Electronic Notes in Theoretical Computer Science

Computing with terms and graphs

TERMGRAPH 2007

Preliminary Proceedings

Braga, Portugal

31 March 2007

Guest Editors:

Ian Mackie and Detlef Plump

Contents

Preface

The *Fourth International Workshop on Computing with Terms and Graphs* (TERMGRAPH 2007) was held in Braga, on Saturday 31 March 2007, as a satellite event of the 10th European Joint Conference on Theory and Practice of Software (ETAPS 2007). The first TERMGRAPH workshop took place in Barcelona, in 2002, as a satellite event of the International Conference on Graph Transformation (ICGT), the second TERMGRAPH workshop took place in Rome 2004, also as a satellite event of ICGT, and the third in Vienna, as a satellite event of ETAPS 2006.

The advantage of computing with graphs rather than terms is that common subexpressions can be shared, improving the efficiency of computations in space and time. Sharing is ubiquitous in implementations of programming languages: many functional, logic, object-oriented and concurrent calculi are implemented using term graphs. Research in term and graph rewriting ranges from theoretical questions to practical implementation issues. Different research areas include: the modelling of first- and higher-order term rewriting by (acyclic or cyclic) graph rewriting, the use of graphical frameworks such as interaction nets and sharing graphs (optimal reduction), rewrite calculi for the semantics and analysis of functional programs, graph reduction implementations of programming languages, graphical calculi modelling concurrent and mobile computations, object-oriented systems, graphs as a model of biological or chemical abstract machines, and automated reasoning and symbolic computation systems working on shared structures.

The aim of this workshop is to bring together researchers working in these different domains and to foster their interaction, to provide a forum for presenting new ideas and work in progress, and to enable newcomers to learn about current activities in term graph rewriting.

Topics of interest include all aspects of term graphs and sharing of common subexpressions in rewriting, programming, automated reasoning and symbolic computation. This includes (but is not limited to): term rewriting, graph transformation, programming languages, models of computation, graph-based languages, semantics and implementation of programming languages, compiler construction, pattern recognition, databases, bioinformatics, and system descriptions.

For TERMGRAPH 2007, the Programme Committee selected 10 papers for inclusion in these proceedings, covering a wide range of the topics.

The Programme Committee consisted of:

- Zena Ariola, University of Oregon, USA
- Andrea Corradini, University of Pisa, Italy
- Maribel Fernández, King's College London, UK
- Bernhard Gramlich, Vienna University of Technology, Austria
- Annegret Habel, University of Oldenburg, Germany

- Claude Kirchner, INRIA & LORIA, Nancy, France
- Jean-Jacques Lévy, INRIA, Rocquencourt, France
- Ian Mackie, King's College London & École Polytechnique (Co-Chair)
- Aart Middeldorp, University of Innsbruck, Austria
- Ugo Montanari, University of Pisa, Italy
- Jorge Sousa Pinto, University of Minho, Braga, Portugal
- Detlef Plump, University of York, UK (Co-Chair)
- Arend Rensink, University of Twente, The Netherlands

We would like to thank all those who contributed to TERMGRAPH 2007. We are grateful to the Programme Committee members for their careful and efficient work in reviewing the submitted papers and selecting the workshop programme.

Ian Mackie and Detlef Plump

1 March 2007

Term-graph rewriting in Tom using relative positions

Emilie Balland and Paul Brauner

UHP & LORIA, INPL & LORIA
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex France

Abstract

In this paper, we present the implementation in Tom of a de Bruijn indices generalization allowing the representation of term-graphs over an algebraic signature. By adding pattern matching and traversal controls to Java, Tom is a well-suited environment for defining program transformations or analyses. As some analyses, e.g. based on control flow, require graph-like structures, the use of this formalism is a natural way of expressing them by graph rewriting.

Key words: term-graph,rewriting,strategic programming

1 Introduction

Program transformation and graph rewriting are strongly related [10]. Indeed, although the structure of a program may be represented by a tree, informations about its execution like data dependencies or control flow are naturally expressed by data-structures inherently using cycles or subterms sharing, in other words by graphs. More precisely, since these graphs are oriented and labelled over an algebraic signature, such transformations are described within the framework of term-graphs [13]. There exists several definitions of term graph rewriting, category-theory oriented [7,11], equationally oriented [2] or implementation-oriented [3].

Since 2001, the Protheo team has been developing the Tom system [12], whose main originality is to be built on top of an existing language Java. Tom provides pattern matching facilities to inspect objects and retrieve values. Moreover, the rewriting steps can be controlled using a powerful strategy language. The main application of the language being program transformation and code analysis, we were interested in extending the Tom language for supporting term-graph transformations.

In this paper, we introduce the notion of relative position inspired from the de Bruijn indices as a way to express paths between two subterms. Then we

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

present an implementation of term-graphs based on this formalism. As Tom provides rewriting strategies, integrating such structures in the language offers strategic graph rewriting for free. After introducing the notion of relative positions, we will explain how the language can be extended to offer facilities for strategic graph rewriting. Finally, we will illustrate the use of this extension by an implementation of lambda-calculus normalization.

2 Term-graph representation

Our goal is to represent term-graphs on top of the term rewriting theory with the fewest possible modifications to this formalism to take advantage of the existing results (confluence, termination) and tools, namely Tom. The main idea of this paper is to raise the notion of position to the level of first-order terms by extending algebraic signatures with an infinite set of constants representing positions. This allows for the description of terms containing some "pointers" to subterms of themselves. As an example, the term $s(a, 1)$ defined over such a signature denotes a term whose second child references the first-one.

The main issue of this representation is that it is context-sensitive. For instance, the position 1.1 references the subterm a in $f(s(a, 1.1))$, but $s(a, 1.1)$ in $f(f(s(a, 1.1)))$. This raises the idea of *relative positions* describing paths inside a term to the referenced subterms. The previous example would then be written $f(s(a, -1.1))$, where -1 indicates one backward step inside the term. This can be seen as a generalization of de Bruijn indices extended to the count of all function symbols, not only abstractions.

In this section, we define more formally this notion of relative position and terms with references before we present an implementation aimed to be used by Tom. We finally discuss the relation between this formalism and term-graphs as well as the associated technical solution.

2.1 Terms with references

As usual, a *position* is a finite sequence of natural numbers. The subterm u of a term t at position ω is denoted $t_{|\omega}$, where ω describes the path from the root of t to the root of u. To emphasize the difference with relative positions, we will sometimes refer to positions as *absolute positions*.

Let us first define relative positions along with their meaning.

Definition 2.1 (Relative position) *The set Rpos of relative positions is the monoid $(\mathbb{Z}^*, .)$ with neutral element Λ where $\mathbb{Z}^* = \mathbb{Z} \setminus \{0\}$.*

We note n, p the elements of $\mathbb{Z}^*$ and ω_r, ω'_r, ... the elements of *Rpos*.

Definition 2.2 (Referenced subterm) *Given an absolute position ω and a relative position ω_r, the absolute position accessed by ω_r from ω is written $pos(\omega, \omega_r)$ and is defined as follows:*

- *if* $\omega_r = \Lambda$, *then* $pos(\omega, \omega_r) = \omega$
- *else, there exists* $p \in \mathbb{Z}^*$ *and* $\omega'_r \in Rpos$ *such that* $\omega_r = p.\omega'_r$ *and*
  - *if* $p > 0$, *then* $pos(\omega, \omega_r) = pos(\omega.p, \omega'_r)$
  - *if* $p < 0$ *and if there exists* ω' *and* ω'' *such that* $\omega = \omega'.\omega''$ *and* $|\omega''| = -p$, *then* $pos(\omega, \omega_r) = pos(\omega', \omega'_r)$

It is undefined everywhere else.

We note $t_{|\omega,\omega_r}$ *the term* $t_{|pos(\omega,\omega_r)}$ *for every* ω *and* ω_r *such that* $pos(\omega, \omega_r)$ *and* $t_{|pos(\omega,\omega_r)}$ *are defined. We name it the subterm of* t *referenced by* ω_r *from* ω.

Intuitively, ω_r describes a path back and forth inside t from ω to $t_{|\omega,\omega_r}$. For example, the relative positions -1.1 and $-2.1.2.-1.1$ reference the same subterm a of $f(s(a, b))$ from the position 1.2.

We can now define the notion of first-order terms with references. It only consists in extending an algebraic signature with an infinite set of constants denoting relative positions.

Definition 2.3 (Term with references) *For every set of first-order terms* $\mathcal{T}(\mathcal{F}, \mathcal{X})$, *the corresponding set of terms with references* $\mathcal{T}_{ref}(\mathcal{F}, \mathcal{X})$ *is the set* $\mathcal{T}(\mathcal{F} \cup Rpos, \mathcal{X})$ *where elements of Rpos have arity* 0.

As an example, $f(s(a, -1.1))$ is a term with references of $\mathcal{T}_{ref}(\{f, s, a\}, \emptyset)$. By abuse of notation, we will say that "-1.1 references a in $f(s(a, -1.1))$", without specifying it occurs at position 1.2.

Problems will inevitably occur when considering undefined relative positions. We define therefore validity as follows.

Definition 2.4 (Term with references validity) *A term with references* $t \in \mathcal{T}_{ref}(\mathcal{F}, \mathcal{X})$ *is valid if for every leaf* $\omega_r = t_{|\omega}$ *such that* $\omega_r \in Rpos$, $t_{|\omega,\omega_r}$ *is defined and is not in Rpos.*

Notice that we forbid relative positions referencing relative positions.

2.2 Implementation of terms with references

Let us now see how this formalism can be transposed to the Tom language. One characteristic of Tom is its data-structure independence. A term can be represented by any Java object as long as the user provides a mapping to see these objects as trees. For easier development, it comes up with a language called Gom [14] which automatically generates from a signature the Java implementation and the mapping. The resulting implementation is efficient in space and time (constant time terms equality test) because of maximal subterm sharing. Readers must pay attention to the difference between the maximal sharing and the notion of sharing in term-graphs. In our case, the maximal sharing is only at implementation level and does not lead to sharing at the term level. A Gom signature contains sorts and their constructors. For example, the signature below defines two sorts `A` and `B` along with their

constructors.

```
A = a()                    B = g(A)
  | f(A)
  | s(A,A)
```

With this signature, we can construct the terms `a()`, `f(a())` or `g(f(a()))` for instance. Our goal is to generate an extended signature for terms with references from an initial Gom one. To achieve this, for every sort `T` of a Gom module, we generate a new constructor of rank `posT(int*)`. The notation `*` is the same as in [4, Section 2.1.6] and can be seen as a family of constructors with arities in $[0, \infty[$. The previous example is extended in this way:

```
A = a()                    B = g(A)
  | f(A)                     | posB(int*)
  | s(A,A)
  | posA(int*)
```

As an example, we can now build the extended term $s(-1.2.1, f(a))$ with the following syntax: `s(posA(-1,2,1),f(a()))`. Then `posA(-1,2,1)` references `a()` in the term `s(posA(-1,2,1),f(a()))`.

