exploiting the complete partial ordered structure of (infinite, partial) terms, a notion of infinite parallel reduction can be defined. Moreover, we proved that this notion can be used to relate term graph rewriting with cycles and term rewriting, formalizing the intuition that a single graph reduction correspond to a (possibly infinite) parallel term reduction. This result is used to show that cyclic graph rewriting is adequate for rational term rewriting, exploiting the notion of adequacy proposed in [23]. As discussed in the introduction, for several approaches to cyclic term graph rewriting it is pretty clear whether this parallel interpretation is more faithful than the sequential one: this point has still to be clarified for the more recent approaches, and it will be addressed in the full version of the paper. Another interesting topic to explore is how far and under which additional restrictions the proposed results could be generalized to non-orthogonal systems.

References

[1] Z.M. Ariola & J.W. Klop (1996): *Equational Term Graph Rewriting*. Fundam. Inform. 26(3/4), pp. 207–240.

[2] A. Arnold & M. Nivat (1980): *The metric space of infinite trees. Algebraic and topological properties*. Fundamenta Informaticae 4, pp. 445–476.

[3] P. Baldan, C. Bertolissi, H. Cirstea & C. Kirchner (2007): *A rewriting calculus for cyclic higher-order term graphs*. Mathematical Structures in Computer Science 17(3), pp. 363–406.

[4] P. Baldan, C. Bertolissi, H. Cirstea & C. Kirchner (2008): *Towards a Sharing Strategy for the Graph Rewriting Calculus*. Electr. Notes Theor. Comput. Sci. 204, pp. 111–127.

[5] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer & M.R. Sleep (1987): *Term graph reduction*. In J.W. de Bakker, A.J. Nijman & P.C. Treleaven, editors: Parallel Architectures and Languages Europe. LNCS 259, Springer Verlag, pp. 141–158.

[6] G. Berry & J.-J. Lévy (1979): *Minimal and Optimal Computations of Recursive Programs*. *J. ACM* 26(1), pp. 148–175.

[7] A. Corradini (1993): *Term Rewriting in CT_Σ*. In M.-C. Gaudel & J.-P. Jouannaud, editors: *Trees in Algebra and Programming*. *LNCS* 668, Springer Verlag, pp. 468–484.

[8] A. Corradini & F. Gadducci (1997): *A 2-Categorical Presentation of Term Graph Rewriting*. In E. Moggi & G. Rosolini, editors: *Category Theory and Computer Science*. *LNCS* 1290, Springer Verlag, pp. 87–105.

[9] A. Corradini & F. Gadducci (1999): *Rewriting on Cyclic Structures: Equivalence between the Operational and the Categorical Description*. *Informatique Théorique et Applications/Theoretical Informatics and Applications* 33, pp. 467–493.

[10] A. Corradini & F. Rossi (1993): *Hyperedge Replacement Jungle Rewriting for Term Rewriting Systems and Logic Programming*. *Theoretical Computer Science* 109, pp. 7–48.

[11] H.B. Curry & R. Feys (1958): *Combinatory Logic Volume I*. Studies in Logic and the Foundations of Mathematics, North-Holland Publishing Company, Amsterdam.

[12] N. Dershowitz & S. Kaplan (1989): *Rewrite, Rewrite, Rewrite, Rewrite, Rewrite...* In: *Proc. POPL'89, Austin*. pp. 250–259.

[13] N. Dershowitz, S. Kaplan & D.A. Plaisted (1989): *Infinite Normal Forms (plus corrigendum)*. In: *Automata, Languages and Programming*. pp. 249–262.

[14] D.J. Dougherty, P. Lescanne & L. Liquori (2006): *Addressed term rewriting systems: application to a typed object calculus*. *Mathematical Structures in Computer Science* 16(4), pp. 667–709.

[15] D. Duval, R. Echahed & F. Prost (2007): *Modeling Pointer Redirection as Cyclic Term-graph Rewriting*. *Electr. Notes Theor. Comput. Sci.* 176(1), pp. 65–84.

[16] H. Ehrig (1987): *Tutorial introduction to the algebraic approach of graph-grammars*. In H. Ehrig, M. Nagl, G. Rozenberg & A. Rosenfeld, editors: *Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*. *LNCS* 291, Springer Verlag, pp. 3–14.

[17] W.M. Farmer, J.D. Ramsdell & R.J. Watro (1990): *A correctness proof for combinator reduction with cycles*. *ACM Transactions on Programming Languages and Systems* 12, pp. 123–134.

[18] W.M. Farmer & R.J. Watro (1991): *Redex capturing in term graph rewriting*. In R.V. Book, editor: *Rewriting Techniques and Applications*. *LNCS* 488, Springer Verlag, pp. 13–24.

[19] J.A. Goguen, J.W. Tatcher, E.G. Wagner & J.R Wright (1977): *Initial Algebra Semantics and Continuous Algebras*. *Journal of the ACM* 24, pp. 68–95.

[20] B. Hoffmann & D. Plump (1991): *Implementing Term Rewriting by Jungle Evaluation*. *Informatique Théorique et Applications/Theoretical Informatics and Applications* 25, pp. 445–472.

[21] G. Huet & J.-J. Lévy (1991): *Computations in orthogonal rewriting systems, I*. In J.-L. Lassez & G. Plotkin, editors: *Computational Logic: Essays in honour of Alan Robinson*. MIT Press, pp. 395–414.

[22] J.R. Kennaway (1991): *Graph rewriting in some categories of partial morphisms*. In: *Graph Grammars and their Application in Computer Science*. *LNCS* 532, Springer Verlag, pp. 490–504.

[23] J.R. Kennaway, J.W. Klop, M.R. Sleep & F.J. de Vries (1994): *On the Adequacy of Graph Rewriting for Simulating Term Rewriting*. *ACM Transactions on Programming Languages and Systems* 16, pp. 493–523.

[24] S. Mac Lane (1971): *Categories for the working mathematician*. Springer Verlag.

[25] G Pacini, Carlo Montangero & Franco Turini (1974): *Graph Representation and Computation Rules for Typeless Recursive Languages*. In J. Loeckx, editor: *ICALP*. *LNCS* 14, Springer, pp. 157–169.

[26] J. Staples (1980): *Computation of graph-like expressions*. *Theoretical Computer Science* 10, pp. 171–195.

[27] D.A. Turner (1979): *A New Implementation Technique for Applicative Languages*. *Softw., Pract. Exper.* 9(1), pp. 31–49.

[28] J. Vuillemin (1974): *Correct and Optimal Implementations of Recursion in a Simple Programming Language*. *J. Comput. Syst. Sci.* 9(3), pp. 332–354.

Nominal Graphs

Joint GT-VMT and TERMGRAPH Invited Talk

Maribel Fernández
King's College London
Department of Computer Science
London WC2R 2LS, U.K.
Maribel.Fernandez@kcl.ac.uk

Nominal syntax is an extension of standard, first-order syntax that provides support for the specification of operators with binding, thanks to an abstraction construct and the axiomatisation of the alpha-equivalence relation. Nominal equivalence/unification/rewriting are generalisations of the standard notions of syntactic equivalence/unification/rewriting, to take into account alpha-equivalence.

In this talk we will explore the relationships between nominal and graph-based calculi, which have also been proposed as a representation of systems with binding. On one hand, we will go from nominal terms to nominal graphs— a graphical representation of nominal terms— and use them to give a quadratic nominal unification algorithm (one of the most efficient algorithms available so far to solve unification problems modulo alpha). On the other hand, we will show that nominal syntax can be used to give a formal, algebraic account of a general class of graph rewriting systems.

Rachid Echahed (Ed.): 6th International Workshop
on Computing with Terms and Graphs (TERMGRAPH 2011)
EPTCS 48, 2011, pp. 19–19, doi:10.4204/EPTCS.48.4

© M. Fernández
This work is licensed under the
Creative Commons Attribution License.

Rule-based transformations for geometric modelling

Thomas Bellet
University of Poitiers, XLIM-SIC CNRS, France
thomas.bellet@univ-poitiers.fr

Agnès Arnould
University of Poitiers, XLIM-SIC CNRS, France
agnes.arnould@univ-poitiers.fr

Pascale Le Gall
Ecole Centrale Paris, MAS, France
pascale.legall@ecp.fr

The context of this paper is the use of formal methods for topology-based geometric modelling. Topology-based geometric modelling deals with objects of various dimensions and shapes. Usually, objects are defined by a graph-based topological data structure and by an embedding that associates each topological element (vertex, edge, face, etc.) with relevant data as their geometric shape (position, curve, surface, etc.) or application dedicated data (e.g. molecule concentration level in a biological context). We propose to define topology-based geometric objects as labelled graphs. The arc labelling defines the topological structure of the object whose topological consistency is then ensured by labelling constraints. Nodes have as many labels as there are different data kinds in the embedding. Labelling constraints ensure then that the embedding is consistent with the topological structure. Thus, topology-based geometric objects constitute a particular subclass of a category of labelled graphs in which nodes have multiple labels.

We previously introduced a formal approach of topological modelling based on graph transformation rules. Topological operations, that only modify the topological structure of objects, can be defined such that the topological consistency of constructed objects is ensured with syntactic conditions on rules. In this paper, we follow the same approach in order to deal with geometric operations, that can modify both the topological structure and the embedding. Thus, we define syntactic conditions on rules to ensure the consistency of the embedding during transformations.

Introduction

Topology-based geometric modelling deals with the manipulation (construction, modification, ...) of objects that are subdivided according to their topological structure. The topological structure is the cell subdivision (vertices, edges, faces, volumes) of objects and the adjacency relations between these cells. Among the existing topological models, we choose in this paper the model of generalized maps [6, 8], also called G-maps. The topological structure of G-maps can be represented by a graph where edges indicate which nodes are neighbours and where edge labels indicate what kind of neighbouring is concerned (*e.g.* connection between faces or between volumes). This graph must satisfy some constraints on the arc labelling to ensure the topological consistency of the topological structure. For example, while their shapes are different, the three objects of Fig. 1 have the same topological structure: a closed face that contains four edges and four vertices.

In addition to the topological structure, objects are defined by an embedding that includes all other kinds of information attached to the topological cells of the object. An evident example of embedding is given by the kinds of information needed to capture the shape of objects. For the objects of Fig. 1, we assume that the associated embedding contains three elements:

- geometric points (defined by 2 dimension coordinates in the case of plane objects) that are attached to topological vertices;

Rachid Echahed (Ed.): 6th International Workshop
on Computing with Terms and Graphs (TERMGRAPH 2011)
EPTCS 48, 2011, pp. 20–37, doi:10.4204/EPTCS.48.5

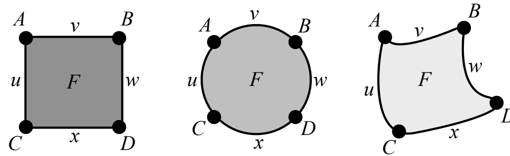


Figure 1: Three objects with a same topological structure

- curves that are attached to edges;
- colors that are attached to faces.

While the topological structure is represented by the arc labels, the different elements of the object embedding can be represented by node labels. Intuitively, a node is labelled by all the embedding elements that are attached to its adjacent cells (vertex, edges, faces, volumes). Let us point out that while the embedding generally contains classical geometric data describing the shape of the objects (e.g. points, curves, surfaces, etc.), the embedding also contains specific data that depend on the targeted application (*e.g.* molecule concentration for biology, rock density for geology, material for architecture, etc.). Actually, nodes have as many labels as there are different kinds of data in the application-oriented definition of the embedding. This fact explains that in a first step, we provide in Section 1 a category of graphs whose nodes can carry multiple labels. This category is defined as a direct extension of the category of partially labelled graphs as defined in [4]. We then define in Section 2 our embedded topological model as particular graphs of this category. Graphs that represent embedded G-maps have to satisfy constraints to ensure both the topological consistency and the embedding consistency.

To define operations on objects represented as embedded G-maps, we choose the graph transformations and more precisely the so-called double-pushout approach [3]. In a previous work [10], we defined a rule-based language dedicated to topology-based modelling. The first interest of this language is that we defined syntactic conditions on rules to ensure by construction, that the application of a rule to a G-map produces a G-map. In other words, objects resulting from applications of well-formed rules on G-maps are systematically well-formed topological objects, that is objects satisfying the topological consistency constraints. In this paper, we define a similar framework for geometric operations that can modify both the topological structure and the embedding. As we need to change labels of nodes and arcs during transformations, we based our work on rules of [4] that allow to rename labels. In Section 3, similarly to the topological conditions introduced in [10], we define syntactic conditions on rules that ensure the preservation of the embedding constraints when rules are applied to embedded G-maps. In Section 4, we provide G-map rule schemes that allow to define generic geometric operations. Actually, rule schemes contain expressions on variables to allow to compute the embedding of the resulting objects. Finally, we provide syntactic conditions on rule schemes to ensure the preservation of embedding consistency by rule application.

1 Transformation rules for I-labelled graphs

1.1 Category of I-labelled graphs

In this section, we define the category of I-labelled graphs as an extension of the one of partially labelled graphs defined in [4]. While in [4] nodes have at most one label, in our case, nodes can have at most $|I|$ labels where I is a chosen set of indexes.

Definition 1 (I-labelled graph) *Let $(\mathscr{C}_{V,i})_{i\in I}$ be a family of node label sets and $\mathscr{C}_E$ be an arc label set. A I*-labelled graph *$G^I = (V,E,s,t,(l_{V,i})_{i\in I},l_E)$ upon $(\mathscr{C}_{V,i})_{i\in I}$ and $\mathscr{C}_E$ is defined as:*

- *a set V of nodes;*
- *a set E of arcs;*
- *two functions* source $s : E \to V$ *and* target $t : E \to V$*. For $e \in E$, $s(e)$ and $t(e)$ are respectively the* source node *and the* target node *of e;*
- *a family of partial functions*[1] *$(l_{V,i} : V \to \mathscr{C}_{V,i})_{i\in I}$ that label nodes. For $v \in V$, when it exists, $l_{V,i}(v)$ is called the i-label of v ;*
- *a partial function $l_E : E \to \mathscr{C}_E$ that labels arcs.*

For a graph $G^I = (V,E,s,t,(l_{V,i})_{i\in I},l_E)$, elements of the tuple can be indexed by G to make explicit the graph name: V_G for V for example. The above definition is a natural extension of partially labelled graphs of [4]. Indeed, instead of a unique partial function l_V that labels nodes, we consider an I-indexed family $(l_{V,i})_{i\in I}$ of partial labelling functions[2]. By extending the definition given in [4] for a unique node labelling function, an I-labelled morphism $g : G^I \to G'^I$ between I-labelled graphs G^I and G'^I is defined by two functions $g_V : V_G \to V_{G'}$ and $g_E : E_G \to E_{G'}$ preserving sources, targets and labels : $s_{G'} \circ g_E = g_V \circ s_G$, $t_{G'} \circ g_E = g_V \circ t_G$, for all x in $Dom(l_{G,E})$, $l_{G',E}(g_E(x)) = l_{G,E}(x)$ and lastly, for all i in I, for all x in $Dom(l_{G,V,i})$, $l_{G',V,i}(g_V(x)) = l_{G,V,i}(x)$. Thus, the only difference with [4] is that for I-labelled graphs, I-labelled morphisms have more labels to preserve. An I-labelled morphism $g : G^I \to G'^I$ is an inclusion if $\forall x \in E_G, g_E(x) = x$ and $\forall x \in V_G, g_V(x) = x$. Such an inclusion is then denoted as $g : G^I \hookrightarrow G'^I$. I-labelled graphs and I-labelled morphisms constitute a category, where morphism composition is defined componentwise as function composition.

For any partially labelled graph $G = (V,E,s,t,l_V,l_E)$, we call the base of G the partially labelled graph defined as $(V,E,s,t,\perp,l_E)$ whose node labelling is totally undefined and denote it by $G_\perp$.

We say that two morphisms $g : G \to G'$ and $h : H \to H'$ between partially labelled graphs have the same base if $G_\perp = H_\perp$, $G'_\perp = H'_\perp$ and $g_V = h_V$, $g_E = h_E$. We note $g_\perp : G_\perp \to H_\perp$ the derived morphism defined by $g_{\perp E} = g_E$ and $g_{\perp V} = g_V$.

We respectively note $\mathscr{G}$, $\mathscr{G}^I$ and $\mathscr{G}^\perp$ the category of partially labelled graphs (as defined in [4]), I-labelled graphs and bases of partially labelled graphs (that is, graphs whose node labelling is the function totally undefined).

For a I-labelled graph $G^I = (V,E,s,t,(l_{V,i})_{i\in I},l_E)$, for an index $i \in I$, the projection $proj_i(G^I)$, also called the i-component, is defined as the partially labelled graph $(V,E,s,t,l_{V,i},l_E)$ according to [4]. Similarly, for an I-labelled morphism $g : G^I \to G'^I$, we call $proj_i(g) : proj_i(G^I) \to proj_i(G'^I)$ the graph morphism that only consider the i-labels of the I-labelled graphs.

From an I-indexed family of partially labelled graphs G_i defined on a common base $(V,E,s,t,\perp,l_E)$ with $l_{V,i}$ as node labelling function, we define by $Prod_{i\in I}G_i$ the I-labelled graph $(V,E,s,t,(l_{V,i})_{i\in I},l_E)$. Similarly, from an I-indexed family of graph morphisms $g_i : G_i \to G'_i$ sharing the same base, we can define an I-labelled morphism $Prod_{i\in I}g_i$, from $Prod_{i\in I}G_i$ to $Prod_{i\in I}G'_i$, that coincides with any g_i on the

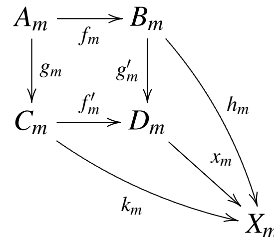
[1] Given X and Y two sets, a partial function f from X to Y is a total function $f : X' \to Y$, from X' a subset of X. X' is called the domain of f, and is denoted by $Dom(f)$. For $x \in X - Dom(f)$, we say that $f(x)$ is undefined, and write $f(x) = \perp$. We also note $\perp : X \to Y$ the function totally undefined, that is $Dom(\perp) = \emptyset$.

[2] To better fit with the frame of [4], one would think to label nodes by a unique label made of a Cartesian product, instead of having a family of labelling functions. But, such an approach would not allow us to have the possibility of labelling a node simultaneously by a defined i-label and by an undefined i'-label for i and i' indexes of I.

node set V_G and the arc set E_G. Obviously, we then get the identities: $G^I = Prod_{i\in I} proj_i(G^I)$ for G a I-labelled graph and $g^I = Prod_{i\in I} proj_i(g^I)$ for g^I an I-labelled morphism.

Since from any partially labelled graphs F, G, H, ... and from any morphisms on them $f : F \to G$, $g : G \to H$, $h : F \to H$, we can derive their corresponding base form, respectively $F_\perp$, $G_\perp$, $H_\perp$, ... $f_\perp : F_\perp \to G_\perp$, $g_\perp : G_\perp \to H_\perp$, $h_\perp : F_\perp \to H_\perp$, and then for any diagram made of morphisms expressed on partially labelled graphs, we can derive a similar diagram on their corresponding base. For example, from the diagram $F \xrightarrow{f} G \xrightarrow{g} H = F \xrightarrow{h} H$, we can derive the diagram $F_\perp \xrightarrow{f_\perp} G_\perp \xrightarrow{g_\perp} H_\perp = F_\perp \xrightarrow{h_\perp} H_\perp$.

Lemma 1 *For $m = 1, 2$, let us consider $f_m : A_m \to B_m$ and $g_m : A_m \to C_m$ two graph morphisms in $\mathscr{G}$ such that g_m is injective and for all x in V_{B_m} (resp. in E_{B_m}), $\{l_{B_m,V}(x)\} \cup l_{C_m,V}(g_{V\,m}(f_{V\,m}^{-1}(x)))$ (resp. $\{l_{B_m,E}(x)\} \cup l_{C_m,E}(g_{E\,m}(f_{E\,m}^{-1}(x)))$) contains at most one element, then there exists a graph D_m and graph morphisms $f'_m : C_m \to D_m$ and $g'_m : B_m \to D_m$ such that the following diagram is a pushout*[3]



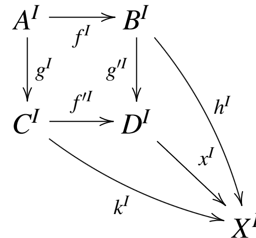
Moreover, if both pushout diagrams have the same underlying base diagram, that is $A_{1\perp} = A_{2\perp}$, $B_{1\perp} = B_{2\perp}$, $C_{1\perp} = C_{2\perp}$, $f_{1\perp} = f_{2\perp}$ and $g_{1\perp} = g_{2\perp}$, then we get $D_{1\perp} = D_{2\perp}$, $f'_{1\perp} = f'_{2\perp}$ and $g'_{1\perp} = g'_{2\perp}$.

Proof. The proof of the existence of pushout is given in [4].

The uniqueness of the base elements $D_{m\perp}$, $f'_{m\perp}$ and $g'_{m\perp}$ comes from the fact that the proof in [4] explicitly constructs the elements $D_{m\perp}$, $f'_{m\perp}$ and $g'_{m\perp}$ in relation to the elements of the base diagram. □

For convenience issues, we note $B_m +_{A_m} C_m$ the graph D_m, occurring in the pushout diagram.

Lemma 2 (Existence of pushouts) *Let $f^I : A^I \to B^I$ and $g^I : A^I \to C^I$ be two I-labelled morphisms in $\mathscr{G}^I$ such that g^I is injective and for all x in V_B (resp. in E_B), for all i in I, $\{l_{B,V,i}(x)\} \cup l_{C,V,i}(g_{V,i}({f_{V,i}}^{-1}(x)))$ (resp. $\{l_{B,E}(x)\} \cup l_{C,E}(g_E({f_E}^{-1}(x)))$) contains at most one element, then there exists a I-labelled graph D^I and two I-labelled morphisms $f'^I : C^I \to D^I$ and $g'^I : B^I \to D^I$ in $\mathscr{G}^I$ such that the following diagram is a pushout:*



Moreover D^I can be defined as $Prod_{i\in I} D_i$ with $D_i = proj_i(B^I) +_{proj_i(A^I)} proj_i(C^I)$

Proof. By lemma 1, we know that $(D_i)_{i\in I}$ have the same base because $(proj_i(A))_{i\in I}$, $(proj_i(B))_{i\in I}$ and $(proj_i(C))_{i\in I}$ have respectively the same base. Thus, $Prod_{i\in I} D_i$ is a well defined I-labelled graph.

[3] A commutative diagram $A \xrightarrow{g} C \xrightarrow{f'} D = A \xrightarrow{f} B \xrightarrow{g'} D$ is a pushout if and only if for every graph X and all morphisms $h : B \to X$ and $k : C \to X$ with $k \circ g = h \circ f$, there is an unique morphism $x : D \to X$ with $x \circ g' = h$ and $x \circ f' = k$.

Moreover, there exist I-labelled morphisms $f'^I : C^I \to D^I$ and $g'^I : B^I \to D^I$ in $\mathscr{G}^I$ ensuring that the diagram is commutative. It suffices to choose : $f'^I = Prod_{i\in I} f'_i$ and $g'^I = Prod_{i\in I} f'_i$ where $f'_i : proj_i(B) \to D_i$ and $g'_i : proj_i(C) \to D_i$ are the underlying morphisms constituting the pushout construction : $D_i = proj_i(B) +_{proj_i(A)} proj_i(C)$.

Let us show the universal property : let us consider $k^I : C^I \to X^I$ and $h^I : B^I \to X^I$ two I-labelled graphs with $h^I \circ f^I = k^I \circ g^I$. By the universal property of D_i, there exists a unique labelled morphism $x_i : D_i \to proj_i(X^I)$ such that $x_i \circ proj_i(f'^I) = x_i \circ proj_i(g'^I)$. Then we can consider $x^I = Prod_{i\in I} x_i : D^I \to X^I$ verifying $x^I \circ f'^I = x^I \circ g'^I$. □

Thus, constructions holding on partially labelled graphs can be replicated at the level of I-labelled graphs. It suffices to work with their i-components, index per index, using the $proj_i$ application and to reconstruct I-labelled graphs or morphisms by applying the $Prod_{i\in I}$ operator on objects sharing the same base.

In the sequel, we take benefit of all results given in [4] : existence of pullbacks, characterisation of natural pushouts[4]. For the purpose of simplicity, we give up the exponent I upon the I-labelled graph (resp. morphism) names and we will use I-labelled inclusions to define rules.

Definition 2 (graph transformation rule) *A* graph transformation rule *$r : L \hookleftarrow K \hookrightarrow R$ over $\mathscr{G}^I$ consists of two I-labelled graph inclusions $K \hookrightarrow L$ and $K \hookrightarrow R$ in $\mathscr{G}^I$ such that:*

1. *for all node $x \in V_L$ and all $i \in I$, $l_{L,V,i}(x) = \perp$ implies $x \in V_K$ and $l_{R,V,i}(x) = \perp$; reciprocally, for all node $x \in V_R$ and all $i \in I$, $l_{R,V,i}(x) = \perp$ implies $x \in V_K$ and $l_{L,V,i}(x) = \perp$;*
2. *for all arc $x \in E_L$, $l_{L,E}(x) = \perp$ implies $x \in E_K$ and $l_{R,E}(x) = \perp$; reciprocally, for all arc $x \in E_R$, $l_{R,E}(x) = \perp$ implies $x \in E_K$ and $l_{L,E}(x) = \perp$;*

Usually, L is called the left-hand side*, R the* right-hand side *and K the* kernel.

Definition 3 (direct transformation) *Let $r : L \hookleftarrow K \hookrightarrow R$ be a graph transformation rule over $\mathscr{G}^I$ and G a I-labelled graph and $m : L \to G$ an injective I-labelled morphism in $\mathscr{G}^I$ called* match morphism.

A direct transformation *$G \overset{r,m}{\Rightarrow} H$ of G into H consists in the following natural double pushout defined over $\mathscr{G}^I$:*

$$\begin{array}{ccccc} L & \hookleftarrow & K & \hookrightarrow & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow \\ G & \hookleftarrow & D & \hookrightarrow & H \end{array}$$

Definition 4 (dangling condition) *An I-labelled morphism $m : L \to G$ satisfies the* dangling condition *with respect to the inclusion $K \hookrightarrow L$, if none node of $m(L)\backslash m(K)$ is source or target of an arc of $G\backslash m(L)$.*

Theorem 1 (Existence and uniqueness of direct transformation) *Let $r : L \hookleftarrow K \hookrightarrow R$ be a rule and $m : L \to G$ a match morphism in $\mathscr{G}^I$, the previous direct transformation $G \Rightarrow^{r,m} H$ exists if and only if m satisfies the dangling condition. Moreover, in this case D and H are unique up to isomorphism.*

As our framework of I-labelled graphs is a direct adaptation of partially labelled graphs as defined in [4], this theorem is directly obtained by the application to I-labelled graphs and I-labelled morphisms of the similar theorem of [4] that consider partially labelled graphs and graph morphisms. Finally, we also inherited from [4] that for a derivation $G \overset{r,m}{\Rightarrow} H$, H is totally labelled if and only if G is totally labelled where a I-labelled graph is said to be totally labelled when each labelling function $l_{V,i}$ is totally labelled. To sum up, graph transformations defined over $\mathscr{G}$ can be easily adapted for $\mathscr{G}^I$ (thus for I-labelled graphs) by preserving all constructions and results.

[4] A natural pushout is both a pushout and a pullback.

2 G-maps

In this section, we introduce the definition of our embedded topological structures as a particular class of I-labelled graphs. First, we consider graphs without node labels to represent the topological structure. Then, we define node labelling functions to represent the embedding. Thus, the topological structure is encoded as the base of the I-labelled graph representing the embedded topological structure.

2.1 The topological graph

As said in the introduction, we choose the topological model of generalized maps (or G-maps) [7]. This model is mathematically well defined. Its first main advantage is the homogeneity in the handling of dimensions: objects of any dimension can be represented in the same manner as graphs. This allows us to use rules for denoting operations defined on embedded G-maps, in an uniform way [11, 10]. The second advantage is that the G-map model comes with consistency constraints. They express conditions to define a topologically consistent object. Obviously, these constraints have to be maintained when operations are applied.

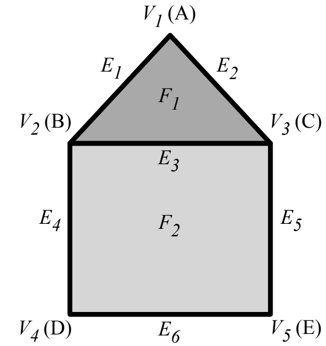


Figure 2: Embedded 2D object

The representation of an object as a G-map comes intuitively from its decomposition into topological cells (vertices, edges, faces, volumes, etc.). For example, the decomposition of the 2D topological object of Fig. 2 into a 2-dimensional G-map is shown on Fig. 3. The object is first decomposed into faces on Fig. 3(a). These faces are *linked* along their common edge E_3 with the relation α_2. In the same way, faces are split into edges connected with the relation α_1 on Fig. 3(b). At last, the edges are split into vertices by relation α_0 to obtain the 2-G-map of Fig. 3(c). Split vertices obtained at the end of the process are the nodes of the G-map graph and the α_i relations are the arcs (For a 2-dimensional G-map, i belongs to $\{0,1,2\}$). Hence, for n a dimension, n-G-maps are particular I-labelled graphs where the arc label set is $\mathscr{C}_E = \{\alpha_0,\dots,\alpha_n\}$ and where arcs are totally labelled. In fact, G-maps are represented by non-oriented graphs, that is, such that for each arc of source v, of target v' and labelled by α_i, there also exists an arc of source v', of target v' and labelled by α_i. As usual, double reversed arcs are represented on pictures by a non oriented arc. Notice that in all figures given in the sequel, we will use the α_i graphical codes of Fig. 3(c) (simple line for α_0, dashed line for α_1 and double line for α_2) in order to be more readable.

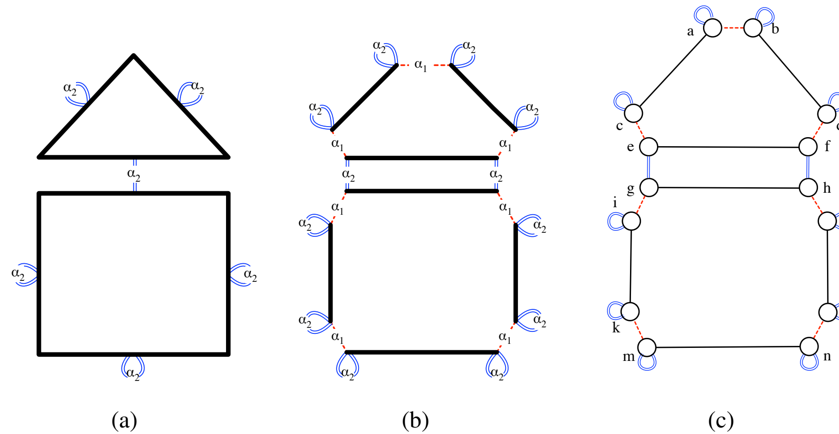


Figure 3: Cell decomposition of an object

Topological cells are not explicitly represented in G-maps but only implicitly defined as subgraphs. They can be computed using traversal of nodes using a given set of neighborhood arcs. For example, on Fig. 4, the e incident 0-cell (or object vertex) is the subgraph which contains e, nodes reachable from e using arcs α_1 and α_2 labelled (nodes c, e, g and i) and the arcs themselves. This subgraph is denoted by $<\alpha_1\alpha_2>(e)$ and models the vertex V_2 of Fig. 2. On Fig. 4, the e incident 1-cell (or object edge) is the subgraph $<\alpha_0\alpha_2>(e)$ containing nodes e, f, g and h, and adjacent α_0 and α_2 arcs. It represents the topological edge E_3. Finally, the e incident 2-cell (or object face) is the subgraph $<\alpha_0\alpha_1>(e)$ and represents the face F_1. More generally, the notion of orbit may be defined.

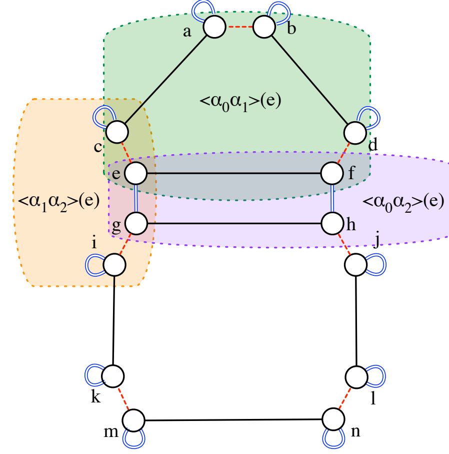


Figure 4: Reconstruction of adjacent cells of e

Definition 5 (n-topological graph and orbit) *A I-labelled graph G is said to be an n-*topological graph *if all arcs are labelled in $\mathscr{C}_E = \{\alpha_0, \ldots, \alpha_n\}$.*

Let us consider o a subword[5] *of $\alpha_0\alpha_1 \ldots \alpha_n$.*

Let $\equiv_{G<o>}$ be the equivalence orbit relation *between G nodes defined as the reflexive, symmetric and transitive closure built from arcs labelled by a label in o, i.e., ensuring that for each arc e of G labelled in o, we have $s(e) \equiv_{G<o>} t(e)$.*

*For any node v of G, the $<o>$-*orbit *(also simply called* orbit*) of G adjacent to v is denoted by $G<o>(v)$ and is defined as the subgraph of G whose set of nodes is the equivalence class of v using $\equiv_{G<o>}$, whose set of arcs are those labelled on o between previous nodes, and such that source, target, labelling functions are the restrictions of the corresponding functions on sets of nodes and arcs of the equivalence class.*

As G-maps are mathematically well defined, they come with consistency constraints.

Definition 6 (Generalised map) *An n-dimension generalized map, or n-G-map, is a n-topological graph G, that satisfies the following topological constraints :*

- **Non-orientation constraint***: G is non-oriented, i.e. for each arc e of G, there exists a reversed arc e' of G, such as $s_G(e') = t_G(e), t_G(e') = s_G(e)$, and $l_{G,E}(e') = l_{G,E}(e)$;*
- **Adjacent arc constraint***: each node is the source node of exactly $n+1$ arcs respectively labelled by α_0 to α_n;*
- **Cycle constraint***: for every α_i and α_j verifying $0 \leq i \leq i+2 \leq j \leq n$, there exists a cycle*[6] *labelled by $\alpha_i\alpha_j\alpha_i\alpha_j$ starting from each node.*

These constraints ensure that objects represented by embedded G-maps are consistent manifolds [8]. In particular, the cycle constraint ensures that in G-maps, two i-cells can only be adjacent along $(i-1)$-cells. For instance, in the 2-G-map of Fig. 3(c), the $\alpha_0\alpha_2\alpha_0\alpha_2$ cycle implies that faces are stuck along topological edges. Let us notice that thanks to loops (see α_2-loops in Fig. 3(c)), these three constraints also hold at the border of objects.

[5] $\alpha_{i_1} \ldots \alpha_{i_k}$ is a subword of $\alpha_0\alpha_1 \ldots \alpha_n$ if $i_1 \ldots i_k$ is a restricted increasing sequence of $[0,n]$.

[6] A node v of a graph G has an adjacent cycle labelled $l_1 \ldots l_k$ if there is a path of arcs $e_1 \ldots e_k$ from v to v such $e_1, \ldots, e_k$ are respectively labelled by $l_1, \ldots, l_k$.

2.2 Embedded generalized maps

We started to define n-G-map as I-labelled graphs where the arc label set is $\mathscr{C}_E = \{\alpha_0, \ldots, \alpha_n\}$. We now complete this definition with a family of node label sets to represent the embedding. Actually, as sketched in the introduction, each kind of embedding label has its own type and is defined on a particular kind of topological cell: for example, a point can be attached to a vertex, a color to a face. Thus, a node labelling function $l_{V,i}$ composing the embedding will be equipped with two static pieces of information: the kind of topological cells that is concerned by $l_{V,i}$ and the type of the data that are described by $l_{V,i}$. Based on algebraic specifications, a node labelling function is characterized by an *embedding operation* $\pi : <o> \rightarrow s$ where π is its operation name, $s \in S$ is its type with S a given set of data types and $<o>$ is its domain given as an n-dimensional orbit type. Hence, for a G-map, the family of node label sets $(\mathscr{C}_{V,\pi})_{\pi \in \Pi}$ is defined by a set Π of embedding operations. For example, for the object of Fig. 2, the set of embedding operations can be $\Pi = \{point : <\alpha_1 \alpha_2> \rightarrow point_type, color : <\alpha_0 \alpha_1> \rightarrow color_type\}$ where *point_type* and *color_type* are supposed to be appropriate data types. In particular, for an embedding operation $\pi : <o> \rightarrow s$, $\mathscr{C}_{V,\pi}$ will be a set of values of type s, according to some algebra interpreting all the sorts involved by the embedding.