This type of terms with references using explicit relative positions constitutes a first extension of a Gom signature. In order to ensure type-preservation and reference correctness, a second representation level consists in expressing references with the help of labels. This notion of labelling can be seen as an implementation of the addressed terms presented in [5]. We have added new constructors to facilitate the use of labels and functions to transform a term with labels into the low-level representation. For every sort `T`, we generate two constructors. The constructor `labT(String,T)` enables the user to label a term with a string and `refT(String)` to reference a labelled term. Thus the term `s(refA("l"),f(labA("l",a())))` corresponds to the low-level term `s(posA(-1,2,1),f(a()))`. This notion of labels can be seen as syntactic sugar for hiding positions to users in order to avoid bad manipulations. Thereby, the constructors `posT` should be private so that users can only construct terms with references by label usage. We provide functions which generate the corresponding low-level terms after verifying that each `refT` corresponds to a `labT` of identical sort. This transformation is itself described using strategic rewriting introduced in section 4.

2.3 Correspondence with term-graphs

Let us see now how a representation of cyclic term-graphs (in the sense of [2] for instance) can be obtained from the terms with references introduced above. For example, the term-graph rooted by `s` whose two children correspond to the shared subterm a may be represented by $s(a, -1.1)$. It may also be represented by $s(-1.2, a)$ though, so we need to define canonical forms. Moreover, we noticed that several relative positions may reference the same subterm from a

given position. Hence, we define canonical relative positions.

Definition 2.5 (Canonical relative position) *Let ω_1, ω_2 be two absolute positions, the canonical relative position $cpos(\omega_1, \omega_2)$ from ω_1 to ω_2 is the smallest relative position with respect to the length such that $pos(\omega_1, cpos(\omega_1, \omega_2)) = \omega_2$.*

Let us remark that $cpos(\omega_1, \omega_2) = q.\omega'$ where $\omega' \in (\mathbb{N}^*, .)$ and $q \in \mathbb{Z}^* \cup \{\Lambda\}$. We can now define the canonical form of terms with references using an order on absolute positions.

Definition 2.6 (Canonical term with references) *Let $\omega_1 = n_1.\omega_1'$ or Λ and $\omega_2 = n_2.\omega_2'$ or Λ be two different absolute positions,*

$$\omega_1 <_\Omega \omega_2 \Leftrightarrow \begin{cases} & \omega_1 = \Lambda \\ or & n_1 < n_2 \\ or & n_1 = n_2 \text{ and } \omega_1' <_\Omega \omega_2' \end{cases}$$

A term t with references is then canonical if and only if t is valid and for every leaf $\omega_r = t_{|\omega}$ such that $\omega_r \in Rpos$, ω_r is canonical and $pos(\omega, \omega_r) <_\Omega \omega$.

Typically, contrary to $s(-1.2, a)$, the term $s(a, -1.1)$ is a canonical representation of a term-graph.

The formalism presented all along this section has been implemented through a plugin for Gom which generates an extended signature with new constructors for positions and construction functions which offer different levels of abstractions (from terms with explicit positions to term-graphs with labels). As illustrated by the Figure 1, a user may provide a labelled representation which is not a canonical form and use the provided construction function to normalize it. Whatever the favored level of the user,

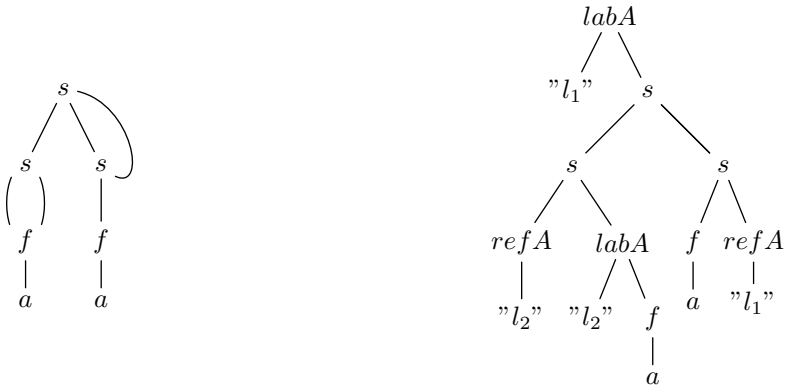


Fig. 1. An example of term-graph and its representation as a labelled term.

the in-memory representation is always based on explicit relative positions. Moreover, due to Gom design and in particular to the maximal sharing, the efficiency in time and space is ensured. For example, the term-graph

presented Figure 1 is automatically translated during the construction into the low-level term with positions depicted in Figure 2. The principle of maximal sharing is also illustrated by a schematic representation of the heap.

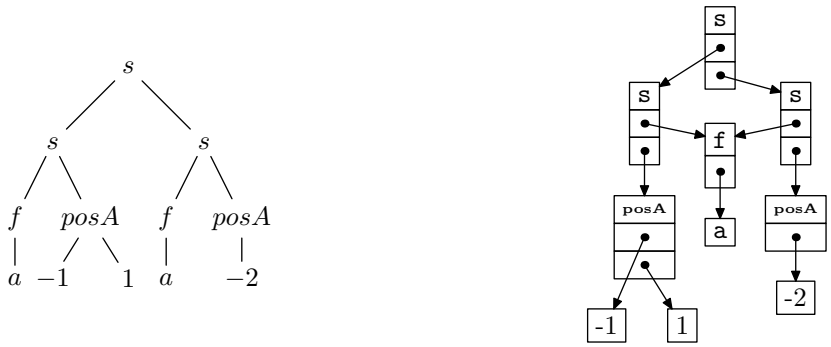


Fig. 2. Generation of relative positions from the labelled representation and maximal subterm sharing in memory.

After defining terms with references rewriting, we will exhibit in the next two sections how the Tom language offers strategic rewriting of these structures.

3 Term-graph matching

The originality of the previous approach is that pattern matching on terms with references built upon $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is simply defined as pattern matching on terms of $\mathcal{T}_{ref}(\mathcal{F}, \mathcal{X})$. There is therefore no need to extend the notion of rewriting, which allows us to reuse existing results and rewriting tools. However, the questions raised by this formalism are situated at another level: we would like the rewrite system to rewrite only valid terms. Giving some non-trivial criterion on rewrite rules implying this property remains an open question for the moment. The next sections of this paper therefore focus on technical aspects of the pattern matching problem implementation.

After introducing the Tom language, we discuss various presentations of graph with references rewriting in this system. Although we cannot statically check that patterns ensure the validity of matched terms, we also propose several solutions to check this property at runtime.

3.1 Tom *pattern matching*

The first mechanism offered by the Tom language is pattern matching on algebraic terms. This feature is similar to the constructs proposed by functional languages like OCaml or Haskell. It is enabled by the `%match` keyword which allows us to match a subject against some pattern and to get the values of the pattern variables into Java ones:

```
A term = `s(f(a()),a());
```

```
%match(term) {
  s(x,y) ->  {
    System.out.println(
      "First child: " + ‘x + ", second child: " + ‘y
    );
    return ‘f(x);
  }
}
```

A subject is then any Java object which is an instance of a class whose description has been provided to Tom via a *mapping*. This mapping indicates to the Tom compiler how to match some class against a pattern, and how to create new algebraic terms implemented by this class via the ‘ construct. Here we are using the classes generated by Gom along with their mappings. Tom also supports associative matching, a.k.a. list matching, as well as anti-patterns [9] and non-linear matching.

Let us elaborate on the mapping mechanism. It provides an algebraic view of some Java object (*e.g.* seeing integers as Peano natural numbers, or seeing an XML tree as a term). It is divided into two parts: the *destructive* part and the *constructive* one. The destructive part is used by the matching algorithm and its main function is to describe how to query a term about its head symbol and how to get its n^{th} child. For instance, the mapping between integers and Peano naturals would be similar to the following schematic code:

```
is_zero(n)                { n == 0 }
is_successor(n)           { n > 0 }
get_successor_child(n)    { n - 1 }
```

On the other hand, the constructive part is used by the compiler to build an algebraic term. It usually consists in calling the constructor of the Java class implementing the term. Although our goal is to work as much as possible on top of classes and mappings generated by Gom, we will punctually adapt some mapping to our needs.

3.2 Matching terms with references

Given these language constructs and the terms described in Section 2.2, there are many ways to express matching against patterns with references. As for term construction, patterns can be expressed at low-level using directly positions or by a syntax based on labelling. In each case, it refers to a stated subterm whose position is well-known. To compare two references by value instead of references, we will introduce a `deref` operator in patterns implemented using Tom mappings.

The simplest way to handle Gom terms with references is to consider the extended signature and perform some standard pattern-matching on it. Since

the `posT(int*)` constructors generate matchable terms, it is possible to write patterns where relative positions are explicitly given. As an example, the term represented Figure 3 matches against the pattern `s(a(),pos(-1,1))`. Notice that this type of pattern denotes exactly the structure of the term: *e.g.* `s(pos(-1,2),a())` would not match the same term. This method allows us to match against any position, even those pointing to an upper term as shown Figure 4. This may still be relevant in case of a procedure carrying some

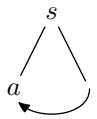


Fig. 3. `s(a(),pos(-1,1))`

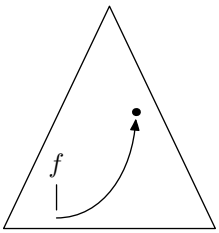


Fig. 4. `f(pos(-n,...))`

contextual information or fetching the position to perform some computation later. It may also be useful to compare two positions without knowing the value of the subterms they are referencing. Figure 5 illustrates this situation. Notice however that this is only possible if the two variables have the same height in the term, as we are comparing *relative* positions.