Moreover, as an embedding operation $\pi : <o> \rightarrow s$ is characterized by its domain cell, it is expected that on an embedded G-map, the π-label, also called π-embedding (that is, the image by $l_{V,\pi}$) is the same for every node belonging to a common $<o>$-orbit. Hence, we represent on Fig. 5 the embedded version of the object of Fig. 2. Let us notice that this graphical representation is a simplification of the full notation. For example, we only label a with its point label A and color it with its color label instead of the full labelling $(point : A, color : dark_grey)$. Hence, for the embedding operation *point*, a and b are labelled by A, c, e, g and i by B, d, f, h and j by C, k and m by D, l and n are labelled by E. For the embedding operation *color*, nodes a to f are labelled with dark grey and nodes g to n are labelled with clear grey. Thus, on Fig. 5, for a domain $<o>$, every node of a $<o>$-orbit has the same label. We express this property by embedding constraints that embedded G-maps have to satisfy.

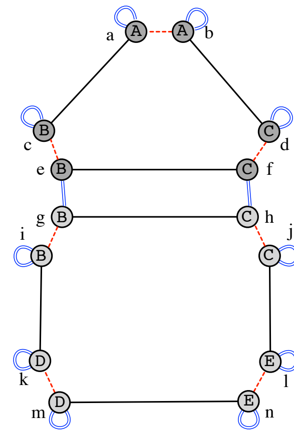


Figure 5: Embedded 2-G-map

Definition 7 (Embedded generalised map) *Let n be a dimension and Π a set of embedding operations. An embedded n-dimentional generalised map on Π, or Π-embedded n-G-map, is an n-G-map G which nodes are labelled by the family $(\mathscr{C}_{V,\pi})_{\pi \in \Pi}$, that satisfies the following embedding constraint :*

Embedding constraint*: for all embedding operations $(\pi : <o> \rightarrow s)$ of Π, all nodes of a given $<o>$-orbit of G are labelled with the same defined π-embedding* i.e. *for all nodes v and w of G, such that $v \equiv_{G<o>} w$ then $l_{V,\pi}(w) \neq \perp$ and $l_{V,\pi}(v) = l_{V,\pi}(w)$.*

Clearly, Π-embedded n-G-maps are Π-labelled graphs. To handle and compute data associated to embedding operations, we define an algebra parameterised by a given Π-embedded n-G-map G. Let us first note $v.\pi$ the access to the π-label $l_{G,V,\pi}(v)$ of a node v of G. For example, on the embedded G-map of Fig. 5, $a.point$ is A and $a.color$ is dark grey. Thanks to the topological adjacent arcs constraint, we can also define link operations on G-map's nodes that from a given node, give access to neighboring nodes. So, for each node v of G and each arc label α_i, $v.\alpha_i$ is the only node v' of G such that there exists an arc e with $s_G(e) = v$, $t_G(e) = v'$ and $l_{G,E}(e) = \alpha_i$. For example, on the embedded G-map of Fig. 5, $a.\alpha_1$ is the b node, and $a.\alpha_0.point$ is $c.point$ *i.e.* B.

In the context of geometric modelling, it is common that operations collect all the π-embedding values that are carried by nodes of a given cell. For example, the triangulation of a face collects all the points associated to the face in order to compute the new point associated to the added center. Thus, we consider the collection of a given embedding operation π carried by a given orbit $<o>(v)$. The notation $\pi\{<o>(v)\}$ will denote the multiset of π-labels of all nodes of $G<o>(v)$, that is, of the $<o>$-orbit incident to node v of G. For example, on the embedded G-map of Fig. 5, $point\{<\alpha_0,\alpha_1,\alpha_2>(a)\}$ is the multiset $\{A,B,C,D,E\}$ containing all points that correspond to *point*-labels of nodes of the $<\alpha_0,\alpha_1,\alpha_2>$-orbit adjacent to the node a. Let us notice that our definition only keeps a point per $<\alpha_1,\alpha_2>$-cell that intersects the initial cell, here the orbit $<\alpha_0,\alpha_1,\alpha_2>(a)$. Thus, even if the point B occurs four times as *point*-embedding of nodes of $<\alpha_0,\alpha_1,\alpha_2>(a)$, that is for the nodes c, e, g and i, there is an unique occurrence of the point B in $point\{<\alpha_0,\alpha_1,\alpha_2>(a)\}$ since c, e, g and i belong to the same 0-cell. To summarize, for an embedding operation $\pi:<o'>\rightarrow s$, the collect operation $\pi\{<o>(v)\}$ only keeps one π-embedding label per $<o'>$-orbit intersecting the $<o>$-orbit adjacent to v. Thus, the collected multiset contains a π-label twice if two different $<o'>$-orbits have the same π-label. In our example (cf. Fig. 5), each vertex has a different *point*-embedding and thus, each point appears only once in the resulting multiset.

Definition 8 (Embedding expressions) *Let Π be a set of embeddings for G-maps of dimension n.*
An embedding signature $\Sigma_\Pi = (S_\Pi, F_\Pi)$ is defined by:

- *a set of embedding sorts S_Π which contains at least, the predefined sort $Node$, the sort s of each embedding $\pi:<o>\rightarrow s$ of Π and the associated sort $Multi(s)$,*
- *a set of embedding operations F_Π such that each operation $f \in F_\Pi$ is equipped with its profile in $S_\Pi^* \times S_\Pi$ denoted $f: s_1 \times \ldots \times s_n \rightarrow s$. F_Π contains at least:*
  - *access operation $_.\pi : Node \rightarrow s$ for each embedding $\pi:<o>\rightarrow s$ of Π,*
  - *link operation $_.\alpha_i : Node \rightarrow Node$ to any arc label α_i,*
  - *and collect operation $\pi\{<o'>(_)\} : Node \rightarrow Multi(s)$ for every embedding $\pi:<o>\rightarrow s$ of Π and any orbit type o' of dimension n.*

Let $\mathscr{T}_\Pi(V)$ be the set of embedding terms built on Σ_Π and a variable set V of sort $Node$.
Let G be a Π-embedded n-G-map. An embedding algebra $\mathscr{A}_G$ is defined by:

- *a set of values A_s for each sort s of S_Π, such that, A_{Node} is the node set of G, and $A_{Multi(s)}$ is the multiset of A_s values,*
- *a function $f^{\mathscr{A}} : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$ for each operation $f: s_1 \times \ldots \times s_n \rightarrow s$ of F_Π, such that:*
  - *$_.\pi^{\mathscr{A}}$ is defined on each node v of G by its π-label $l_{G,V,\pi}(v)$,*
  - *$_.\alpha_i^{\mathscr{A}}$ is defined on each node v of G by the target $t_G(e)$ of the only arc e of G such $s_G(e) = v$ and $l_{G,E}(e) = \alpha_i$,*
  - *and $\pi\{<o'>(_)\}^{\mathscr{A}}$ is defined on each node v of G by the multiset[7] $\{l_{V,\pi}(w) \mid w \in W/\equiv_{G<o'>}\}$ where W is the node set of $G<o'>(v)$ and $W/\equiv_{G<o'>}$ the quotient set.*

The interpretation $eval_\sigma(t)$ of terms t of $\mathscr{T}_\Pi(V)$ using an assignment σ of variables V on G nodes, is canonically defined with the interpretation functions of $\mathscr{A}_G$.

We suppose that usual data types as *point_type* or *color_type* are provided with usual operations as the addition operation $+$, ... In the sequel, such operations are used without explicit definition. For example, the operation *mean* computes the center of gravity of a multiset of points (type $Multi(point)$).

[7] Thanks to the embedding constraint verified by the embedded G-map G and equivalence relationship properties, this collect interpretation is well defined.

3 G-maps rules

As G-maps are a particular class of Π-labelled graphs, we now investigate how operations can be defined using graph transformation rules over $\mathscr{G}^I$ (see Section 1). For example, the transformation of Fig. 6 adds a new vertex to the central edge of the previous object. To be consistent, rules on embedded G-maps need to preserve both the topological consistency and the embedding consistency. In this section, we will give some conditions on rules to ensure the preservation of constraints in relation with topology and embedding. In particular, this will allow us to state that the rule of Fig. 6 can be safely applied to any embedded *G*-map, since the resulting graph is also an embedded G-map by construction. These conditions will be extended in Section 4 to allow the user to use variables in order to handle rules that are generic with respect to the embedding values.

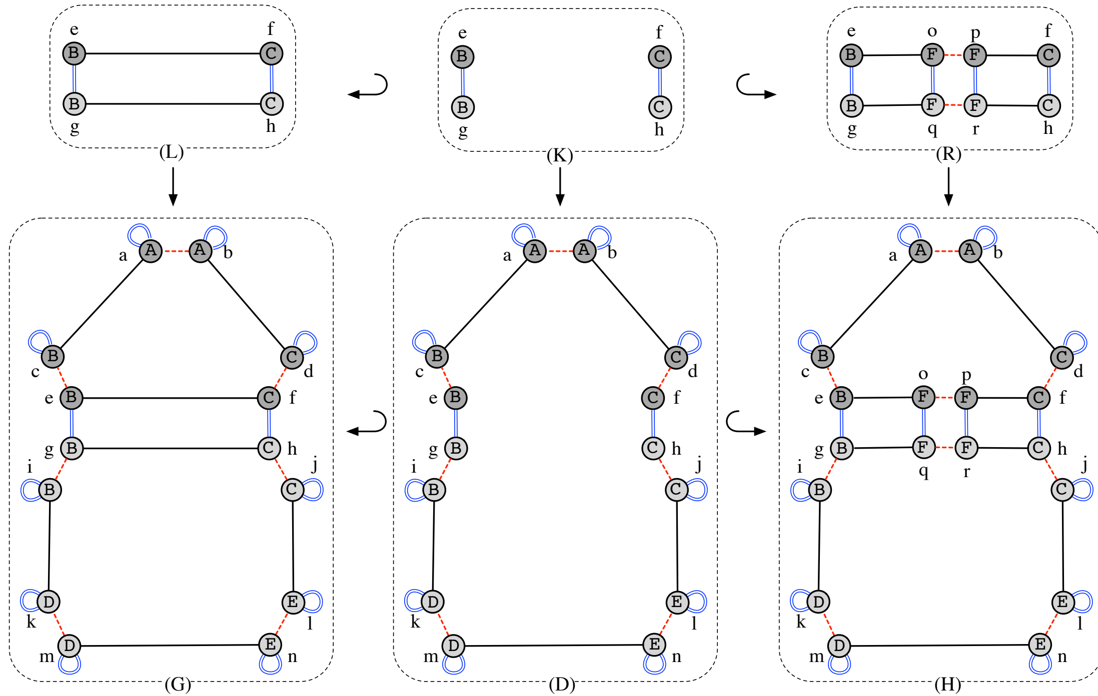


Figure 6: A simple G-map transformation

To ensure the topological consistency, we have defined in [9] the following syntactic conditions on rules.

Definition 9 (Topological consistency preservation) *For a rule $r : L \hookleftarrow K \hookrightarrow R$ over $\mathscr{G}^\perp$, the conditions of topological consistency preservation are:*

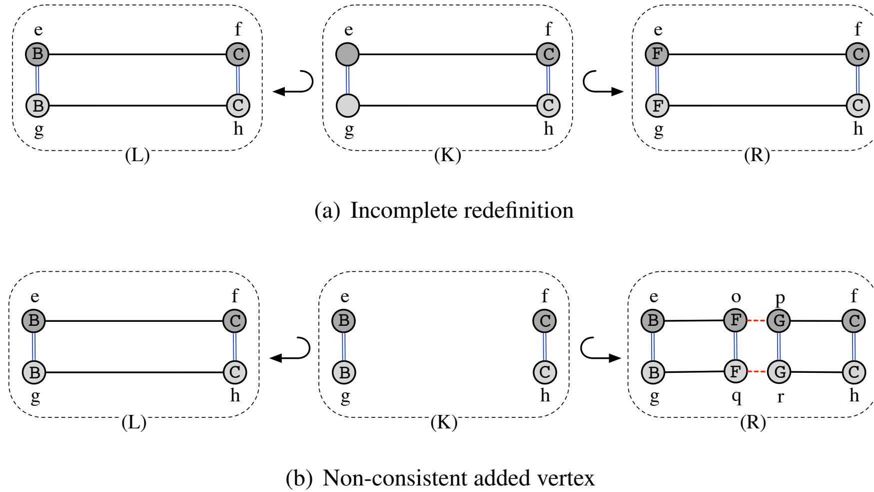
- *Non-orientation condition: both L, K and R are non-oriented graphs;*
- *Adjacent arcs condition:*
  - *adjacent arcs of preserved nodes of K have the same labels on both the left-hand side and right-hand side;*
  - *removed nodes of $L \backslash K$ and added nodes of $R \backslash K$ must have exactly $n+1$ adjacent arcs respectively labelled with α_0 to α_n;*

- *Cycles condition:*
  - *an added node of $R \backslash K$ must have with all $\alpha_i\alpha_j\alpha_i\alpha_j$-labelled cycle for $0 \leq i \leq i+2 \leq j \leq n$;*
  - *if a preserved node of K belongs to a $\alpha_i\alpha_j\alpha_i\alpha_j$-labelled cycle in L, it must belong to an $\alpha_i\alpha_j\alpha_i\alpha_j$-labelled cycle in R;*
  - *if a preserved node of K belongs to an incomplete $\alpha_i\alpha_j\alpha_i\alpha_j$-labelled cycle in L, then its α_i and α_j-labelled arcs are preserved in R.*

In the following, only rules that satisfy these topological conditions are considered. Below, we introduce syntactic conditions that ensure the embedding consistency of constructed objects.

Theorem 2 (preservation of the embedding consistency) *Let $r : L \hookleftarrow K \hookrightarrow R$ be a graph transformation rule over $\mathcal{G}^I$ that satisfies conditions of topological consistency preservation, G a Π-embedded G-map and $m : L \to G$ a match morphism. The direct transformation $G \Rightarrow^{r,m} H$ produces an Π-embedded G-map H if the following* conditions of embedding consistency preservation *are satisfied, for all embedding $\pi : <o> \to s \in \Pi$:*

- *All nodes of an $<o>$-orbit of R are labelled with the same π-embedding, defined or not - i.e. for all nodes v and w of R such that $v \equiv_{R<o>} w$, either $l_{R,V,\pi}(v) = l_{R,V,\pi}(w)$ with $l_{R,V,\pi}(v) \neq \perp$, or they are both not labelled $l_{R,V,\pi}(v) = \perp$ and $l_{R,V,\pi}(w) = \perp$.*
- *If a node v of R is an added node of $R \backslash K$ or a preserved node of K such that its π-label is changed, then $R <o> (v)$ is a complete orbit - i.e. if $v \in V_R \backslash V_K$ or $v \in V_K$ with $l_{L,V,\pi}(v) \neq l_{R,V,\pi}(v)$, then every node of $R <o> (v)$ is the source of exactly one arc labelled by α_i for each label α_i of o.*



(a) Incomplete redefinition

(b) Non-consistent added vertex

Figure 7: Two non-consistent rules, not satisfying conditions of Th. 2.

These conditions prevent the partial redefinition of an embedding. For example, the rule of Fig. 7(a) tries to redefine the point B by F. But the topological vertex (defined as a $<: \alpha_1, \alpha_2 >$-orbit) is not fully matched by the rule (α_1 is missing) and so it cannot be applied on the G-map of Fig. 5 without breaking the embedding constraints. Indeed, if the rule was applied, node e and g would be labelled by point F while c and i would still be labelled by point B. In the same way, the rule of Fig. 7(b) would add to the G-map a non-consistent new vertex embedded with two different points F and G.

Proof. The proof of this theorem can be found in the technical report [1] which contains the full length version of this paper. □

4 G-map rule schemes

Simple rules on G-maps are quite limited. Actually, in the general context of graph transformations, rules without variables are sufficient if it is possible to write all possible transformations. In the context of geometric modeling, both the topological graph structure and the embedding node labelling are not predefined. The topological transformation depends on the original shape of the cell to transform (its number of vertices, edges, etc.). This issue has been solved by [10, 9] with the introduction of rule schemes based on topological variables. These variables allow us to represent both the matched topological cells and their transformations. For example, a topological variable of type $< \alpha_0, \alpha_1 >$ can represent any arbitrary 2-cell such that the topological triangulation operation can be applied to a triangle, a square or a pentagon. A topological rule scheme is then instantiated according to a substitution of the given variable by a 2-cell of the G-map to be transformed. Such an instantiation builds a transformation rule that meets the conditions of topological consistency preservation (provided that the scheme rule also meets some conditions given in [10, 9]). In the same way, the embedding transformation depends on the original embedding of the matched cell. For example, usually, when a face is triangulated, the central position of the added vertex depends on the positions of existing vertices. With the simple framework of Section 3, there should be as many rules as possible vertex positions. We introduce embedding variables to get rule schemes that will be instantiated according to the different possible values associated to the variables.

These variables are based on the notion of *attributed variables* introduced by [5]. The variables label nodes of the left-hand side of rules in order to match the existing labels of the object. In the right-hand side, new labels are defined as expressions upon these variables. These algebraic expressions are then interpreted when rules are applied. For example in Fig. 8, the variables x and y of the left-hand side can match any labels and the expression $x+y$ of the right-hand side should be evaluated according to the values provided by the match morphism in order to define the label of the new node 4. To apply this rule to an object, we instantiate the variables of the rule with the corresponding values of the matched object to obtain a classical rule that is applied as a direct transformation.

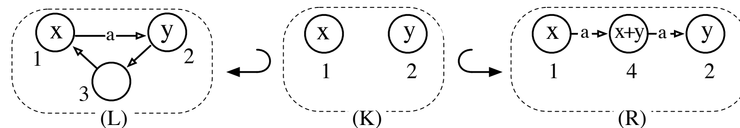
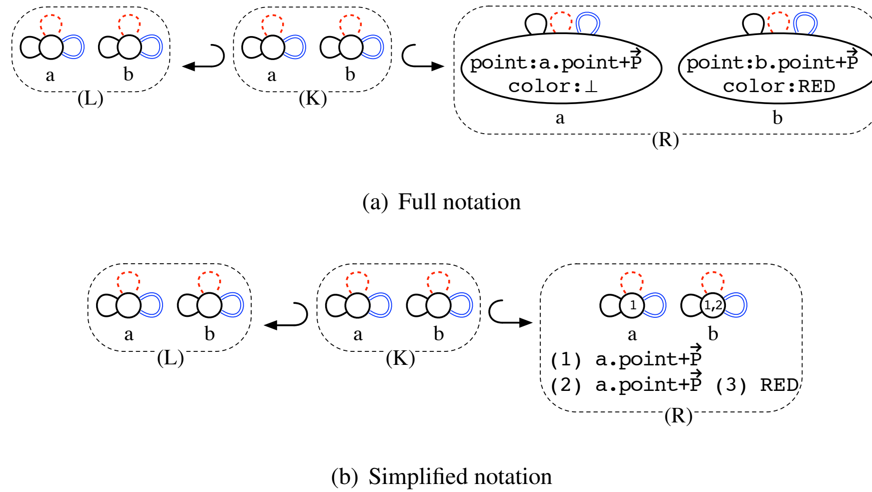


Figure 8: A rule with attributed variables

As in our case, nodes have multiple labels, rules can have a variable per node and per embedding operation to define transformations. To simplify computations on embedding values, we use embedding expressions introduced in Section 2. For example, on Fig. 9(a), the rule translates by a vector $\vec{P}$ the points associated to the nodes a and b. The color associated to node b is redefined while the color associated to a is not matched by the rule and, as a consequence, not transformed. On Fig. 9(b) we use a simplified notation. As there is no ambiguity on the type of the expressions, they are not explicitly typed. In the same way, the unmatched color of a is not represented. Moreover, for lack of space, the expressions will often be placed below the graph and referenced by a number. For example, the node a is labelled by the number 1 that represents the expression $a.point + \overrightarrow{P}$ associated to (1).

Let us notice that in the example of Fig. 9, this notation allows us to not explicitly label both the left-hand side and the kernel of the rule in order to match the embedding. Expressions on variable names allow us to directly compute new labels in the right-hand side. For example, on Fig. 10(a), when



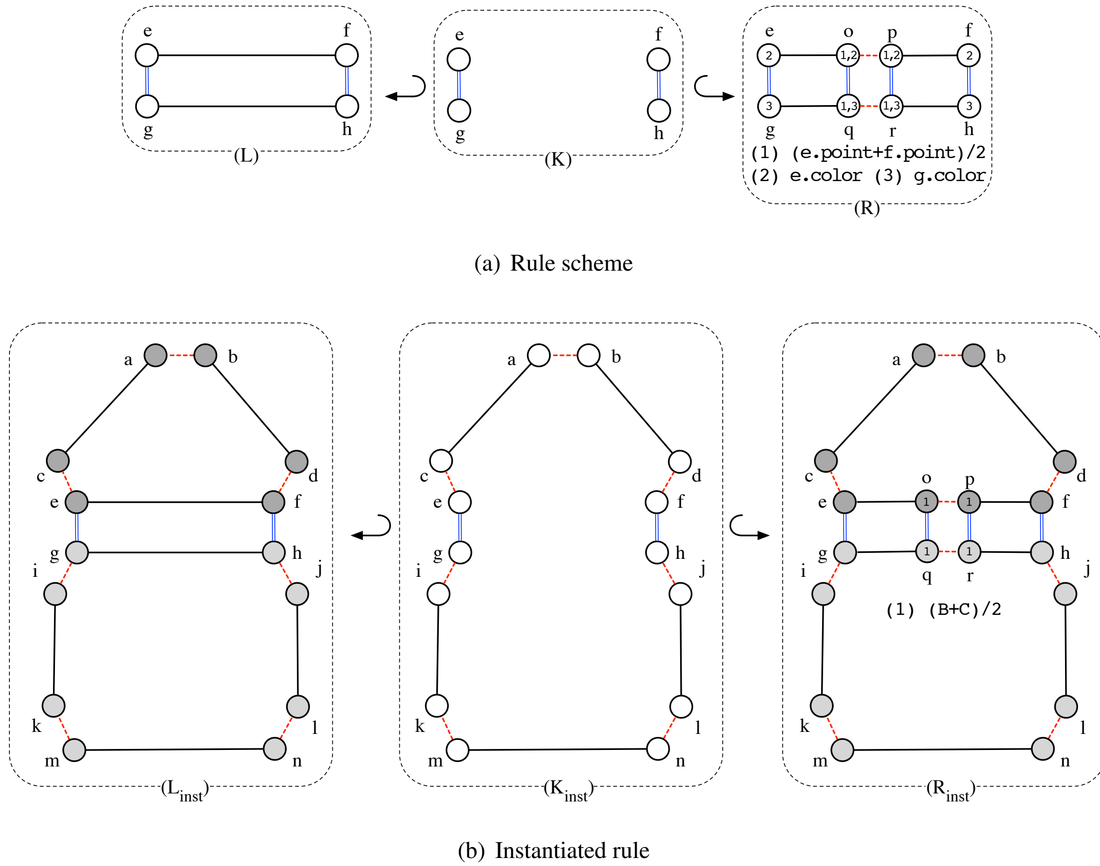
(a) Full notation

(b) Simplified notation

Figure 9: Translation of an isolated vertex

the edge is split, the center is computed with the expression $(e.point + f.point)/2$ while the preserved nodes keep their original embedding. In order to apply the rule of Fig. 10(a) to object of Fig. 5 along the inclusion match morphism, the variables have to be instantiated and expressions computed. For example on Fig. 10(b), $e.color$ and $g.color$ are respectively instantiated by dark grey and light grey and the new point is computed as $(B+C)/2$. However, even with such evaluation and computation mechanisms, the rule cannot be directly applied. The instantiation mechanism has also to complete the orbits of redefined embedding values. Indeed, rule schemes describe the modification in a minimal way. In particular, for an embedding operation $\pi : <o> \to s$, we have to deal with indirect modifications for nodes belonging to an $<o>$-orbit of a node whose π-embedding is modified by the rule. For example, as the *color*-embedding labels are redefined for the node e, f, g and h (in the present case they remain the same), then, potentially, the *color*-embedding of all nodes that belong to an $<\alpha_0, \alpha_1>$-orbit of one of these nodes can be modified by the transformation rule application. For this reason, for a given match morphism, the instantiation mechanism will both substitute the embedding variables and complete the pattern under modification to include all possible indirect modifications (in Fig. 10(b), the completion mechanism will consider the full triangle and the full square in order to redefine colors). The application of the instantiated rule to the object is then the classical rule application (as described in Section 3).

The rule schemes allow us to compute new embedding values by using expressions introduced in Section 2. For example, the rule scheme of Fig. 11(a) defines the triangulation of a triangle. A vertex is added at the center of the face, and its associated point is defined by the expression $mean(point\{<\alpha_0\alpha_1>(a)\})$ as the mean of the points of the face. This expression is interpreted by $mean\{A,B,C\}$ when rule is instantiated on Fig. 11(b) to be applied on object Fig. 2. Simultaneously, the colors of faces created by triangulation are defined as the mean between the original face color and the color of their respective adjacent faces. For example, the left/up side face color is defined as $(a.color + a.\alpha_2.color)/2$ where the expression $a.\alpha_2$ represents a node of the adjacent face (or the node itself if there is no adjacent face). When this rule is instantiated on Fig. 11(b), $a.\alpha_2.color$ is instantiated by the color of a, $b.\alpha_2.color$ by the color of b and $e.\alpha_2.color$ by the color of g. Let us notice that for this instantiation, the face is fully matched by the rule scheme and so the face orbit does not have to be completed to define the color properly. At the opposite, the vertex orbits corresponding to the embedded points B and C are not fully matched but they have to be completed with g, h, i and j by the instantiation mechanism since *point*-embeddings are redefined for the nodes e and f.



(a) Rule scheme

(b) Instantiated rule

Figure 10: Edge splitting scheme

Definition 10 (Graph scheme) *Let G be a Π-embedded n-G-map. Let us consider an embedding signature Σ_Π and its corresponding embedding algebra $\mathscr{A}_G$.*

A graph scheme H on $\mathscr{T}_\Pi(V)$ is a Π-labelled graph on terms of $\mathscr{T}_\Pi(V)$.

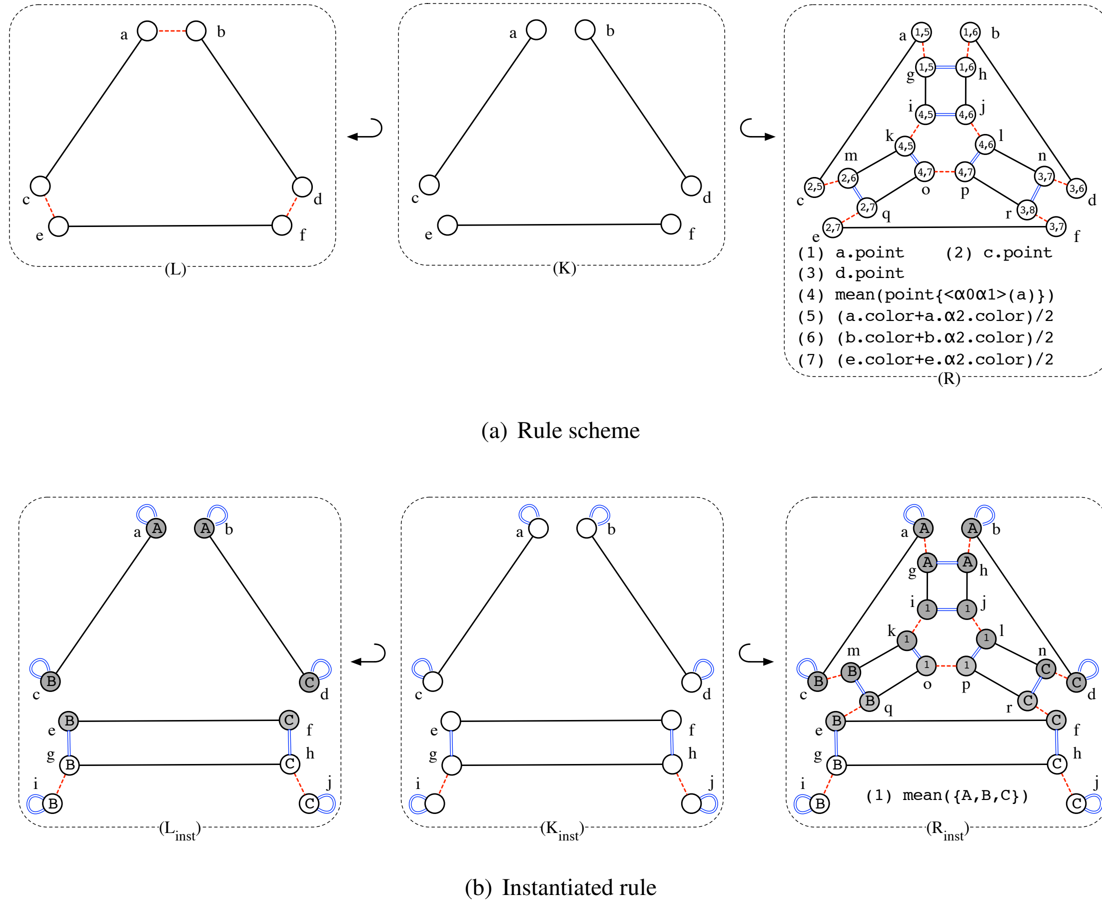
Let $\sigma : V \rightarrow V_G$ be an interpretation of the variables, the evaluation $eval_\sigma(H)$ of the graph H is the Π-labelled graph that has the same base ($eval_\sigma(H)_\perp = H_\perp$) such as for each embedding operation π of Π, $l_{eval_\sigma(H),V,\pi} = l_{H,V,\pi} \circ eval_\sigma$.

Definition 11 (Rule scheme) *Let Π be a set of embedding operations of dimension n and Σ_Π an embedding signature.*

A rule scheme $r_\mathscr{T} : L_\mathscr{T} \hookleftarrow K_\mathscr{T} \hookrightarrow R_\mathscr{T}$ on Σ_Π is defined by two inclusion morphisms $K_\mathscr{T} \hookrightarrow L_\mathscr{T}$ and $K_\mathscr{T} \hookrightarrow R_\mathscr{T}$ between the graph schemes $L_\mathscr{T}$, $K_\mathscr{T}$ and $R_\mathscr{T}$ on $\mathscr{T}_\Pi(V_L)$ such that:

- *node labels of $L_\mathscr{T}$ (and so, labels of $K_\mathscr{T}$) are undefined* - i.e. $L_\mathscr{T} = \Pi_{\pi\in\Pi}(proj_\pi(L_\mathscr{T})_\perp)$;
- *$R_\mathscr{T}$ satisfies the embedding constraints of Definition 7.*

The instantiation mechanism of a rule scheme is constructive and based on the match morphism $m : L_\mathscr{T} \rightarrow G$ between the left-hand side of the scheme rule $L_\mathscr{T}$ and the embedded G-map G on which the rule schema is applied. The main underlying idea is basically to build from the considered pattern ($L_\mathscr{T}$, $K_\mathscr{T}$ or $R_\mathscr{T}$) and from the match morphism m, a graph completed with all nodes (and arcs) belonging to orbits whose embedding values can potentially be modified by the application of the rule. The resulting graphs are respectively denoted as $L[m]$, $K[m]$ and $R[m]$.



(a) Rule scheme

(b) Instantiated rule

Figure 11: Triangulation scheme

- the left hand-side $L[m]$ of the instantiated rule will consist of all matched nodes together with nodes whose embedding values can be indirectly modified and of all associated embedding values.
- similarly, the kernel $K[m]$ will be built following the same construction, but without node labels.
- the right hand-side $R[m]$ will include $K[m]$ and be completed with added parts and labels of $R_{\mathscr{T}}$ that are evaluated.

Definition 12 (Rule scheme instantiation) *Let Π be a set of embedding of dimension n and Σ_Π an embedding signature. Let $r_{\mathscr{T}} : L_{\mathscr{T}} \hookleftarrow K_{\mathscr{T}} \hookrightarrow R_{\mathscr{T}}$ be a rule scheme on Σ_Π, $m : L_{\mathscr{T}} \to G$ be a match morphism on a Π-embedded n-G-map G, and $\mathscr{A}_G$ be a Σ_Π-embedding algebra.*

The instantiated rule $r[m] : L[m] \hookleftarrow K[m] \hookrightarrow R[m]$ is defined by

- $L[m] = Lsat_{\Pi \times V_{K_{\mathscr{T}}}}(L_{\mathscr{T}})$,
- $K[m] = Ksat_{\Pi \times V_{K_{\mathscr{T}}}}(K_{\mathscr{T}})$,
- *and* $R[m] = Rsat_{\Pi \times V_{K_{\mathscr{T}}}}(R_{\mathscr{T}})$,

where the saturation operators Lsat, Ksat and Rsat are recursively defined on $\Pi \times V_{K_{\mathscr{T}}}$.

Let us define the saturation operators Lsat, Ksat and Rsat by the following induction principle over the elements of the set $\Pi \times V_{K_{\mathscr{T}}}$:

- **base case** $\Pi \times V_{K_{\mathscr{T}}} = \emptyset$.

  Let $\sigma_m : V_{L_{\mathscr{T}}} \to V_G$ *be the substitution that associates to each node* v *of* $L_{\mathscr{T}}$ *its image* $m(v)$ *along the match morphism* m.

  $Lsat_{\emptyset}(L_{\mathscr{T}})$, $Ksat_{\emptyset}(K_{\mathscr{T}})$ *and* $Rsat_{\emptyset}(R_{\mathscr{T}})$ *are the graphs respectively isomorphic to* $m(L_{\mathscr{T}})$ *(the node images with all their embedding values and arcs issued from* $L_{\mathscr{T}}$*),* $Prod_{\pi\in\Pi}(proj_{\pi}(m(K_{\mathscr{T}}))_{\perp})$ *and* $eval_{\sigma_m}(R_{\mathscr{T}})$ *such that the following inclusions exist:* $Lsat_{\emptyset}(L_{\mathscr{T}}) \hookleftarrow Ksat_{\emptyset}(K_{\mathscr{T}}) \hookrightarrow Rsat_{\emptyset}(R_{\mathscr{T}})$.

  Let $h_{Lsat_{\emptyset}} : L_{\mathscr{T}} \to Lsat_{\emptyset}(L_{\mathscr{T}})$ *be the morphism that associates each node* v *of* $L_{\mathscr{T}}$ *to the node of* $Lsat_{\emptyset}$ *isomorphic to* $m(v)$.

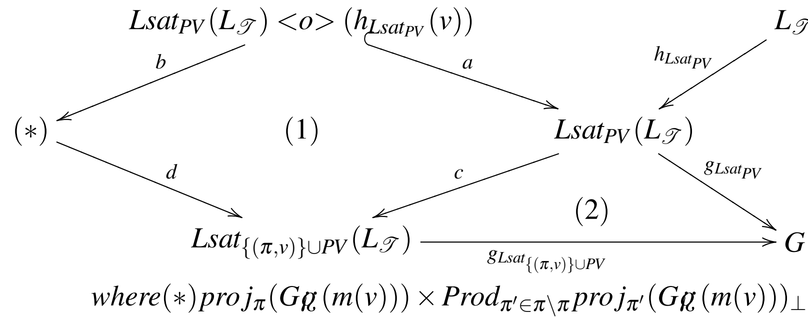
  Let $g_{Lsat_{\emptyset}} : Lsat_{\emptyset}(L_{\mathscr{T}}) \to G$ *that associates each node of* $Lsat_{\emptyset}(L_{\mathscr{T}})$ *that is isomorphic to* $m(v)$ *to* $m(v)$ *itself. In particular, for all node* v *of* $L_{\mathscr{T}}$, $g_{Lsat_{\emptyset}}(h_{Lsat_{\emptyset}}(v)) = m(v)$.

- **induction step** $\Pi \times V_{K_{\mathscr{T}}} \neq \emptyset$

  Let note a subset $PV \subset \Pi \times V_{K_{\mathscr{T}}}$*, a* Π*-labelled rule* $Lsat_{PV}(L_{\mathscr{T}}) \hookleftarrow Ksat_{PV}(K_{\mathscr{T}}) \hookrightarrow Rsat_{PV}(R_{\mathscr{T}})$*, and two morphisms* $h_{Lsat_{PV}} : L_{\mathscr{T}} \to Lsat_{PV}(L_{\mathscr{T}})$ *and* $g_{Lsat_{PV}} : Lsat_{PV}(L_{\mathscr{T}}) \to G$.

  Let $\pi : <o> \to s \in \Pi$ *and* $v \in V_{K_{\mathscr{T}}}$ *with* $(\pi, v) \notin PV$.

  Let us construct $Lsat_{\{(\pi,v)\}\cup PV}(L_{\mathscr{T}})$ *with the appropriate morphisms.*



$$where(*) proj_{\pi}(G\ell(m(v))) \times Prod_{\pi'\in\pi\setminus\pi} proj_{\pi'}(G\ell(m(v)))_{\perp}$$

Let us define the morphisms

- $a : Lsat_{PV}(L_{\mathscr{T}}) <o> (h_{Lsat_{PV}}(v)) \hookrightarrow Lsat_{PV}(L_{\mathscr{T}})$
- *and* $b : Lsat_{PV}(L_{\mathscr{T}}) <o> (h_{Lsat_{PV}}(v)) \to proj_{\pi}(G\ell(m(v))) \times Prod_{\pi'\in\Pi\setminus\pi} proj_{\pi'}(G\ell(m(v)))_{\perp}$

such that for all node or arc x *of* $Lsat_{PV}(L_{\mathscr{T}}) <o> (h_{Lsat_{PV}}(v))$, $b(x) = g_{Lsat_{PV}}(x)$.

Let us define $Lsat_{\{(\pi,v)\}\cup PV}(L_{\mathscr{T}})$ *as the pushout (1) of a and b defined by*

- $c : Lsat_{PV}(L_{\mathscr{T}}) \to Lsat_{\{(\pi,v)\}\cup PV}(L_{\mathscr{T}})$
- *and* $d : proj_{\pi}(G\ell(m(v))) \times Prod_{\pi'\in\Pi\setminus\pi} proj_{\pi'}(G\ell(m(v)))_{\perp} \to Lsat_{\{(\pi,v)\}\cup PV}(L_{\mathscr{T}})$.

Let $h_{Lsat_{\{(\pi,v)\}\cup PV}} = c \circ h_{Lsat_{PV}}$.

And let $g_{Lsat_{\{(\pi,v)\}\cup PV}} : Lsat_{\{(\pi,v)\}\cup PV} \to G$ *be the morphism such as diagram (2) is commutative and* $g_{Lsat_{\{(\pi,v)\}\cup PV}} \circ d$ *be the identity for all node or arc x of* $proj_{\pi}(G\ell(m(v))) \times Prod_{\pi'\in\Pi\setminus\pi} proj_{\pi'}(G\ell(m(v)))_{\perp}$ *(that is always possible because (1) and (2) are commutative).*

In particular, for all node v *of* $L_{\mathscr{T}}$, $g_{Lsat_{\{(\pi,v)\}\cup PV}}(h_{Lsat_{\{(\pi,v)\}\cup PV}}(v)) = m(v)$ *because (2) commutes and the induction hypothesis on* $Lsat_{PV}$ *and its associated morphism.*

The construction of $Ksat_{\Pi,V_{K_{\mathscr{T}}}}(K_{\mathscr{T}})$ *and* $Rsat_{\Pi,V_{K_{\mathscr{T}}}}(R_{\mathscr{T}})$ *is similar with a difference in the labelling along b and d. For the kernel, as we want no label, we use* $Prod_{\pi\in\Pi} proj_{\pi}(G\ell(m(v)))_{\perp}$ *instead of* $proj_{\pi}(G\ell(m(v))) \times Prod_{\pi'\in\Pi\setminus\pi} proj_{\pi'}(G\ell(m(v)))_{\perp}$. *In the same way, for the right hand-side,*

as we want expression interpretations as node labels, we use $(proj_{\pi}(G_{\not{\kappa}}(m(v))))_{eval_{\sigma m}(l_{R_{\mathscr{T}},V,\pi}(v))} \times Prod_{\pi' \in \Pi \setminus \pi} proj_{\pi'}(G_{\not{\kappa}}(m(v)))_{\perp}$ *instead of* $proj_{\pi}(G_{\not{\kappa}}(m(v))) \times Prod_{\pi' \in \Pi \setminus \pi} proj_{\pi'}(G_{\not{\kappa}}(m(v)))_{\perp}$.
The following inclusions hold: $Lsat_{\{(\pi,v)\} \cup PV}(L_{\mathscr{T}}) \hookleftarrow Ksat_{\{(\pi,v)\} \cup PV}(K_{\mathscr{T}}) \hookrightarrow Rsat_{\{(\pi,v)\} \cup PV}(R_{\mathscr{T}})$.
Finally, the result match morphism $m^* : L[m] \to G$ *is* $m^* = g_{Lsat_{\Pi \times V_{K_{\mathscr{T}}}}}$.

Let us note that the inclusion morphism a always exists, by definition of orbits. For the left-hand and kernel parts, it is clear that b exists, since all added graphs during saturation are included in G. For the right-hand part, existence of b depends on the condition imposed on $R_{\mathscr{T}}$ by the rule scheme definition. Thus, the saturation with (π,v) and (π,w) for two nodes that belong to the same orbit (*ie* $v \equiv_{R_{\mathscr{T}}<o>} w$ where $<o>$ is the domain of π) adds the same graph. Especially, $l_{R_{\mathscr{T}},V,\pi}(v) = l_{R_{\mathscr{T}},V,\pi}(w)$, and thus $proj_{\pi}(G_{\not{\kappa}}(m(v)))_{eval_{\sigma m}(l_{R_{\mathscr{T}},V,\pi}(v))} = proj_{\pi}(G_{\not{\kappa}}(m(w)))_{eval_{\sigma m}(l_{R_{\mathscr{T}},V,\pi}(w))}$.

At each saturation step, graphs added to the left-hand side, to the kernel, and to the right-hand side have the same base. Thus, the double inclusion always exists with an adequate choice of node names and arc names.

The saturation order of (π,v) couples does not matter, because the construction of the morphism b guarantees an unique addition of nodes and arcs of G (or their isomorphisms) to the instantiated rule.

Theorem 3 (preservation of embedded G-map's consistency) *Let* Π *be a set of embedding of dimension n,* $r_{\mathscr{T}} : L_{\mathscr{T}} \hookleftarrow K_{\mathscr{T}} \hookrightarrow R_{\mathscr{T}}$ *be a rule scheme and* $m : L_{\mathscr{T}} \to G$ *be a match morphism on a* Π*-embedded n-G-map G. If* $r_{\mathscr{T}}$ *satisfies the conditions of topological consistency preservation for m, the direct transformation* $G \Rightarrow^{r[m],m^*} H$ *with the instantiated rule* $r[m]$ *exists and produces a* Π*-embedded n-G-map H.*

Proof. The proof of this theorem can be found in the technical report [1]. □

Conclusion

In this article, in the context of topology-based geometric modelling, we have proposed a representation of embedded n-dimensional objects as a particular class of I-labelled graphs. Nodes have as many labels as there are different kinds of data to represent the geometric embedding. The category of I-labelled graphs is defined as a natural extension of the partially labelled graphs defined in [4]. Considering the modelling operations, we extend a rule-based language [10] used to define topological operations. We introduce embedding variables and expressions on rule node labels to deal with the computation of the embedding of constructed objects. The resulting language allows to define geometric operations in an easy and safe way, as constraints on rules ensure both topological and geometric consistency.

Moreover, we have already designed a first prototype of a topology-based geometric modeler, but only for pure topological operations described with rule schemes based on topological variables [9]. As previously mentioned, these variables allow us to define topological operations independently from the size of cells, that is, from the number of nodes constituting the cell to be filtered. For example, it allows us to define the topological triangulation of a triangle, a square, or any face with a single generic rule. The tool can be seen as a rule-application engine dedicated to our topological transformation rules [2]. It allows us to quickly design and implement a modeler by specifying both its topological dimension and its set of application dedicated rules. For usual topological operations, the prototype efficiency is comparable to other topology-based geometric modelers based on G-maps. An unquestionable benefit of our approach is that topological operations can be quickly designed and implemented and that prototyped modelers are easily and safely extensible [2]. We are now extending this first prototype with embedding variables to deals with geometric operations. The combination of the two kind of variables has still to be formalized but the first developments attest of their compatibility.

References

[1] T. Bellet, A. Arnould & P. Le Gall (2011): *Rule-based transformations for geometric modeling*. Research Notes 2011-1, XLIM-SIC, UMR CNRS 6172, University of Poitiers.

[2] T. Bellet, M. Poudret, A. Arnould, L. Fuchs & P. Le Gall (2010): *Designing a topological modeler kernel: a rule-based approach*. In: *Shape Modeling International (SMI'10) Shape Modeling International (SMI'10)*. Aix-en-Provence, France.

[3] H. Ehrig, K. Ehrig, U. Prange & G. Taentzer (2006): *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc. Secaucus, NJ, USA.

[4] A. Habel & D. Plump (2002): *Relabelling in Graph Transformation*. In: *Graph Transformation, First International Conference, ICGT*. *Lecture Notes in Computer Science* 2505, Springer, pp. 135–147.

[5] B. Hoffmann (2005): *Graph transformation with variables*. *Formal Methods in Software and System Modeling* 3393, pp. 101–115.

[6] P. Lienhardt (1989): *Subdivision of n-dimensional spaces and n-dimensional generalized maps*. In: *Annual Symposium on Computational Geometry SCG'89*. ACM Press, Saarbruchen, Germany, pp. 228–236.

[7] P. Lienhardt (1991): *Topological models for boundary representation: a comparison with n-dimensional generalized maps*. *Computer-Aided Design* 23(1), pp. 59–82.

[8] P. Lienhardt (1994): *N-dimensional generalised combinatorial maps and cellular quasimanifolds*. *International Journal on Computational Geometry and Applications (IJCGA)* (3), pp. 275–324.

[9] M. Poudret (2009): *Transformations de graphes pour les opérations topologiques en modélisation géométrique, Application à l'étude de la dynamique de l'appareil de Golgi*. Thèse, Université d'Évry val d'Essonne, Programme Epigénomique.

[10] M. Poudret, A. Arnould, J.-P. Comet & P. Le Gall (2008): *Graph Transformation for Topology Modelling*. In: *4th International Conference on Graph Transformation (ICGT'08)*. *LNCS* 5214, Springer, Leicester, United Kingdom, pp. 147–161.

[11] M. Poudret, J.-P. Comet, P. Le Gall, A. Arnould & P. Meseure (2007): *Topology-based Geometric Modelling for Biological Cellular Processes*. In: *1st International Conference on Language and Automata Theory and Applications (LATA 2007)*. Tarragona, Spain. Http://grammars.grlmc.com/LATA2007/proc.html.

Dependently-Typed Formalisation of Typed Term Graphs

Wolfram Kahl
McMaster University, Hamilton, Ontario, Canada,
kahl@cas.mcmaster.ca

We employ the dependently-typed programming language Agda2 to explore formalisation of untyped and typed term graphs directly as set-based graph structures, via the gs-monoidal categories of Corradini and Gadducci, and as nested **let**-expressions using Pouillard and Pottier's NotSoFresh library of variable-binding abstractions.

1 Introduction

The Coconut project [AK09a, AK09b] uses "code graphs" [KAC06], a variant of term graphs in the spirit of "jungles" [HP91, CR93], as intermediate presentation for the generation of highly optimised assembly code. This is currently implemented in Haskell, and we use the Haskell type system in an embedded domain-specific language (EDSL) for creating such code graphs via what appears to be standard Haskell function definitions, with **let**-definitions introducing sharing, and with functions representing assembly-level operations constructing hyperedges [AK09a]. However, since Haskell does not support full dependent typing, the intermediate term graph datatype interface, supporting graph navigation, traversal, and manipulation operations, cannot preserve the connection with the Haskell-level typing of the assembly operations. Therefore, although EDSL-created code graphs are *well-typed by construction, as certified by the type checker*, this does not hold anymore for code graphs that are the result of internal operations. Those internal operations either require separate proof that they preserve well-typedness, or they need to perform run-time checks, at considerable run-time cost.

In addition, our code-graph-creation EDSL has a second "simulator" implementation, which turns the EDSL expressions into Haskell functions that implement a "machine simulation". Since the code graph representation has lost its connection with the Haskell-level typing, it is "unintuitively hard" to use the simulation machinery for code graphs that result from code graph manipulation operations.

Mainly for these reasons, we are now exploring implementation of code graphs in a dependently typed programming language, where there is no need to "loose" the type information when moving to a graph representation, and where even stronger assertions about operations on code graphs than just type preservation can be proven *inside* the implementing system.

We start, in Sect. 2, with a quick introduction to the dependently typed programming language (and proof checker) Agda [Nor07]. This is followed by formalisations of set-based mathematical definitions of untyped (Sect. 3) and typed (Sect. 4) term graphs, and then a summary of the gs-monoidal category view on these term graphs in Sect. 5. Finally, we present two formalisations of acyclic term graphs as (differently structured) nested **let**-expressions (Sections 6 and 7).

2 Introduction to Agda: Types, Sets, Equality

The Agda home page[1] states:

[1] http://wiki.portal.chalmers.se/agda/

Rachid Echahed (Ed.): 6th International Workshop
on Computing with Terms and Graphs (TERMGRAPH 2011)
EPTCS 48, 2011, pp. 38–53, doi:10.4204/EPTCS.48.6

© Wolfram Kahl

> **Agda is a dependently typed functional programming language.** It has inductive families, i.e., data types which depend on values, such as the type of vectors of a given length. It also has parametrised modules, mixfix operators, Unicode characters, and an interactive Emacs interface which can assist the programmer in writing the program.
>
> **Agda is a proof assistant.** It is an interactive system for writing and checking proofs. Agda is based on intuitionistic type theory, a foundational system for constructive mathematics developed by the Swedish logician Per Martin-Löf. It has many similarities with other proof assistants based on dependent types, such as Coq, Epigram, Matita and NuPRL.

Syntactically and "culturally", Agda is quite close to Haskell. However, since Agda is strongly normalising and has no $\bot$ values, the underlying semantics is quite different. Also, since Agda is dependently typed, it does not have the distinction that Haskell has between terms, types, and kinds (the "types of the types"). The Agda constant Set corresponds to the Haskell kind *; it is the type of all "normal" datatypes. For example, the Agda standard library defines the type Bool as follows:

```
data Bool : Set where true  : Bool
                      false : Bool
```

Since Set needs again a type, there is Set_1, with Set : Set_1, etc., resulting in a hierarchy of "universes". Since version 2.2.8, Agda supports *universe polymorphism*, with universes Set i where i is an element of the following special-purpose variant of the natural numbers:

```
data Level : Set where zero : Level
                       suc : (i : Level) → Level
```

With this, the conventional usage turns into syntactic sugar, so that Set is now Set zero, and Set_1 = Set (suc zero). For example, the standard library includes the following universe-polymorphic definition for the parameterised Maybe type:

```
data Maybe {a : Level} (A : Set a) : Set a where just : (x : A) → Maybe A
                                                 nothing : Maybe A
```

Maybe has two parameters, a and A, where dependent typing is used since the type of the second parameter depends on the first parameter. The use of {...} flags a as an *implicit parameter* that can be elided where its type is implied by the call site of Maybe. This happens in the occurrences of Maybe A in the types of the data constructors just and nothing: In Maybe A, the value of the first, implicit parameter of Maybe can only be a, the level of the set A.

The same applies to implicit function arguments, and in most cases, implicit arguments or parameters are determined by later arguments respectively parameters. Frequently, implicit arguments correspond quite precisely to that part of the context of mathematical statements that is frequently left implicit by mathematicians, so that the reader may be advised to skip implicit arguments at first reading of a type, and return to them for clarification where necessary for understanding the types of the explicit parameters.

While the Hindley-Milner typing of Haskell and ML allows function definitions without declaration of the function type, and type signatures without declaration of the universally quantified type variables, in Agda, almost all types and variables need to be declared, but implicit parameters and the type checking machinery used to resolve them alleviate that burden significantly. For example, the original definition writes only Maybe {a} (A : Set a) : Set a, since the type of a will be inferred from a's use as argument to Set.

The "programming types" like Maybe can be freely mixed with "formula types", inspired by the Curry-Howard-correspondence of "formulae as types, proofs as terms". The formula types of true formulae contain their proofs, while the formula types of false formulae are empty.

The standard library type of propositional equality has (besides two implicit parameters) one explicit parameter and one explicit argument; the definition therefore gives rise to types like the type "$2 \equiv 1 + 1$", which can be shown to be inhabited using the definition of natural numbers 1 and 2 and natural number addition $+$, and the type "$2 \equiv 3$", which is an empty type, since it has no proof.

```
data _≡_ {a : Level} {A : Set a} (x : A) : A → Set a where refl : x ≡ x
```

The underscore characters occurring in the name _≡_ declare mixfix syntax with argument positions for explicit parameters and arguments; this mixfix syntax is already used in the type of the single constructor. The definition introduces types x ≡ y for any x and y of type A, but only the types x ≡ x are inhabited, and they contain the single element refl {a} {A} {x}.

In Agda, as in other type theories without quotient types, sets with equality are typically modelled as *setoids*, that is, carrier types equipped with an equivalence. This closely corresponds to the non-primitive nature of the "equality" test (==) : Eq a ⇒ a → a → Bool in Haskell. A setoid is a dependent record consisting of a Carrier set, a relation _≈_ on that carrier, and a proof that the relation _≈_ is an equivalence relation:

```
record Setoid c l : Set (suc (c⊔l)) where
  field Carrier : Set c
        _≈_   : Rel Carrier l
        isEquivalence : IsEquivalence _≈_
  open IsEquivalence isEquivalence public
```

An Agda record is also a module that may contain other material besides its **field**s; the "**open**" clause makes the fields of the equivalence proof available as if they were fields of Setoid. This language feature enables incremental extension of smaller theories to larger theories at very low notational cost.

Whenever we allow arbitrary node or edge sets, and we want to prove, for example, isomorphism of certain graphs, we actually need setoids and not just sets. For such contexts, we introduce the following abbreviation for extracting the carrier set from a setoid:

```
⌊_⌋ : {c l : Level} → Setoid c l → Set c
⌊ s ⌋ = Setoid.Carrier s
```

3 Set-Based Term Graphs

We now present a simple definition of term graphs that is intentionally kept close to conventional mathematical formulations. To reduce complexity and improve readability of this initial formalisation, we present untyped term graphs here; a typed variant will be shown in Sect. 4.

In the context of an arity-indexed label type Label : ℕ → Set, we first define a type DHG_1 of directed hypergraphs with one putput per edge, indexed by input and output arities of the whole graph, with the following components (since Agda records are also modules, they can contain additional material besides their **field**s):

- A setoid Inner of non-input nodes. (For simplicity, we do not emply universe polymorphism here, and all our setoids are of type Setoid zero zero.)
  For technical reasons, we find it more convenient to have the non-input nodes separate from the input nodes. Otherwise we would have had to include an explicit injection from the input positions to the complete node set.
- The setoid Node of all nodes is then derived as the disjoint union of Inner with the setoid of input positions, which is obtained from Fin m, the set of natural numbers smaller than m.
- The second **field** is the n-element vector of output nodes, which can be either input nodes or inner nodes.
- For symmetry, we also provide the m-element vector of input nodes, constructed using allFin m which is the vector (i.e., array) containing all m elements of the set Fin m in sequence, i.e., 0, 1, . . . , m - 1.
- Edge is the setoid of hyperedges.
- eInfo maps each edge to a dependent tuple consisting of an arity k, a k-ary label, and a k-element vector of edge input nodes.
- eOut maps each edge to its output node, which cannot be an input node of the Jungle, and therefore has to be an Inner node. (The function arrow between setoids is optically not distinguishable from the general function type arrow, but is technically a different symbol. Since setoids cannot be used as types, no confusion can arise.)
- We derive the function eLabel that maps each edge e to its edge label. Since the arity of that label is not known in advance, the function eLabel returns a dependent pair consisting of the label arity k and a k-ary label.
- We also derive the function eIn that maps each edge e to the vector of input nodes of e; the type of this vector depends on the arity of e, which is the first component ($proj_1$) of the dependent tuple eLabel e.

```
record DHG₁ (m n : ℕ) : Set₁ where
  field Inner : Setoid zero zero
  Node = Fin.setoid m ⊎⊎ Inner
  field output : Vec ⌊ Node ⌋ n

  input : Vec ⌊ Node ⌋ m
  input = Vec.map inj₁ (allFin m)

  field Edge : Setoid zero zero
        eInfo : ⌊ Edge ⌋
              → Σ [k : ℕ] (Label k × Vec ⌊ Node ⌋ k)

        eOut : Edge ⟶ Inner

  eLabel : ⌊ Edge ⌋ → Σ [k : ℕ] Label k
  eLabel e = Product.map id proj₁ (eInfo e)

  eIn : (e : ⌊ Edge ⌋) → Vec ⌊ Node ⌋ (proj₁ (eLabel e))
  eIn = proj₂ ∘ proj₂ ∘ eInfo
```

```
record Jungle (m n : ℕ) : Set₁ where
  field Inner : Setoid zero zero
  Node = Fin.setoid m ⊎⊎ Inner
  field output : Vec ⌊ Node ⌋ n

  input : Vec ⌊ Node ⌋ m
  input = Vec.map inj₁ (allFin m)

  field Edge : Setoid zero zero
        eInfo : ⌊ Edge ⌋
              → Σ [k : ℕ] (Label k × Vec ⌊ Node ⌋ k)
        EOut : Inverse Edge Inner
  eOut : Edge ⟶ Inner
  eOut = Inverse.to EOut
  producer : Inner ⟶ Edge
  producer = Inverse.from EOut

  eLabel : ⌊ Edge ⌋ → Σ [k : ℕ] Label k
  eLabel e = Product.map id proj₁ (eInfo e)

  eIn : (e : ⌊ Edge ⌋) → Vec ⌊ Node ⌋ (proj₁ (eLabel e))
  eIn = proj₂ ∘ proj₂ ∘ eInfo
```

In this DHG_1 definition, eOut does not have to be surjective, which means that there may be "undefined nodes", and eOut also does not have to be injective, which means that there may be "join nodes" in the sense of [KAC06]. If bijectivity of eOut is desired, we can replace the setoid mapping with an inverse pair of mappings, and extract eOut and the producer mapping for inner nodes from that, as shown above to the right.

These jungles are isomorphic to conventional termgraphs, where inputs (as arguments) and labels are attached directly to inner nodes:

```
record TermGraph (m n : ℕ) : Set₁ where
  field Inner : Setoid zero zero
  Node = Fin.setoid m ⊎⊎ Inner
  field output : Vec ⌊ Node ⌋ n
  input : Vec ⌊ Node ⌋ m
  input = Vec.map inj₁ (allFin m)
  field label : ⌊ Inner ⌋ → Σ [k : ℕ] Label k
        args : (n : ⌊ Inner ⌋) → Vec ⌊ Node ⌋ (proj₁ (label n))
```

The following basic constructor functions are highly similar for DHG_1, Jungle, and TermGraph; we show them here for Jungle.

Using the one-element setoid ⊤ (with element tt), we can define primitive jungles consisting of a single hyperedge:

```
prim : {k : ℕ} → Label k → Jungle k 1
prim {k} f = record
  {Inner   = ⊤
  ; output = [inj₂ tt]
  ; Edge   = ⊤
  ; eInfo  = λ _ → (k, (f, Vec.map inj₁ (allFin k)))
  ; EOut   = Inverse.id
  }
```

For wiring graphs, we need empty sets (⊥) of edges and inner nodes:

```
wire : {m n : ℕ} → Vec (Fin m) n → Jungle m n
wire {m} {n} v = record
  {Inner   = ⊥
  ; output = Vec.map inj₁ v
  ; Edge   = ⊥
  ; eInfo  = E.⊥-elim
  ; EOut   = Inverse.id
  }
```

With this, we can easily construct the standard wiring graphs required for defining a gs-monoidal category (see Sect. 5) of Jungles:

```
idJungle : {m : ℕ} → Jungle m m
idJungle = wire (allFin _)
dupJungle : {m : ℕ} → Jungle m (m + m)
dupJungle {m} = wire (allFin m ⁺ allFin m)
termJungle : {m : ℕ} → Jungle m 0
termJungle = wire []
exchJungle : (m n : ℕ) → Jungle (m + n) (n + m)
exchJungle m n = wire (Vec.map (raise m) (allFin n) ⁺ Vec.map (inject+ n) (allFin m))
```

Separating the inner nodes from the inputs in particular has the advantage that for sequential composition, we can just use the disjoint union of the two Inner node sets:

```
seqJungle : {k m n : ℕ} → Jungle k m → Jungle m n → Jungle k n
seqJungle {k} {m} {n} g₁ g₂ = let
    open Jungle
    h₁ : ⌊ Node g₁ ⌋ → Fin k ⊎ (⌊ Inner g₁ ⌋ ⊎ ⌊ Inner g₂ ⌋)
    h₁ = Sum.map id inj₁
    h₂ : ⌊ Node g₂ ⌋ → Fin k ⊎ (⌊ Inner g₁ ⌋ ⊎ ⌊ Inner g₂ ⌋)
    h₂ = [(λ i → h₁ (Vec.lookup i (output g₁))), inj₂ ∘ inj₂]′
  in record
    {Inner = Inner g₁ ⊎⊎ Inner g₂
    ; output = Vec.map h₂ (output g₂)
    ; Edge = Edge g₁ ⊎⊎ Edge g₂
    ; eInfo = [productMap₂₂ (Vec.map h₁) ∘ eInfo g₁, productMap₂₂ (Vec.map h₂) ∘ eInfo g₂]′
    ; EOut = EOut g₁ ⊕⊕ EOut g₂
    }
```

Parallel composition works similarly; here the input positions need to be adapted.

```
parJungle : {m₁ n₁ m₂ n₂ : ℕ} → Jungle m₁ n₁ → Jungle m₂ n₂ → Jungle (m₁ + m₂) (n₁ + n₂)
parJungle {m₁} {n₁} {m₂} {n₂} g₁ g₂ = let
    open Jungle
    h₁ : ⌊ Node g₁ ⌋ → Fin (m₁ + m₂) ⊎ (⌊ Inner g₁ ⌋ ⊎ ⌊ Inner g₂ ⌋)
    h₁ = Sum.map (inject+ m₂) inj₁
    h₂ : ⌊ Node g₂ ⌋ → Fin (m₁ + m₂) ⊎ (⌊ Inner g₁ ⌋ ⊎ ⌊ Inner g₂ ⌋)
    h₂ = Sum.map (raise m₁) inj₂
  in record
    {Inner = Inner g₁ ⊎⊎ Inner g₂
    ; output = Vec.map h₁ (output g₁) ⁺ Vec.map h₂ (output g₂)
    ; Edge = Edge g₁ ⊎⊎ Edge g₂
    ; eInfo = [productMap₂₂ (Vec.map h₁) ∘ eInfo g₁, productMap₂₂ (Vec.map h₂) ∘ eInfo g₂]′
    ; EOut = EOut g₁ ⊕⊕ EOut g₂
    }
```

4 Typed Code Graphs

Coconut code graphs [KAC06] have types associated with nodes, and hyperedges may have not only multiple inputs, but also multiple outputs, to be able to model operations that yield multiple results; the typing of the input and output nodes needs to be compatible with the operations indicated by the edge labels.

For simplicity, we assume here a global set Type : Set of node types, and dispense with using setoids in this section. An edge label is now indexed by vectors of input and output types, so we assume Label : $\{m\ n : \mathbb{N}\} \rightarrow$ Vec Type m $\rightarrow$ Vec Type n $\rightarrow$ Set, and also define the dependent record type EdgeType for collecting these indices:

```
record EdgeType : Set where
  field inArity : ℕ
        outArity : ℕ
        inTypes : Vec Type inArity
        outTypes : Vec Type outArity
```

An edge label then is such an index collection together with a label drawn from the corresponding label set; the **open** declaration makes the EdgeType fields available for EdgeLabel elements as if this was a record extension:

```
record EdgeLabel : Set where
  field eType : EdgeType
        label : Label (EdgeType.inTypes eType) (EdgeType.outTypes eType)
  open EdgeType eType public
```

For typed term graphs, there are many different ways to deal with node typing, and for any given way, different views are useful in different contexts. We will keep a node typing function as a **field**, and derive from this an indexed view of typed nodes, using the following general construct: Given a set A and a typing function type for A, the Type-indexed set Typed A type associates with every type ty all elements of A that have type ty; formally, an element of Typed A type ty is a dependent pair consisting of an element a : A together with a proof that type a ≡ ty:

```
Typed : (A : Set) → (A → Type) → Type → Set
Typed A type ty = Σ [a : A] (type a ≡ ty)
```

Since the Agda standard library does not provide a variant of Vec where the element types may depend on their positions, we directly use dependently typed functions starting from these positions instead, producing "typed vectors" with elements type according to the argument type vector v:

```
TypedVec : (A : Set) → (A → Type) → {k : ℕ} → Vec Type k → Set
TypedVec A type {k} v = (i : Fin k) → Typed A type (Vec.lookup i v)
```

The EdgeInfo associated with each hyperedge then contains, besides an EdgeLabel, two such "typed node vectors", typed according to the label's typing information (for modularity, this definition is kept outside the code graph definition and parameterised with the type Nodes for "typed node vectors" to be supplied there):

```
record EdgeInfo (Nodes : {k : ℕ} → Vec Type k → Set) : Set where
  field eLab    : EdgeLabel
        eInput  : Nodes (EdgeLabel.inTypes eLab)
        eOutput : Nodes (EdgeLabel.outTypes eLab)
  open EdgeLabel eLab public
```

A CodeGraph is now defined roughly analogous to a Jungle, with the following differences worth pointing out:

- Code graphs can be considered as "generalised hyperedges", and therefore have an EdgeType derived from the CodeGraph type parameters. Keeping the current parameters eases the implementation of the categorical view, in comparison with using the EdgeType as a parameter instead.
- We only need to explicitly represent the typing of the inner nodes; from this we can derive the typing of all Nodes by looking up the typing of the input positions in inTypes.
- A TypedNode ty is a Node with type ty; an element of TypedNodes v is a "typed node vector" according to the type vector v.
- The CodeGraph field output and each individual edge interface use TypedNode "vectors".
- We can still provide lower-level interfaces to edges; we show functions that extract the edge label, edge input arity, and edge input Node vectors (discarding the type information), both dependently-typed and existentially-typed with respect to the vector length. (The corresponding functions eOut etc.are not shown.)

```
record CodeGraph {m n : ℕ} (inTypes : Vec Type m) (outTypes : Vec Type n) : Set₁ where
  cgType : EdgeType
  cgType = record {inArity   = m
                  ; outArity  = n
                  ; inTypes   = inTypes
                  ; outTypes  = outTypes}
  field Inner : Set
        iType : Inner → Type
  Node = Fin m ⊎ Inner
  nType : Node → Type
  nType = [(λ i → Vec.lookup i inTypes), iType] ′
  TypedNode : Type → Set
  TypedNode = Typed Node nType
  TypedNodes : {k : ℕ} → Vec Type k → Set
  TypedNodes = TypedVec Node nType
  field output : TypedNodes outTypes
  input : TypedNodes inTypes
  input = λ i → (inj₁ i, refl)
  field Edge : Set
        eInfo : Edge → EdgeInfo TypedNodes
  eLabel : Edge → EdgeLabel
  eLabel = EdgeInfo.eLab ∘ eInfo
  eInArity : Edge → ℕ
  eInArity = EdgeInfo.inArity ∘ eInfo
  eIn : (e : Edge) → Vec Node (eInArity e)
  eIn e = mkVec (proj₁ ∘ EdgeInfo.eInput (eInfo e))
  eIn′ : Edge → Σ [k : ℕ] (Vec Node k)
  eIn′ e = eInArity e, eIn e
```

Again, eOut is not guaranteed to reach all nodes, and, due to the possibility of multi-output operations, this cannot be amended by joining the Inner and Edge sets as in jungles. This and other degrees of generality contained in this definition can be useful for certain purposes, but also can be forbidden for other purposes by adding appropriate constraints.

We show the function for producing primitive one-edge code graphs:

```
prim : (l : EdgeLabel) → CodeGraph (EdgeLabel.inTypes l) (EdgeLabel.outTypes l)
prim l = record
  {Inner   = Fin (EdgeLabel.outArity l)
  ; output = λ i → (inj₂ i, refl)
  ; Edge   = ⊤
  ; eInfo  = λ _ → record {eLab     = l
                          ; eInput  = λ i → (inj₁ i, refl)
                          ; eOutput = λ i → (inj₂ i, refl)}}
```

While type-checking the three propositional equality proofs refl in here, Agda actually proves that the mentioned types are indeed equal: An Agda program can only produce CodeGraph values that are correctly typed, both on the external interface, and internally at each port of each edge.

5 GS-Monoidal Categories

Corradini and Gadducci proposed *gs-monoidal categories* for modelling acyclic term graphs [CG99]; extended discussion of how code graphs fit into this framework is contained in [KAC06]. Here we only present a quick summary, and tie this into the formalisation in Sect. 3.

In a category theory context, we write "$f : \mathscr{A} \to \mathscr{B}$" to declare that morphism f goes from object $\mathscr{A}$ to object $\mathscr{B}$, and use "$;$" as the associative binary *composition* operator; composition of two morphisms $f : \mathscr{A} \to \mathscr{B}$ and $g : \mathscr{B}' \to \mathscr{C}$ is defined iff $\mathscr{B} = \mathscr{B}'$, and then $(f \mathbin{;} g) : \mathscr{A} \to \mathscr{C}$. Furthermore, the identity morphism for object $\mathscr{A}$ is written $\mathbb{I}_{\mathscr{A}}$.

Jungle can be seen to define morphisms of an untyped term graph category where objects are natural numbers. (For CodeGraph, the collection of Objects is $\Sigma\,[\mathsf{k} : \mathbb{N}]\ (\mathsf{Vec\ Type\ k})$.)

In the Jungle category, a morphism from m to n is an element of Jungle m n, that is, a term graph with m input nodes and n output nodes. More precisely, such a morphism is an isomorphism class of jungles, since node and edge identities do not matter; we will define a Setoid where the Carrier is Jungle m n and equivalence proofs are Jungle isomorphisms.

Composition $F \mathbin{;} G$ "glues" together the output nodes of F with the respective input nodes of G, as we have implemented in seqJungle. The identity on n consists only of n input nodes which are also, in the same sequence, output nodes, and no edges, and is therefore constructed as a wiring graph:

```
idJungle : {m : ℕ} → Jungle m m
idJungle = wire (allFin _)
```

Definition 5.1 A *symmetric strict monoidal category* [ML71] consists of a category $\mathbf{C}_0$, a strictly associative monoidal bifunctor $\otimes$ with $\mathbb{1}$ as its strict unit, and a transformation $\mathbb{X}$ that associates with every two objects $\mathscr{A}$ and $\mathscr{B}$ an arrow $\mathbb{X}_{\mathscr{A},\mathscr{B}} : \mathscr{A} \otimes \mathscr{B} \to \mathscr{B} \otimes \mathscr{A}$ with:

$$(F \otimes G) \mathbin{;} \mathbb{X}_{\mathscr{C},\mathscr{D}} = \mathbb{X}_{\mathscr{A},\mathscr{B}} \mathbin{;} (G \otimes F)\ , \qquad \mathbb{X}_{\mathscr{A},\mathscr{B}} \mathbin{;} \mathbb{X}_{\mathscr{B},\mathscr{A}} = \mathbb{I}_{\mathscr{A}} \otimes \mathbb{I}_{\mathscr{B}}\ ,$$
$$\mathbb{X}_{\mathscr{A}\otimes\mathscr{B},\mathscr{C}} = (\mathbb{I}_{\mathscr{A}} \otimes \mathbb{X}_{\mathscr{B},\mathscr{C}}) \mathbin{;} (\mathbb{X}_{\mathscr{A},\mathscr{C}} \otimes \mathbb{I}_{\mathscr{B}})\ , \qquad \mathbb{X}_{\mathbb{1},\mathbb{1}} = \mathbb{I}_{\mathbb{1}}\ . \qquad \square$$

For Jungle, the unit object $\mathbb{1}$ is the natural number 0, and $\otimes$ on objects is addition. On morphisms, $\otimes$ forms the disjoint union of code graphs, concatenating the input and output node sequences, as implemented in parJungle. $\mathbb{X}_{m,n}$ differs from $\mathbb{I}_{m+n}$ only in the fact that the two parts of the output node sequence are swapped:

```
exchJungle : (m n : ℕ) → Jungle (m + n) (n + m)
exchJungle m n = wire (Vec.map (raise m) (allFin n) ⁺ Vec.map (inject+ n) (allFin m))
```

Definition 5.2 A *strict gs-monoidal category* is a symmetric strict monoidal category where in addition ! associates with every object $\mathscr{A}$ of $\mathbf{C}_0$ an arrow $!_{\mathscr{A}} : \mathscr{A} \to \mathbb{1}$, and ∇ associates with every object $\mathscr{A}$ of $\mathbf{C}_0$ an arrow $\nabla_{\mathscr{A}} : \mathscr{A} \to \mathscr{A} \otimes \mathscr{A}$, such that $\mathbb{I}_{\mathbb{1}} = !_{\mathbb{1}} = \nabla_{\mathbb{1}}$, and the following axioms hold:

$$\nabla_{\mathscr{A}} \mathbin{;} (\mathbb{I}_{\mathscr{A}} \otimes \nabla_{\mathscr{A}}) = \nabla_{\mathscr{A}} \mathbin{;} (\nabla_{\mathscr{A}} \otimes \mathbb{I}_{\mathscr{A}}) \qquad \nabla_{\mathscr{A}} \mathbin{;} \mathbb{X}_{\mathscr{A},\mathscr{A}} = \nabla_{\mathscr{A}} \qquad \nabla_{\mathscr{A}} \mathbin{;} (\mathbb{I}_{\mathscr{A}} \otimes !_{\mathscr{A}}) = \mathbb{I}_{\mathscr{A}}$$
$$\nabla_{\mathscr{A}\otimes\mathscr{B}} \mathbin{;} (\mathbb{I}_{\mathscr{A}} \otimes \mathbb{X}_{\mathscr{B},\mathscr{A}} \otimes \mathbb{I}_{\mathscr{B}}) = \nabla_{\mathscr{A}} \otimes \nabla_{\mathscr{B}} \qquad !_{\mathscr{A}\otimes\mathscr{B}} = !_{\mathscr{A}} \otimes !_{\mathscr{B}} \qquad \square$$

In Jungle, the "terminator" $!_n$ differs from $\mathbb{I}_n$ only in the fact that the output node sequence is empty.

```
termJungle : {n : ℕ} → Jungle n 0
termJungle = wire []
```

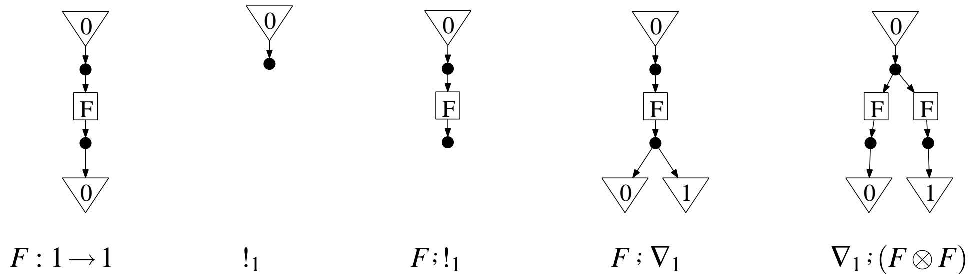
The "g" of "gs-monoidal" stands for "garbage": all edges of a term graph $G : m \to n$ are garbage in the term graph $G ; !_n$.