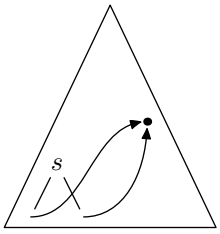


Fig. 5. `s(x,x)`

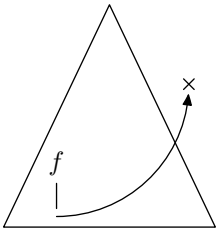


Fig. 6. `f(pos(-n,...))`

This first simple manner of matching graphs with references presents two issues: the main one, depicted by Figure 6, is that a relative position may be undefined. These patterns should therefore be considered as a kind of unsafe assembly language for matching terms with references. The second one is that the explicit notation of positions is not mandatory and may be easily avoided with some syntactic sugar.

Thereby we propose to slightly modify the Tom compiler to address them. The first change consists in integrating labels capturing and denoting positions of subterms into the patterns syntax in order to avoid any explicit position matching. As an example, the term represented in Figure 3 would match against the pattern `s(x:a(),x)`. The translation of this kind of patterns to the former one is trivial: each occurrence of a label `lab` is replaced by the relative position from its position to the position of the subterm labelled by `lab`.

The second modification aims at reinforcing the patterns safety. As explained in section 2.2, we do not want the user to be able to recover a position by matching the term of figure 3 against `s(_,x)` for instance. This can be achieved by inhibiting the generation of mappings for position constructors, so that the matching algorithm fails on such patterns. Another less restrictive way of dealing with the undefined relative positions problem would be to have the patterns similar to `s(_,x)` match only valid terms. This could be achieved by checking at runtime that every relative position in `x` references an accessible term. This is easily done with the help of strategies presented in section 4. In both cases, we cannot avoid some modifications of the pattern-matching algorithm, thus of the compiler.

The two previous kinds of patterns focus on the positions themselves as matchable objects. Another approach would be to have the patterns express constraints about the value of the referenced subterms. The mapping mechanism presented in Section 3.1 offers the necessary features to achieve this via the writing of an *ad hoc* destructor. We wrote this `deref` destructor which acts like a proxy between the pattern matching algorithm and the destructor of the value referenced by a position. As an example, the term represented by

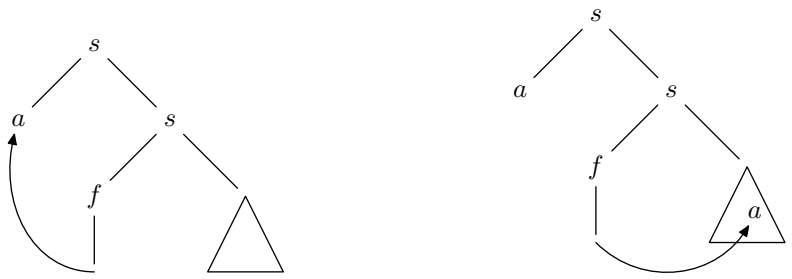


Fig. 7. `deref(a())` ambiguity

Figure 3 matches against the pattern `s(a(),deref(a()))`. It is important to note that the patterns are now an abstraction of the term so we do not match the graph *structure* anymore. For instance, the two terms of Figure 7 match against the same pattern `s(a,s(f(deref(a())),_))`. In particular, it is not possible anymore to use non-linear pattern matching in order to check that two positions are referencing the same sub-term, as depicted by Figure 8 which shows the ambiguity of the `s(s(deref(x),deref(x)),_)` pattern. Again, matching terms with references in this way is not safe. Indeed the subject may contain positions referencing terms above its root. However this time, checking the validity of a term does not require any change to the compiler since the test can be transfered to the destructor. The latter aborts the matching process by returning `false` if accessing the pointed term raises an exception.

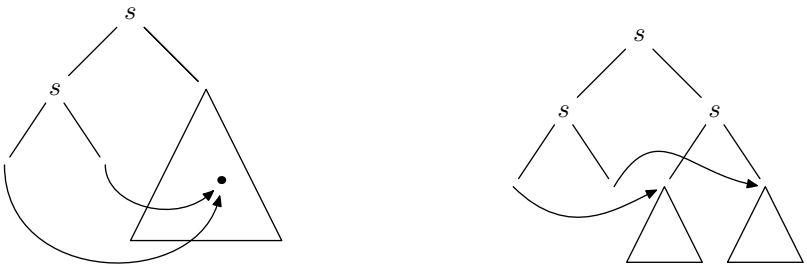


Fig. 8. `deref(x),deref(x)` ambiguity

3.3 Matching term-graphs

Contrary to Gom terms with references, the usual term-graph definition does not differentiate two types of children. Therefore, it may be convenient to have the patterns `s(x:a(),x)` and `s(x,x:a())` match either `s(a(),pos(-1,1))` or `s(pos(-1,2),a())`. The normal form mentioned in Section 2.2 enables such a feature: it is sufficient to maintain normalization of both terms at runtime and patterns at compile time to ensure this behavior. It requires some minor changes of the Tom compiler though.

As recalled in Section 2.2, one main application of term-graphs is the representation of subterms sharing in the purpose of gaining space and computation time. However, this structure (the sharing) does not reflect the structure of the represented term (typically a λ-term) and it is therefore desirable to manipulate it modulo this encoding. The basic idea is to interweave `deref` constructors inside the patterns, so that `s(a(),a())` is translated into `deref(s(deref(a()),deref(a())))` and thus matches the graph of figure 3. It only requires to confer some lazy behavior to the `deref` destructor, which should act as if not existing in case of a direct subterm (not a position).

Even if the classical [3] representation of term-graphs by a labelled graph is similar to ours, the conditions on rewrite rules are more restrictive (the left-hand side of a rule is limited to trees). For now, term-graph rewriting in Tom is expressed by syntactic term rewriting. Contrary to [3], there is no garbage collection phase and referenced subterms can disappear or change, leading to invalid terms. One solution would be to integrate this garbage collection phase in the Tom matching. An other attractive approach would be to implement the formalism presented in [6] where the right-hand side of the rewriting rules consists in a set of actions on the pointers.

4 Strategic programming with term-graphs

Tom provides a powerful strategy language inspired by ELAN and Stratego. The purpose of strategies is to describe how transformation rules should be applied. In case of terms with references, the strategy language must be extended in such a way that we can traverse them as graphs.

4.1 Tom *strategy language*

Elementary strategies are composed of the two basic strategies `Identity()` and `Fail()` as well as type-preserving user-defined rewrite rules specializing their behaviour:

```
%strategy Eval() extends Fail() {
  visit A {
    s(x,a()) -> { return `f(x); }
    s(x,y)   -> { return `y; }
  }
}
```

When applied to a node of sort `A`, a transformation is performed if one of the patterns matches the node. Otherwise, the default `Fail` strategy is applied.

More complex strategies can be built on top of elementary ones, involving basic combinators introduced in ELAN [8] and extended in [15]: `Sequence(s1,s2)`, `Choice(s1,s2)`, `All(s)`, `One(s)`, *etc.* We can therefore build strategies such as `` `Choice(Eval(),Identity()) `` which tries to apply `Eval()` to the current node and returns it unchanged if `Eval()` failed (*i.e.* none of the patterns matched the current node).

Besides, the strategy language allows the declaration of recursive parametrized strategies, enabling the definition of higher-level constructs. For example, the fix-point operator can be expressed by `Repeat(s)` $\triangleq \mu$`x.Choice(Sequence(s,x), Identity())`, where μ denotes a recursion operator, `x` a variable, and `s` a parameter of the strategy. In Tom, we raised the recursion operator to the object level, allowing the definition of complex strategies in a truly algebraic manner:

```
Strategy Repeat(Strategy v) {
  return `mu(MuVar("x"),
             Choice(Sequence(v,MuVar("x")),Identity()));
}
```

Finally, Gom generates a congruence strategy `_f` for each constructor `f` of an algebraic signature. Using the notation `s[t]` to express the application of the strategy `s` to the term `t`, `_f(`s_1`,...,`s_n`)[f(`c_1`,...,`c_n`)]` returns `f(`s_1`[`c_1`],...,`s_n`[`c_n`])` and fails if the head symbol of the subject is not `f`. This allows to perform pattern matching "on the fly" during term traversal.

One noticeable property of strategic programming with Tom is that it is possible to get the current absolute position inside the visited term during a traversal. This allows for instance to collect in one pass the set of reduced forms of a term for a given rewrite system. In our case, we will make use of this feature in the next section to collect the positions of bounded variables occurences under an abstraction.

4.2 Extension of Tom strategy language

In order to traverse terms with references, we enrich the strategy language of Tom with one new strategy combinator `Ref` whose semantics is defined as follows:

$$\texttt{Ref(s)[t]} = \begin{cases} \texttt{s[t']} \text{ if } \texttt{t'} \text{ is the term referenced by } \texttt{t} \\ \texttt{s[t]} \text{ otherwise} \end{cases}$$

This new basic combinator can be used everywhere in a composed strategy. One important characteristic of the Tom strategy language is that every composed strategy is itself a term and therefore can be traversed and rewritten. Adapting a strategy term for graphs with references consists in weaving the `Ref` combinator ahead every elementary strategy inside a strategy term. For example, `Sequence(s1,s2)` where `s1` and `s2` are elementary strategies will be rewritten into `Sequence(Ref(s1),Ref(s2))`.

5 Application to the lambda-calculus

Let us see now some application of our programming framework through the implementation of a basic λ-calculus interpreter. The graph with references will encode variable bindings, acting as de Bruijn indices, while the strategy language will translate the usual evaluation strategies of λ-calculus.

We work with a minimalist Gom signature:

```
LT = App(LT, LT)
   | Abs(LT)
```

The chosen representation of λ-terms makes use of terms with references by replacing variables with positions pointing to the corresponding binder. For instance, the term $\lambda f.\lambda x.(f\,x)$ will be encoded by the Gom term `Abs(Abs(App(posLT(-3),posLT(-2))))`. This encodes a kind of de Bruijn indices counting not only abstractions but also every node in the syntactic tree of the λ-term.