The duplicator ∇_n in Jungle differs from $\mathbb{I}_n$ only in the fact that the output node sequence is the concatenation of the input node sequence with itself:

```
dupJungle : {n : ℕ} → Jungle n (n + n)
dupJungle {n} = wire (allFin n ⁺ allFin n)
```

The "s" of "gs-monoidal" stands for "sharing": every input of $\nabla_k ; (F \otimes G)$ is shared by $F : k \to m$ and $G : k \to n$.

Code graphs (and term graphs) over a fixed edge label set form a gs-monoidal category, but not a *Cartesian* category, where in addition ! and ∇ are *natural* transformations, i.e., for all $F : \mathscr{A} \to \mathscr{B}$ we have $F ; !_{\mathscr{B}} = !_{\mathscr{A}}$ and $F ; \nabla_{\mathscr{B}} = \nabla_{\mathscr{A}} ; (F \otimes F)$. To see how these naturality conditions are violated by term graphs, the following five Jungles correspond to the expressions below them (we draw jungles and code graphs from the inputs on top to the outputs at the bottom, with numbered triangles marking input and output positions, and rectangles enclosing edge labels).



Formalising (symmetric gs-) monoidal categories in Agda is a straight-forward extension of the standard type-theoretic formalisation of category theory deriving essentially from Kanda's "effective categories" [Kan81]; this uses setoids of morphisms, but not of objects. This approach is also used by Huet and Saïbi [HS98, HS00] for their formalisation of category theory in Coq, and by Gonzalía [Gon06] for his formalisation of Freyd and Scedrov's allegory hierarchy [FS90] in Alf, a predecessor of Agda.

This approach also corresponds to the general practice in category theory to consider objects only up to isomorphism, not up to equality. However, the definition of strict monoidal categories runs counter to this approach, by assuming an object-level operation $(\otimes)$ satisfying non-trivial object-level equations. Therefore we directly formalise what MacLane calls "relaxed" monoidal categories, with natural isomorphisms $\alpha : \mathscr{A} \otimes (\mathscr{B} \otimes \mathscr{C}) \to (\mathscr{A} \otimes \mathscr{B}) \otimes \mathscr{C}$ and $\lambda : \mathbb{1} \otimes \mathscr{A} \to \mathscr{A}$ and $\rho : \mathscr{A} \otimes \mathbb{1} \to \mathscr{A}$.

This explicit approach also has advantages for moving between different levels of data nesting without requiring additional features; this is important for example for reasoning about the effect of SIMD operations together with SIMD vector manipulations on individual scalar values, which is necessary for verifying numerous high-performance "tricks", see e.g.[AK08].

6 Term Graphs as Let Constructs

The code graph representation of Sect. 4 essentially is a typed variant of the current internal representation of Coconut code graphs, but, as mentioned in the introduction, we essentially write Haskell definitions to initially create code graphs.

In lazy functional programming implemented by graph reduction, since at least KRC [Tur82], local definitions (via **let** or **where**) are understood to introduce *sharing*. In a mathematical context, [AK94] represents cyclic term graphs as systems of mutually recursive equations, and [MOW98] presents sharing in the call-by-need λ-calculus via **let**-expressions.

In the following, we present two formalisations of term graphs defined by non-recursive nested **let**-expressions. For the sake of readability, we restrict ourselves to untyped term graphs and single-output primitives.

With **let**-expressions, we automatically have to deal with the complications of bound variables, involving scoping, renaming to avoid variable clashes, etc. The Agda library NotSoFresh by Pouillard and Pottier [PP10] allows us to abstract from these concerns to a large degree, at the cost of following the discipline of their World-based programming interface. At the core of their approach, there are Worlds in which different variables are in scope; for a world α, the set of usable names is Name α. Introducing a new name happens via a "world extension link"; an element of $\alpha \leftharpoondown \beta$ is a *weak link* that provides a variable in β that might be shadowing one of the variables in α, while an element of $\alpha \longleftharpoondown\!\!\rightarrow \beta$ is a *strong link* that provides, in β, a variable that is *fresh* with respect to all variables in α.

For programming and in mathematics, we are used to working in a context of weak links, while symbol manipulation systems, including theorem provers and compilers, frequently disambiguate names so that they can work with strong links exclusively. To enable both settings, we will parameterise over these "world Extension relation" with a parameter E : World $\rightarrow$ World $\rightarrow$ Set.

We first present the type TG that formalises **let**-expressions with arbitrary nesting; this type is only a slight modification of the λ-term datatype Tm from [PP10].

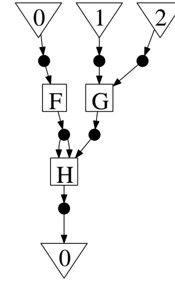
A value of type TG E α m n is, in the context of m input nodes and of a world α providing already existing inner nodes, a term graph "suffix" producing n output nodes:

- The input node at position i can be produced as an output node by Input i.
- An existing node x : Name α is produced as an output node by V x.
- The empty suffix is called ε.
- Given two suffixes t and u of output lengths n_1 and n_2, their union, with concatenated output lists, is t $\triangledown$ u. The symbol $\triangledown$ reads "fork", as in the fork algebras of [HFBV97]; it is related with the duplicator ∇ via the equation t $\triangledown$ u $= \nabla_m ; (t \otimes u)$.
- A primitive f can only be invoked while applying it to the outputs of a term graph suffix t and while at the same time creating a new node x in an expression of the shape Let x f t u, which, in more conventional notation, would read "**let** x = f (t) **in** u".
  If the primitive f expects k inputs, the argument term graph suffix t, which may not use the new name x because it is in the "old" world α, has to have k outputs.
  The term graph suffix u may use also the new name x, and its outputs will be the outputs of the "Let x f t u" expression.

```
data TG (E : World → World → Set) (α : World) (m : ℕ) : ℕ → Set where
  Input : (i : Fin m)     → TG E α m 1
  V     : (x : Name α) → TG E α m 1
  ε     : TG E α m 0
  _▽_   : {n₁ n₂ : ℕ} → TG E α m n₁ → TG E α m n₂ → TG E α m (n₁ + n₂)
  Let   :  {β : World} {k n : ℕ}
         → (x : E α β)                          -- let x
         → (f : Label k) → (t : TG E α m k)     -- = f (t)
         → (u : TG E β m n)                     -- in u
         → TG E α m n
```

Without additional support, defining term graphs using this interface is somewhat inconvenient — the following assumes a unary label F, a binary label G, and a ternary label H:

```
TG0 : Label 1 → Label 2 → Label 3 → TG _↼−_ ∅ 3 1
TG0 F G H = let f0 = fresh∅                  -- a strong link
                x0 = FreshPack.weakOf f0     -- weak view of f0
                n0 = FreshPack.nameOf f0     -- Name of f0
  in Let x0 H
    (Let x0 F (Input zero) (V n0 ▽ V n0)
      ▽
      Let x0 G (Input (suc zero) ▽ Input (suc (suc zero))) (V n0)
    )
    (V n0)
```



Using slightly more conventional notation, this corresponds to the following, relatively readable version, with “i” prefixing inputs and “n” prefixing node names:

```
let n0 = H ((let n0 = F (i0) in (n0 ▽ n0))
             ▽
             (let n0 = G (i1 ▽ i2) in n0)
           ) in n0
```

Either by adding more notational support, or by defining a separate input language, this can provide an interface that comes reasonably close to Haskell-style programming.

The real point of the definition of TG however is that it not only provides an input language, but also a representation of term graphs that can be manipulated and transformed by programs. For example, we can turn a TG with name shadowing (i.e., using weak links) into one with strong links by replacing all node names with fresh names relative to their respective worlds:

```
strengthenTG : {α α′ : World} → Fresh α′ → CEnv (Name α′) α
             → {m n : ℕ} → TG _↼−_ α m n → TG _↼−→_ α′ m n
strengthenTG _ _ ε          = ε
strengthenTG fr Γ (t ▽ u) = (strengthenTG fr Γ t) ▽ (strengthenTG fr Γ u)
strengthenTG _ _ (Input i) = Input i
strengthenTG fr Γ (V x)    = V (lookupCEnv Γ x)
strengthenTG fr Γ (Let x f t u)
  = let Γ′ = mapCEnv importWith Γ, x ↦ nameOf
     in Let strongOf f (strengthenTG fr Γ t) (strengthenTG nextOf Γ′ u)
    where open FreshPack fr
```

Parallel composition is also easy to program, using fork after embedding, respectively shifting, the inputs:

```
parTG : {E : _} {α : _} {m₁ n₁ m₂ n₂ : ℕ}
      → TG E α m₁ n₁ → TG E α m₂ n₂ → TG E α (m₁ + m₂) (n₁ + n₂)
parTG {E} {α} {m₁} {n₁} g₁ {m₂} {n₂} g₂ = extendTG m₂ g₁ ▽ shiftTG m₁ g₂
```

Sequential composition is much harder to implement directly, since the output nodes of the first argument may have been defined in separate worlds and combined with fork, and now need to be brought into a common world, which in general requires renaming and restructuring. A convenient “canonical form” for such **let**-expressions has no Let at argument positions, and no Let below fork, and therefore degenerates into a sequence of Let declarations each binding a new node to the application of some primitive

to existing nodes. When dealing with any kind of canonical forms, especially in a dependently-typed setting, it is frequently worth while declaring this as a separate datatype so that it becomes easier to exploit its properties. For this canonical form of TG, we introduce a separate datatype with additional restructuring below.

7 Term Graphs with Sequential Node Declaration

According to our explanation of TG term graphs, $\triangledown$ with ε obviously forms a monoid, but the monoid laws do not come for free in TG. Moving to the Vec container type instead provides us with the monoid laws in the standard library, and makes for a more canonical representation. With this change, and with strictly linearised node declaration, the term graph TG0 shown above could be written in a somewhat conventional notation as follows (without fully specifying the number of inputs):

```
let n0 = F i0
let n1 = G i1 i2
let n2 = H n0 n0 n1
in [n2]
```

We introduce the type Arg for individual nodes, either existing inner nodes, or input positions, and a type synonym Args for their vectors:

```
data Arg α (m : ℕ) : Set where
  Input : (i : Fin m) → Arg α m
  V     : (x : Name α) → Arg α m
Args α m n = Vec (Arg α m) n
```

The datatype TG′ has the same reading as TG, but a simpler structure:

- If all nodes have been declared, Output as assembles the vector of output nodes.
- Let x f v u, which, in more conventional notation, would read “**let** x = f (v) **in** u”, binds a new node x to an edge labelled f with input nodes v, and makes x visible in the remaining term graph suffix u.

```
data TG′ E α (m : ℕ) : ℕ → Set where
  Output : {n : ℕ} → Args α m n → TG′ E α m n
  Let    :   {β : World} {k n : ℕ}
         →  (x : E α β)                       -- let x
         →  (f : Label k) (v : Args α m k)    -- = f (v)
         →  (u : TG′ E β m n)                 -- in u
         →  TG′ E α m n
```

We first show that primitive and wiring graphs are easily programmed:

```
prim : {k : ℕ} → Label k → TG′ _↼→_ ø k 1
prim {k} f = Let strongOf f (Vec.map Input (Vec.allFin k)) (Output [V nameOf])
  where open FreshPack freshø
wire : {k n : ℕ} {E : _} {α : World} → Vec (Fin k) n → TG′ E α k n
wire v = Output (Vec.map Input v)
idWire : {k : ℕ} {E : _} {α : World} → TG′ E α k k
idWire {k} = wire (Vec.allFin k)
dup : {k : ℕ} {E : _} {α : World} → TG′ E α k (k + k)
```

```
dup {k} = wire (Vec.allFin k ++ Vec.allFin k)
term : {k : ℕ} {E : _} {α : World} → TG′ E α k 0
term = wire []
```

With these definitions, we can reconstruct the term graph TG0 from above via the gs-monoidal interface, with sequential composition seqTG′ and parallel composition parTG′ defined below:

```
tg0 = seqTG′ (parTG′ (seqTG′ (prim F) dup) (prim G)) (prim H)
```

For the analogous function to strengthenTG, which replaces each link x in a Let construct with a fresh link, we present an easy generalisation to serve dual purposes:

- Starting from weak links, strengthenTG′ {_↼−_} id is proper strengthening;
- starting from strong links, strengthenTG′ {_↼−→_} StrongPack.weakOf is renaming with fresh names with respect to the new world α'.

```
strengthenTG′ : {E : _} → (E ⇒ _↼−_)
              → {α α′ : World} → Fresh α′ → CEnv (Name α′) α
              → {m n : ℕ} → TG′ E α m n → TG′ _↼−→_ α′ m n
strengthenTG′ weak fr Γ (Output as) = Output (mapVarArgs (lookupCEnv Γ) as)
strengthenTG′ weak fr Γ (Let x f as u)
  = let Γ′ = mapCEnv importWith Γ, weak x ↦ nameOf
    in Let strongOf f (mapVarArgs (lookupCEnv Γ) as) (strengthenTG′ weak nextOf Γ′ u)
  where open FreshPack fr
```

Both sequential and parallel composition are implemented by inserting the material of one graph between the innermost Let and the Output of the other graph. We define a general helper function for this purpose:

```
inLet′ :  {α β : World} → (s : α *↼−→ β) → Fresh β → {m n n′ : ℕ}
      → ({γ : World} → (s′ : α *↼−→ γ) → Fresh γ
          → Args γ m n → TG′ _↼−→_ γ m n′)
      →      TG′ _↼−→_ β m n → TG′ _↼−→_ β m n′
inLet′ s fr F (Let x f t u) = Let x f t (inLet′ (s ▷ x) fr′ F u) where fr′ = StrongPack.nextOf x
inLet′ s fr F (Output as) = F s fr as
```

We first implement fork, which walks the only primitively available fresh link freshø past all the Lets of g_1, uses the resulting fresh link fr to rename g_2, and afterwards adapts the output list as_1 of g_1 to the inner world of the renamed g_2, so that the two output lists can be concatenated:

```
forkTG′ : {m n₁ n₂ : ℕ}
        → TG′ _↼−→_ ø m n₁
        → TG′ _↼−→_ ø m n₂
        → TG′ _↼−→_ ø m (n₁ + n₂)
forkTG′ {m} {n₁} {n₂} g₁ g₂ = inLet′ ε freshø
  (λ {γ} s′ fr as₁ → inLet′ ε fr
    (λ s″ _ as₂ → Output (mapVarArgs (import⊆ (*↼−→-⊆ s″)) as₁ ++ as₂))
    (strengthenTG′ {_↼−→_} StrongPack.weakOf fr emptyCEnv g₂)
  ) g₁
```

The implementation of parallel composition then relies on fork in the same way as that for TG:

```
parTG′ : {m₁ n₁ : ℕ}  → TG′ _↼−→_ ø m₁ n₁
       → {m₂ n₂ : ℕ} → TG′ _↼−→_ ø m₂ n₂
```

```
                          → TG′ _↼—→_ ø (m₁ + m₂) (n₁ + n₂)
parTG′ {m₁} g₁ {m₂} g₂ = forkTG′ (extendTG′ m₂ g₁) (shiftTG′ m₁ g₂)
```

Sequential composition follows the same pattern as forkTG′, and first traverses the declarations of g_1, which are preserved, but uses the helper function mapArgsTG′ to properly replace any occurrence of inputs in argument and output lists of the renamed g_2 with the corresponding output nodes of g_1, after adapting them to the respective nested world.

```
seqTG′ : {k m n : ℕ}
       → TG′ _↼—→_ ø k m
       → TG′ _↼—→_ ø m n
       → TG′ _↼—→_ ø k n
seqTG′ g₁ g₂ = inLet′ ε freshø
  (λ {γ} s′ fr as₁ → mapArgsTG′ ε
    (λ s″ as → seqArgs (mapVarArgs (import⊆ (*↼—→-⊆ s″)) as₁) as)
    (strengthenTG′ {_↼—→_} StrongPack.weakOf fr emptyCEnv g₂)
  ) g₁
```

Finally, it is also reasonably easy to convert a TG′ term graph into a Jungle with Fin k as Inner node set and as Edge set, where k is the number of Let declarations.

8 Conclusion and Outlook

Formalising mathematical definitions of term graphs and their operations in Agda is a remarkably straightforward exercise, and, due to the dependent typing of Agda, also carries over to typed term graphs much more easily than in the more restricted type systems of Haskell or higher-order logic.

The remarkable abstract interface to variable binding provided by Pouillard and Pottier's NotSoFresh Agda library [PP10] also makes name-binding representations of term graphs conveniently accessible to mechanised reasoning and programmed manipulation. Typing is easily added to our TG and TG′ datatypes — the original Tm datatype provided as NotSoFresh example includes typing, but we omitted it here to improve readability.

Implementing additional term graph operations, manipulations, and conversion functions, and proving the algebraic properties of the term graph operations is ongoing work.

Future work will strive to base code-graph based optimised-code generation algorithms for the Coconut project [AK09a] on our Agda formalisations of code graphs, with a fully verifying tool chain as ultimate goal.

References

[AK94] Zena M. Ariola & Jan Willem Klop (1994): *Cyclic Lambda Graph Rewriting*. In: *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Paris, France, pp. 416–425.

[AK08] Christopher Kumar Anand & Wolfram Kahl (2008): *Code Graph Transformations for Verifiable Generation of SIMD-Parallel Assembly Code*. In Andy Schürr, Manfred Nagl & Albert Zündorf, editors: *Applications of Graph Transformations with Industrial Relevance, AGTIVE 2007*. *LNCS* 5088, pp. 217–232, doi:10.1007/978-3-540-89020-1.

[AK09a] Christopher K. Anand & Wolfram Kahl (2009): *An Optimized Cell BE Special Function Library Generated by Coconut*. *IEEE Transactions on Computers* 58(8), pp. 1126–1138, doi:10.1109/TC.2008.223.

[AK09b] Christopher K. Anand & Wolfram Kahl (2009): *Synthesizing and Verifying Multicore Parallelism in Categories of Nested Code Graphs*. In Michael Alexander & William Gardner, editors: *Process Algebra for Parallel and Distributed Processing*, chapter 1. *CRC Computational Science Series* 2, Chapman & Hall, pp. 3–45.

[CG99] Andrea Corradini & Fabio Gadducci (1999): *An Algebraic Presentation of Term Graphs, via GS-Monoidal Categories*. *Applied Categorical Structures* 7(4), pp. 299–331.

[CR93] Andrea Corradini & Francesca Rossi (1993): *Hyperedge replacement jungle rewriting for term-rewriting systems and logic programming*. In B. Courcelle & G. Rozenberg, editors: *Selected Papers of the International Workshop on Computing by Graph Transformation, Bordeaux, France, March 21–23, 1991*. Elsevier, pp. 7–48, doi:10.1016/0304-3975(93)90063-Y. Theoretical Computer Science **109** (1–2).

[FS90] Peter J. Freyd & Andre Scedrov (1990): *Categories, Allegories*. *North-Holland Mathematical Library* 39, North-Holland, Amsterdam.

[Gon06] Carlos Gonzalía (2006): *Relations in Dependent Type Theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg University. Technical Report No. 14D.

[HFBV97] Armando Haeberer, Marcelo Frias, Gabriel Baum & Paulo Veloso (1997): *Fork Algebras*. In Chris Brink, Wolfram Kahl & Gunther Schmidt, editors: *Relational Methods in Computer Science*, chapter 4. Advances in Computing Science, Springer, Wien, New York, pp. 54–69.

[HP91] Berthold Hoffmann & Detlef Plump (1991): *Implementing Term Rewriting by Jungle Evaluation*. *Informatique théorique et applications/Theoretical Informatics and Applications* 25(5), pp. 445–472.

[HS98] Gérard Huet & Amokrane Saïbi (1998): *Constructive Category Theory*. In: *Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg*. doi:10.1.1.39.4193.

[HS00] Gérard Huet & Amokrane Saïbi (2000): *Constructive Category Theory*. In Gordon D. Plotkin, Colin Stirling & Mads Tofte, editors: *Proof, language, and interaction: Essays in honour of Robin Milner*. Foundations Of Computing Series, MIT Press, pp. 239–275.

[KAC06] Wolfram Kahl, Christopher Kumar Anand & Jacques Carette (2006): *Control-Flow Semantics for Assembly-Level Data-Flow Graphs*. In Wendy McCaull, Michael Winter & Ivo Düntsch, editors: *8th Intl. Seminar on Relational Methods in Computer Science, RelMiCS 8, Feb. 2005*. *LNCS* 3929, Springer, pp. 147–160.

[Kan81] Akira Kanda (1981): *Constructive Category Theory (No. 1)*. In Jozef Gruska & Michal Chytil, editors: *Mathematical Foundations of Computer Science, MFCS '81*. *LNCS* 118, Springer, pp. 563–577, doi:10.1007/3-540-10856-4_125.

[ML71] Saunders Mac Lane (1971): *Categories for the Working Mathematician*. Springer-Verlag.

[MOW98] John Maraist, Martin Oderski & Philip Wadler (1998): *The Call-by-Need Lambda Calculus*. *J. Functional Programming* 8(3), pp. 275–317.

[Nor07] Ulf Norell (2007): *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology. Available at http://www.cs.chalmers.se/~ulfn/papers/thesis.html.

[PP10] Nicolas Pouillard & François Pottier (2010): *A fresh look at programming with names and binders*. In: *ICFP 2010, Intl. Conf. on Functional Programming*. ACM, New York, NY, USA, pp. 217–228, doi:10.1145/1863543.1863575. Available at http://nicolaspouillard.fr/publis/pouillard-pottier-fresh-look-agda-2010/.

[Tur82] David A. Turner (1982): *Recursion Equations as a Programming Language*. In J. Darlington, editor: *Functional Programming and its Applications: An Advanced Course*. Cambridge Univ. Press, pp. 1–27.

PORGY: Strategy-Driven Interactive Transformation of Graphs*

Oana Andrei[†]
oana.andrei@glasgow.ac.uk

Maribel Fernández [‡]
maribel.fernandez@kcl.ac.uk

Hélène Kirchner [§]
helene.kirchner@inria.fr

Guy Melançon [§]
guy.melancon@inria.fr

Olivier Namet [‡]
olivier.namet@kcl.ac.uk

Bruno Pinaud [§]
bruno.pinaud@inria.fr

This paper investigates the use of graph rewriting systems as a modelling tool, and advocates the embedding of such systems in an interactive environment. One important application domain is the modelling of biochemical systems, where states are represented by port graphs and the dynamics is driven by rules and strategies. A graph rewriting tool's capability to interactively explore the features of the rewriting system provides useful insights into possible behaviours of the model and its properties. We describe PORGY, a visual and interactive tool we have developed to model complex systems using port graphs and port graph rewrite rules guided by strategies, and to navigate in the derivation history. We demonstrate via examples some functionalities provided by PORGY.

1 Introduction

Graphical formalisms are widely used for describing complex structures in a visual and intuitive way, *e.g.*, UML diagrams, representation of proofs, microprocessor design, XML documents, communication networks, data and control flow, neural networks, biological systems, *etc.* Graph transformation (or graph rewriting) is a fundamental concept in concurrency and computational models, as well in modelling the dynamics of complex system in general. From a theoretical point of view, graph rewriting has solid logic, algebraic and categorical foundations [10, 25], and from a practical point of view, graph transformations have many applications in specification, programming, and simulation [12, 13]. Several graph-transformation languages and tools have been developed, such as PROGRES [26], AGG [14], Fujaba [22], GROOVE [24], GrGen [19] and GP [23], only to mention a few.

When the graphs are large or growing via transformations, or when the number of transformation rules is important, being able to directly interact with the rewriting system becomes crucial to understand the changes in the graph structure. From a naïve point of view, the output of a graph rewriting system is a dynamic graph: a sequence of graphs obtained through a series of topological modifications (addition/deletion of nodes/edges). However, the study of a rewriting system is actually much more complex. Reasoning about the system's properties actually involves testing various rewriting scenarios, backtracking to a previously computed graph, possibly updating rules, etc. In this paper, we address these issues. Our main contribution is a solution to these problems via a strategy-driven interactive environment for the specification of graph rewriting systems: PORGY.

*Partially supported by INRIA's *Associate team program* (see `http://gravite.labri.fr/?Projects_%2F_Grants:Porgy`) and the French National Research Agency project EVIDEN (ANR 2010 JCJC 0201 01).

[†]School of Computing Science, University of Glasgow, Glasgow G12 8RZ, UK
[‡]King's College London, Department of Informatics, Strand, London WC2R 2LS, UK
[§]INRIA Bordeaux Sud-Ouest, Université Bordeaux 1, CNRS UMR 5800, LaBRI, 33405 Talence Cedex, France

Rachid Echahed (Ed.): 6th International Workshop
on Computing with Terms and Graphs (TERMGRAPH 2011)
EPTCS 48, 2011, pp. 54–68, doi:10.4204/EPTCS.48.7

© Andrei *et al.*
This work is licensed under the
Creative Commons Attribution License.

Our work emerged from the necessity to assemble different views on the rewriting system and relevant interactions. First of all, an appropriate formalism and associated structures are needed to represent and manipulate graph rewriting systems and rewriting sequences. Our approach is based on the use of port graphs and port graph rewriting rules [1, 4]. We support our claim on the generality of this concept by using port graph transformations for modelling biochemical interactions that take part in the regulation of cell proliferation and transformation. This case study illustrates the highly dynamic context and some interesting challenges for graph visualisation provided by biochemical systems. PORGY provides support for the initial task of defining a set of graph rewriting rules, and the graph representing the initial state of the system (the "initial model" in PORGY's terminology), using a visual editor.

Other crucial issues concern when and where rules are applied. To address this problem, PORGY provides a strategy language to constrain the rewriting derivations, generalising the control structures used in PROGRES, GP and rewrite-based programming languages such as Stratego and ELAN. In particular, the strategy language includes control structures that facilitate the implementation of graph traversal algorithms, thanks to the explicit definition of "positions" in a graph, where rules can be applied (we refer the reader to [17] for examples of graph programs in PORGY, and to [16] for the formal semantics of the strategy language).

Rewriting derivations can also be visualised, and used in an interactive way, using PORGY's interface. Designing a graph transformation system is often a complex task, and the analysis and debugging of the system involves exploring how rules operate on graphs, analysing sequences of transformations, backtracking and changing earlier decisions. For this purpose, PORGY's visual environment offers a view on the rewriting history and ways to select time points in the history where to backtrack.

The organisation of this paper is as follows. In Section 2, we recall the concept of port graph and port graph transformations, and use this formalism to model the scaffold protein AKAP in the process of mediating a crosstalk between the cAMP and the Raf-1/MEK/ERK signalling pathway. In Section 3 we describe PORGY's strategy language. In Section 4, we focus on the visualisation and interaction features designed to better understand the model and its behaviour. Section 5 discusses related work and compares PORGY to the similar approaches we are aware of. Finally, Section 6 concludes and describes future work.

2 Port Graph Rewriting

The basic constructs of the PORGY environment are the concept of port graph and the port graph rewriting relation that we recall in this section.

Informally, a *port graph* is a graph where nodes have explicit connection points called *ports* for the edges and a *p-signature* is a mapping which associates a set of port names to a node name.

Definition 1 (P-Signature[1, 5]) *Let $\nabla_{\mathscr{N}}$ be a set of constant node names, $\nabla_{\mathscr{P}}$ a set of constant port names, and $\mathscr{X}_{\mathscr{P}}$ and $\mathscr{X}_{\mathscr{N}}$ two sets of port name variables and node name variables, respectively. A* p-signature *is a pair of sets of names $\nabla^{\mathscr{X}} = \langle \nabla_{\mathscr{N}} \cup \mathscr{X}_{\mathscr{N}}, \nabla_{\mathscr{P}} \cup \mathscr{X}_{\mathscr{P}} \rangle$ such that each node name $N \in \nabla_{\mathscr{N}} \cup \mathscr{X}_{\mathscr{N}}$ comes with a finite set of port names $Interface(N) \subseteq \nabla_{\mathscr{P}} \cup \mathscr{X}_{\mathscr{P}}$.*

Definition 2 (Port Graph [1, 5]) *Given a fixed p-signature $\nabla^{\mathscr{X}}$, a* labelled port graph over *$\nabla^{\mathscr{X}}$ is a tuple $G = \langle V_G, E_G, lv_G, le_G \rangle$ where:*

- *V_G is a finite set of nodes;*
- *E_G is a finite multiset of edges,*
  $E_G \subseteq \{\langle (v_1, p_1), (v_2, p_2) \rangle \mid v_i \in V_G, p_i \in Interface(lv_G(v_i))\}$;

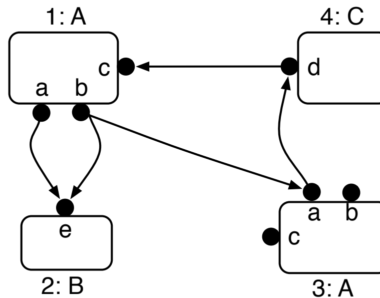


Figure 1: A port graph over the p-signature $\nabla = \langle \{A,B,C\},\{a,b,c,d,e\}$ with $Interface(A) = \{a,b,c\}$, $Interface(B) = \{e\}$, $Interface(C) = \{d\}$, and $1,2,3,4$ the node identifiers.

- $lv_G : V_G \rightarrow \nabla_{\mathscr{N}} \cup \mathscr{X}_{\mathscr{N}}$ *is a node-labelling function associating to a node a name and a set of ports according to the p-signature;*
- $le_G : E_G \rightarrow (\nabla_{\mathscr{P}} \cup \mathscr{X}_{\mathscr{P}}) \times (\nabla_{\mathscr{P}} \cup \mathscr{X}_{\mathscr{P}})$ *is an edge-labelling function associating to an edge the pair of port names where it connects to the nodes, i.e.* $le_G(\langle (v_1, p_1), (v_2, p_2) \rangle) = (p_1, p_2)$.

A simple example of a port graph is depicted in Fig. 1. Port graphs were first identified as an abstract view of proteins and molecular complexes resulting from the protein interactions in a biochemical setting. From a biochemical perspective, a protein is characterised by a collection of functional domains also called *sites*. Two proteins may interact by binding on complementary sites. Then a protein with binding sites is graphically modelled by a node with ports, and a bond between two proteins by an edge in a port graph. A port can also just carry some information from a set of attributes instead of being a connection or binding point. For instance, two proteins may interact just by changing the attribute information of a site, from phosphorylated (P or “+”) to unphosphorylated (U or “−”) and vice versa. This view is at the origin of several formalisms for biology, such as the κ-calculus [11] and BioNetGen [15]; see also [8, 3], where graph models have been designed to simulate a chemical reactor using rule-based systems and strategies.

Example 3 (AKAP **model: species as port graphs)** *We illustrate port graphs and port graph rewriting for modelling a biochemical network in a simplified model of the scaffold-mediated crosstalk between the cyclic adenosine monophosphate (cAMP) and the Raf/MEK/ERK pathway [2]. This interaction has an important role in the regulation of cell proliferation, transformation and survival. Let us call this simplified biochemical model the* AKAP model*. The chemical species occurring in the* AKAP *model are: scaffold protein* AKAP *with three binding sites; nucleotide* cAMP *with one binding site; protein* PKA *with one site for binding to the scaffold and one site for binding to* cAMP*; enzyme* PDE8 *with one site for binding to the scaffold and one phosphorylation site;* Raf-1 *protein with two sites: one for binding to the scaffold and the other for phosphorylation; signal protein* SA*. The* AKAP *scaffold protein binds the three molecules* PKA*,* PDE8 *and* Raf-1*. Although these molecules are not all proteins, we model them as nodes with ports in port graphs by abstracting the signal transfer as binding actions between their ports. In Fig. 2 we show a port graph representation of a state of the* AKAP *model.*

Definition 4 (Port graph rewrite rule [1, 5]) *A* port graph rewrite rule $L \Rightarrow R$ *is a port graph consisting of two port graphs L and R over the same p-signature and one special node* $\Rightarrow$*, called* arrow node *connecting them. L and R are called the* left- *and* right-*hand side respectively. The arrow node has*

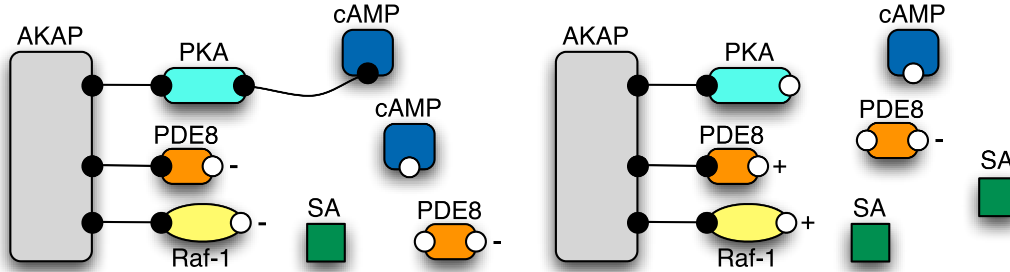


Figure 2: Port graph representation for the chemical species in the AKAP model where a port with a plus sign stands for a phosphorylated site and a minus sign for unphosphorylated. In this graphical representation we omitted the port names as there is no risk of confusion.

the following characteristics: for each port p in L, to which corresponds a non-empty set of ports $\{p_1,\ldots,p_n\}$ *in R, the arrow node has a unique port r and the incident directed edges* (p,r) *and* (r,p_i)*, for all* $i=1,\ldots,n$*; all ports from L that are deleted in R are connected to the* black hole *port of the arrow node.*

A port graph rewrite system $\mathscr{R}$ *is a finite set of port graph rewrite rules.*

Intuitively, the arrow node together with its adjacent edges embed the correspondence between elements of L and elements of R. When the correspondence between ports in the left- and right-hand side of the rule is obvious we omit the ports and edges involving the arrow node.

Let G and H be two port graphs defined over the same p-signature. A *port graph morphism* $f: G \to H$ relates the elements of G to elements of H by preserving sources and targets of edges, constant node names and associated port name sets up to a variable renaming. We say that G and H are *homomorphic* when any two ports are connected in G if and only if their f-images are connected in H.

We now informally recall the *port graph rewriting relation* from [1]. Let $L \Rightarrow R$ be a port graph rewrite rule and G a port graph such that there is an injective port graph morphism g from L to G. By replacing the subgraph $g(L)$ of G by $g(R)$ and connecting it with the rest of the graph as indicated by the interface of the rule, we obtain a port graph G' representing a result of *one-step rewriting* of G using the rule $L \Rightarrow R$, written $G \to_{L\Rightarrow R} G'$. Several injective morphisms g from L to G may exist leading to possibly different rewriting results. These are built as solutions of a *matching* problem from L to a subgraph of G. If there is no such injective morphism, we say that G is *irreducible* with respect to $L \Rightarrow R$. Given a set $\mathscr{R}$ of rules, a port graph G *rewrites* to G', denoted by $G \to_{\mathscr{R}} G'$, if there is a port graph rewrite rule r in $\mathscr{R}$ such that $G \to_r G'$. This induces a transitive relation on port graphs. Each *rule application* is a rewriting step and a *derivation* is a sequence of rewriting steps, also called a computation. A port graph is in *normal form* if no rule can be applied on it. Rewriting is intrinsically non-deterministic since several subgraphs of a port graph may be rewritten under a set of rules.

Example 5 (AKAP **model: reactions as port graph rewrite rules)** *We consider the following four chemical reactions:*

($\mathbf{r_1}$) cAMP *activates* PKA *through binding;*

($\mathbf{r_2}$) *active* PKA *phosphorylates* PDE8 *and* Raf-1 *on the same scaffold and becomes inactive;*

($\mathbf{r_3}$) *phosphorylated* PDE8 *degrades free* cAMP *and becomes unphosphorylated as well as* Raf-1 *at which point* Raf-1 *sends an activation signal* SA*;*

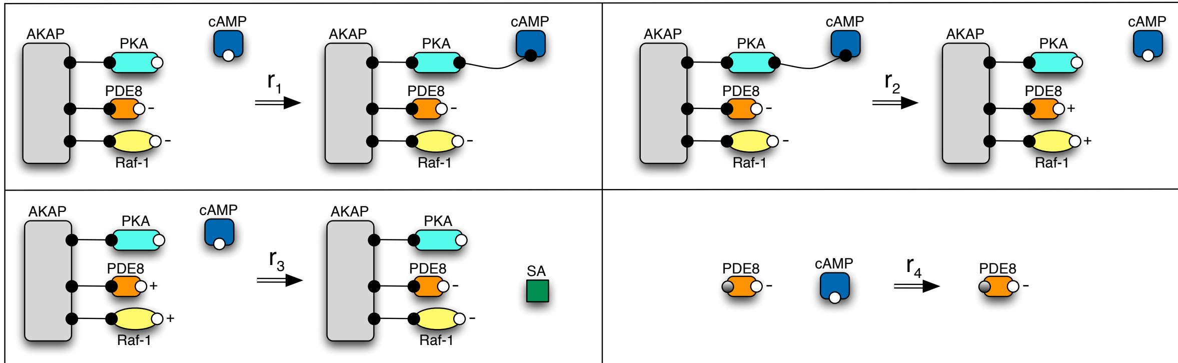


Figure 3: Port graph representation for the biochemical reactions in the AKAP model.

($\mathbf{r_4}$) *unphosphorylated* PDE8 *degrades free* cAMP *(we represent the binding site for* PDE8 *in a shade of grey meaning that unphosphorylated* PDE8 *has the same behaviour in the presence of a* cAMP *molecule if it is bound or not to the* AKAP *protein).*

These reactions are graphically represented as port graph rewrite rules in Fig. 3. A phosphorylation action activates a site (or port) and we represent this graphically by changing the attribute "−" of a site (or port) into "+"; an unphosphorylation event does the opposite. In Fig. 3 we only give a schematic representation of the rule, whereas in Fig. 4 we detail the port graph rewrite rule correspoding to reaction **r4***: the arrow node is included together with two ports for showing correspondence between the two sites of* PDE8 *on both sides of the rule, as well as the black hole (bh) port which is responsable for deleting the only one site of* cAMP *and, in consequence, the entire molecule* cAMP.

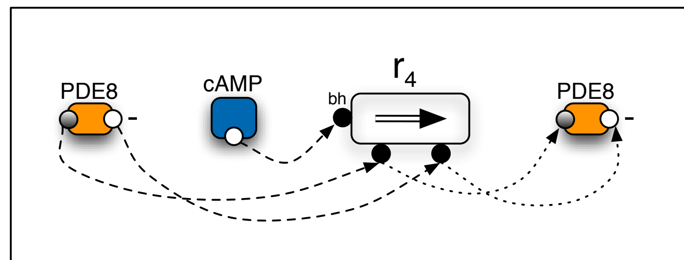


Figure 4: Reaction **r4** as a port graph rule with explicit arrow node and *black hole (bh)* port.

An overview of AKAP*'s behaviour is as follows. Free* cAMP *activates* PKA *by binding to its free port. When* PKA *becomes active, it phosphorylates* PDE8 *and* Raf-1 *if all bound to the same* AKAP *protein. The amount of* PDE8 *may be greater than the amount of* AKAP *which means that the* PDE8 *proteins not on the scaffold protein will never be phosphorylated.* PDE8 *in either state degrades* cAMP*, but phosphorylated* PDE8 *degrades approximately three times more* cAMP *than unphosphorylated* PDE8 *does. This information is expressed as the following statement:* rule $\mathbf{r_3}$ is three times faster than $\mathbf{r_4}$ *in a stochastic context, or* rule $\mathbf{r_3}$ has 75% chances of occurrence whereas $\mathbf{r_4}$ only 25% chances. *As* Raf-1 *becomes unphosphorylated, the pathway* Raf-1/MEK/ERK *is activated and the signalling cascade begins; we model this process by the creation of one signal protein* SA.

Time courses from the laboratory experiments suggest the presence of a pulsating behaviour in the system. The pulsations ensure that the state of the Raf-1/MEK/ERK *pathway alternates between active and inactive (note: very long periods of activity or inactivity may increase the risk of disease). This expected property of the model translates into alternating short periods of time where the number of the signal* SA *increases (active pathway) and times where it remains constant (inactive pathway).*

At this point, a comparison of the structure of port graphs with other graphical formalism needs to be included. A port graph can be seen as a multigraph, thus inheriting the theoretical results available for graph transformations, such as confluence, or parallel independency of rewriting steps; for more details see [1, 5]. A graph structure similar to port graphs but with a more restrictive definition of rewrite rules and rewriting relation can be found in the context of reserved graph grammars (RGG) [28] – a class of context-sensitive graph grammars well suited for describing and efficiently implementing *diagrammatic* visual programming languages. The graphs used in RGGs are based on two-level nodes: a node is made up of vertices and vertices can be interconnected; one vertex is distinct for representing the node itself, called *super vertex*. The application domain of RGGs imposes the graph rewrite rules to be locally confluent and vertices in a RGG rule can be marked which means at matching the incidence degree has to match as well, ensuring that no dangling edges will occur after a graph transformation.

In [1] a suitable (strategic) rewriting relation and an abstract calculus have been defined on port graphs. This formalism is powerful enough to model biochemical applications and generation of biochemical networks, as well as self-management properties of autonomic systems [4, 5].

In the study of biochemical networks, the PORGY environment allows controlled network generation and in silico experiments with different priorities or probabilities of rule applications. We expect to be able to answer typical questions asked by biologists, such as, in the previous example, find the derivations leading to a constant alternation between active and inactive pathway.

Another main application domain for port graph rewriting and the PORGY environment, besides the biochemical networks, is the Interaction Nets [21, 17]. These applications, amongst others, motivate our choice of port graphs as a basis for modelling complex systems and for prototyping their evolution (port graph rewriting provides executable specifications). In order to support the various tasks involved in the study of a port graph rewriting system, we suggest to combine different points of view on the system:

- to explore a derivation tree with all possible derivations,
- to perform on-demand reduction using a strategy language which permits to restrict or guide the reductions,
- to track the reduction throughout the whole tree,
- to navigate in the tree, for instance, backtracking and exploring different branches.

These capabilities are developed in the remaining of this paper.

3 Strategies

In this section, we present a strategy language that controls the application of rules from a graph rewriting system [17]. This language was designed to cover a wide range of graphs, but with the main application domains of port graphs in mind, such as biochemical networks and Interaction Nets.

In the PORGY environment, the user creates a *graph-program* from a strategy over a graph rewriting system, a graph and a position. In the following, we first review the strategy language and then give examples of graph-programs.

Let G, L, R be graphs, P a position, ρ a property, m, n integers, $p_i \in [0,1]$.

$$
\begin{array}{rcll}
\textbf{(Focusing)} & F & := & \texttt{CrtGraph} \mid \texttt{CrtPos} \mid \texttt{AllSuc} \mid \texttt{OneSuc} \mid \texttt{NextSuc} \\
& & \mid & \texttt{Property}(\rho, F) \mid F \cup F \mid F \cap F \mid \overline{F} \mid F \setminus F \\
\textbf{(Applications)} & A & := & \texttt{Id} \mid \texttt{Fail} \mid (L \Rightarrow R)_P \mid (A \parallel A) \mid (A \parallel\!| A) \mid A^{\parallel(m,n)} \\
\textbf{(Strategies)} & S & := & F \mid A \mid S;S \mid S+S \mid \texttt{ppick}(S_1/p_1, \ldots, S_i/p_i) \\
& & \mid & \texttt{while}\ (S)\ \texttt{do}(S)\ \texttt{min}(m)\ \texttt{max}(n) \\
& & \mid & \texttt{if}\ (S)\ \texttt{then}(S)\ \texttt{else}(S) \mid \texttt{Empty(F)} \mid \langle S \rangle \mid \texttt{SetPos}(F)
\end{array}
$$

Figure 5: Syntax of the strategy language.

In term rewriting, by default the rewriting is performed at the *root* position of a term. Then one may use term traversal strategies such as top-down, bottom-up, *etc.*, in order to apply rules or strategies at particular positions in a term. Graphs generalise the tree-like structure of terms and lose the notion of root as absolute position. In order to overcome this, we consider a generalisation of the notion of position, and define *located graphs*.

Definition 6 (Located graph) *Let G be a graph and P a subgraph of G representing the* position *where a strategy is to be applied. Then the structure $G[P]$ is called a* located graph*.*

Thus, any subgraph of a graph G can be used as a position P to apply a rewrite rule. The notion of position allows us to *focus* on specific parts of a graph under study, in order to apply rewrite rules. Considering a located graph $G[P]$ to be rewritten, we require that the homomorphic image of the left-hand side of the rule has a non-empty intersection with the subgraph P. A simple and intuitive example is to define P as consisting of a specific node in G, and in this case, only the rules having this node in the homomorphic image of their left-hand sides are allowed to be applied. The application of a strategy on a graph G may change the subgraph P in G: our strategy language includes operators not only to select rules and the position where the rules are to be applied, but also to change the focus of the rewriting engine along a derivation.

Figure 5 shows the grammars F, A and S for generating expressions to define positions, to apply rules and to define general strategies. A *focusing expression* defines a position subgraph, whereas an *application* may change both the graph and the positions. A *strategy* embeds the previous constructs and combines them using sequential composition, iteration and conditionals. In the following we describe informally the semantics of each operator in the language and give examples.

Focusing. The expressions generated by F allow us to focus on different parts of the graph during the rewriting process. These constructs are functions from located graphs to port graphs: an expression F applies to a located graph $G[P]$ to produce a new graph P'. They are used in strategy expressions to change the position P where rules apply and to specify different types of graph traversals. `CrtPos` returns the position in the current located graph. `AllSuc` returns immediate successors of all nodes in the current position, where an immediate successor of a node v is a node u with a port connected to a port of v. `OneSuc` looks for all the immediate successors of all nodes in the current position and picks one of those non-deterministically. `NextSuc` computes successors of nodes in the current position using a function that designates a specific port for each node. $\texttt{Property}(\rho, Y)$ is a filtering construct, that returns a subgraph of G containing only the nodes from Y that satisfy the decidable property ρ (Y would

generally be P or G, but can be any graph returned by an expression F). ρ typically tests a property on nodes allowing us, for example, to select the subgraph of red nodes. The set theory operators *union*, *intersection*, *complement* and *substraction* apply to positions, considering that those graphs are sets of nodes and edges.

Applications. The application of `Id` on a located graph never fails and leaves the graph unchanged whereas `Fail` always fails (it leaves the graph unchanged and returns failure). $(L \Rightarrow R)_Q$ where Q is a subgraph of R, represents the application of the rule $L \Rightarrow R$ at the current position P in a located graph $G[P]$ where the morphism g is chosen such that $g(L) \cap P$ is not empty; the current position becomes $(P \setminus g(L)) \cup g(Q)$. If more than one application is possible, one is non-deterministically selected from the set of possible results. The other results are then used if there is a backtrack, that is, if a failure arises. $A \parallel A'$ represents simultaneous application of A and A' on *disjoint* (i.e. not connected) subgraphs of G and returns `Id` only if both applications are possible and `Fail` otherwise. $A \,|||\, A'$ is a weaker version of $A \parallel A'$ as it returns `Id` if at least one application of A or A' is possible. $A^{\|(m,n)}$ applies A simultaneously a minimum of m and a maximum of n times. If the minimum is not satisfied then `Fail` is returned and `Id` otherwise. If n is a negative integer then no maximum is considered. In the current implementation, these concurrent applications are done for rules only, *i.e.* for an elementary kind of strategies. Extending these constructions to full strategies needs further exploration.

Strategies. The expression $S;S'$ represents sequential application of S followed by S', and $S+S'$ applies whatever S or S' that returns `Id`: if both fail then `Fail` is returned and if both are successful then one of them is picked non-deterministically. When probabilities $p_1, \ldots, p_n \in [0,1]$ are associated to strategies $S_1, \ldots, S_n$ such that $p_1 + \ldots + p_n = 1$, the strategy $\texttt{ppick}(S_1/p_1, \ldots, S_n/p_n)$ non-deterministically picks one of the strategies for application, according to the given probabilities. For iterations, we have expressions of the form $\texttt{while}\ (S)\ \texttt{do}(S')\ \texttt{min}(m)\ \texttt{max}(n)$ which keep on sequentially applying S' for as long as the expression S rewrites to `Id`; if the minimum of m successful applications of S' is not satisfied then it returns `Fail` or else `Id` is returned. Similar to $A^{\|(m,n)}$, setting n to a negative integer eliminates the maximum. The strategy $\texttt{if}\ (S)\ \texttt{then}(S')\ \texttt{else}(S'')$ checks if the application of S to a located graph $G[P]$ returns `Id` in which case S' is applied to $G[P]$ otherwise S'' is applied. S is only tested on $G[P]$ and does not actually change the located graph. `Empty` returns `Id` if the current position is empty and `Fail` otherwise. This can be used for instance inside the condition of an `if` or `while`, to check if the application of the strategy makes the current position empty or not, instead of checking if the strategy itself can be applied. The strategy $\langle S \rangle$ applies S and considers S as one atomic rewriting step in the derivation tree. This is useful to abstract several reduction steps as one for visualisation purposes. $\texttt{SetPos}(P)$ changes the current position to a new position P.

Definition 7 (Graph-program) *A* graph-program *is a pair* $[S_{\mathscr{R}}, G[P]]$ *where* $S_{\mathscr{R}}$ *is a strategy expression built over a graph rewriting system* $\mathscr{R}$ *and* $G[P]$ *a located graph. The result of the execution of a terminating graph-program is another graph-program of the form* $[\texttt{Id}, G'[P']]$ *or* $[\texttt{Fail}, G'[P']]$ *such that* $G'[P']$ *has been obtained by the application of* $S_{\mathscr{R}}$ *on* $G[P]$.

The semantics of the strategy constructors defined by the grammars in Fig. 5 has been formally defined in [16] using rewrite rules that reduce a graph-program $[S, G[P]]$.

The notion of graph-program defined above is very general, and the language allows programmers to define high-level algorithms in a variety of application domains. For examples of programs developed using this language, we refer the reader to [17]. Below we describe an example that exploits the features of the language to simulate the behaviour of a biochemical system.

Example 8 (Strategy for the AKAP model) *In the AKAP model introduced in Sect. 2 the initial port graph G_0 to be rewritten consists of: 200 unbound* cAMP *molecules; 10 structures built upon an* AKAP *protein binding an inactive* PKA*, an unphosphorylated* PDE8 *and an unphosphorylated* Raf-1*; and 3 unphosphorylated* PDE8 *proteins not bound to an* AKAP *scaffold protein.*

From the lab experiments, the biologists concluded that phosphorylated PDE8 *degrades three times more* cAMP *than the unphosphorylated* PDE8*, in other words reaction rule r_3 is three times faster than reaction rule r_4 in a stochastic setting. We model this behaviour in a probabilistic setting via the strategy construct* `ppick` *which applies the rule r_3 with probability 0.75 and the rule r_4 with probability 0.25. We remark that r_1 and r_4 have a critical pair, therefore we use the strategy* `ppick` *with probability 0.5 for each of the rules; the same reasoning goes for r_2 and r_4 in the presence of a free* cAMP *molecule. In order to generate the biochemical network, we repeat the applications of all rules according to their application probabilities until we reach a normal form. Therefore the strategy for the* AKAP *model has the following form:*

$$S_{\mathsf{AKAP}} = \texttt{repeat}_*(\texttt{ppick}(r1/0.5, r4/0.5),\ \texttt{ppick}(r2/0.5, r4/0.5),\ \texttt{ppick}(r3/0.75, r4/0.25))$$

where $\texttt{repeat}_*(S)$ *is syntactic sugar for* $\texttt{while}(S)\ \texttt{do}(S)\ \texttt{min}(1)\ \texttt{max}(-1)$*. We show the result of the application of S_{AKAP} on G_0 in Sect. 4 as well as a procedure to analyse the results by counting the number of signal proteins* SA *during the execution.*

4 Working with the PORGY platform

One of the goals of PORGY is to allow the user to interact and experiment with a port graph rewriting system in a visual and interactive way. Ideally, a visual environment should offer different views on each component of the rewriting system: the current graph being rewritten, the derivation tree and the rules as shows the overview (for the AKAP model) in Figure 6. The application of rules is performed based on an ad hoc matching algorithm finding instances of left-hand sides of rules in a port graph. Our matching algorithm is based on the work of Ullman [27] and Cordella *et al.* [9]. A detailed discussion of our matching algorithm is out of scope here.

As the figure shows, the normal form of G_0 is found, after the strategy S_{AKAP} has been successfully applied with S_{AKAP} and G_0 defined in Example 8. Due to its design, the strategy terminates on a failure because no more rules apply. The failure is made explicit by showing a red node as part of the trace tree (the graph shown on the left pane). We make use of a quotient graph to embed the states of G into nodes of the trace tree (nodes G327 to G332). Thus, when scanning graphs along a branch, one can read the evolution of the graph being rewritten and explore the effect and properties of a strategy. A local and more detailed view allows a closer examination of a particular state of G (graph G332 in the lower right pane).

Typically, the user may be interested in plotting the evolution of a parameter computed out of each intermediary state. For example, going back to the AKAP model (Example 5) the behaviour of the SA protein, as predicted by the biologists, can be examined by plotting the curve of the evolution of the number of SA protein throughout the rewriting process (Figure 7).

The interactive features of PORGY simplify the study of the rewriting system partly due to a synchronisation between the different views. For example, selecting points on the plot view (Fig. 7) triggers the selection of the corresponding nodes in the trace tree. Such a mechanism obviously helps to track properties of the output graph along the rewriting process. Alternate strategies can be applied from any current or past state (see Fig. 8).

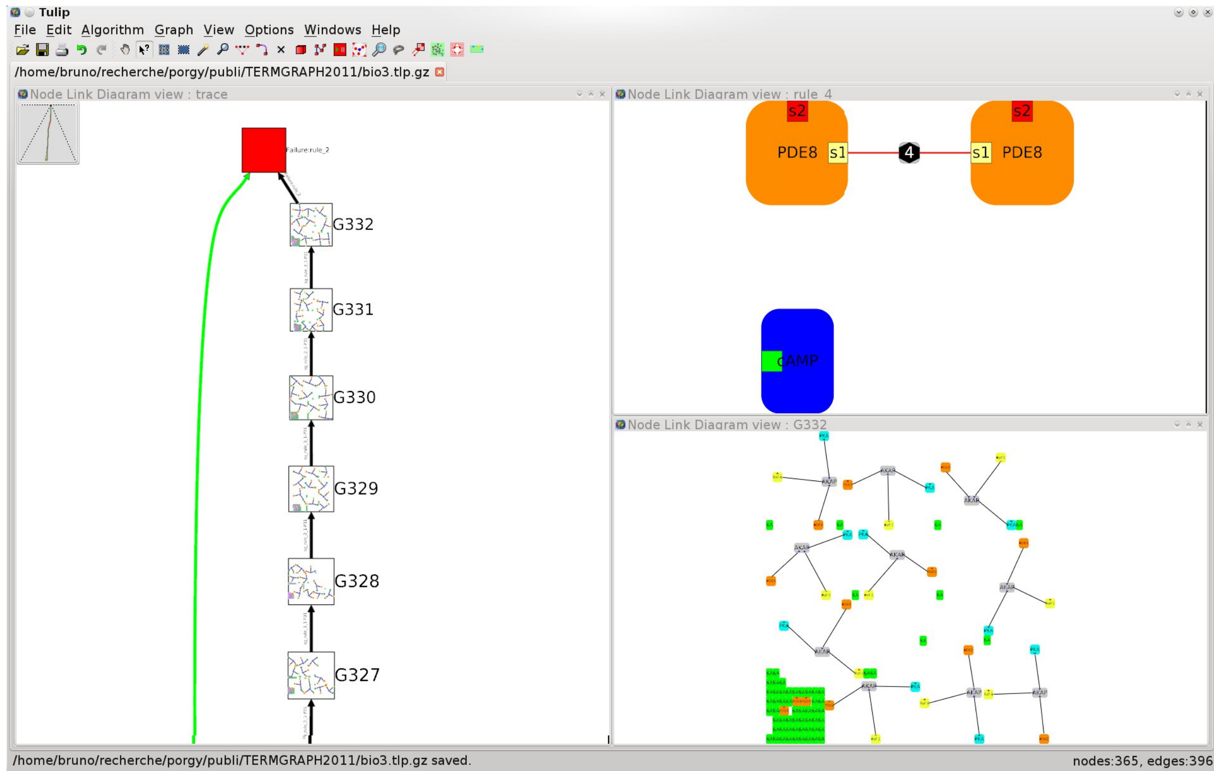


Figure 6: Overview of the PORGY environment. The graph at the bottom right ($G332$) corresponds to the normal form of G_0 after application of the strategy S_{AKAP} (see Example 8). The left panel shows part of a derivation tree, with nodes of the tree containing the last intermediate states of G. The top right panel corresponds to the rule $\mathbf{r_4}$.

The system we are currently developing is built on top of the Graph Visualisation framework Tulip [6][1]. Specific plugins have been designed and developed to implement the various interaction required to simulate and study port graph rewriting systems. Because Tulip does not handle port graph directly, each port node is built from several standard nodes, while implementation details exploiting Tulip's built-in graphical features have been kept hidden to the user.

5 Related Work

There are several tools available for editing graphs, of which some allow users to model graph transformations. In this section we review the ones that are similar in scope with PORGY.

GROOVE [24] is centered around the use of simple graphs for modelling the design-time, compile-time, and run-time structure of object-oriented systems. The GROOVE tool set includes an editor for creating graph production rules, a simulator for visually computing the graph transformations induced by a set of graph production rules, a generator for automatically exploring state spaces, a model checker for analysing the resulting graph transformation systems and an imaging tool for converting graphs to images. Visualisation is not its main objective, and after each rewrite step the user must update the

[1] See also `http://www.tulip-software.org`.

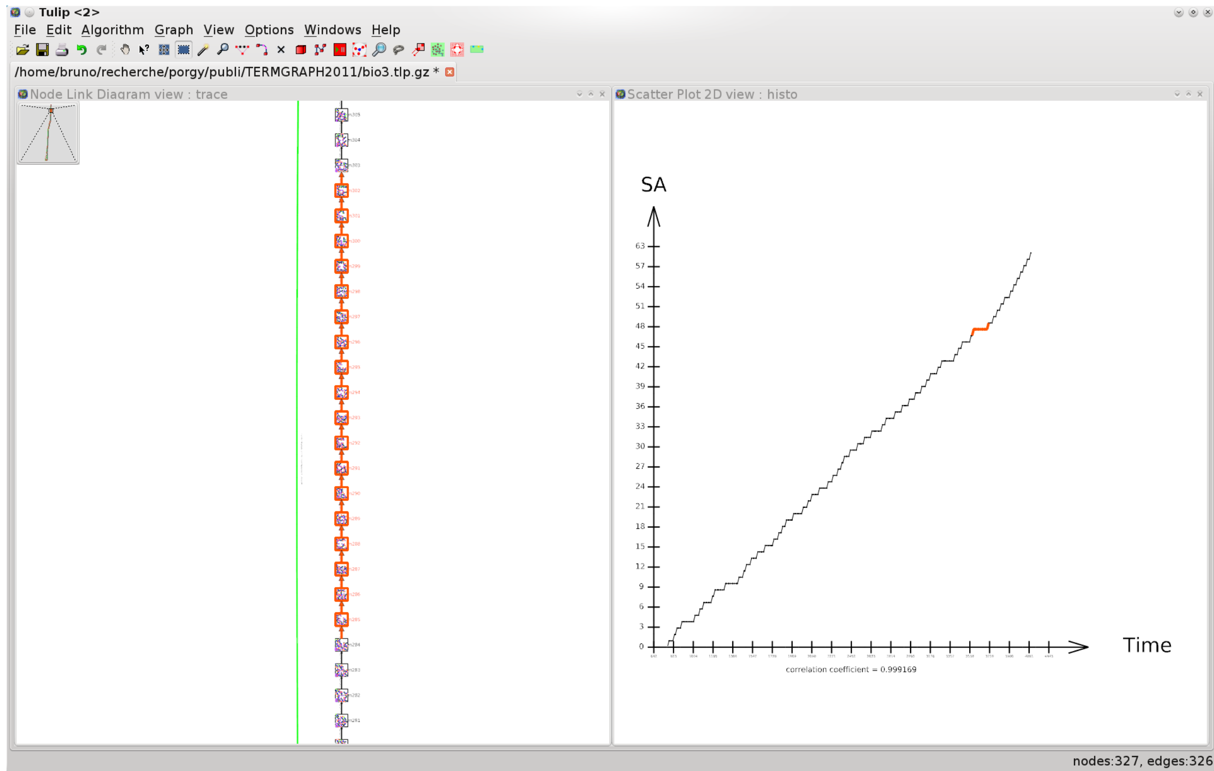


Figure 7: Evolution of the number of signal protein SA when applying the strategy presented in Example 8 – until a graph in normal form is found.

layout of the graph by hand. The model transformations and the operational semantics are based on graph transformations. GROOVE permits to control the application of rules, via a control language with sequence, loop, random choice, conditional and simple (non recursive) function calls. These are similar to PORGY's constructs, but the main difference is that GROOVE's language does not include the notion of position, it is not possible to specify a position for the application of rules within the language. Tracing is possible through state space exploration.

Fujaba [22] Tool Suite is an Open Source CASE tool providing developers with support for model-based software engineering and re-engineering. It combines UML class diagrams, UML activity diagrams, and a graph transformation language and offers a formal, visual specification language that can be used to completely specify the structure and behaviour of a software system under development. Graphs and rules are used to generate Java code. Fujaba has a basic strategy language, including conditionals, sequence and method calls. There is no parallelism, and again one of the main differences with PORGY is that Fujaba does not include a notion of position to guide the rule application.

AGG [14] is a rule-based visual language supporting an algebraic approach to graph transformation, and is aimed at specifying and implementing applications with complex graph-structured data. AGG may be used as a general purpose graph transformation engine in high-level JAVA applications employing graph transformation methods. Rule application can be controlled by defining layers and then iterating through and across these layers. Again, the position can not be specified and there is no control on the search for redexes.

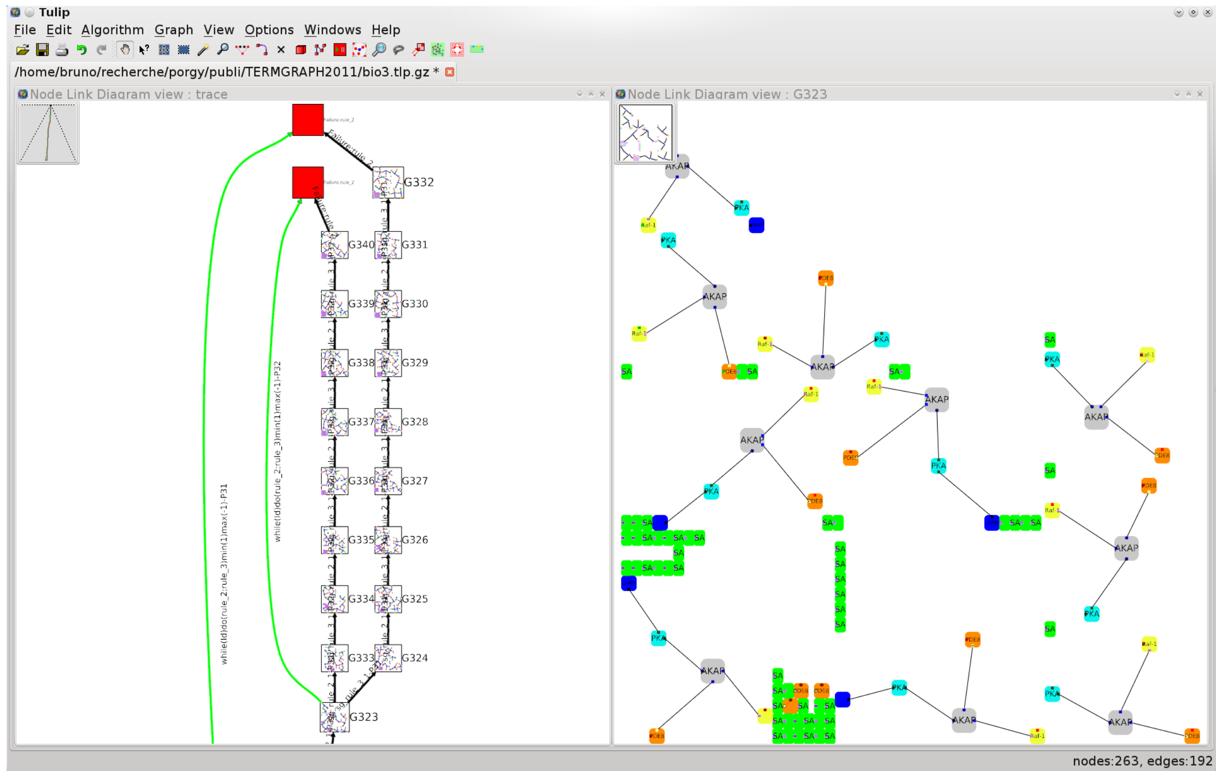


Figure 8: The green edges in the derivation tree (the graph on the left and the green edges are the edges on the left of this graph) represent the start and end of previously applied strategies. The other edges of the derivation tree represent the application of a single rewriting rule between two states of G.

PROGRES [26] project works on the theoretical foundations as well as the practical implementation of an executable specification language based on graph rewriting systems (graph grammars). It combines EER-like and UML-like class diagrams for the definition of complex object structures with graph rewrite rules for the definition of operations on these structures. PROGRES allows users to define the way rules are applied (it includes non-deterministic constructs, sequence, conditional and looping) but it does not allow users to specify the position where the rule is applied. It is a very expressive language and also includes a tracing functionality through backtracking.

GrGen.NET [19] is a programming tool for graph transformation designed to ease the transformation of complex graph structured data as required in model transformation, computer linguistics, or modern compiler construction, for example. It is comparable to other programming tools like parser generators which ease the task of formal language recognition. GrGen.Net has a rule application language with constructs for sequential, logical and iterative application.

GP [23] is a rule-based, non-deterministic programming language. Programs are defined by sets of graph rewriting rules and a textual expression that describes the way in which rules should be applied to a given graph. The simplest expression is a set of rules, and this means that any of the rules can be applied to rewrite the graph. The language has three main control constructs: sequence, repetition and conditional (if-then-else), and it has been shown to be complete. It uses a built-in Prolog-like backtracking technique: if at some point no rule can be applied, it backtracks to the nearest point where there was a choice of redex (users cannot easily handle the derivation tree or change the backtracking algorithm).

GReAT [7] is a tool for building model transformation tools. First, one has to specify the metamodels of the input and target models (using UML style class diagrams) and give rules to specify the transformation. Rules are pairs of typed, attributed graphs. Then, the pattern-matching algorithm always starts from specific nodes called "pivot nodes". Rule execution is sequenced and there are conditional and looping structures.

PORGY and its strategy language allow a higher expressive power with its focus on position. Strategies are not limited to picking random applications but can travel through the graph in a dynamic and strategic manner to apply rules and sub-strategies. PORGY has also a strong focus on visualisation and scale, thanks to the TULIP back-end which can handle large graphs with millions of elements and comes with powerful visualisation and interaction features. Some of Tulip's built-in functionalities, such as selecting some nodes in a visualisation for highlighting the equivalent nodes in the trace tree, give the user an immediate visual feedback (Fig 7).

6 Conclusion and Future Work

The PORGY environment provides an interactive visual environment for graph transformations. In this paper we presented the main concepts underlying PORGY: port graph rewriting and strategies for graph rewriting.

The PORGY environment is yet under development. A first implementation of the strategy language is available but needs further improvement. From the visualisation point of view, we are working on enhancing the algorithms for drawing rules and models.

Verification and debugging tools for avoiding conflicting rules or non termination for instance are also planned in the future. Moreover we will address other application domains: for instance linguistics analysis [18] shares some of the features of biological networks and we expect to be able to handle linguistic models in PORGY without much difficulty. PORGY already provides information about the number of matching solutions for the application of a rule. Based on this information we plan to extend a port graph rewriting system with a stochastic semantics [20] which is very useful for developing biochemical models.

References

[1] Oana Andrei (2008): *A Rewriting Calculus for Graphs: Applications to Biology and Autonomous Systems*. Ph.D. thesis, Institut National Polytechnique de Lorraine. Available at `http://tel.archives-ouvertes.fr/tel-00337558/fr/`.

[2] Oana Andrei & Muffy Calder (2010): *A Model and Analysis of the* AKAP *Scaffold*. In: *Proc. of CS2Bio'10*. To appear in Electronic Notes in Theoretical Computer Science.

[3] Oana Andrei, Liliana Ibanescu & Hélène Kirchner (2006): *Non-intrusive Formal Methods and Strategic Rewriting for a Chemical Application.* In Kokichi Futatsugi, Jean-Pierre Jouannaud & José Meseguer, editors: *Algebra, Meaning, and Computation*. *Lecture Notes in Computer Science* 4060, Springer, pp. 194–215.

[4] Oana Andrei & Hélène Kirchner (2009): *A Higher-Order Graph Calculus for Autonomic Computing*. In M. Lipshteyn, V.E. Levit & R.M. McConnell, editors: *Graph Theory, Computational Intelligence and Thought*. *Lecture Notes in Computer Science* 5420, Springer, pp. 15–26.

[5] Oana Andrei & Hélène Kirchner (2009): *A Port Graph Calculus for Autonomic Computing and Invariant Verification*. In: *Proceedings of the Fifth International Workshop on Computing with Terms and Graphs (TERMGRAPH 2009)*. *Electronic Notes in Theoretical Computer Science* 253, pp. 17–38.

[6] David Auber (2003): *Tulip – A huge graph visualization framework*. In P. Mutzel & M. Jünger, editors: Graph Drawing Software. Mathematics and Visualization Series, Springer Verlag, pp. 105–126.

[7] Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk & Gabor Karsai (2006): *The Graph Rewriting and Transformation Language: GReAT*. ECEASST 1.

[8] Olivier Bournez, Guy-Marie Côme, Valérie Conraud, Hélène Kirchner & Liliana Ibanescu (2003): *A Rule-Based Approach for Automated Generation of Kinetic Chemical Mechanisms*. In Robert Nieuwenhuis, editor: RTA. Lecture Notes in Computer Science 2706, Springer, pp. 30–45.

[9] L. P. Cordella, P. Foggia, C. Sansone & M. Vento (2004): *A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs*. IEEE Trans. on Pattern Analysis and Machine Intelligence 26(10), pp. 1367–1372.

[10] Bruno Courcelle (1990): *Graph Rewriting: An Algebraic and Logic Approach*. In J. van Leeuwen, editor: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics. Elsevier Science Publishers and MIT Press, pp. 193–242.

[11] Vincent Danos & Cosimo Laneve (2004): *Formal Molecular Biology*. Theoretical Computer Science 325(1), pp. 69–110.

[12] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors (1997): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*. World Scientific.

[13] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari & Grzegorz Rozenberg, editors (1997): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 3: Concurrency, Parallelism, and Distribution*. World Scientific.