Let us write a `beta` strategy wich performs one β-reduction step on a redex. As mentioned in the previous section, it is possible to get the current position inside a visited term during its traversal by a strategy. Thereby, knowing the position of λ inside the visited redex $(\lambda x.f\ a)$ will allow us to find all the occurences of x in f, *i.e.* relative positions pointing to λ. The `beta` strategy then simply consists in four steps when applied to an application $(\lambda x.f\ a)$:

(i) collecting the position of λ;

(ii) collecting a;

(iii) replacing all the occurences of relative positions pointing to λ by a in f;

(iv) replacing the redex by the modified f.

Assuming we have a mutable structure `info` (a Java class here) which can store both informations of the first and second steps, this is achieved by the following strategy:

```
Strategy beta = `Sequence(
        _App(Identity(),collectTerm(info)),
        _App(
            Sequence(
                collectPosition(info),
                _Abs(μx.Choice(substitute(info),All(x)))),
            Identity()),
        clean());
```

We can notice the presence of four user defined strategies: `collectTerm`, `collectPosition`, `substitute` and `clean`. They respectively perform the four steps described above. Their code is obvious and one line long, except for the `substitute` strategy which has to compute the absolute position referenced by the current term to compare it with the position of λ stored in `info`. Then it performs the necessary shifts on bounded variables (relative positions) inside a before returning it. The whole strategy itself is an overlapping of congruence strategies. The `μx.Choice(substitute(info),All(x))` construct means that we do not go down further inside the term if the substitution succeeded.

We shall now apply this `beta` strategy on a λ-term with some evaluation strategy until we reach a fixpoint. `beta` being a strategy, it can be combined with other strategies to perform reductions. In particular, the `TopDown` and `Innermost` strategies respectively encode call-by-name and call-by-value evaluation strategies modulo some fixpoint computation encoded by the provided `RepeatId` strategy. They are themselve expressed using elementary strategies:

```
  TopDown(s) = μx.Sequence(s,All(x));
Innermost(s) = μx.Sequence(AllRL(MuVar(x)),Try(Sequence(s,x)))
```

Where `AllRL` applies `s` to all the childs of the current node from right to left. Substituting `s` by `beta` inside one these enables the expected evaluation behaviour.

Let us briefly see how a typical use of term-graphs, namely subterms shared evaluation, can be implemented by a slight modification of the previous example. We now assume that many bounded variables are represented by shared subterms where "shared" is meant in the sense of term-graphs semantics. For example, the λ-term $\lambda x.(x\,x)$ will be represented by `Abs(App(posLT(-2),posLT(-1,1)))` instead of `Abs(App(posLT(-2),posLT(-2)))`. The previous `beta` strategy is then

still mainly valid since this modification only affect the situations where the second child of an application is a variable, *i.e.* a relative position. Hence, changing the line `_App(Identity(),collectTerm(info))` by `_App(Identity(),Ref(collectTerm(info)))` suffices to adapt the strategy to the new λ-terms representation. This modification is of course relevant in case of a call-by-name strategy.

Finally we shall notice that termgraphs are sometime used to represent cyclic λ-terms [1]. This raises the question of the representation of terms cycling on an abstraction like $\langle x \mid x = \lambda y.(x\ y)\rangle$ with our de Bruijn encoding. Indeed, both y and x variables are then references denoting the root of the λ-term. This is easily handled by the use of "colored" references, implemented by two different `posLT` constructors: `Abs(App(PosLT1(-2),PosLT2(-2)))`.

The discussed implementation is available in the Tom subversion repository [1], under the `examples/termgraph` path.

6 Conclusion

To the best of our knowledge, we have presented here a new way of representing terms with references which presents strong similarities with the term-graph formalism. Using the Tom language as a programming background, we have discussed the various advantages and drawbacks of such an approach at different levels: memory representation, pattern matching and strategic traversal. We finally presented an application of this framework via the writing of a simple λ-calculus interpreter making an heavy use of strategies.

A major part of the presented propositions has been implemented. We are now working on the definition of a rewriting step similar to the one of [2]. Another field of investigation would be the writing of `Ref` strategies aborting infinite loops appearing during the traversal of a graph with cycles. This could be achieved by some map associating counters to visited nodes.

As shown by the last section, this model has interesting applications and opens promising perspectives in terms of program transformation and code analysis. Besides, the normal form described in section 2.2 makes it a solid basis for experimenting transformations on term-graphs in a concise and expressive manner.

References

[1] Ariola, Z. M. and J. W. Klop, *Cyclic lambda graph rewriting*, in: *Logic in Computer Science*, 1994, pp. 416–425.

[1] Compilation instructions are detailed in the Tom documentation at `http://tom.loria.fr/docs.php`

[2] Ariola, Z. M. and J. W. Klop, *Equational term graph rewriting*, Fundam. Inf. **26** (1996), pp. 207–240.

[3] Barendregt, H. P., M. van Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer and M. Sleep, *Term graph rewriting*, in: *PARLE Parallel Architectures and Languages Europe*, LNCS **259** (1987), pp. 141–158.

[4] Comon, H. and J.-P. Jouannaud, *Les termes en logique et en programmation* (2003), master lectures at Univ. Paris Sud.
URL http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud/articles/cours-tlpo.pdf

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

[6] Echahed, R. and N. Peltier, *Narrowing data-structures with pointers.*, in: *Third International Conference on Graph Transformations*, LNCS **4178**, 2006, pp. 92–106.

[7] Kennaway, R., *On graph rewritings*, Theoretical Computer Science **52** (1987), pp. 37–58.

[8] Kirchner, C., H. Kirchner and M. Vittek, *Designing constraint logic programming languages using computational systems*, in: F. Orejas, editor, *Proceedings of the 2nd CCL Workshop, La Escala (Spain)*, 1993.

[9] Kirchner, C., R. Kopetz and P. Moreau, *Anti-pattern matching*, in: *Proceedings of the 16th European Symposium on Programming*, 2007.

[10] Lacey, D. and O. de Moor, *Imperative program transformation by rewriting*, in: *Proceedings of the 10th International Conference on Compiler Construction* (2001), pp. 52–68.

[11] Löwe, M., *Algebraic approach to single-pushout graph transformation*, Theoretical Computer Science **109** (1993), pp. 181–224.

[12] Moreau, P.-E., C. Ringeissen and M. Vittek, *A Pattern Matching Compiler for Multiple Target Languages*, in: G. Hedin, editor, *Proceedings of the 12th International Conference on Compiler Construction*, LNCS **2622** (2003), pp. 61–76.

[13] Plump, D., *Term graph rewriting, handbook of graph grammars and computing by graph transformation*, G. Rozenber, World Scientific Publishing, 3-61 (1999).

[14] Reilles, A., *Canonical abstract syntax trees*, in: *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications* (2006), to appear.

[15] Visser, E. and Z.-e.-A. Benaissa, *A core language for rewriting*, in: C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications*, ENTCS **15** (1998).

Hard combinators

Denis Béchet [1] [2]

LINA
Université de Nantes
Nantes, France

Sylvain Lippi[3]

I3S
Université de Nice
Sophia Antipolis, France

Abstract

We present a simple system of four symbols and seven rules that can be used to translate a subclass of graph relabeling systems called hard interaction nets.

Keywords: Interaction nets, asynchronous circuits, universal machines, graph rewriting

1 Introduction

Interaction nets introduced by Yves Lafont [3] can be considered as a generalization of linear logic multiplicative proof nets. Syntactically they are presented as graph rewriting systems where rules are applied on pairs of nodes (called *cells*) connected by an "active edge" called *cut* by logicians. Lafont presented in [4] a system of three symbols and six rules called *interaction combinators* that is *universal*: any interaction system can be translated (in a sense that we shall detail below) into the system of the combinators. Interaction nets have been successfully used to implement various reduction strategies for the λ-calculus ([7] and [5]) and several interpreters (in particular a parallel one by [8] and a graphical one by [6]) for interaction nets have been proposed. More recently, non-deterministic extensions have been studied.

In this paper, we shall focus on a restriction called *hard interaction nets* where the geometry of the net is invariant during reduction and propose a universal system (called *hard combinators*) for such systems. The translation of an arbitrary

[1] Thanks to the referees and to Yves Lafont for their useful advises.

[2] Email: denis.bechet@univ-nantes.fr

[3] Email: lippi@i3s.unice.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

hard interaction system into hard combinators has a quite different character from the corresponding translation for interaction nets where the key technical point is implementing the duplication of some nets.[4] Here we shall represent nodes as binary words and calculate the transformations with boolean functions. The name *hard* interaction nets is well-chosen, since they are a form of abstract hardware. In this perspective, it is interesting to sum up the important rules and give the basic components that can be used to construct asynchronous circuits. For example, it should be possible to build an asynchronous computer simply by following classical Von Neumann computer architecture and using hard combinators [1].

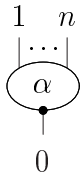
Notation. The domain of some variables is implicitly given by their names with the following conventions: $x, y, z, x_0, x_1, x_2, ...$ are binary digits, p, q, r, s, t are binary words and σ, ρ and τ are signatures ($+$ or $-$). Concatenation of p and q is noted pq so xy is a word with two digits and the scalar product of x and y is explicitly noted $x \times y$. x^n denotes the word $x...x$ with n letters. $|p|$ is the length of p. The set of boolean values $\{0, 1\}$ is noted $\mathbb{B}$ and the set of natural numbers $\mathbb{N}$.

2 Hard interaction nets

We present hard interaction nets informally from scratch without any reference to linear logic or even to interaction nets. A hard interaction system (or hard system for short) is composed with a set of *symbols* and their corresponding arity and with a set of *interaction rules*.

2.1 Cells, Ports, Nets and Cuts

Occurrence of symbols are called *cells* and have $n + 1$ *ports* where n is the corresponding arity. Each cell has exactly one principal port (pictured with a blob) and n auxiliary ones:



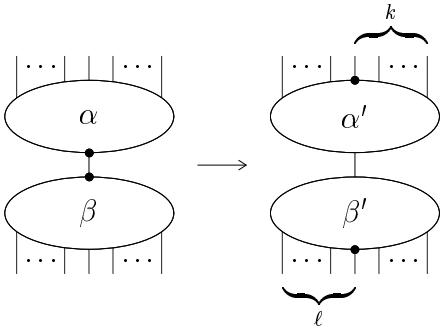
Nets are build with a set of cells and *free ports* where ports (principal, auxiliary and free ones) are connected pairwise. *Cuts* are particular nets composed of two cells connected by their principal ports.

2.2 Interaction rules

The difference between the principal port and the auxiliary ones is essential since rewriting (or *interaction*) can be applied only on cuts. In other words, the left member of an interaction rule is composed of two cells connected by their principal ports. Interaction consists in relabeling cells and changing the orientation of the

[4] more precisely principal nets for the connoisseur.

principal ports; we shall say that the cell is turning. To sum up, an interaction rule is pictured as follows,

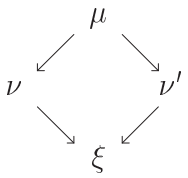


and we say that if an α-cell interacts with a β-cell it becomes α' and turns k times counter clockwise. Similarly, β-cell becomes β' and turns ℓ times. Note we are interested only in *deterministic* hard interaction systems so there is at most one interaction rule for each pair of symbols.

2.3 Reduction

Starting from an initial net containing cuts, we can apply an interaction rule obtaining another net and so on until an irreducible net if the reduction finishes. Hard interaction systems are very simple since the computation is local (only two cells are involved in a reduction) and the geometry of the net is invariant. However one can show [4] that it is complete from a computational point of view *i.e* one can define a hard interaction system that simulate a Turing Machine. Let us finish with an essential property due to the local synchronization.

Proposition 2.1 (strong confluence) *If a net μ reduces in one step to ν and ν', with $\nu \neq \nu'$, then ν and ν' reduce in one step to a common net ξ.*



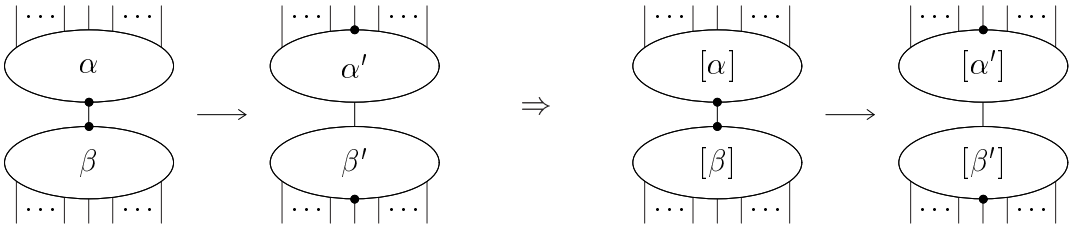
Proof The left member of an interaction rule is a cut and $\nu \neq \nu'$. Consequently the above reductions are applied on two different instances of cuts. Two instances of cuts are necessarily disjoint (a cell is in one cut at most) so the corresponding interaction rules can be applied independently. □

Consequently reduction is deterministic in a strong way: any reduction strategy gives the same result with the same number of steps.

Corollary 2.2 (reduction) *If a net μ reduces to an irreducible net ν in n steps, then any reduction starting from μ eventually reaches ν in n steps.*

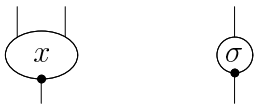
3 A universal system: hard combinators

We present a particular hard system called *hard combinators* with four symbols and seven rules that is sufficient to simulate all other hard systems. More precisely, we can translate each cell α by a net $[\alpha]$ built with hard combinators such that,



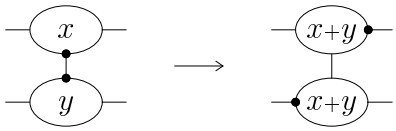
3.1 Cells

Our system is composed of four different symbols: two binary ones, 0 and 1, and two unary ones, + and −.

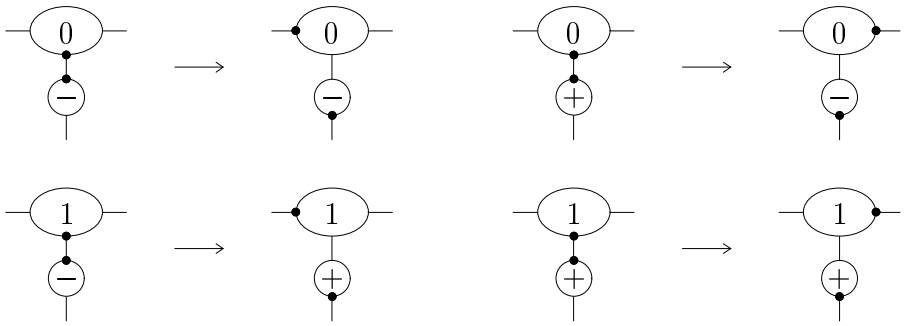


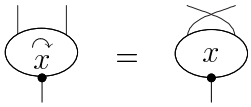
3.2 Rules

There are seven rules that can be split into two groups: three rules between binary cells and four rules between unary and binary cells. There is no rules between unary cells. Binary rules are also called *uniform rules* because the principal port "turns in the same direction" (counter clockwise) for each interaction. The three uniform rules can be summed up by the following schema where + denotes sum modulo 2.

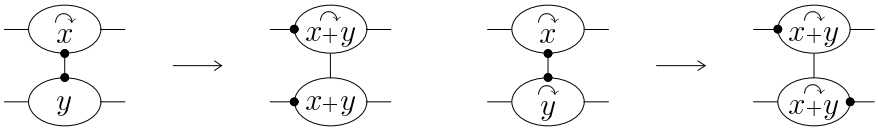


Consequently, the four other rules are called *non-uniform rules* because the orientation of a binary cell depends on the unary cell interacting with it. Intuitively, (+)-cells let binary cells turn counter clockwise and (−)-cells force them to turn clockwise.



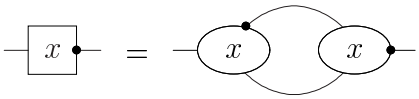
Definition 3.1 [clocks] for any bit x, 

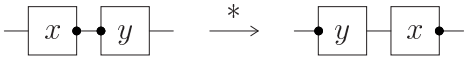
Clocks are introduced for graphical convenience to avoid complicated crossing of wires. They are noted $\overset{\frown}{x}$ because they interact as binary cells except their principal port turns clockwise. For example, we have the following reductions.



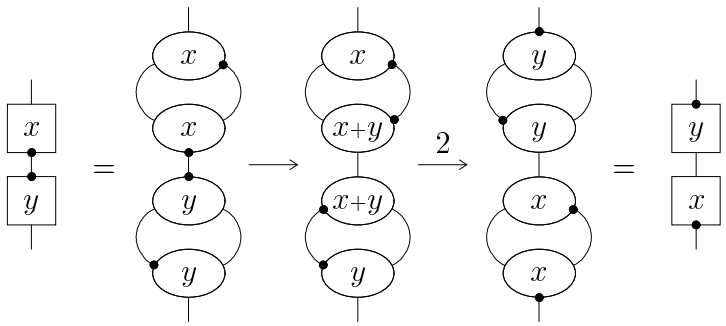
4 Uniform components

In this section, we consider the subsystem composed only with the two binary cells and the corresponding three uniform rules. Surprisingly, non trivial functions can be built in this restriction and, indeed it is a decisive step in the construction of a universal translation.

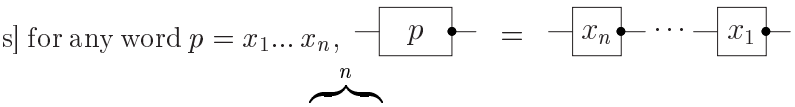
Definition 4.1 [binary pipes] for any bit x, 

Lemma 4.2 *for any bits x and y,* 

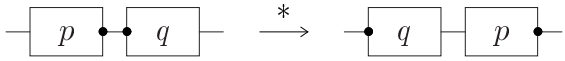
Proof We apply uniform rules and the equality $x + x + y = y \mod 2$



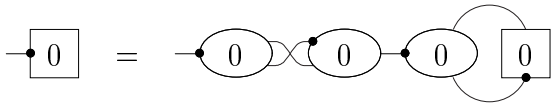
□

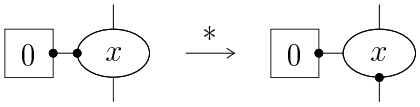
Definition 4.3 [pipes] for any word $p = x_1 ... x_n$, 

Notation. We also picture an *unknown pipe*  for pipes corresponding to any word of size n or simply  if there is no ambiguity. Those blank representations come from the idea that if one does not know what is stored in a pipe then, the place is free !

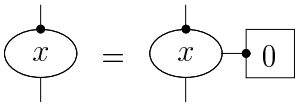
Lemma 4.4 *for any words p and q,* 

Proof by induction on p and q. □

Definition 4.5 [zero] 

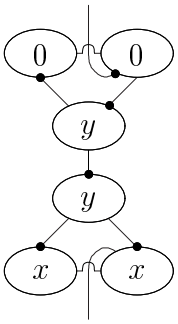
Lemma 4.6 *for any bit x,* 

Proof The above reduction can be easily checked with the binary rules. □

Definition 4.7 [seesaws] for any bit x, 

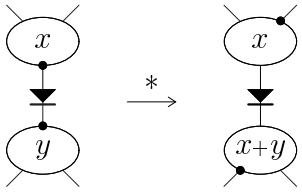
As clocks, seesaws are introduced to simplify the definitions of the other nets and do not have any functional property. Seesaws interact as binary cells: they change their principal port and their symbol is summed with the interacting cell.