[14] Claudia Ermel, Michael Rudolf & Gabriele Taentzer (1997): *The AGG Approach: Language and Environment*. In: [12]. World Scientific, pp. 551–603.

[15] James R. Faeder, Michael L. Blinov & William S. Hlavacek (2009): *Rule-Based Modeling of Biochemical Systems with BioNetGen*. In Ivan V. Maly, editor: Systems Biology. Methods in Molecular Biology, Humana Press, pp. 133–168.

[16] Maribel Fernández, Hélène Kirchner & Olivier Namet (2010): *A strategy language for graph rewriting*. Submitted, see `http://www.oliviernamet.co.uk/Publications.html`.

[17] Maribel Fernández & Olivier Namet (2010): *Strategic Programming on Graph Rewriting Systems*. In: Proc. of the 1st International Workshop on Strategies in Rewriting, Proving, and Programming.

[18] Chris Fox, Maribel Fernández & Shalom Lappin (2008): *Lambda Calculus, Type Theory, and Natural Language II*. J. Log. Comput. 18(2), p. 203.

[19] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack & Adam Szalkowski (2006): *GrGen: A Fast SPO-Based Graph Rewriting Tool*. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro & Grzegorz Rozenberg, editors: ICGT. Lecture Notes in Computer Science 4178, Springer, pp. 383–397.

[20] Jean Krivine, Robin Milner & Angelo Troina (2008): *Stochastic Bigraphs*. Electr. Notes Theor. Comput. Sci. 218, pp. 73–96.

[21] Yves Lafont (1990): *Interaction Nets*. In: Proc. of the 17th ACM Symposium on Principles of Programming Languages (POPL'90). ACM Press, pp. 95–108.

[22] Ulrich Nickel, Jörg Niere & Albert Zündorf (2000): *The FUJABA environment*. In: ICSE. pp. 742–745.

[23] Detlef Plump (2009): *The Graph Programming Language GP*. In Symeon Bozapalidis & George Rahonis, editors: CAI. Lecture Notes in Computer Science 5725, Springer, pp. 99–122.

[24] Arend Rensink (2003): *The GROOVE Simulator: A Tool for State Space Generation*. In John L. Pfaltz, Manfred Nagl & Boris Böhlen, editors: AGTIVE. Lecture Notes in Computer Science 3062, Springer, pp. 479–485.

[25] Grzegorz Rozenberg, editor (1997): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific.

[26] Andy Schürr, Andreas J. Winter & Albert Zündorf (1997): *The PROGRES Approach: Language and Environment.* In: *[12]*. World Scientific, pp. 479–546.

[27] J. R. Ullmann (1976): *An Algorithm for Subgraph Isomorphism*. *JACM* 23(1), pp. 31–42.

[28] Daqian Zhang & Kang Zhang (1997): *Reserved Graph Grammar: A Specification Tool for Diagrammatic VPLs*. In: *Proc. of VL'97*. IEEE Computer Society, p. 284.

A new graphical calculus of proofs

Sandra Alves
DCC - Faculty of Science & LIACC
University of Porto
R. do Campo Alegre 1021/55,
4169-007, Porto, Portugal

Maribel Fernández
Department of Informatics
King's College London
Strand, London, WC2R 2LS U.K

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

We offer a simple graphical representation for proofs of intuitionistic logic, which is inspired by proof nets and interaction nets (two formalisms originating in linear logic). This graphical calculus of proofs inherits good features from each, but is not constrained by them. By the Curry-Howard isomorphism, the representation applies equally to the lambda calculus, offering an alternative diagrammatic representation of functional computations.
Keywords: Intuitionistic logic, lambda-calculus, visual representation, proof theory

1 Introduction

There are many different ways to write proofs in a given logic, for instance, natural deduction, sequent calculus and Hilbert systems are well-known proof systems (we refer the reader to [10] for details). Each syntax has advantages and disadvantages. For example, classical logic works well in sequent calculus because it allows the symmetry of the connectives to be seen; a natural deduction presentation of classical logic is considered artificial since rules are needed which do not correspond to the introduction or elimination of a connective. Intuitionistic logic, which links exceptionally well with computation, works better in natural deduction, where proofs correspond to programs and there is a notion of canonical proof (which is not the case in sequent calculus).

In this paper we focus on intuitionistic logic, and we aim at designing a syntax that can both

- facilitate the visualisation and understanding of proofs, and
- serve as a basis for the implementation of the simplification rules in the logic.

We choose intuitionistic logic as a basis for this work because of the computational significance of the logic through the Curry-Howard isomorphism: proofs in this logic correspond to functional programs; logic formulas correspond to types of programs; and proof normalisation corresponds to computation. Thus we are not looking for a visual representation of *truth*, but rather that of a *proof*. Furthermore, we are not interested in just representing logics, but also in studying the reduction process (normalisation) which corresponds to computation through the Curry-Howard isomorphism.

Linear logic [8] comes equipped with a graphical syntax called proof nets. One of the motivations for the adoption of a graphical syntax is that traditional syntaxes for logic, such as the sequent calculus, have a lot of constraints to do with the formalism—and not with the logic. To illustrate this, we borrow a well-known example from Girard [9]. Let (r) and (s) be two logical rules, and consider a cut working on auxiliary formulas (not the main formulas of the rules r and s):

$$\dfrac{\dfrac{\vdash \Gamma, A}{\vdash \Gamma', \mathbf{A}}\,(r) \qquad \dfrac{\vdash \neg A, \Delta}{\vdash \neg\mathbf{A}, \Delta'}\,(s)}{\vdash \Gamma', \Delta'}\,(\mathsf{Cut})$$

Rachid Echahed (Ed.): 6th International Workshop
on Computing with Terms and Graphs (TERMGRAPH 2011)
EPTCS 48, 2011, pp. 69–84, doi:10.4204/EPTCS.48.8

If we permute the cut rule up through r and s, then there are two possible choices depending on whether we first permute through s or through r. These two choices are represented by:

$$\dfrac{\dfrac{\dfrac{\vdash \Gamma, \mathbf{A} \qquad \vdash \neg\mathbf{A}, \Delta}{\vdash \Gamma, \Delta}\,(\mathsf{Cut})}{\vdash \Gamma', \Delta}\,(r)}{\vdash \Gamma', \Delta'}\,(s) \qquad\qquad \dfrac{\dfrac{\dfrac{\vdash \Gamma, \mathbf{A} \qquad \vdash \neg\mathbf{A}, \Delta}{\vdash \Gamma, \Delta}\,(\mathsf{Cut})}{\vdash \Gamma, \Delta'}\,(s)}{\vdash \Gamma', \Delta'}\,(r)$$

This demonstrates that permutation of rules is inherent in the syntax. In addition, there is no canonical representation of proofs, even for cut-free proofs. For example, consider the following two alternative proofs (in classical logic), of the formula: $\neg A, C, A \wedge \neg B, B \wedge \neg C$, which differ only by the order in which axioms are combined:

$$\dfrac{\dfrac{}{\vdash \neg A, A}\,(\mathsf{Ax}) \qquad \dfrac{\dfrac{}{\vdash \neg B, B}\,(\mathsf{Ax}) \quad \dfrac{}{\vdash \neg C, C}\,(\mathsf{Ax})}{\vdash C, \neg B, B \wedge \neg C}\,(\wedge)}{\vdash \neg A, C, A \wedge \neg B, B \wedge \neg C}\,(\wedge) \qquad \dfrac{\dfrac{\dfrac{}{\vdash \neg A, A}\,(\mathsf{Ax}) \quad \dfrac{}{\vdash \neg B, B}\,(\mathsf{Ax})}{\vdash \neg A, B, A \wedge \neg B}\,(\wedge) \qquad \dfrac{}{\vdash \neg C, C}\,(\mathsf{Ax})}{\vdash \neg A, C, A \wedge \neg B, B \wedge \neg C}\,(\wedge)$$

A graphical syntax will free us from these inessential permutations. To illustrate the point, the previous two proofs are given by the following proof net:

$\neg A \qquad A \wedge \neg B \qquad B \wedge \neg C \qquad C$

In this visual representation, the nodes represent the connectives and the edges are labelled by formulas. Unlike the sequent calculus, we do not need to impose an order on the introduction of $\wedge$. Moreover, many of the permutation equivalences become identities in proof nets.

In recent years, advances in this area have produced a better understanding of the notion of proof, and a pleasing mathematical theory has been developed. However, it is not clear that this is the best way of visualising proofs or mechanising proof transformations. Proof nets work very well for some fragments of the logic, but less well for others where extra structure is required (boxes for instance). Related systems, such as interaction nets [12] have been used to provide implementations of proof nets. Indeed, they can be seen as a generalisation of proof nets. However, the implementation nature of interaction nets often leads to cluttering proofs with low-level details, which do not aid the understanding of proofs.

In this paper we propose to take features from both proof nets and interaction nets to build a hybrid system that takes some of the good features of each, but is not limited by either of them. We put the case forward for intuitionistic logic, and by the Curry-Howard isomorphism, we also get results for the λ-calculus. This hybrid notation allows us to choose the level of detail that we want to include in the proof. We can give either a high level, visual description of the dynamics of proof simplification, or a low level description which is suited for the implementation or for fine-grained analyses of the cut elimination process.

This graphical notation can be formally defined as a system of port graphs [1, 3]. The visualisation tools available for port graphs (see [2]) can then be used to study proofs and proof transformations.

Summarising, the main contributions of this paper are:

- A new graphical notation for proofs in intuitionistic logic where one can choose different levels of abstraction depending on the kind of analysis that will be done on proofs.

- By the Curry-Howard isomorphism, the previous point gives a graphical notation for the λ-calculus with the same characteristics.

Related work In addition to proof nets and interaction nets, several graphical representations of proofs have been proposed in the literature. We can cite for instance the deduction graphs defined by Geuvers and Loeb [7] and Lamping's sharing graphs [13]. Deduction graphs are a generalisation of natural deduction and Fitch style flag deduction; they have both nodes and boxes. The latter are collections of nodes that form a node themselves, and in this sense they are related to Milner's bigraphs [14], where the place graph describes the nesting of nodes. However, deduction graphs do not have an explicit way to represent sharing; they are not intended as a notation for fine-grained analysis of resource management in proof normalisation. Sharing graphs (introduced in [13] and further developed by Asperti and Guerrini [4]) were presented as a solution for the implementation of Lévy's notion of optimal reduction in the λ-calculus, and, as their name suggests, emphasise the sharing of subexpressions: sharing is explicit.

The rest of this paper is organised as follows. In the following section we briefly recall intuitionistic logic, proof nets, interaction nets and the notions of port graph and port graph rewriting. In Sections 3 and 4 we give the visual representation of the logic and the λ-calculus, respectively. In section 5 we discuss the significance of this work, and speculate on future work.

2 Background

In this section we recall (minimal) intuitionistic logic and sketch some of the ideas behind proof nets and interaction nets, on which our work is built upon. To formalise the notation, we recall the notion of port graph and port graph rewriting from [1, 3]. For more details on linear logic and proof nets, we refer the reader to [8]. For details and examples of interaction nets we refer to [12].

Intuitionistic logic In Figure 1 we give the natural deduction in sequent form presentation of the logic. We give explicitly the structural rules, which helps understanding the graphical notation later. Adding $\vee$ is straightforward, but not included in this paper.

Identity and Structural Group:

$$\frac{}{A \vdash A}(\mathsf{Ax}) \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}(X) \qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B}(W) \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}(C)$$

Logical Group:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B}(\wedge\mathscr{I}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}(\wedge\mathscr{E}_1) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}(\wedge\mathscr{E}_2)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}(\Rightarrow\mathscr{I}) \qquad \frac{\Gamma \vdash A \Rightarrow B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}(\Rightarrow\mathscr{E})$$

Figure 1: Intuitionistic Natural Deduction

The main computational interest for us is the normalisation procedure, which transforms proofs by eliminating redundancies (called detours by Prawitz [15]). The main cases are defined below, showing how the introduction of a connective followed by the elimination of that same occurrence of the connective can be transformed into a proof without the two rules. For simplicity, we have ignored details of permutations of rules that might need to be applied so that one of the following rules can be applied.

Definition 1 *(One Step Normalisation)*

- $(\wedge\mathscr{I})$ *followed by* $(\wedge\mathscr{E}_1)$*: below the double line indicates that (W) may be applied zero or many times. There is a similar case for* $(\wedge\mathscr{I})$ *followed by* $(\wedge\mathscr{E}_2)$*.*

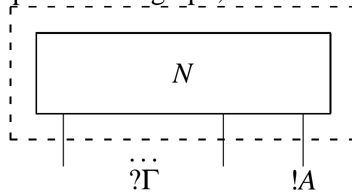
$$\dfrac{\dfrac{\dfrac{\pi_1}{\Gamma \vdash A} \qquad \dfrac{\pi_2}{\Delta \vdash B}}{\Gamma, \Delta \vdash A \wedge B}\,(\wedge\mathscr{I})}{\Gamma, \Delta \vdash A}\,(\wedge\mathscr{E}_1) \quad \textit{becomes} \quad \dfrac{\dfrac{\pi_1}{\Gamma \vdash A}}{\Gamma, \Delta \vdash A}\,(W)$$

- $(\Rightarrow\mathscr{I})$ *followed by* $(\Rightarrow\mathscr{E})$*:* π_1' *is the proof* π_1 *where all axioms* $A \vdash A$ *are replaced (substituted) by a proof of* π_2*.*

$$\dfrac{\dfrac{\dfrac{\pi_1}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B}\,(\Rightarrow\mathscr{I}) \qquad \dfrac{\pi_2}{\Delta \vdash A}}{\Gamma, \Delta \vdash B}\,(\Rightarrow\mathscr{E}) \quad \textit{becomes} \quad \dfrac{\pi_1'}{\Gamma, \Delta \vdash B}$$

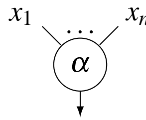
In this presentation, the notion of substitution is important, and is quite difficult to understand. An advantage of the visual representation given below is that it clarifies this notion.

Proof Nets and Interaction Nets Proof nets were introduced as the graphical syntax for linear logic. One of the motivations for the study of graphical presentations is to free us from inessential permutations in proofs, as mentioned in the Introduction. Proof nets work very well for the multiplicative fragment of the logic, but less well for the other fragments. For instance, for the exponentials, more complicated machinery is needed, which takes us away from a uniform visual notation. More precisely, exponentials are represented using boxes to group parts of the graph, shown as a dotted line in the diagram below:



Boxes work at a different level to the other nodes in the graph, leading to a two-level syntax. Interaction nets, on the other-hand, encode the box machinery in the same notation, as shown below.

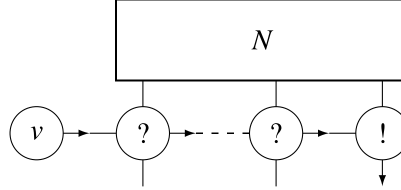
An interaction net system [12] is specified by giving a set Σ of symbols, and a set $\mathscr{R}$ of interaction rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*. An occurrence of a symbol $\alpha \in \Sigma$ will be called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a distinguished one called the *principal port* depicted by an arrow, and n *auxiliary ports* labelled $x_1, \ldots, x_n$ corresponding to the arity of the symbol. Such an agent will be drawn in the following way:



A net N built on Σ is a graph (not necessarily connected) with agents at the vertices. The edges of the graph connect agents together at the ports such that there is only one edge at every port.

A pair of agents $(\alpha,\beta) \in \Sigma \times \Sigma$ connected together on their principal ports is called an *active pair*; the interaction net analog of a redex. An interaction rule $((\alpha,\beta) \Longrightarrow N) \in \mathscr{R}$ replaces an occurrence of the active pair (α,β) by a net N. The rule must satisfy two conditions: all free ports are preserved during reduction, and there is at most one rule for each pair of agents.

Boxes are encoded in interaction nets using extra nodes, as shown below:



Due to the constraints in the rules, interaction nets are easy to implement, but extra rules are needed for management: the details of copying and erasing for instance must be given in full detail. In addition, each node in an interaction net has a unique principal port, and this is fixed for each kind of agent, which means that the reduction system is fixed as part of the encoding.

Our approach in this paper is to put forward a hybrid notation between proof nets and interaction nets to get the best from each. We will be able to use interactive tools developed for interaction nets, such as PORGY [2] in order to visualise the encodings of proofs. In fact, PORGY deals with port graphs, a class of graphs that is more general than interaction nets, and which can be used to formalise our hybrid notation. We recall port graphs below.

Port graphs A port graph [1] is a graph where nodes have explicit connection points for edges, called *ports*. Port graphs were first identified as an abstract view of proteins and molecular complexes.

Let $\mathscr{N}$ and $\mathscr{P}$ be two disjoint sets of node names and port names respectively. A *p-signature* over $\mathscr{N}$ and $\mathscr{P}$ is a mapping $\nabla : \mathscr{N} \to 2^{\mathscr{P}}$ which associates a finite set of port names to a node name. A p-signature can be extended with variables: $\nabla^{\mathscr{X}} : \mathscr{N} \cup \mathscr{X}_{\mathscr{N}} \to 2^{\mathscr{P} \cup \mathscr{X}_{\mathscr{P}}}$, where $\mathscr{X}_{\mathscr{P}}$ and $\mathscr{X}_{\mathscr{N}}$ are two disjoint sets of port name variables and node name variables respectively. A *labelled port graph* over a *p-signature* $\nabla^{\mathscr{X}}$ is a tuple $G = \langle V_G, lv_G, E_G, le_G \rangle$ where: V_G is a finite set of nodes; $lv_G : V_G \to \mathscr{N} \cup \mathscr{X}_{\mathscr{N}}$ is an injective labelling function for nodes; $E_G \subseteq \{\langle (v_1,p_1),(v_2,p_2) \rangle \mid v_i \in V_G, p_i \in \nabla(lv_G(v_i)) \cup \mathscr{X}_{\mathscr{P}}\}$ is a finite multiset of edges; $le_G : E_G \to (\mathscr{P} \cup \mathscr{X}_{\mathscr{P}}) \times (\mathscr{P} \cup \mathscr{X}_{\mathscr{P}})$ is an injective labelling function for edges such that $le_G(\langle (v_1,p_1),(v_2,p_2) \rangle) = (p_1,p_2)$. A port may be associated to a state (for instance, active/inactive or principal/auxiliary); this is formalised using a mapping from ports to port states. Similarly, nodes can also have associated properties (like colour or shape that can be used for visualisation purposes).

Let G and H be two port graphs over the same p-signature $\nabla^{\mathscr{X}}$. A *port graph morphism* $f : G \to H$ maps elements of G to elements of H preserving sources and targets of edges, constant node names and associated port name sets, up to variable renaming. We say that G and H are *isomorphic* if $f : V_G \times \nabla(lv_G(V_G)) \to V_H \times \nabla(lv_H(V_H))$ is bijective.

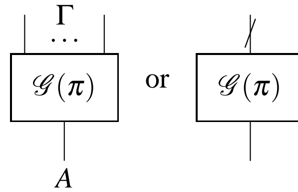
A *port graph rewrite rule* $L \Rightarrow R$ is itself represented as a port graph consisting of two port graphs L and R over the same p-signature and one special node $\Rightarrow$, called *arrow node* connecting them. L and R are called the *left-* and *right-hand side* respectively. The arrow node is used to represent the interface of the rule; it has the following characteristics: for each port p in L, to which corresponds a non-empty set of ports $\{p_1,\ldots,p_n\}$ in R, the arrow node has a unique port r and the incident directed edges (p,r) and (r,p_i), for all $i = 1,\ldots,n$; all ports from L that are deleted in R are connected to the *black hole* port

of the arrow node. When the correspondence between ports in the left- and right-hand side of the rule is obvious we omit the ports and edges involving the arrow node. In this way, we avoid dangling edges after rewriting (for more details on graph rewriting we refer the reader to [11, 6]).

Let $L \Rightarrow R$ be a port graph rewrite rule and G a port graph such that there is an injective port graph morphism g from L to G; hence $g(L)$ is a subgraph of G. A *rewriting step* on G using $L \Rightarrow R$, written $G \rightarrow_{L \Rightarrow R} G'$, transforms G into a new graph G' obtained from G by replacing the subgraph $g(L)$ of G by $g(R)$, and connecting $g(R)$ to the rest of the graph as specified in the arrow node. We call $g(L)$ a redex. If there is no such injective morphism, we say that G is *irreducible* by $L \Rightarrow R$. Given a finite set $\mathscr{R}$ of rules, a port graph G *rewrites* to G', denoted by $G \Rightarrow_{\mathscr{R}} G'$, if there is a rule r in $\mathscr{R}$ such that $G \Rightarrow_r G'$. This induces a transitive relation on port graphs, denoted by $\Rightarrow^*_{\mathscr{R}}$. A port graph on which no rule is applicable is in *normal form*.

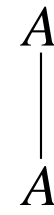
3 Graphs from Proofs

In this section we give a graphical representation of proofs in intuitionistic logic. The general idea is to interpret a proof π of $\Gamma \vdash A$ as a port graph $\mathscr{G}(\pi)$ with edges representing formulas in the following way (the second alternative borrows a notation from electronic circuits):

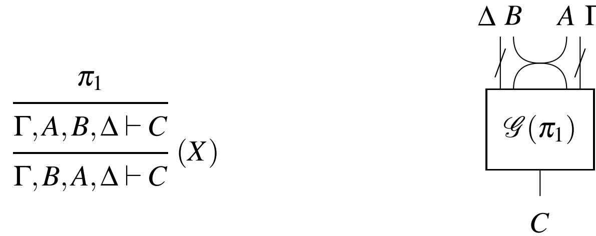


The nodes of the graph represent rules in the logic, edges will be attached to ports which are optionally labelled with formulas. We will explicitly distinguish the conclusion port when it is relevant. Node names will be introduced on demand in the translation below. Later we will see that we need some additional control nodes, so that there will not be a 1-1 correspondence between logical rules and node names. We give a translation inductively over the structure of the proof, and refer to Figure 1 for the rules that we are translating.

- If π is an Axiom $A \vdash A$, then $\mathscr{G}(\pi)$ is simply a node Ax with two ports, both with label A. in the diagrams we omit this node and draw simply a line as it is often done in proof nets.

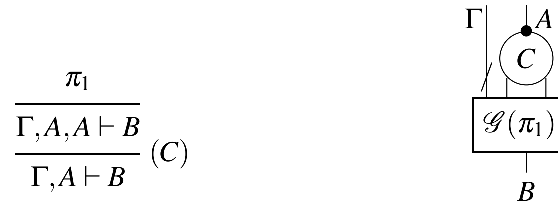


- Exchange. If π_1 is a proof ending in $\Gamma, A, B, \Delta \vdash C$, then we can build a proof π of $\Gamma, B, A, \Delta \vdash C$, using the exchange rule, and a graph $\mathscr{G}(\pi)$ where the exchange rule is encoded by exchanging two edges:

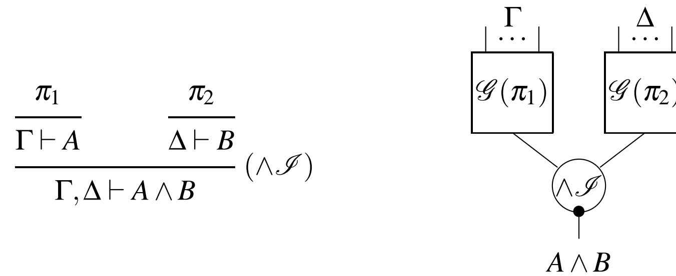


- Weakening. If π_1 is a proof ending in $\Gamma \vdash B$, then we can build a proof π of $\Gamma, A \vdash B$ using the weakening rule, and a graph $\mathscr{G}(\pi)$ as follows, where we explicitly mark the erasing port in the node W:

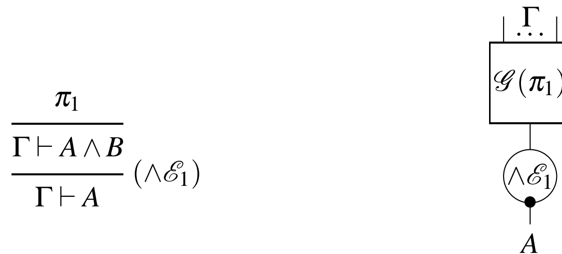
- Contraction. If π_1 is a proof ending in $\Gamma, A, A \vdash B$, then we can build a proof π of $\Gamma, A \vdash B$ using the contraction rule, and a graph $\mathscr{G}(\pi)$, where we explicitly mark the copying port in the node C:



- If π_1 is a proof ending in $\Gamma \vdash A$ and π_2 is a proof ending in $\Delta \vdash B$, then we can build a proof π ending with $\Gamma, \Delta \vdash A \wedge B$ using the $\wedge\mathscr{I}$ rule, and a graph $\mathscr{G}(\pi)$, where we introduce a new node $\wedge\mathscr{I}$ corresponding to the rule, and explicitly mark its conclusion port:



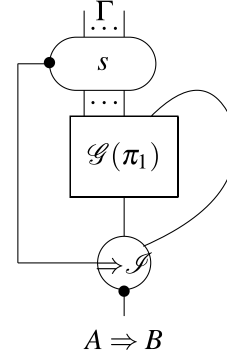
- If π_1 is a proof ending in $\Gamma \vdash A \wedge B$, then we can build a proof π of $\Gamma \vdash A$ using the $\wedge\mathscr{E}_1$ rule, and a graph $\mathscr{G}(\pi)$, where we introduce a new node $\wedge\mathscr{E}_1$:



- If π_1 is a proof ending in $\Gamma \vdash A \wedge B$, then we can build a proof π of $\Gamma \vdash B$ using the $\wedge\mathscr{E}_2$ rule, and a graph $\mathscr{G}(\pi)$ where we introduce a new node $\wedge\mathscr{E}_2$. The graph is similar to the previous case, except that we use a node $\wedge\mathscr{E}_2$ instead of $\wedge\mathscr{E}_1$, and conclude B instead of A.

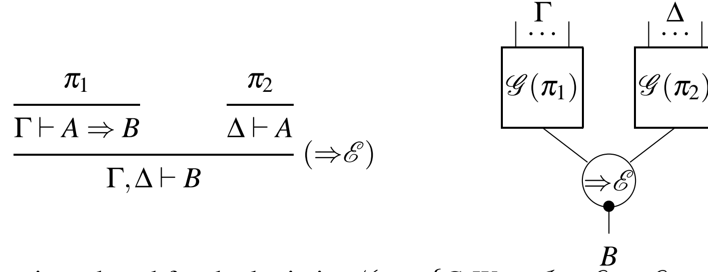
- If π_1 is a proof ending in $\Gamma, A \vdash B$ then we can build a proof π of $\Gamma \vdash A \Rightarrow B$ using the $\Rightarrow\mathscr{I}$ rule, and a graph $\mathscr{G}(\pi)$:

$$\dfrac{\dfrac{\pi_1}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B}\,(\Rightarrow\mathscr{I})$$



where we have introduced a new node $\Rightarrow\mathscr{I}$ corresponding to the rule, and a second node s with variable arity (formally, a family of nodes) that represents the scope of the rule: when the assumption A is discharged we mark all the other assumptions. This structure plays a role to visually represent scope, but more importantly, it will play a crucial role in the dynamics of cut-elimination that we give later. We remark that if Γ is empty (i.e. the proof is closed), then we do not need this extra node, and will just draw the $\Rightarrow\mathscr{I}$ node. Note also that the node s does not correspond to a connective.

- If π_1 is a proof ending in $\Gamma \vdash A \Rightarrow B$ and π_2 is a proof ending in $\Delta \vdash A$, then we can build a proof π of $\Gamma, \Delta \vdash B$ using the $\Rightarrow\mathscr{E}$ rule, and a graph $\mathscr{G}(\pi)$, where we introduce a new node $\Rightarrow\mathscr{E}$:

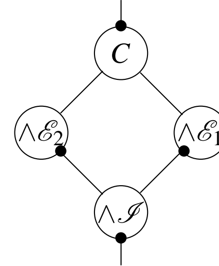


The set of node names introduced for the logic is $\mathscr{N}_{\mathscr{L}} = \{C, W, \wedge\mathscr{I}, \wedge\mathscr{E}_1, \wedge\mathscr{E}_2, \Rightarrow\mathscr{I}, \Rightarrow\mathscr{E}, s\}$.

3.1 Example

To illustrate the translation, we give here an example proof and the corresponding graph. The proof is of $A \wedge B \vdash B \wedge A$, which shows that $\wedge$ is commutative, and in the graph the formulae on the edges can be read from the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A\wedge B \vdash A \wedge B}\,(\mathsf{Ax})}{A \wedge B \vdash B}\,(\wedge\mathscr{E}_2) \qquad \dfrac{\dfrac{}{A\wedge B \vdash A \wedge B}\,(\mathsf{Ax})}{A \wedge B \vdash A}\,(\wedge\mathscr{E}_1)}{A \wedge B, A \wedge B \vdash B \wedge A}\,(\wedge\mathscr{I})}{A \wedge B \vdash B \wedge A}\,(C)$$

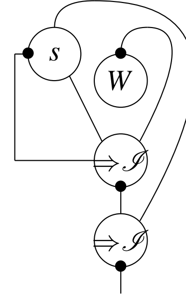


In this example, we can see the true structure of the underlying proof. Contraction was the last rule used in the derivation and is the last rule in the diagram, at the top since the structural rules work on the left of the $\vdash$ symbol and at the top of the diagrams (see the translation above).

Note that we have chosen to use the multiplicative presentation of intuitionistic logic in the sequent calculus, additive is an alternative, and this would lead to a different graphical representation. An essential issue is that the graphical representation is faithful to this choice. In this example, we invite the reader to apply one more rule $(\Rightarrow\mathscr{I})$ to give a proof of $\vdash A \wedge B \Rightarrow B \wedge A$.

A second example illustrates one of the axioms of intuitionistic logic: $A \Rightarrow B \Rightarrow A$. The proof and the graph are the following, where we demonstrate the two different translations of $\Rightarrow\mathscr{I}$, one closed using just an $\Rightarrow\mathscr{I}$ node, and one with free variables using an s node.

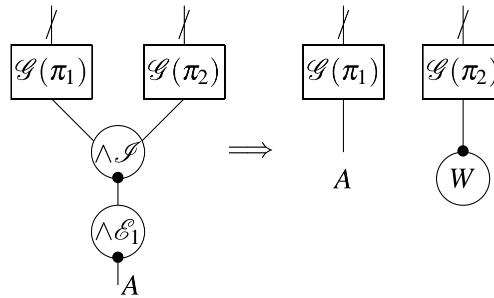
$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,(\mathsf{Ax})}{A, B \vdash A}\,(W)}{A \vdash B \Rightarrow A}\,(\Rightarrow\mathscr{I})}{\vdash A \Rightarrow B \Rightarrow A}\,(\Rightarrow\mathscr{I})$$



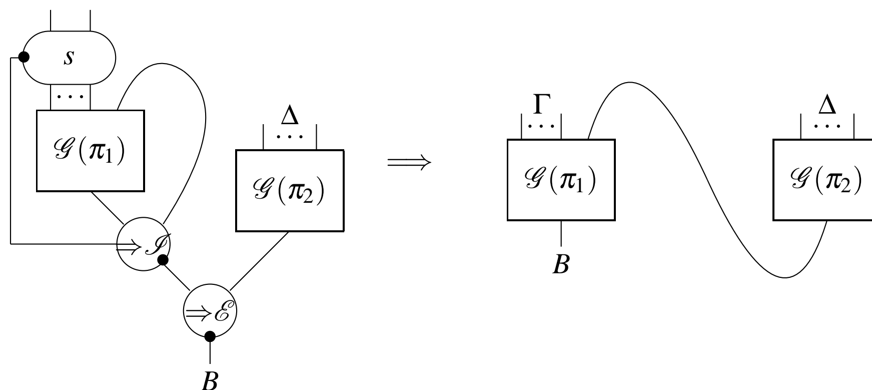
3.2 Normalisation

Next we turn to the normalisation process in this graphical setting. Our aim is to show graph transformations, formalised as port graph rewrite rules, for each of the normalisation steps given in Definition 1.

- $(\wedge\mathscr{I})$ followed by $(\wedge\mathscr{E}_1)$: we attach the weakening node W to the graph representing π_2. There is a symmetric case for $(\wedge\mathscr{I})$ followed by $(\wedge\mathscr{E}_2)$.



- $(\Rightarrow\mathscr{I})$ followed by $(\Rightarrow\mathscr{E})$:

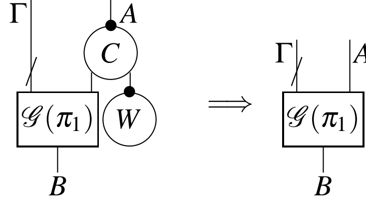


This rule performs a substitution. One of the most beautiful aspects of the notation is the clear explication of the substitution process, which can be seen as replacing an axiom in the proof π_1 with the proof π_2.

We will also consider the following simplification rule, optimising proofs when C is followed by W on the same formula. Thus, the following proof:

$$\dfrac{\dfrac{\Gamma, A \vdash B}{\Gamma, A, A \vdash B}\,(W)}{\Gamma, A \vdash B}\,(C)$$

will be represented and simplified by the following:

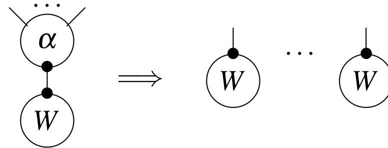


3.3 Copy and Erase

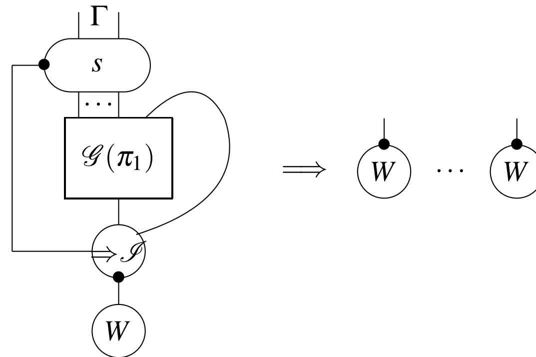
The copy and erase nodes, which correspond to the contraction and weakening rules, require specific rules in order to copy or erase proofs. This is one of the points where interaction net and proof net representations differ. In interaction nets, the copy and erasing processes are performed by traversing the net after it has been normalised (in this way, redexes cannot be copied, which ensures a more efficient implementation). On the other hand, the low level details of the copy and erase rules obscure the understanding of the logic. In the syntax described above, we are free to give global rules for copy and erase as in proof nets (but the price to pay for this is a more involved notion of graph rewriting, with an expensive matching algorithm). We are also free to choose low level, interaction net style rules, which have a simple matching algorithm and are better if we need to analyse the cost of the normalisation process.

Erasing Here we show that the nets arising from the translation function can be erased, either by global steps on W nodes or using the ε agent to erase locally, thus showing that the weakening cut elimination step above can be fully simulated.

- Erasing a node $\alpha \in \mathcal{N}_{\mathcal{L}}$, $\alpha \neq \Rightarrow\mathcal{I}$:

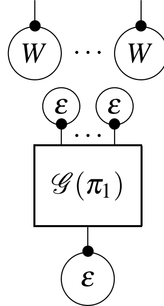


- Erasing $\Rightarrow\mathcal{I}$:

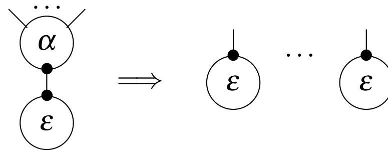


This transformation is a global reduction step: it requires that we identify the graph for π_1, which in turn relies on a notion of pattern matching that is not easy to implement (cf. subgraph isomorphism [16]).

Alternatively, one can use ε nodes that perform small-step erasing, in which case the pattern matching algorithm is trivial. In this case, the previous rule has a left-hand side consisting of just W, $\Rightarrow\mathscr{I}$ and s, and we have a reduction to:

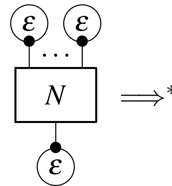


In this way, we can provide a low level definition of weakening which is better adapted for fine grained analysis of the erasing process. The rules for ε, with $\alpha \in \mathscr{N}_{\mathscr{L}} \cup \{\varepsilon\}$, are below (note that if α is ε the right hand side is an empty graph):



In this case the cost of erasing the graph (i.e., the number of rewriting steps involved) depends on its size.