Remark 4.8 *Do not confuse between pipes (binary words in a square box), seesaws (bits in a round box) and unary cells (signatures in a round box).*

Definition 4.9 [diodes]  =


Remark 4.10 *Unlike pipes or zero, diodes correspond to a* set of nets *not to a unique one. Indeed, bits x and y in the above definition can have any binary values so there are four different representation of a diode. We shall use this kind of definition for other components.*

Lemma 4.11 *For any bits x and y,*

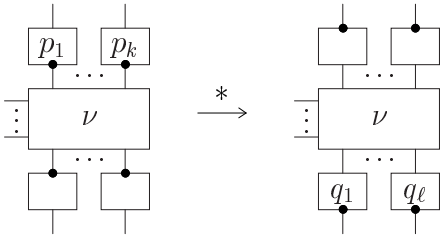


Proof The above reduction can be easily checked with the uniform rules. □

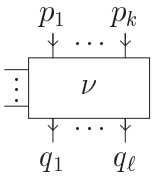
Remark 4.12 *According to remark 4.10, the above lemma should be read "starting from any representation of the diode in the left member, we obtain another (possibly different) representation of the diode in the right member.*

5 Invariant nets

Definition 5.1 [Invariant nets] Let us consider a net ν where free ports are partitioned into three sets: *inputs*, pictured with an in-going arrow, *outputs*, pictured with an out-going arrow, and unused, pictured with no arrow. We say that ν is invariant on inputs $p_1, ..., p_k$ and produces outputs $q_1, ..., q_\ell$ when we have the following reduction,



where the length of the "input" pipes are respectively $|p_1|$, ..., $|p_k|$ and the length of the "output" ones $|q_1|$, ..., $|q_\ell|$. We shall use the following notation for invariant nets,



Remark 5.2 *We do not mention where are the principal ports of ν. Indeed, the important point is to identify the inputs and the outputs and to know how they interact with pipes.*

As explained in remark 4.10, the net ν corresponds to a class of nets and the reduction above means that the right member is in the same class of nets as the left member. For example, in definition 5.6 , x_0 and y_0 range over $\{0, 1\}$ and σ ranges over $\{+, -\}$ so there are eight different representations.

Remark 5.3 *According to the previous definition, an invariant net is a pair composed of a net and a partition of its free ports and there may be several invariant*

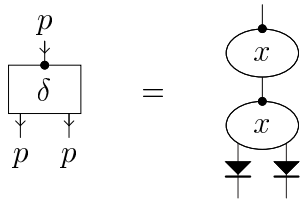
nets corresponding to a unique net. However, we also say that a net is invariant when such a partition exists.

Remark 5.4 *In the previous section we introduced unknown pipes and zero which are invariant. More precisely,* $p \to \boxed{} \to p$ *and* $\boxed{0} \to 0$.

5.1 Duplicator and arithmetic operations

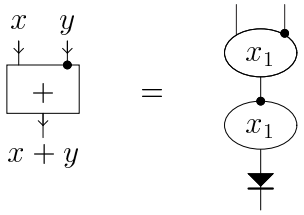
To avoid cumbersome repetitions, we give the definition and the corresponding invariance property of the following nets in one shot. For example, the net δ is defined by the right member of the equality and we show that it is invariant on input p and produces output p twice.

Definition-Lemma 5.1 (duplicator)



Proof We apply lemma 4.11 for the diode and the uniform rules. □

Definition-Lemma 5.2 (plus)

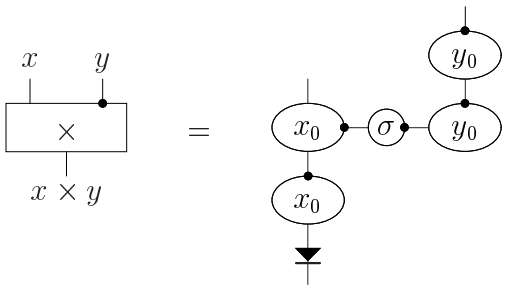


Remark 5.5 $+$ *denotes the sum modulo 2.*

Proof We apply lemma 4.11 for the diode and the uniform rules. □

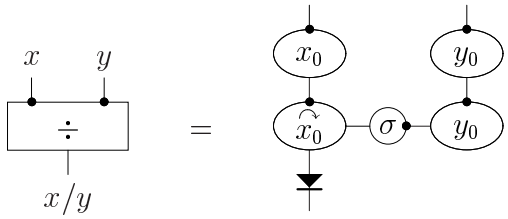
In the uniform subsystem, we have defined constants, pipes, duplication and plus. So one may wonder if it is possible to define product as well in this subsystem. The answer is probably negative. Indeed, the plus operation (binary `xor`) is weaker than binary addition that is computing the sum and but also the carry. Moreover, one can prove that is impossible to build a uniform system that is universal.

Definition 5.6 [sequential product]



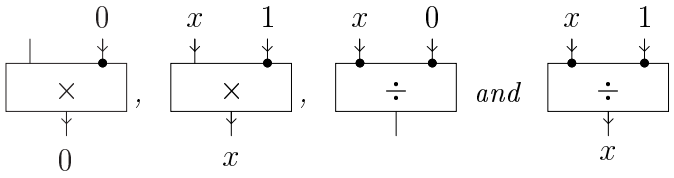
The *sequential product* use input y first. If y is zero the result is directly returned and input x is not used.

Definition 5.7 [partial quotient]



The *partial quotient* can be considered as the dual of the sequential product. Both inputs are used but it returns no result when input y is zero.

Lemma 5.8 *We have the following invariants for sequential product and partial quotient,*



Proof Trivial with uniform but also non-uniform rules. □

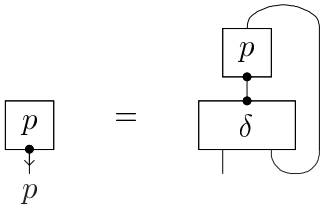
5.2 Composition

The first steps, building invariant nets from scratch can be compared to bootstrap in the sense that the difficult part is only to build the very first components (constant zero, duplicator, product). It is now easy to compose invariant nets with pipes and build other more complicated nets.

However, for synchronizations reasons, it is not always possible to compose two invariant nets by plugging directly outputs of the first one with inputs of the second one. To avoid this problem, outputs of invariant nets are connected to unknown pipes. It is not difficult to verify that such "buffered" invariant nets can be freely composed. In some cases, we can suppress those "output pipes" but the proof of the invariance property is tedious. Consequently, from now on, all outputs of invariant nets are connected to pipes when they are composed with other invariant nets.

A first application is to implement binary word constants.

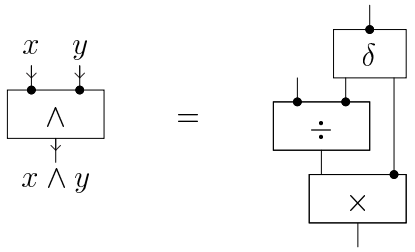
Definition-Lemma 5.3 (constant)



Remark 5.9 *For clarity, constants are defined with non-reduced nets. We can verify that we can reduce them and by the confluence property, we can use the reduced form.*

Let us give an invariant net for boolean *and*.

Definition-Lemma 5.4 (boolean and)

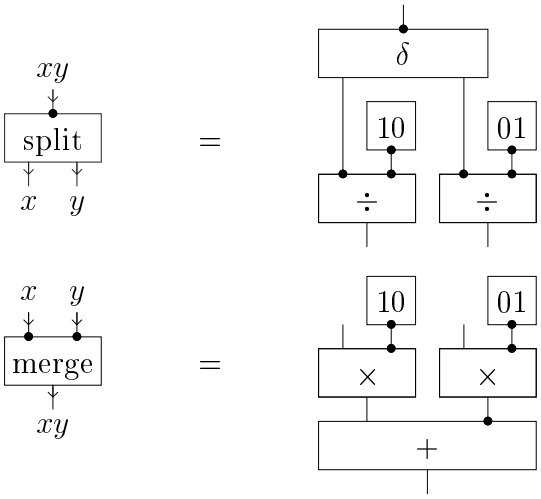


Proof We consider two cases: $y = 0$ and $y = 1$ and apply composition. □

Remark 5.10 *$x \wedge y = x \times y$ so the difference between sequential product and boolean and is that boolean and always uses its two inputs.*

In the same way, we can define invariant nets with several inputs and outputs for vectorial boolean functions on several inputs. Eventually, those invariant nets can be used to build the corresponding functions on binary words. To that purpose, the nets spit and merge can be composed to build some kind of parallel/serial adaptators.

Definition-Lemma 5.5 (split and merge)



Proof By composition. □

6 The Translation

Now we are ready to translate a given hard interaction system into the system of hard combinators presented in section 3. Symbols are numbered and represented by binary words of a fixed length N. A first idea is to represent the set of rules that we want to encode by a partial function $\varphi : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{B}^N \times \mathbb{N}$ where $\varphi(p, q) = (p', k)$ if p interacts with q, becomes p' and turns k times. Let us remark that we need the values of $\varphi(p, q)$ and $\varphi(q, p)$ to compute the reduction between p and q.

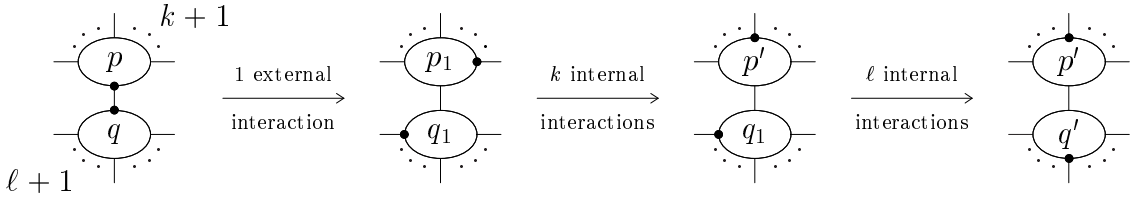
In fact, we choose a slightly different representation and introduce *stable cells* that interact with another (stable) cell and *unstable cells* that interact internally

reaching eventually a stable state. Each interaction is decomposed into one *external interaction* between two *stable cells* followed by several (possibly zero) *internal interactions* inside each *unstable cell*. This way we can impose that a cell turns (uniformly !) exactly once at each (external or internal) interaction. Consequently, the set of rules is represented by a partial function $\psi : \mathbb{B}^N \times \mathbb{B}^N \to \mathbb{B}^N$ where $\psi(p,q) = p'$ if p interacts with q, becomes p' and turns exactly once.

Let us define ψ from φ. For each couple of (stable) symbols p and q such that $\varphi(p,q) = (p', k+1)$ [5] we introduce k new (unstable) symbols $p_1, ..., p_k$ and set,

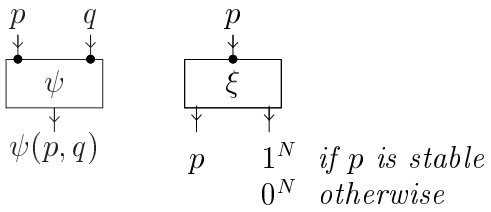
$$\begin{cases} \psi(p,q) & = p_1 \\ \psi(p_1, 0^N) & = p_2 \\ \vdots & \\ \psi(p_{k-1}, 0^N) & = p_k \\ \psi(p_k, 0^N) & = p' \end{cases}$$

Since unstable cells do not interact with another one, we arbitrarily fix the value of the second argument of ψ to 0^N. Here is the graphical representation of an interaction between p and q where $\varphi(p,q) = (p', k+1)$ and $\varphi(q,p) = (q', \ell+1)$,



Let us introduce two invariant nets. The first one corresponds to the function ψ that computes the new symbol after an (internal or external) interaction. The second one called discriminant ξ, says if a cell is stable or not.

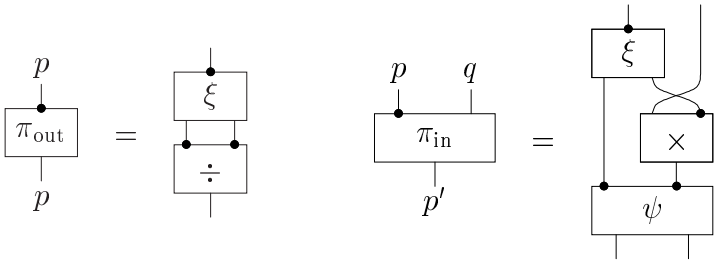
Definition 6.1 [transition and discriminant]



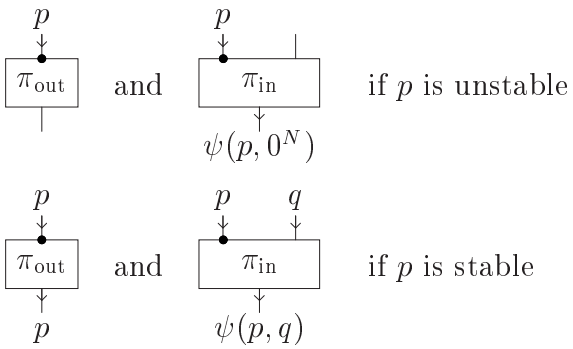
Now we can give the translation of the port of a cell into two parts: π_{in} and π_{out}. The important idea is that π_{in} computes the next symbol p' without any interaction with q in the case p is not stable. In the same way π_{out} gives the current symbol p only if p is stable.

[5] If the principal port remains unchanged after reduction, we say that it turns $a+1$ times where a is the arity of the cell.

Definition 6.2

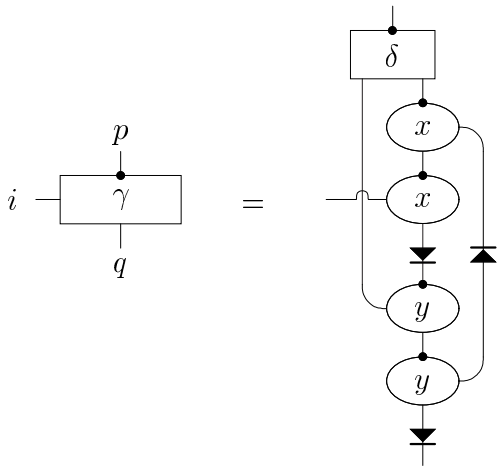


Lemma 6.3



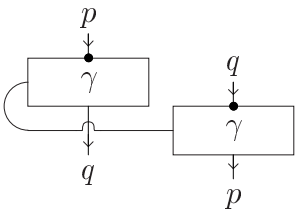
Invariant nets are easy to use and compose because we feel "at home" with inputs/outputs. However this notion is not mandatory for general interaction nets. Indeed, in the translation of a port, we need some kind of "full/duplex" connection since a cell outputs its current symbol to another cell but also inputs the symbol of the cell with whom it is interacting ! This is exactly what is done by the net γ.

Definition 6.4 [gamma]



Port p corresponds to an input, port q to an output and i to the "full/duplex" interface. Each port of a cell corresponds to a γ-cell; when two cells interact, the input of a γ-cell is reproduced on the output of the other γ-cell. This property is summed up in the following lemma.

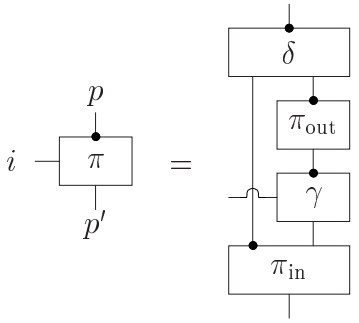
Lemma 6.5



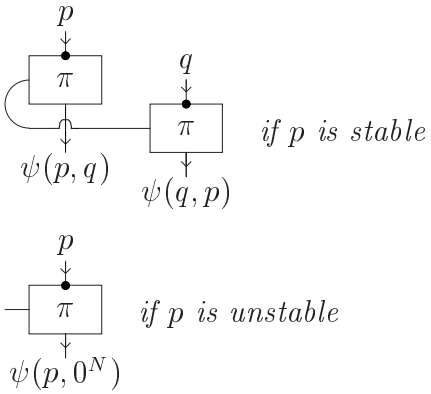
Remark 6.6 *Let us remark that surprisingly γ is built only with uniform cells.*

Now we can compose, π_{in}, π_{out} and γ and give the translation of a port π. According to the previous paragraph, port i (interface) is both an input and an output.

Definition 6.7



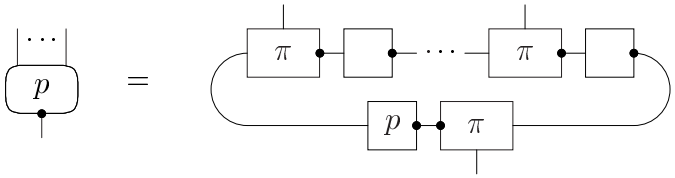
Lemma 6.8 (external and internal interaction)



Proof By composition. □

The above lemma details two cases: two stable cells interact with one another or an unstable cell interact internally. Consequently, port i is unused or plugged to the interface of another π net.

Definition 6.9 [translation of a cell]

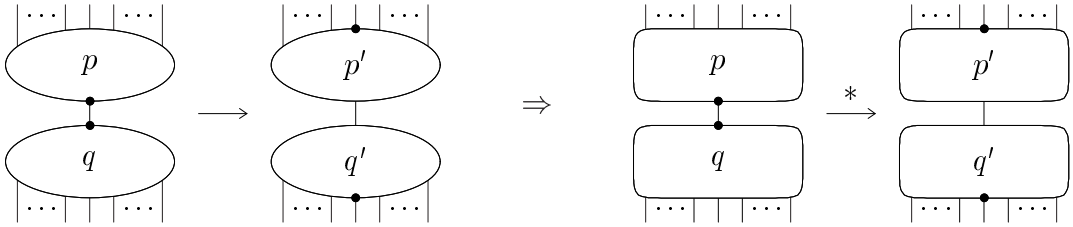


where the length of the pipes is $|p| = N$

By analogy with computer architectures, π corresponds to a form of Arithmetic and Logical Unit (ALU) and pipe to a register. Then this basic architecture (a net π composed with a pipe) is repeated for the translation of each port of the cell. Another possibility is to "centralize" the transition function for the whole cell. The advantage is we do not have to introduce unstable cells but on the other side we have to implement a more complicated component for the interface part.

Finally, it is now easy to verify that our translation simulates the rules of a given hard system.

Theorem 6.10



Proof Apply lemma 6.8 and definition 6.9. See appendix A for the detailed reduction. □

7 Conclusion

The system we propose seems to be a good candidate for a universal hard system. However this work is a first step in the domain of hard interaction nets. Indeed many questions related to fundaments as well as applications remain still open.

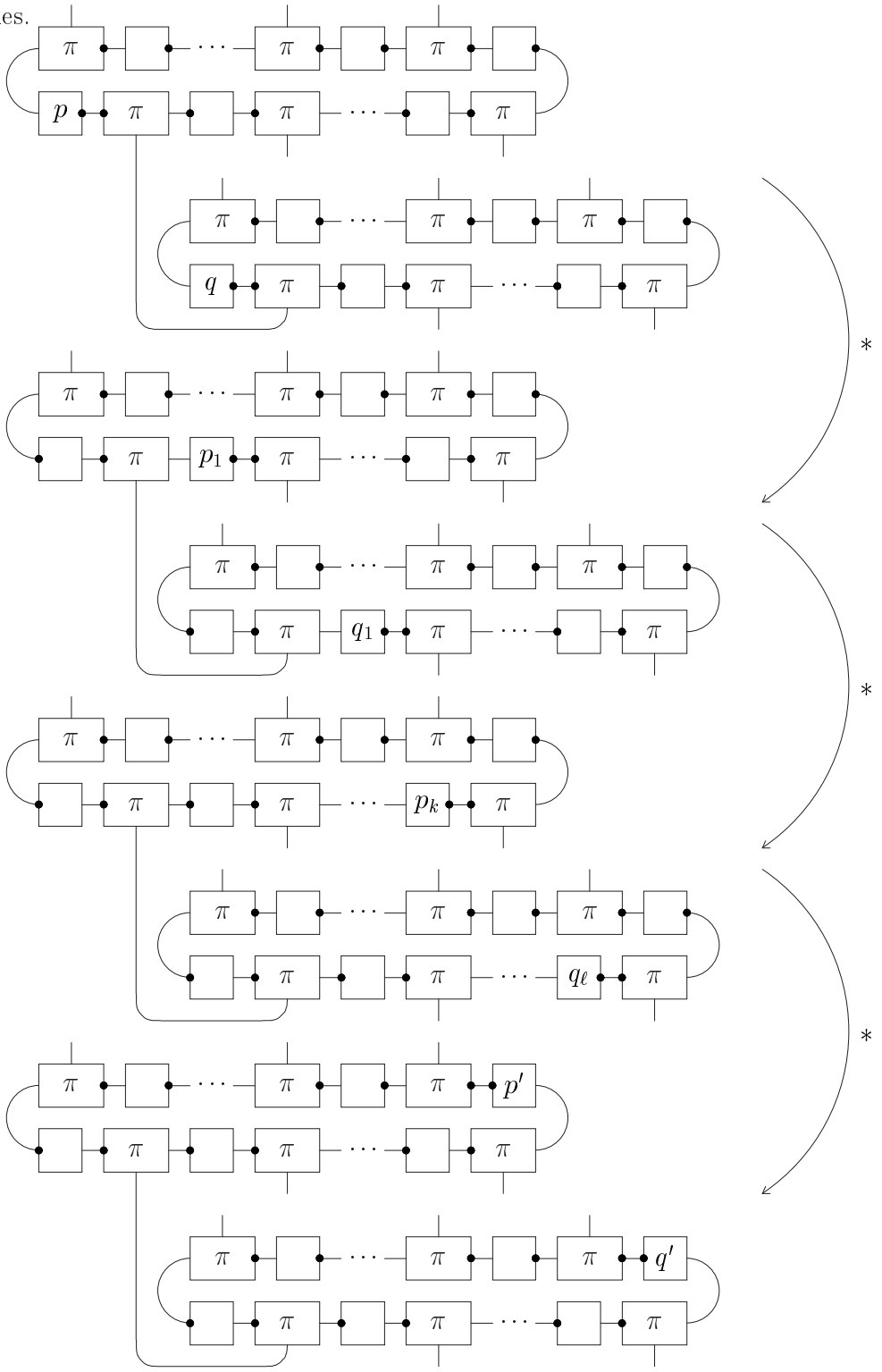
- The first one concerns the minimality of such a system; is it possible to give a simpler universal system with fewer symbols or rules? For instance, it is not easy to know whether three symbols would be sufficient. We only know that a system composed only of uniform rules cannot be universal.
- There is a correctness criterion for interaction nets imported from linear logic to prevent deadlocks. It is important to reformulate this criterion for the particular case of hard interaction nets since it is an opportunity to simplify and perhaps to refine it.
- Although (general) interaction nets cannot be translated into hard interaction nets, it is interesting to see if there could be a compilation process for some