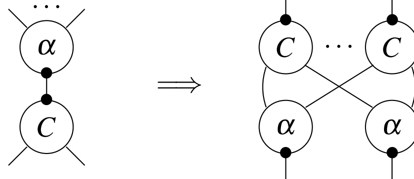
Lemma 1 (Erasing) *Let $N = \mathscr{G}(\pi)$, for any proof π of $A_1, \ldots, A_{n-1} \vdash A_n$. Then using n ε nodes there is a sequence of rewriting steps that erase N, as shown in the following diagram, where the right-hand side is the empty graph:*



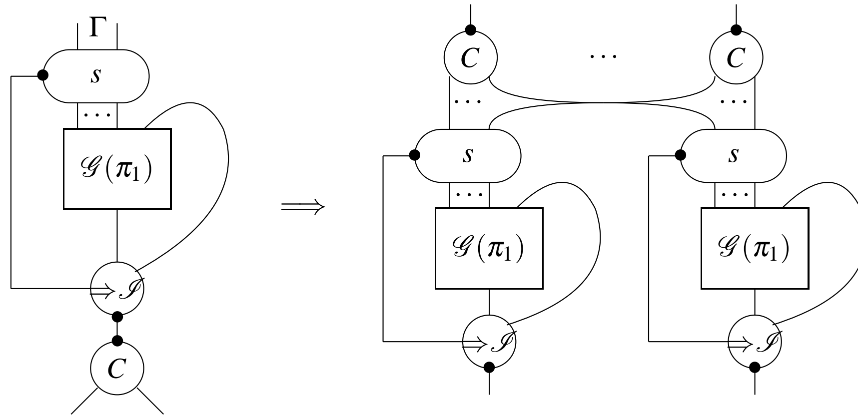
Proof: By induction on π. The proof follows a similar structure to the Duplication Lemma (see Lemma 2 below) that we shall give later. (The case for duplication is slightly more interesting.) □

Duplication Next we address the issue of duplication: specifically, we show that the graphs arising from the translation function can be copied, either by global steps on C nodes or by using the δ agent to copy step-by-step. We first show the rules for the C node.

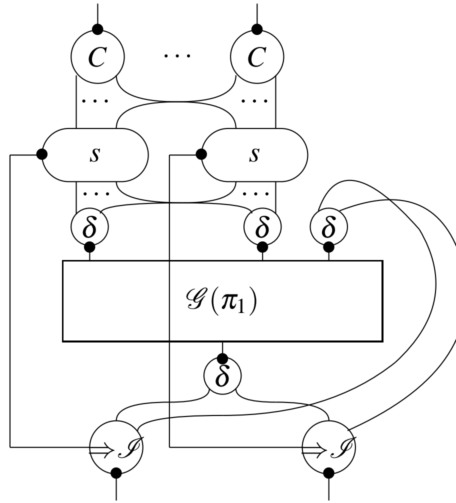
- Copying a node $\alpha \neq \Rightarrow\mathscr{I}$, $\alpha \in \mathscr{N}_{\mathscr{L}}$:



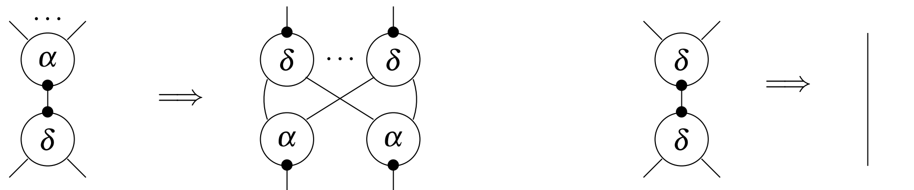
- Copying $\Rightarrow\mathscr{I}$:



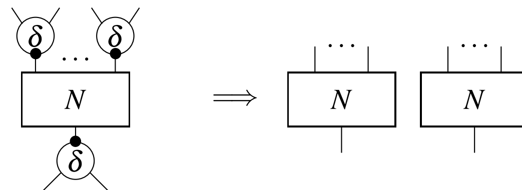
Alternatively, one can use δ nodes to perform small-step copying. In this case the left-hand side of the rule is just C with $\Rightarrow\mathscr{I}$ and s, and we have a reduction to:



We now introduce the rewrite rules for the δ nodes. Let α be any node in $\mathscr{N}_{\mathscr{L}}$. Then δ copies the node and propagates itself to copy the rest of the graph, as shown below. The rule for δ with δ ends the duplication process.

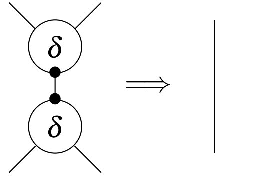


Lemma 2 (Duplication) *Let $N = \mathscr{G}(\pi)$, for any proof π of $A_1,\ldots,A_{n-1} \vdash A_n$. Then using n δ nodes there is a sequence of rewriting steps that duplicates N, as shown in the following diagram:*

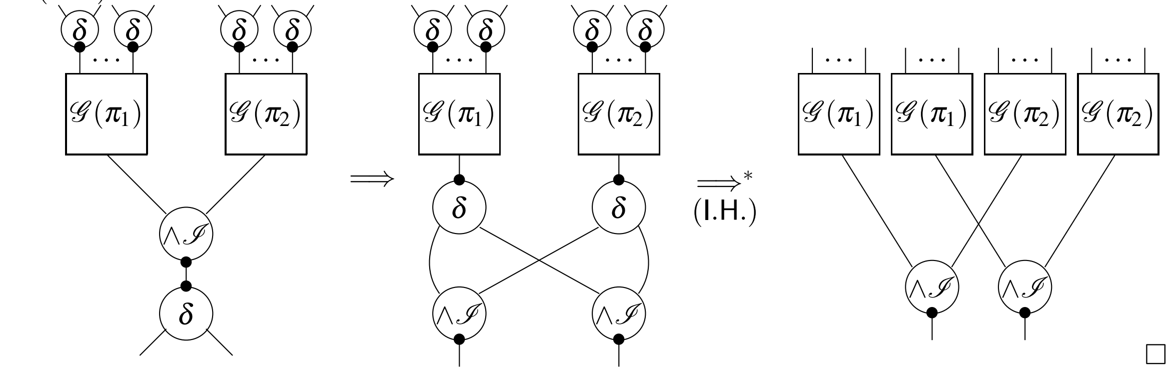


Proof: By induction on the depth of the proof π. We show the cases for Axiom and $(\wedge\mathscr{I})$.

- Axiom:



- $(\wedge\mathscr{I})$:



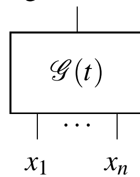
□

Proposition 1 *(Correctness) For each normalisation step transforming π to π', there is a transformation from $\mathscr{G}(\pi) \Longrightarrow^* \mathscr{G}(\pi')$.*

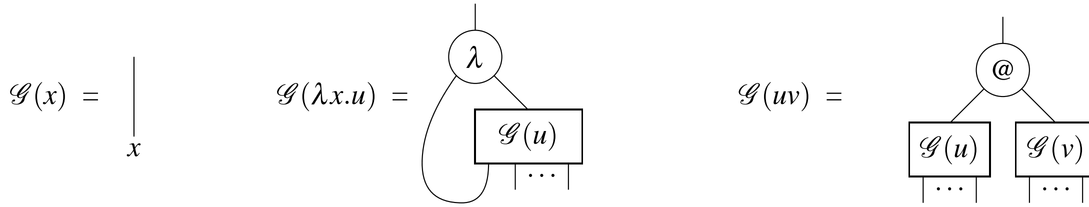
Proof: For the first normalisation rule, π is a proof build from π_1 and π_2 using a $(\wedge\mathscr{I})$ followed by $(\wedge\mathscr{E}_1)$ (similar for $(\wedge\mathscr{E}_2)$), and $\mathscr{G}(\pi')$ is a graph consisting of $\mathscr{G}(\pi_1)$ and a W node attached to each hypothesis in Δ (note that Δ are the hypothesis for π_2). Using the normalisation step in Section 3.2, from $\mathscr{G}(\pi)$ one obtains a graph consisting of $\mathscr{G}(\pi_1)$ and a W node attached to the conclusion port of $\mathscr{G}(\pi_2)$. By induction on π_2, and relying on Lemma 1, one can proof that a graph consisting to a W node attached to the conclusion port of $\mathscr{G}(\pi_2)$, reduces to a graph consisting of a W node attached to each hypothesis in Δ. For the second normalisation rule, we rely on Lemma 2 to prove a similar result for C. □

4 Graphs for the Linear λ-calculus

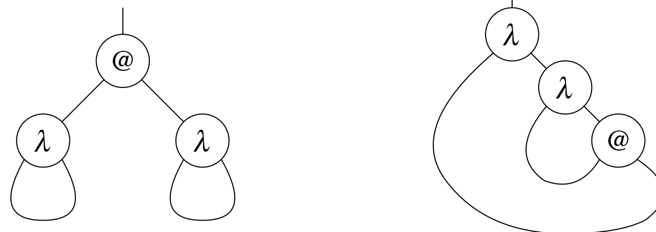
Now that we have seen what graph reduction is for the logic, we briefly look at it again, but from a different perspective. There are standard ways of representing the λ-calculus as graphs, and the reduction mechanism as a graph rewriting system. We restrict this section to the linear case which is simpler, and generalises using the structural rules of the previous section. The general form of the translation generating a graph from a term t is the following:



where the free variables of t are $x_1,\ldots,x_n$. The translation $\mathscr{G}(\cdot)$ is given inductively over the structure of the linear term t. We shall often drop the labelling of the edges when there is no ambiguity. For abstractions we assume (without loss of generality) that the (unique occurrence of the) variable x occurs in the leftmost position of the free variables of $\mathscr{G}(u)$. Notice also that, in the case of applications there will not be any common free variables between the graphs for u and v by the linearity constraint.

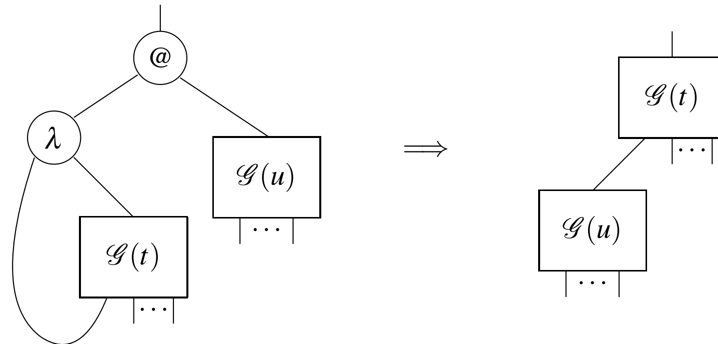


As examples consider the graphs for: $(\lambda x.x)(\lambda x.x)$ and $(\lambda xy.yx)$.

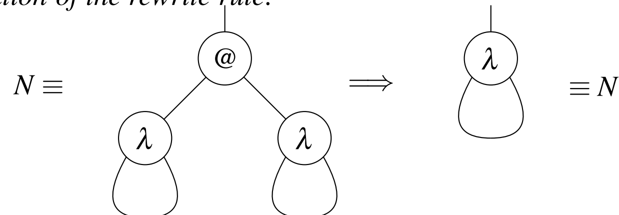


It is not difficult to see that replacing $\Rightarrow\mathscr{I}$[1] by λ and $\Rightarrow\mathscr{E}$ by @, and changing the orientation of these graphs we obtain exactly the same system of graph reduction given for the logic. This all leads to visual confirmation of the Curry-Howard isomorphism, where we can think of graphs corresponding to proofs, types corresponding to formulas, and graph reduction to normalisation.

We now turn to reduction in the linear λ-calculus for these graphs, and set up a notion of *linear graph reduction*. The idea is quite simple: we will draw the graph for the term $(\lambda x.t)u$ and another for $t[u/x]$ and try to deduce the corresponding graph reduction step(s). The required reduction is given by:

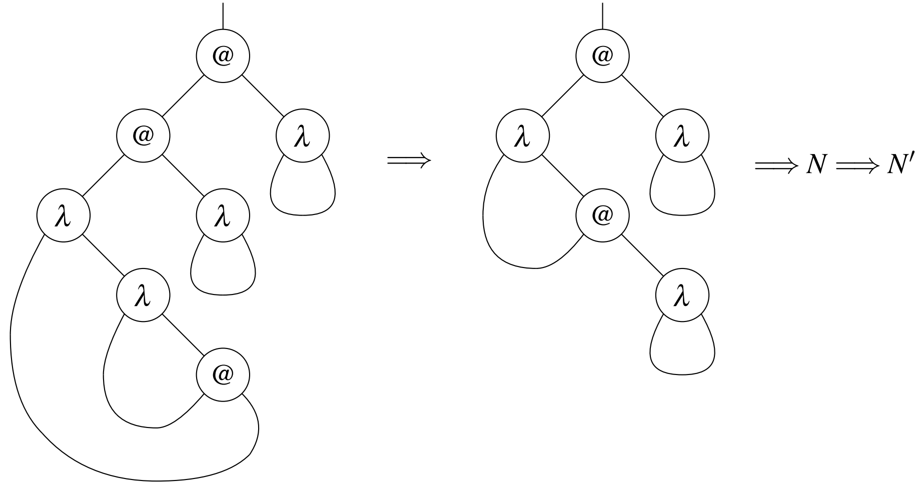


Example 1 *Here are a couple of examples of linear graph reduction. The first example is the identity applied to the identity function. Now $\mathscr{G}((\lambda x.x)(\lambda x.x))$ (which we call N) reduces to $\mathscr{G}(\lambda x.x)$ (which we call N') with one application of the rewrite rule:*



Note that there was only one graph reduction step required here: the β-reduction step, together with the substitution, was captured in a single rewrite. The advantage of this particular system of graph rewriting for the λ-calculus is that substitution is always done for free. As a larger example, consider the term $(\lambda xy.yx)(\lambda x.x)(\lambda x.x)$. As a graph reduction, this simply becomes the following:

[1]Simplified to the linear case.



where N and N' are the nets of the previous example. Thus, the total number of graph rewrite steps is just three, corresponding exactly to the number of β-reductions performed at the level of syntax. One can see that there are a lot of benefits from a graphical notation for this simple calculus: the graph rewriting process is local*, meaning that at any time we only rewrite the part of the graph connecting the application and abstraction; the rest of the graph remains unchanged.*

5 Discussion and Conclusion

Our goal was to provide a notation for intuitionistic logic (also for the λ-calculus through the Curry-Howard isomorphism) that shares some of the advantages of previous graphical notions such as proof nets and interaction nets, but also simplifies and alleviates some of the constraints.

- The graphical notation brings out the structure of the proof visually, close to the abstract syntax, and consequently we believe it to be quite natural.
- This notation is preserved under normalisation (computation) which means that we can animate the process. As part of this, we can better understand the process of normalisation and substitution.
- As the examples show, the diagrams also alleviate much of the syntactic clutter which helps to bring out the structure of the underlying proof.
- We have established that normalisation preserves the graphical notation, but we have assumed that proofs always come from a natural deduction proof (i.e., through a translation). We have deliberately avoided the question as to when an arbitrary graph built from the nodes given is a valid proof however. These questions are difficult to solve, and until we have established the usefulness of the notation, we need not invest effort into this. However, it remains a very important question that will need to be addressed.
- For the λ-calculus, our approach and motivation is similar to that of [5]. Our graphs are closer to the abstract syntax trees, and we believe this is easier to relate to the syntax in addition to allowing the process of substitution to be controlled precisely.
- Since the λ-calculus is the foundational calculus underlying functional programming, this gives a starting point for a visual approach for this paradigm.

References

[1] O. Andrei. *A Rewriting Calculus for Graphs: Applications to Biology and Autonomo us Systems*. PhD thesis, Institut National Polytechnique de Lorraine, http://tel.archives-ouvertes.fr/tel-00337558/fr/, 2008.

[2] O. Andrei, M. Fernández, H. Kirchner, G. M. a nd O. Namet, and B. Pinaud. Porgy: Strategy driven interactive transformation of graphs. In *Proceedings of TERMGRAPH 2011, Saarbrucken*. EPTCS, 2011. In this volume.

[3] O. Andrei and H. Kirchner. A Rewriting Calculus for Multigraphs with Ports. In *Proceedings of RULE'07*, volume 219 of *Electronic Notes in Theoretical Computer Science*, pages 67–82, 2008.

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

[5] W. Citrin, R. Hall, and B. Zorn. Programming with visual expressions. In *In Proceedings of the 11th IEEE Symposium on Visual Languages*, pages 294–301. IEEE Computer Society Press, 1995.

[6] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 163–246. World Scientific, 1997.

[7] H. Geuvers and I. Loeb. Natural deduction via graphs: formal definition and computation rules. *Mathematical Structures in Computer Science*, 17(3):485–526, 2007.

[8] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[9] J.-Y. Girard. Linear logic : its syntax and semantics. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, number 222 in London Mathematical Society Lecture Note Series, pages 1–42. Cambridge University Press, 1995.

[10] J. Goubault-Larrecq and I. Mackie. *Proof Theory and Automated Deduction*, volume 6 of *Applied Logic Series*. Kluwer Academic Publishers, Dordrecht, May 1997.

[11] A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.

[12] 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.

[13] 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.

[14] R. Milner. Axioms for bigraphical structure. *Mathematical Structures in Computer Science*, 15(6):1005–1032, 2005.

[15] D. Prawitz. *Natural Deduction, A Proof-Theoretical Study*. Almqvist and Wiskell, Stockholm, 1965.

[16] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23:31–42, January 1976.

Repetitive Reduction Patterns in Lambda Calculus with `letrec` (*Work in Progress*)*

Jan Rochel
Utrecht University
Utrecht, The Netherlands
Department of Information and Computing Sciences
Information and Software Systems
J.Rochel@cs.uu.nl

Clemens Grabmayer
Utrecht University
Utrecht, The Netherlands
Department of Philosophy
Theoretical Philosophy
Clemens.Grabmayer@phil.uu.nl

For the λ-calculus with `letrec` we develop an optimisation, which is based on the contraction of a certain class of 'future' (also: *virtual*) redexes.

In the implementation of functional programming languages it is common practice to perform β-reductions at compile time whenever possible in order to produce code that requires fewer reductions at run-time. This is, however, in principle limited to redexes and created redexes that are 'visible' (in the sense that they can be contracted without the need for unsharing), and cannot generally be extended to redexes that are concealed by sharing constructs such as `letrec`. In the case of recursion, concealed redexes become visible only after unwindings during evaluation, and then have to be contracted time and again.

We observe that in some cases such redexes exhibit a certain form of repetitive behaviour at run time. We describe an analysis for identifying binders that give rise to such repetitive reduction patterns, and eliminate them by a sort of predictive contraction. Thereby these binders are lifted out of recursive positions or eliminated altogether, as a result alleviating the amount of β-reductions required for each recursive iteration.

Both our analysis and simplification are suitable to be integrated into existing compilers for functional programming languages as an additional optimisation phase. With this work we hope to contribute to increasing the efficiency of executing programs written in such languages.

In this extended abstract we report on work in progress carried out within the framework of the NWO project *Realising Optimal Sharing*. Instead of discussing optimal reduction in the λ-calculus, however, here we are concerned with a static analysis of λ_{letrec}-terms which aims at removing β-redexes that are concealed by recursion constructs and cause cyclic migration of arguments during evaluation. We have to stress that our research on this particular topic is still in an early phase.

1 Introduction

In this work we study terms in λ_{letrec}, i.e. in λ-calculus with an explicit `letrec`-construct for recursive definitions, that exhibit a form of repetitive reduction pattern when evaluated. We try to identify a class of such terms for which this behaviour can be avoided by a transformation into a term with, in some sense, the same semantic denotation. Even though the presented optimisation can be described directly for λ_{letrec}-terms, and hence is applicable in all functional languages of which λ_{letrec} is a meaningful abstraction, we will use Haskell to denote examples of such terms and their optimised equivalents. Additionally, we depict terms as λ-graphs with explicit sharing-nodes (*multiplexers*) as used, for example, in [2].

*Funded by the NWO-project *Realising Optimal Sharing*

Rachid Echahed (Ed.): 6th International Workshop
on Computing with Terms and Graphs (TERMGRAPH 2011)
EPTCS 48, 2011, pp. 85–100, doi:10.4204/EPTCS.48.9

© Rochel, Grabmayer
This work is licensed under the
Creative Commons Attribution License.

A function well-known to Haskell programmers is the *repeat* function that generates an infinite, constant stream of the supplied argument. A definition is easily found, namely:

$$repeat\ x = x : repeat\ x$$

An experienced Haskell programmer, however, would spot a 'space leak', which refers to an $O(n)$ memory consumption for generating *n* stream elements while $O(1)$ is possible, due to lazy evaluation. Therefore in the Haskell standard libraries that function is defined as:

$$repeat\ x = \mathbf{let}\ xs = x : xs\ \mathbf{in}\ xs$$

The exact reasons for this difference in efficiency involve the characteristics of the deployed Haskell compiler and run-time system. A more direct and theoretical explanation can be attempted within the formal framework of λ_{letrec}: The improved variant of *repeat* does not require any β-reductions to produce further stream elements. That becomes apparent by the λ-graphs and their infinite unfoldings (Fig. 1). We use a rewriting relation arrow indexed by a triangle ($\rightarrow_{\triangledown}$) to mark unfolding, regardless of whether sharing is expressed by a multiplexer or a `letrec`-binding.

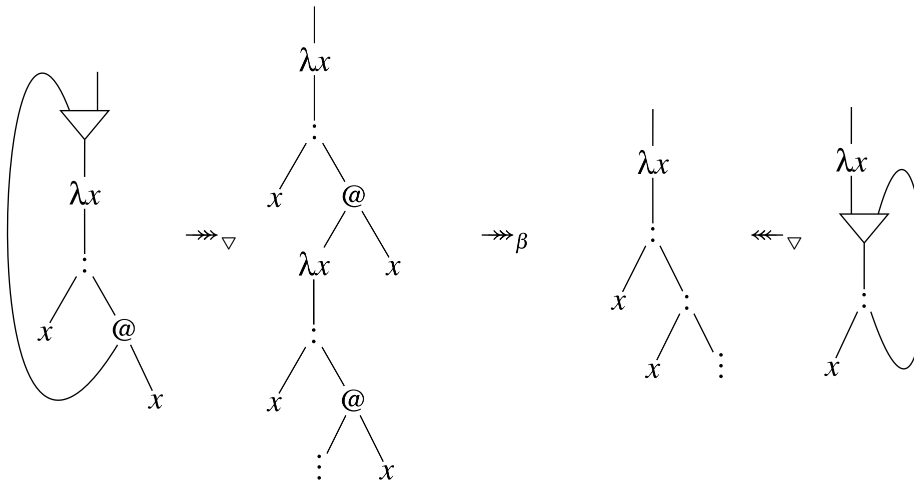


Figure 1: Term graph representation of the two *repeat* implementations and their unfoldings

From a software engineering perspective it is unsatisfactory that the programmer has to recognise and mitigate such cases. One might even consider the unoptimised version superior with respect to code clarity. Therefore we propose an analysis and transformation method to automate the optimisation, which then can be integrated into the compiler pipeline of existing functional language implementations. In the following sections we will work with simple examples to develop this method and successively generalise it for wider applicability.

2 Preliminaries

The method we describe applies to the λ_{letrec}-calculus, which is a higher-order rewrite system. Still, in this work-in-progress report we primarily intend to motivate our research and outline the approaches developed so far. Hence, for the moment we give a largely informal description, and resort to first-order formulations, λ-graphs, or Haskell, whichever seems more suited.

Definition 1 (First-order representation of λ_{letrec}) Let V be a set of variables. Then a λ_{letrec}-term is defined as follows:

$$
\begin{array}{llcll}
(\textit{term}) & T & ::= & \lambda V.T & (\textit{abstraction}) \\
 & & | & T\ T & (\textit{application}) \\
 & & | & V & (\textit{variable}) \\
 & & | & \texttt{letrec}\ \textit{Defs}\ \texttt{in}\ T & (\textit{letrec}) \\
(\textit{definitions}) & \textit{Defs} & ::= & v_1 = T\ \dots\ v_n = T & (\textit{equations}) \\
 & & & v_1,\dots,v_n \in V \text{ all distinct} &
\end{array}
$$

But ultimately only a higher-order formulation can be formally satisfactory, thus we propose the following representation as a higher-order rewrite system (HRS) [13] for λ_{letrec}.

Definition 2 (Higher-order representation of λ_{letrec}) Let *Var* be a set of variables, and *BTypes* a set of base types that induce the set *Types* of simple types. The *terms* of λ_{letrec} are simply-typed higher-order terms over the HRS-signature that for all $n \in \mathbb{N}$, and all types $\tau_0, \tau_1, \dots, \tau_n \in \textit{Types}$ contains a symbol $\texttt{let}_n\texttt{_in}$ of type:

$$\texttt{let}_n\texttt{_in} : (\tau_1 \times \dots \times \tau_n \to \tau_0 \times \tau_1 \dots \times \tau_n) \to \tau_0$$

Product types are only used for better readability. We will use the symbols t, s, u for terms in λ_{letrec}.

Based on this notion of terms, the λ_{letrec}-calculus consists of the rewrite relations: β-reduction, η-reduction, and `letrec`-unfolding. Additionally, we use the concept of generalised β-reduction [11].

For example, `letrec`-unfolding on the informal λ_{letrec}-terms according to the grammar above could be described by the following rewrite rules:

$$\texttt{let}\ f_1 = s_1(\vec{f}),\dots,f_n = s_n(\vec{f})\ \texttt{in}\ t \ \to_{\triangledown}\ t \qquad (\text{if } f_1,\dots,f_n \text{ not free in } t)$$

$$\texttt{let}\ \underbrace{f_1 = s_1(\vec{f}),\dots,f_n = s_n(\vec{f})}_{\textit{Defs}}\ \texttt{in}\ t(\vec{f}) \ \to_{\triangledown}\ t\big(\texttt{let}\ \textit{Defs}\ \texttt{in}\ s_1(\vec{f}),\dots,\texttt{let}\ \textit{Defs}\ \texttt{in}\ s_n(\vec{f})\big)$$

which, if translated into HRS-rules (using the signature defined in Def. 2), can take the following form:

$$\texttt{let}_n\texttt{_in}\,\lambda f_1 \dots f_n.(Y, Z_1(\vec{f}),\dots,Z_n(\vec{f})) \ \to_{\triangledown}\ Y$$

$$
\begin{aligned}
\texttt{let}_n\texttt{_in}\,\lambda f_1 \dots f_n.(Y(\vec{f}), Z_1(\vec{f}),\dots,Z_n(\vec{f})) \ \to_{\triangledown} & \\
Y(\texttt{let}_n\texttt{_in}\,\lambda f_1 \dots f_n.(Z_1(\vec{f}), Z_1(\vec{f}),\dots,Z_n(\vec{f}))) \dots & \\
\dots(\texttt{let}_n\texttt{_in}\,\lambda f_1 \dots f_n.(Z_n(\vec{f}), Z_1(\vec{f}),\dots,Z_n(\vec{f}))) &
\end{aligned}
$$

Notation 3 (Rewrite relations in λ_{letrec}) On λ_{letrec}-terms we consider the following rewrite relations: β-reduction denoted by $\to_\beta$; generalised β-reduction denoted by $\to_{g\beta}$; η-reduction denoted by $\to_\eta$; `letrec`-unfolding denoted by $\to_{\triangledown}$.

For each of these rewrite relations $\to$, the *many-step rewrite relation* with respect to $\to$ will be written as $\twoheadrightarrow$, and the (strongly convergent) *infinite rewrite relation* as $\twoheadrightarrow\!\!\!\!\!\rightarrow$.

With the transformation from Section 1, which is further developed in the next sections, we aim to convert a given term t with repetitive reduction patterns into a term t' that does not require these reductions to be performed any more, but such that t and t' are 'operationally equivalent', in a sense that guarantees that important properties observable during evaluation are preserved.

One candidate for a precisely defined notion of operational equivalence is the extension to λ_{letrec}-terms of 'applicative bisimulation' on λ-terms due to Abramsky [1]. Two λ_{letrec}-terms M and N are called *applicative bisimilar* (symbolically: $M \sim^B N$) if M and N behave in the same way under all possible series $E_0, E_1, E_2, \ldots$ of 'experiments' of the following kind: on a starting term M_0 the first experiment E_0 consists in finding out whether or not M_0 reduces to an abstraction (a weak head normal form); if the outcome M_i of the previous experiment E_i is indeed an abstraction $\lambda x.N_i$, then for experiment E_{i+1} an arbitrary term P_{i+1} is chosen, and it is determined whether or not the redex $(\lambda x.N_i)P_{i+1}$ reduces to an abstraction.

While applicative bisimulation has been frequently used to justify optimising transformations for functional programming languages, there may be a host of other interesting notions of operational equivalence. Since, for the moment, we do not want to commit ourselves to a particular notion of operational equivalence, we will use a syntactically defined notion of equivalence between terms instead. In fact, we will define this syntactic notion as the convertibility relation with respect to rewrite relations that we use for motivating and justifying the optimising transformation in, for example, Fig. 1 and Fig 2. There, we use, in addition to infinite convertibility with respect to $\rightarrow_{\triangledown}$ and $\rightarrow_{g\beta}$, a restricted form of 'vector η-reduction' that is defined by the following rewrite rule:

$$\lambda x_1 \ldots x_n.Mx_1 \ldots x_n \quad \rightarrow \quad M \qquad (\text{if } x_1, \ldots, x_n \text{ distinct, and not free in } M)$$

The induced rewrite relation $\rightarrow_{\vec{\eta}}$ extends η-reduction, but can be mimicked with η-steps, and therefore has the same many-step relation. However, neither for η-reduction nor for vector η-reduction it holds in generality that the source and the target of a step are applicative bisimilar: for example, in the η-reduction step $\lambda x.yx \rightarrow_{\eta} y$ the source is an abstraction, but the target is not.

Since we want to obtain a syntactically defined notion of operational equivalence that is stronger than applicative bisimilarity, we define a restriction $\rightarrow_{\vec{\eta}_0}$ of $\rightarrow_{\vec{\eta}}$, and a variant $\rightarrow_{\vec{\eta}_0^{\text{per}}}$ of $\rightarrow_{\vec{\eta}_0}$, both of which serve our purposes and, importantly, only allow steps between applicative bisimilar terms. The converse rewrite relation $\leftarrow_{\vec{\eta}_0}$ of $\rightarrow_{\vec{\eta}_0}$ performs a copying operation for λ-abstraction prefixes in terms; and the converse rewrite relation $\leftarrow_{\vec{\eta}_0^{\text{per}}}$ of $\rightarrow_{\vec{\eta}_0^{\text{per}}}$ both copies a λ-abstraction prefix and carries out a permutation in it.

Definition 4 (Restriction and variant of $\rightarrow_{\vec{\eta}}$) The restricted version $\rightarrow_{\vec{\eta}_0}$ of the rewrite relation $\rightarrow_{\vec{\eta}}$ on λ_{letrec}-terms is defined by the rule:

$$\lambda x_1 \ldots x_n.(\lambda x_1 \ldots x_n.M)x_1 \ldots x_n \quad \rightarrow \quad \lambda x_1 \ldots x_n.M \qquad (\text{if } x_1, \ldots, x_n \text{ distinct, and not free in } M)$$

And the extension $\rightarrow_{\vec{\eta}_0^{\text{per}}}$ of $\rightarrow_{\vec{\eta}_0}$ with respect to permuting variable names in abstraction prefixes is defined by the rewrite rule:

$$\lambda x_1 \ldots x_n.(\lambda x_{\pi(1)} \ldots x_{\pi(n)}.M)x_{\pi(1)} \ldots x_{\pi(n)} \quad \rightarrow \quad \lambda x_1 \ldots x_n.M$$
$$(\text{if } x_1, \ldots, x_n \text{ distinct, and not free in } M, \text{ and } \pi \text{ is a permutation on } \{1, \ldots, n\})$$

It is easy to verify that left- and right-hand sides of these rules are applicative bisimilar.

Now we define the syntactic notion of equivalence on which we base our transformation.

Definition 5 (Equivalence relation $=^{\infty}_{\triangledown,g\beta}$) *Infinite convertibility with respect to* $\rightarrow_{\triangledown}$ and $\rightarrow_{g\beta}$, extended by finitely many $\rightarrow_{\vec{\eta}_0^{\text{per}}}$-reduction steps, is the following relation on λ_{letrec}-terms:

$$=^{\infty}_{\triangledown,g\beta} \;:=\; \left(\twoheadleftarrow_{\vec{\eta}_0^{\text{per}}} \cup \twoheadleftarrow\!\!\!\twoheadleftarrow_{\triangledown} \cup \twoheadleftarrow\!\!\!\twoheadleftarrow_{g\beta} \cup \twoheadrightarrow\!\!\!\twoheadrightarrow_{g\beta} \cup \twoheadrightarrow\!\!\!\twoheadrightarrow_{\triangledown} \cup \twoheadrightarrow_{\vec{\eta}_0^{\text{per}}}\right)^*$$

Since source and targets of each of the rewrite relations $\to_{\triangledown}$, $\to_{g\beta}$, and $\to_{\vec{\eta}_0^{\text{per}}}$ are applicative bisimilar, and since applicative bisimilarity is a contextual congruence [1], the following proposition can be proved, which states that the syntactical equivalence from Definition 5 is at least as strong as, i.e. is contained in, applicative bisimulation.

Proposition 6 *For all λ_{letrec}-terms M,N it holds:* $M =^{\infty}_{\triangledown,g\beta} N \Rightarrow M \sim^{B} N$.

3 Further Examples

The *repeat* function shows that for some cases it is possible to lift parameters out of recursive positions and thereby improve run-time efficiency. That raises the question of when this is possible. What is the pattern that allows for such an optimisation?

What strikes the eye are the occurrences of the syntactic element *repeat x* on both the left-hand and the right-hand sides of the function definition. That suggests a sort of common subexpression elimination that takes into account both sides of the equation. This formulation, however, cannot cover the following example, which differs from *repeat* essentially only by an additional parameter n on the left-hand side, and the argument $n-1$ on the right.

$$\begin{aligned} &\mathit{replicate}\ 0\ x = [\,] \\ &\mathit{replicate}\ n\ x = x : \mathit{replicate}\ (n-1)\ x \end{aligned}$$

As it was the case for *repeat*, again it is possible to lift the parameter x out of the recursion.

$$\begin{aligned} \mathit{replicate}\ n\ x = \ &\mathbf{let}\ \mathit{rec}\ 0\ = [\,] \\ &\qquad \mathit{rec}\ n = x : \mathit{rec}\ (n-1) \\ &\mathbf{in}\ \mathit{rec}\ n \end{aligned}$$

Again we regard both variants and their infinite unfolding (Fig. 2) to understand the transformation. For reasons of clarity, however, we completely leave out the scrutinisation of parameter n and the subsequent case discrimination and concentrate on the recursive pattern.

In comparison with the previous example we observe two differences. First, note the 'header' $\lambda n.\lambda x.\ [\ ]\ n\ x$ attached on top of the second graph, which we obtain by two η-expansions. This allows us to produce an optimised term that does not comprise a duplicated function body. Furthermore, instead of relying on ordinary β-reduction we have to add *generalised beta-reductions* ($g\beta$-reduction) [11] to our arsenal.

The key pattern shared by the two presented examples that permits optimisation is a parameter p that is being passed through unchanged in the recursive application. Consequently, once the function is called from the outside with some argument a in p's position, while recursively evaluating that call, p can never again be bound to another value than a or a descendant of a. In that sense one might call p a 'constant parameter'. It suggests itself, that components that are 'constant' to a recursive construct can be lifted out of the recursion.

4 A rewrite rule for simple recursive patterns

The examples in Section 1 and Section 3 suggest that there are many similar situations in which optimisations of the kind as described can be carried out. A first attempt to obtain general formulations of such simplification steps would be to use schemata, and in effect, rewrite rules on λ_{letrec}-terms.

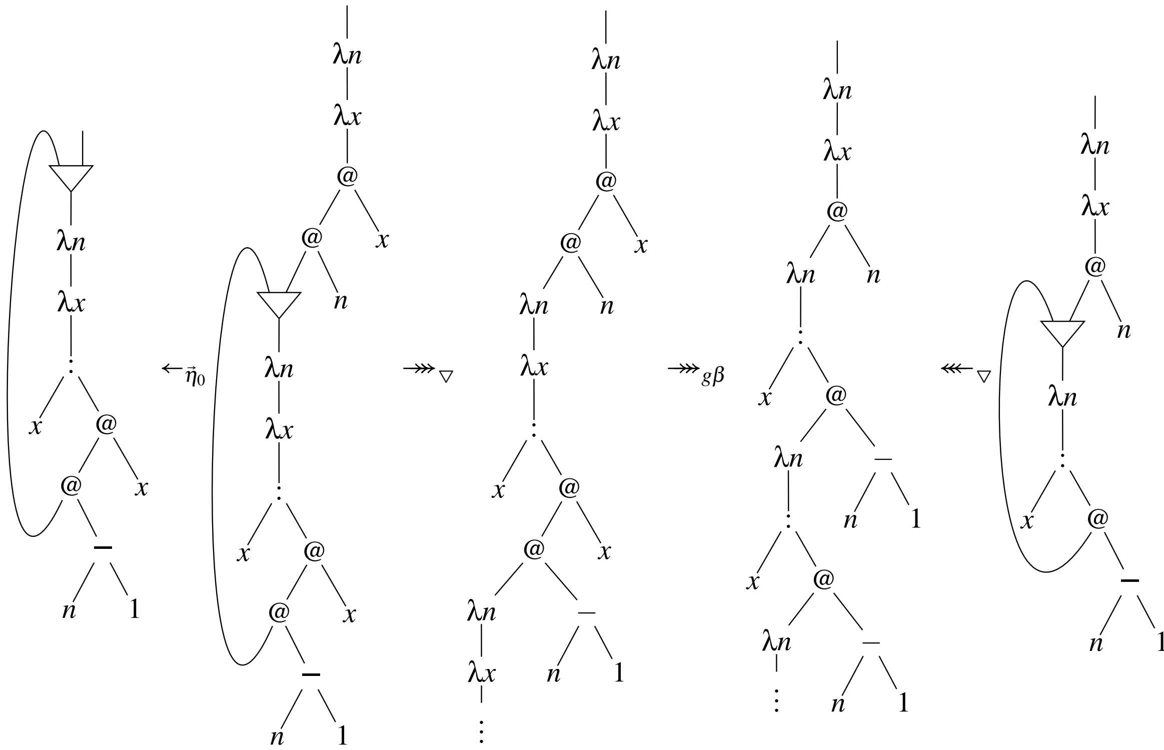


Figure 2: Operational equivalence of the two *replicate* variants depicted in graph notation

As an example, let us consider the recursive definition of a function f in which the $(n+1)$th parameter y is passed on to all recursive calls of f as the $(n+1)$th argument. In this case the transformation that eliminates the recurrent parameter y can be described by the following first-order rewrite rule on λ_{letrec}

$$\mathbf{let}\ f = \lambda x_1.\ ...\ \lambda x_n.\lambda y.C\ [f\ t_1\ ...\ t_n\ y]\ \mathbf{in}\ f$$
$$\rightarrow$$
$$\lambda x_1.\ ...\ \lambda x_n.\lambda y.\mathbf{let}\ f' = \lambda x_1.\ ...\ \lambda x_n.C\ [f'\ t_1\ ...\ t_n]\ \mathbf{in}\ f'\ x_1\ ...\ x_n$$

where C is a λ_{letrec}-context with possibly more than one occurrence of the context-hole $[]$, f and f' do not occur in C, and y does not get bound during hole-filling. Here the recurrent parameter y in the recursive definition of f is lifted out of the recursion, and the number of arguments in the recursive call of the function in the `let`-construct has decreased by one after the transformation. Note, that context C might start off with initial lambdas, and therefore also covers the case in which y is followed by further parameters.

To cover situations with multiple recursive calls to f with (possibly) varying arguments, we can generalise the rewrite rule as follows, as long as the argument in question, y, is the same.

$$\mathbf{let}\ f = \lambda x_1.\ ...\ \lambda x_n.\lambda y.C\ [f\ t_1^1\ ...\ t_n^1\ y, ..., f\ t_1^m\ ...\ t_n^m\ y]\ \mathbf{in}\ f$$
$$\rightarrow$$
$$\lambda x_1.\ ...\ \lambda x_n.\lambda y.\mathbf{let}\ f' = \lambda x_1.\ ...\ \lambda x_n.C\ [f'\ t_1^1\ ...\ t_n^1, ..., f'\ t_1^m\ ...\ t_n^m]\ \mathbf{in}\ f'\ x_1\ ...\ x_n$$

C is a context with m sort of holes $[]_1, \ldots, []_m$, in which holes of each sort may occur more than once, where f and f' do not occur in C, and the parameter y does not get bound during hole-filling.

In order to enhance its application to a bigger class of λ_{letrec}-terms, the second rule can be further generalised to cover situations in which f is only one amongst many functions defined in a `letrec`-construct, or there are also other recursive calls to f that are not of the 'good' form. Namely, if a definition like:

$$f = \lambda x_1.\ ...\ \lambda x_n.\lambda y.C\ [f\ t_1^1\ ...\ t_n^1\ y, ..., f\ t_1^m\ ...\ t_n^m\ y]$$

occurs somewhere in a let-binding, then it can be substituted by:

$$f = \lambda x_1.\ ...\ \lambda x_n.\lambda y.\mathbf{let}\ f' = \lambda x_1.\ ...\ \lambda x_n.C\ [f'\ t_1^1\ ...\ t_n^1\ y, ..., f'\ t_1^m\ ...\ t_n^m\ y]\ \mathbf{in}\ f'\ x_1\ ...\ x_n$$

There might be calls of f in C that are of 'bad' shape. These remain unchanged. On λ_{letrec}-terms, this more general transformation can be expressed by the rewrite rule:

$$\mathbf{let}\ E\ [f = \lambda x_1.\ ...\ \lambda x_n.\lambda y.C\ [f\ t_1^1\ ...\ t_n^1\ y, ..., f\ t_1^m\ ...\ t_n^m\ y]]\ \mathbf{in}\ M$$
$$\rightarrow$$
$$\begin{array}{l}\mathbf{let}\ f = \lambda x_1.\ ...\ \lambda x_n.\lambda y. \\ \quad \mathbf{let}\ E\ [f' = \lambda x_1.\ ...\ \lambda x_n.C\ [f'\ t_1^1\ ...\ t_n^1\ y, ..., f'\ t_1^m\ ...\ t_n^m\ y]]\ \mathbf{in}\ f'\ x_1\ ...\ x_n \\ \mathbf{in}\ M\end{array}$$

where E is a context of the form $D_1, \ldots, D_{i-1}, [], D_i, \ldots, D_l$ consisting of definitions $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_l$ and a single hole $[]$.

4.1 Rewriting the Haskell Prelude

To demonstrate the above rewriting rule, we apply it to straightforward implementations of some well-known functions from the Haskell Prelude.

$$\begin{aligned} &\mathit{map}\ _\ [\,] &&= [\,] \\ &\mathit{map}\ f\ (x : \mathit{xs}) &&= f\ x : \mathit{map}\ f\ \mathit{xs} \end{aligned}$$

$$\mathit{until}\ p\ f\ x = \mathbf{if}\ p\ x\ \mathbf{then}\ x\ \mathbf{else}\ \mathit{until}\ p\ f\ (f\ x)$$

$$\begin{aligned} &(+\!\!+)\ [\,] && \mathit{ys} = \mathit{ys} \\ &(+\!\!+)\ (x : \mathit{xs}) && \mathit{ys} = x : \mathit{xs} +\!\!+ \mathit{ys} \end{aligned}$$

For the optimised counterparts the amount of β-reduction steps saved per recursive call amounts to one for *map* and (++), and to two for *until*.

$$\begin{aligned} \mathit{map}\ f = \mathbf{let}\ &\mathit{rec}\ [\,] &&= [\,] \\ &\mathit{rec}\ (x : \mathit{xs}) &&= f\ x : \mathit{rec}\ \mathit{xs} \\ \mathbf{in}\ &\mathit{rec} \end{aligned}$$

$$\begin{aligned} \mathit{until}\ p\ f\ x = \ &\mathbf{let}\ \mathit{rec}\ x = \mathbf{if}\ p\ x\ \mathbf{then}\ x\ \mathbf{else}\ \mathit{rec}\ (f\ x) \\ &\mathbf{in}\ \ \mathit{rec}\ x \end{aligned}$$

$$\begin{aligned} (+\!\!+)\ \mathit{xs}\ \mathit{ys} = \mathbf{let}\ &\mathit{rec}\ [\,] &&= \mathit{ys} \\ &\mathit{rec}\ (x : \mathit{xs}) &&= x : \mathit{rec}\ \mathit{xs} \\ \mathbf{in}\ &\mathit{rec}\ \mathit{xs} \end{aligned}$$

In practise, however, the amount of β-reductions is only one of many factors for the run time that is necessary to evaluate a piece of code. Therefore it is to be expected that when executed with a system like the Glasgow Haskell Compiler (GHC) the obtained functions would not necessarily lead to better performance. Depending on the compiler version and the flags provided in the invocation of GHC, simple benchmarks yield mixed results, but never resulting in severe degradation (more than an increase of 5% in run time) and with one of the functions (*until*) consistently reaching a speed-up of over 200%.

Let us conclude that before integrating the transformation into a compiler for practical purposes an analysis on how it interacts with other optimisations remains yet to be done.

4.2 Limitations of the Rewrite Rules

While the most general rewrite rule above has proven to be applicable in a number of situations that occur in practice, it also has some severe limitations: First and foremost it applies only to patterns with immediate recursion, thus it fails to capture the repetitive reduction pattern in the evaluation of the term in Fig. 3. If f and g are not used at further positions, once in the recursion initiated by $f\ a$ both x and y will never be bound to a different value than a. The property leading to this behaviour is the relation between the parameters of f and g. In f, if g is called, then x is passed as an argument and thereby bound to y. Conversely, y is bound to x in the call of f in g. Thus, we observe a relation between parameters that is cyclic. Also for all of the previous example such a *parameter cycle* exists, however comprising only a single parameter. In the following section we shall elaborate further on the idea of parameter cycles.

$$\begin{aligned}&\textbf{let}\ f\ x = \ ...\ g\ x\ ...\\&\quad\ \ g\ y = \ ...\ f\ y\ ...\\&\textbf{in}\ ...\ f\ a\ ...\end{aligned}$$

Figure 3: Schematic term involving mutual recursion

5 Binding Analysis

In this section we shall develop an analysis that statically recognises repetitive reduction patterns indicated by parameter cycles, and show how this allows us to eliminate those binders that are part of such a cycle. Parameter cycles describe the possibility of a parameter being passed on from function to function unchanged, finally arriving at its original position. To detect such cycles we need to analyse which subterm might be bound to which variable during the evaluation of a term.

5.1 Binding Graph

To this end we introduce a *binding relation* $\multimap \subseteq V \times T$ on variables V and subterms T of a term. It is a conservative approximation of which bindings might occur during the evaluation of the term and does not distinguish between different descendants of its components.

Since we need to take a global vantage point, i.e. to uniquely identify the term's syntactic elements we assume globally unique variable naming. To this end we could also use positional information, but only along with additional technicalities. Therefore, without loss of generality we henceforth assume that each abstraction binds a distinct variable (variables in the term are 'distinctly bound'), and no variable name has both a free and a bound occurrence (the term observes 'Barendregt's Variable Convention' [3, 2.1.13, p.26]). That allows us to unmistakably address a specific binding λx by the variable x it binds.[1]

For example for some context C in the reduction of term $t = C[(\lambda x.e_1)\ e_2]$ the β-redex $(\lambda x.e_1)\ e_2$ might be contracted and by consequence a descendant of e_2 be bound to an instance of x. Therefore t implies $x \multimap e_2$. This principle carries over to gβ-reduction, and more importantly, to sharing.

In order to obtain the binding relation for a term t one has to identify all terms in argument position that might ever match up with λx in its reduction. (More precisely: terms whose *descendants* might ever match up with *descendants* of λx.) This could be naively accomplished by a search that starts at each abstraction λx and from there travels upwards the spine segment of λx. When an applicator with a as an argument is encountered that matches λx according to the semantics of gβ-reduction $x \multimap a$ is noted down. When a multiplexer is encountered the search is pursued for each of the incoming edges.

The use of higher-order functions makes it impossible to enumerate the binding relation completely. Thus, if during the search the end of the spine is reached one cannot make a safe assumptions on 'future' arguments for that branch. Therefore we employ an additional 'artificial' *blackhole* node $\bullet$ that represents an unknown term on the right hand side of the binding relation. According to this, the occurrence of $e_1\ (\lambda x.e_2)$ implies $x \multimap \bullet$. Note, that the blackhole node is not needed to identify parameter cycles, but is however required for the domination property introduced later on.

Let us revisit the examples presented so far and enlist their binding relation. In the λ-graph of *repeat* (Fig. 1) the search starting from the only abstraction λx branches at the multiplexer above. The left

[1] Even if that forbids distinguishing between different occurrences of the term x, it turns out not to be necessary for our needs.

branch yields a matching application with x in argument position, so we obtain $x \multimap x$. At right branch the spine immediately ends yielding $x \multimap \bullet$. The directed graph that is induced by this relation features a single-node parameter cycle.

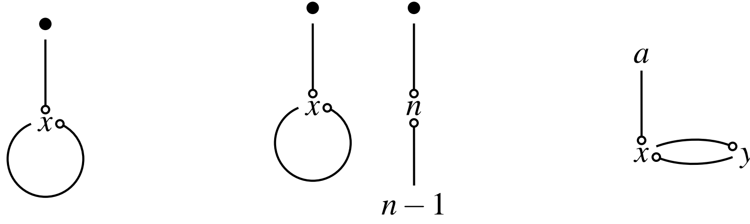


Figure 4: Binding graphs of *repeat*, *replicate*, and the term from Fig. 3

The binding-graph of *replicate* is very similar featuring the same kind of parameter cycle, only does it involve an additional parameter. Also we have to apply the idea of gβ-reduction when searching upwards from λx. First we encounter another abstraction λn, which according to the notion of gβ-reduction requires us to leave out the next application node. In the left branch of the multiplexer this is the one with $n-1$ in argument position. Therefore we end up with x as the argument of the matching application node, by which we obtain $x \multimap x$. Parameter cycles of greater length would occur for scenarios involving mutual recursion such as the term in Fig. 3.

5.2 Inference Rules

To properly define the binding relation we formulate it using inference rules. Since it follows the structure of typing rules, we first give rules for a simply-typed λ_{letrec}-calculus (Fig. 5) and then decorate these typing rules to also yield a term's binding graph, by which we hope to provide an easier access for those who are already familiar with typing rules for the λ-calculus. Type variables are denoted by Greek letters.

$$\textsc{Var}\ \frac{x:\tau \in \Gamma}{\Gamma \Rightarrow x:\tau} \qquad \textsc{Abs}\ \frac{\Gamma \cup \{x:\tau\} \Rightarrow e:\sigma}{\Gamma \Rightarrow \lambda x.e:\tau \to \sigma} \qquad \textsc{Letrec}\ \frac{\forall i \in \{0,\dots,n\}: \Gamma \cup \{f_1:\tau_1,\dots,f_n:\tau_n\} \Rightarrow e_i:\tau_i}{\Gamma \Rightarrow \texttt{letrec}\ f_1 = e_1\ \dots\ f_n = e_n\ \texttt{in}\ e_0\ :\ \tau_0}$$

$$\textsc{App}\ \frac{\Gamma \Rightarrow e_1:\tau \to \sigma \qquad \Gamma \Rightarrow e_2:\rho \qquad \tau = \rho}{\Gamma \Rightarrow e_1\ e_2:\sigma}$$

Figure 5: Typing rules for the simply-typed λ_{letrec}-calculus

In order to infer the binding relation (Fig. 6), at any application $e_1\ e_2$ one has to determine to which variable e_2 might be bound to in e_1. That we accomplish by annotating the types of terms with the names of the variables that are bound by their abstractions. The annotations are in superscript position, so for some annotated types $x:\tau$, $y:\sigma$, $e_3:\rho$, the annotated type of $e_1 = \lambda x.\lambda y.e_3$ is $e_1 : (\tau \to (\sigma \to \rho)^y)^x$. We use ε as an annotation to indicate that we cannot determine the associated variable (due to the use of higher-order functions).

The binding relation is denoted on the left-hand side of the turnstile symbol. Both upper-case and lower-case Greek letters denote annotated type variables, but the latter do not include the outermost annotation (which are then added explicitly).

$$\textsc{Var}\quad \frac{x : T \in \Gamma}{\varnothing \vdash \Gamma \Rightarrow x : T} \qquad \textsc{Abs}\quad \frac{B \vdash \Gamma \cup \{x : \tau^{\varepsilon}\} \Rightarrow e : \Sigma}{B \vdash \Gamma \Rightarrow \lambda x.e : (\tau^{\varepsilon} \to \Sigma)^{x}}$$

$$\textsc{Letrec}\quad \frac{\forall i \in \{0,\dots,n\} : B_i \vdash \Gamma \cup \{f_1 : T_1,\dots,f_n : T_n\} \Rightarrow e_i : T_i}{B_0 \cup \dots \cup B_n \vdash \Gamma \Rightarrow \texttt{letrec}\ f_1 = e_1\ \dots\ f_n = e_n\ \texttt{in}\ e_0 : T_0}$$

$$\textsc{App}\quad \frac{B_1 \vdash \Gamma \Rightarrow e_1 : (T \to \Sigma)^{x} \qquad B_2 \vdash \Gamma \Rightarrow e_2 : R \qquad bare(T) = bare(R)}{B_1 \cup B_2 \cup bind(x, e_2) \cup blackholes(R) \vdash \Gamma \Rightarrow e_1\ e_2 : \Sigma}$$

$$blackholes(T) = \begin{cases} blackholes(R) \cup \{x \circ\!\!- \bullet\} & \text{if } T = (\Sigma \to R)^{x} \\ blackholes(R) & \text{if } T = (\Sigma \to R)^{\varepsilon} \\ \varnothing & \text{otherwise} \end{cases}$$

$$bind(x, e_2) = \begin{cases} \varnothing & \text{if } x = \varepsilon \\ \{x \circ\!\!- e_2\} & \text{otherwise} \end{cases}$$

Figure 6: Inductive definition of the binding relation based on annotated types

In the ABS rule the abstraction puts a new variable x, which is annotated by an ε because we do not make assumptions on the variables exposed by higher-order functions. The type of $\lambda x.e$ is annotated by x, which is exposed. To compare types APP employs a function $bare : A \to T$, which maps annotated types A to bare types T by removing all annotations. Two further functions, *bind* and *blackholes*, are used to enumerate the elements to be added to the binding relation. In case the argument e_2 is a higher-order function the expressions cannot be determined that are bound to the variables it exposes, therefore *blackholes* binds a blackhole to each of them.

Proposition 7 *Every derivation $\mathcal{D}$ in the type system in Fig. 5 with conclusion $\varnothing \Rightarrow M : \tau$, which justifies the assignment of type τ to the λ_{letrec}-term M, can be decorated effectively with as result a derivation $\tilde{\mathcal{D}}$ in the proof system in Fig. 6 with conclusion $B \vdash \varnothing \Rightarrow M : \tilde{\tau}$, by which the binding graph B for M is obtained and justified.*

The proof of this proposition consists in describing an effective algorithm that, given a derivation $\mathcal{D}$ in the type system in Fig. 5, proceeds as follows: First it constructs, in a step by step manner, variable decorations for the types in $\mathcal{D}$ (by concentrating on 'spine loops' of M that correspond to cyclic threads in $\mathcal{D}$, and starting with the decorations at formulas on such threads where the types have minimal length) in such a way that the rule instances are correct for the system in Fig. 6 when the leading binding graph annotations of the form $B \vdash$ are neglected. Second, after the first step is concluded, the algorithm constructs the binding graph annotations according to the rules in Fig. 6 in a top-down manner.

6 Transformation

Using the binding graph we will now develop a more general version of the optimisation previously formulated as rewriting rules restricted to directly recursive functions.

An edge $x \multimapinv e$ in the binding graph of a term t indicates a (possibly infinite) class of gβ-redexes on the infinite unfolding T of t. While all these redexes could be at once contracted on T this does not automatically carry over to its finite representation t.[2] Let us for the present restrict our attention to cases where x has no further incoming edges.

Proposition 8 Let t be a λ_{letrec}-term with infinite unfolding T and a binding graph featuring a node x with a *single incoming* edge $x \multimapinv e$. If we label the transition that contracts all gβ-redexes with involving a λx-abstraction *red*, and the substitution of all occurrences of t by e and the subsequent elimination of all vacuous λx-abstractions *trans*, then the following diagram commutes.

$$\begin{array}{rcl} t & \twoheadrightarrow_{\triangledown} & T \\ trans\downarrow & & \downarrow red \\ t' & \twoheadrightarrow_{\triangledown} & T' \end{array}$$

This follows from the following considerations. Since we assume unique variable naming, the infinitely unfolding t without any renaming is semantics preserving. [8][3] Every occurrence of λv in t gives rise to one ore more gβ-redexes in T each having λv with p as an argument. This follows from $x \multimapinv e$ being the sole incoming edge of x. By both paths from t to T' one finds T' to have the same shape: There are no occurrences of neither λx nor x. Evidently this not a very strong argument and we hope to improve on it by means of higher-order reasoning.

The restriction above to only consider nodes with a single predecessor prevents us from dealing with any of the examples shown so far since they involve cyclic binding graphs. Fortunately, it can be relaxed to a much less restrictive property.

Definition 9 (Domination, and strong domination) Let a $G = \langle V, \rightarrowtail \rangle$ be a directed graph, and u and v be vertices of G. We say that v *dominates* w (v is a *dominator* for w, symbolically: $dom_G(v,w)$) if either $v = w$ or $v \neq w$ and for every path π in G that leads to w but does not contain v it holds that the start vertex of π is reachable from v; more formally, if:[4]

$$v = w \vee \big(v \neq w \wedge \forall u_0, \ldots, u_n \in V \setminus \{v\} \big[u_0 \rightarrowtail u_1 \rightarrowtail \ldots \rightarrowtail u_n = w \implies v \rightarrowtail^* u_0 \big] \big) \quad (1)$$

Note that $v \rightarrowtail^* w$ holds if v dominates w.

And we say that v *strongly dominates* w (v is a *strong dominator* for w, symbolically: $sdom_G(v,w)$) if $v \neq w$ and for every path π in G that leads to w but does not contain v it holds that the start vertex u_0 of π is reachable from v, but does not reside on a common cycle with v, more formally, if:

$$v \neq w \wedge \forall u_0, \ldots, u_n \in V \setminus \{v\} \big[u_0 \rightarrowtail u_1 \rightarrowtail \ldots \rightarrowtail u_n = w \implies v \rightarrowtail^* u_0 \wedge u_0 \not\rightarrowtail^* v \big] \quad (2)$$

[2] In fact, the reduct of an unfolded λ_{letrec}-term does not have to be expressible as a finite term in λ_{letrec} in general.

[3] Even if this has only been proven for μ-unfolding but it is assumed that it also holds for λ_{letrec}. The unfolding does *not* preserve unique naming.

[4] The condition 1 could be simplified by taking it to be just the subformula starting with the universal quantification (that subformula is true if $v = w$); the longer condition is used here to increase readability.

Remark 10 The standard definition of a 'v dominates w' for control-flow graphs (see e.g. [9]) requires that each path from the start node to w has to pass through v. The definition above is a generalisation to directed graphs that does not depend on the existence of a designated start node. Our definition of 'v strongly dominates w' excludes self-domination (i.e. makes the relation irreflexive), and adds the restriction that for all paths from v to w that do not repeatedly pass through v it holds no vertex except the starting vertex v resides on a common cycle with v.

The following proposition suggests an alternative, co-recursive definition of strong domination between vertices, which proceeds stepwisely by examining predecessors of the strongly dominated vertex.

Proposition 11 Let $G = \langle V, \rightarrowtail \rangle$ be a directed graph. Then for all $v, w \in V$ it holds:

$$sdom_G(v,w) \iff v \neq w \wedge v \rightarrowtail^+ w \wedge w \not\rightarrowtail^+ v \wedge \forall u \in V \big(u \rightarrowtail w \wedge u \neq v \Rightarrow sdom_G(v,u) \big)$$

For the definition of the optimising transformation, we choose a formulation different from the one using rewrite rules described in Section 4, one that is particularly easy to express. Such as β-reduction can be decomposed into substitution of individual occurrences of variables (local β-reduction) and the elimination of vacuous bindings (AT-removal) as described in [4], for the transformation we have the possibility of expressing the transformation with an arbitrary level of granularity. The following rule encompasses the substitution of all occurrences of one dominated variable. It could have been formulated more fine-grained by substituting only individual occurrences, or less fine-grained by including the elimination of the bindings that have become vacuous.

Proposition 12 (Main proposition) *In a λ_{letrec}-term t occurrences of a variable that is dominated by an expression d in t's binding graph can be substituted by d.*

$$\frac{B \vdash \varnothing \Rightarrow t : \Sigma \qquad B \Rightarrow dom_B(d,x)}{t =^{\infty}_{\triangledown, g\beta} t\langle x := d \rangle}$$

7 Advanced Examples

There are interesting examples to which the presented transformation cannot be directly applied or does not lead to satisfactory results. Only in combination with a number well-directed unfoldings the desired effect can be obtained. Let us consider two schematic examples:

$$\lambda y. \ldots \textbf{let}\ rec = \lambda x.C\ [rec\ x]\ \textbf{in} \ldots rec \ldots$$
$$\textbf{let}\ f = \lambda x.\lambda y.D\ [f\ x\ b, f\ a\ y]\ \textbf{in} \ldots f \ldots$$

The first one exhibits dominated parameters only after a single let-unfolding (Fig. 7). Also it features repetitive reduction pattern, without having a parameter cycle. This because the argument in question is bound outside of the recursion. Still, after the unfolding the presented optimisation does cure the term.

The second of the two term is particularly intricate since it can be unfolded and then transformed in many different ways with considerable differences in the amount of concealed gβ-redexes. Most solutions involve more than one recursive call with more than two arguments. The ideal transformation with respect to the number of concealed redexes of both terms is depicted in Fig. 8.

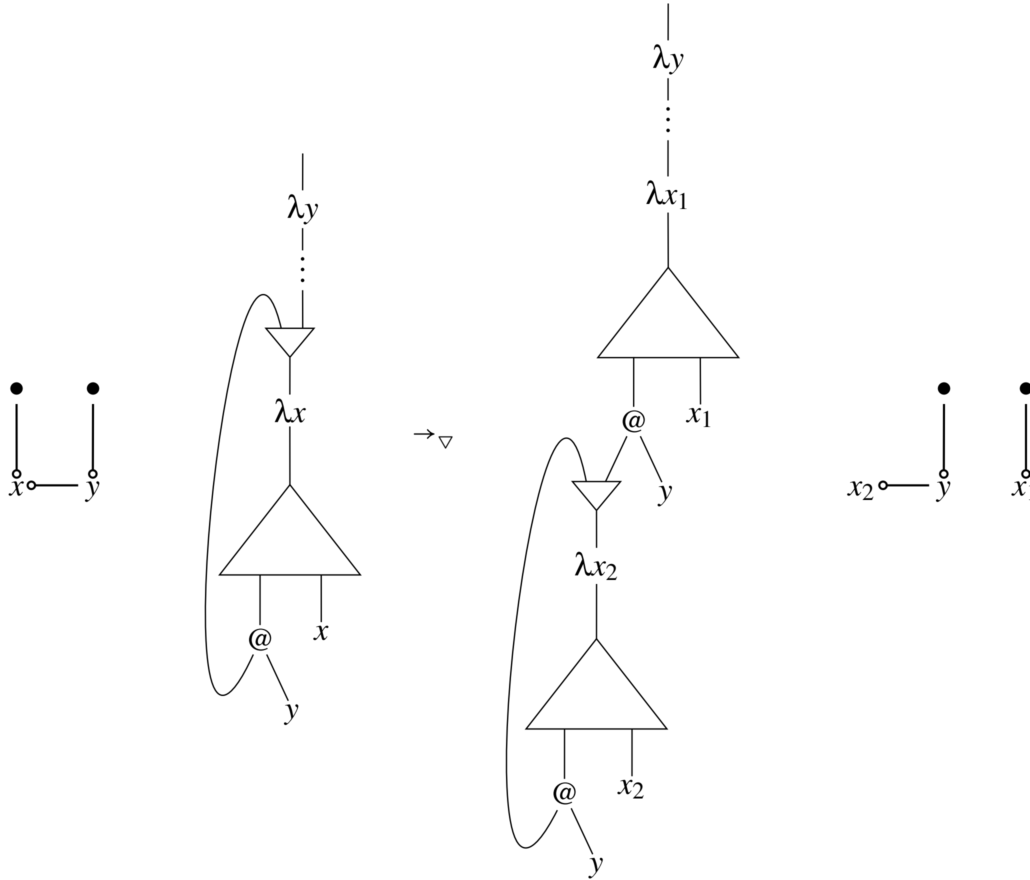


Figure 7: A term, which after one unfolding exhibits a dominated parameter

8 Status quo

Presently this work consists mostly of an extended problem description, which is to a some extent rather informal. Ideas for resolution have been presented, but yet lack both precision and genericity. Therefore, outstanding issues to continue the investigation suggest themselves. Currently we are working on the following problems:

- Properly formalising the concepts introduced in this report. This involves putting grammars and rewrite rules into the higher-order setting of HRSs.
- Investigate whether there are other interesting notions of ‘operational equivalence’, apart from applicative bisimulation, with respect to which we could show correctness of our optimising transformation. (We think of notions that are in line with the semantics of functional programming languages, but nevertheless are language independent, and also of theoretical interest.)
- Expressing the presented transformation as a higher-order rewrite system and proving its correctness with respect to that equivalence relation.

Once these fundamental issues are resolved, we intend to tackle the following questions:

- We have seen that unfolding a λ_{letrec}-term in order to facilitate the optimisation can be effected in

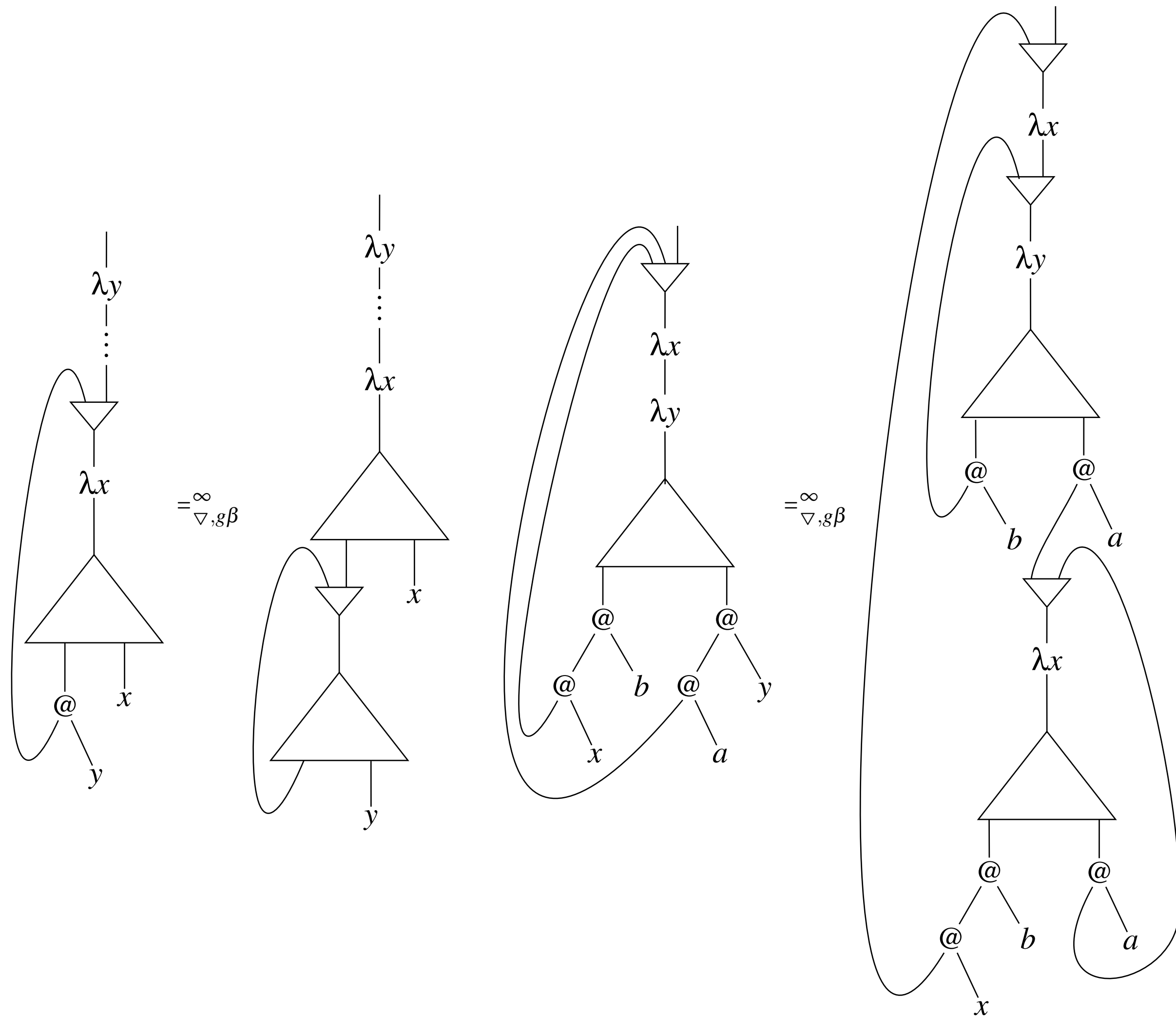


Figure 8: Optimisation of two terms that are not covered by the presented rewriting rules

different ways, leading to terms of different quality. To obtain the most efficient terms one has to provide a procedure to select the most suitable unfolding for each situation.

- This requires an adequate measure for efficiency.
- We would like to learn more about the rewrite properties of the transformation: is it possible to find a formulation that guarantees confluence and normalisation?
- In some of the easy examples we studied, our transformation seemed to be closely connected with the concept of 'lambda-dropping' [7, 5], and hence also with its converse, 'lambda-lifting' [10, 12, 6]. We want to understand that relationship in detail.
- Once the analysis has been optimised in terms of genericity, i.e. that it recognises as many cases as possible for which the transformation is correct, it would be interesting to assess the frequency in which these cases occur in existing systems, such as functional programming libraries or intermediate code generated by compilers.

- The remaining question is, how the optimisation actually affects the run-time efficiency of real-world systems like Haskell programs.

Acknowledgment. The incentive to investigate the presented optimisation was provided by Doaitse Swierstra. We thank him and Vincent van Oostrom for many insightful discussions and hints.

References

[1] Samson Abramsky (1990): *The Lazy Lambda Calculus*. In: *Research Topics in Functional Programming*. Addison-Wesley, pp. 65–116. Updated version (2006) available at `http://web.comlab.ox.ac.uk/people/Samson.Abramsky/lazy.pdf`.

[2] Andrea Asperti & Stefano Guerrini (1998): *The Optimal Implementation of Functional Programming Languages*. *Cambridge Tracts in Theoretical Computer Science* 45, Cambridge University Press.

[3] H.P. Barendregt (1984): *The Lambda Calculus: Its Syntax and Semantics*, 2nd edition. *SLFM* 103, Elsevier.

[4] N.G. de Bruijn (1987): *Generalizing Automath by Means of a Lambda-Typed Lambda Calculus*. Technical Report AUT092 (AUTOMATH archive `http://www.win.tue.nl/automath/`), Technische Universiteit Eindhoven, Eindhoven, the Netherlands. Available at `http://alexandria.tue.nl/repository/freearticles/597608.pdf`.

[5] Olivier Danvy (1999): *An Extensional Characterization of Lambda-Lifting and Lambda-Dropping*. In Aart Middeldorp & Taisuke Sato, editors: *Functional and Logic Programming*. *LNCS* 1722, Springer Berlin / Heidelberg, pp. 241–250, doi:10.1007/10705424_16.

[6] Olivier Danvy & Ulrik Schultz (2002): *Lambda-Lifting in Quadratic Time*. In Zhenjiang Hu & Mario Rodríguez-Artalejo, editors: *Functional and Logic Programming*. *LNCS* 2441, Springer Berlin / Heidelberg, pp. 134–151, doi:10.1007/3-540-45788-7_8.

[7] Olivier Danvy & Ulrik P. Schultz (1997): *Lambda-dropping: transforming recursive equations into programs with block structure*. In: *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. PEPM '97, ACM, New York, NY, USA, pp. 90–106, doi:10.1145/258993.259007.

[8] Jörg Endrullis, Clemens Grabmayer, Jan Willem Klop & Vincent van Oostrom (2010): *On Equal μ-Terms*. Submitted, currently under review, to be published in 2011.

[9] M. S. Hecht & J. D. Ullman (1974): *Characterizations of Reducible Flow Graphs*. *JACM* 21, pp. 367–375, doi:10.1145/321832.321835.

[10] Thomas Johnsson (1985): *Lambda lifting: Transforming programs to recursive equations*. In Jean-Pierre Jouannaud, editor: *Functional Programming Languages and Computer Architecture*. *LNCS* 201, Springer Berlin / Heidelberg, pp. 190–203, doi:10.1007/3-540-15975-4_37.

[11] Fairouz Kamareddine & Rob Nederpelt (1995): *Refining reduction in the lambda calculus*. *Journal of Functional Programming* 5(4), pp. 637–651, doi:10.1017/S0956796800001507.

[12] Simon L. Peyton Jones (1987): *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[13] Terese (2003): *Term Rewriting Systems*. *Cambridge Tracts in Theoretical Computer Science* 55, Cambridge University Press.