subclass of interaction nets. Interaction nets would be the high level programming language whereas hard interaction nets would be the target (low level) language. In the same spirit, interpreters have been developed for interaction nets. Would it be possible to physically implement components for hard combinators? In other words, we can consider hard combinators as components for electronic asynchronous circuits?

- As interaction nets can be compared to graph rewriting systems, hard interaction nets can be compared to graph relabeling. These techniques have been particularly successful in the study of graph election algorithms [2]. It would be interesting to implement such algorithms with hard interaction nets and this way take benefit from the confluence property! More generally, it would be interesting to compare hard interaction nets with other existing rewriting techniques.
- The fixed geometry of hard interaction nets gives them a very similar flavour to cellular automata, or a generalization of cellular automata to non-rectangular grids and there are universality results for cellular automata so it should be interesting to compare those rewriting systems.

A Simulation of hard interaction rules

We detail the proof of theorem 6.10. We consider the interaction between a cell p and q where p becomes p' and turns $k+1$ times and q becomes q' and turns $\ell+1$ times.



References

[1] Cecile Germain, Daniel Etiemble. Architecture des Ordinateurs. Cours de Licence Informatique, Université d'Orsay, 2005.

[2] Emmanuel Godard, Yves Métivier, M. Mosbah, and A. Sellani. Termination detection of distributed algorithms by graph relabelling systems. *In proceedings of the first Conference on Graph Transformation*, 2002.

[3] Yves Lafont. Interaction nets. *In proceedings of the 17th Annnual ACM Symposium on Principles of Programming Languages, Orlando (Fla., USA)*, pages 95–108, 1990.

[4] Yves Lafont. Interaction combinators. *Information and Computation*, 137(1):69–101, 1997.

[5] Sylvain Lippi. Encoding left reduction in the lambda-calculus with interaction nets. *Mathematical Structures in Computer Science*, 12(6), December 2002.

[6] Sylvain Lippi. in²: a graphical interpreter for the interaction nets. In *Proceedings of Rewriting Techniques and Applications (RTA '02)*. Springer Verlag, 2002.

[7] Ian Mackie. Yale: Yet another lambda evaluator based on intetaction nets. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*. ACM Press, 1998.

[8] Jorge Sousa Pinto. *Implantation parallèle avec la logique linéaire (applications des réseaux d'interaction et de la géométrie de l'interaction)*. PhD thesis, Ecole Polytechnique, 2001.

Universal Boolean Systems

Denis Béchet[1]

LINA
Université de Nantes
Nantes, France

Sylvain Lippi[2]

Université de Nice
Sophia Antipolis, France

Abstract

Boolean interaction systems and hard interaction systems define nets of interacting cells. They are based on the same local interaction principle between two cells as interaction nets but do not allow that the structure of nets may evolve. With boolean nets, it is not possible to create or destroy cells or links between existing cells. They are very similar to hardware circuits but based on an implicit rendez-vous information exchange mechanism.
If we want to implement such systems using hardware circuits, it is important to define a set of universal combinators that reduces this task to the implementation of a fixed number of known agents. Here, we show how we can simulate every hard interaction system by a universal boolean interaction system composed of three combinators: a duplicator, a NAND gate and a three-state input/output channel.

Keywords: interaction net, hard interaction system, boolean interaction system, combinator, universal system.

1 Introduction

Interaction nets [6] are a programming paradigm inspired by Girard's proof nets for *linear logic* [3]. Some translations from λ-calculus into interaction nets [9,5,10] or from proof nets [7,12,2,13] show that interaction systems are interesting for computation. They are a special case of a hypergraph replacement systems [14] or of graph relabelling systems [11] but are strongly confluent. In fact, interaction net reductions are purely local and confluent. Moreover, the number of steps that are necessary to reduce completely a net is independent of the way one may choose. From the point of view of λ-calculus, translations used in [4,5] captures optimal reduction.

[1] Email: Denis.Bechet@univ-nantes.fr
[2] Email: lippy@unice.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Hard interaction systems are, in fact, a variant of interaction systems where rules are constrained in such a way that the structure of nets can not change. Rules do not create or destroy cells or links between cells. They can only change the symbol of agents and the port that is principal.

In [8], Lafont introduces a universal interaction system with only three different symbols γ, δ and ϵ. δ and ϵ are respectively a duplicator and an eraser and γ is a constructor. This system preserves the complexity of computation for a particular system. The number of steps that are necessary to reduce a simulated interaction net is just (at most) the number of steps of the original interaction net multiplied by a constant (which depends only on the simulated system and not on the size of the original interaction net). [1] shows that there exists a universal system with only two symbols.

However, both systems can not be considered as universal hard interaction systems because the rules that define the systems do not preserve the structure of nets. The paper investigates this problem and shows how we can simulate every hard interaction system by a *universal boolean interaction system*. In fact, boolean interaction systems are hard interaction systems where information that are exchange between agents are binary like hardware circuits connected by a wire can only communicate binary information.

We thing that this result is interesting if we want to implement (eventually with hardware circuits) such system using a finite set of combinators. This result also shows the main principles behind hard interaction system: duplication (the system is linear), computing (something must be done) and conditionnal input/output interaction (the cells must choose to whom they want to interact to).

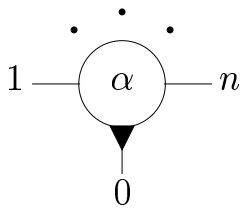
This paper is organized as follows: after an introduction to interaction nets and hard and boolean interaction systems, the notions of interaction net homomorphism, simulations and universal hard interaction systems are presented. Section 4 shows how to translate a system to a universal system.

2 Hard interaction system

Interaction nets are a model of computing introduced by Yves Lafont in [6]. We briefly presents interaction nets and hard interaction systems are. Boolean interaction systems are presented in the Section 4.

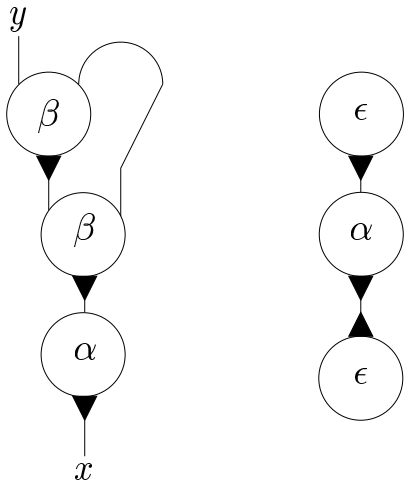
2.1 Agents and nets

An *interaction net* is a set of agents linked together through their ports. An individual agent is an instance of a particular symbol which is characterized by its *name* α and its *arity* $n \geq 0$. The arity defines the number of auxiliary ports associated to each agent. In addition to auxiliary ports, an agent owns a *principal port*. Graphically, an agent is represented by a circle :



In fact, the ports form a circular list that are represented on the circle. The principal port is marked by a triangle and the name is put inside the circle. The (dynamic) state of an agent is only determined by its name and the position of the port that is principal.

An interaction net is a set of agents where the ports are connected two by two. The ports that are not connected to another one are the *free ports* of the net and are distinguished by a name. The set of names of the free ports of a net is the *interface* of this net. Below, the interface is $\{y, x\}$. α has one auxiliary port, β has two and ϵ has none.

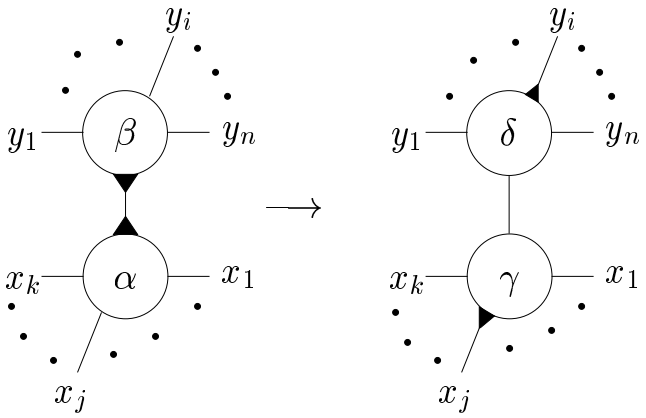


2.2 Hard interaction rule and hard interaction system

An interaction net can evolve when two agents are connected through their principal ports. An *interaction rule* is a rewriting rule where the left member is constituted of only two agents connected through their principal ports and the right member is any interaction net with the same interface. For hard interaction system, the rule must preserve the structure of nets. Thus the right member of a *hard interation rule* is also constituted of two agents with the same arities as the agents of the left member of the rule and they must be connected by a link that corresponds to the same ports as for the left member. In fact, the right member of a rule is the same as its left member except that names may be different and the ports that are principal may be different (at least one principal port must be different).

The right member of a hard interaction rule can be characterized for each interacting agent by the new name of the agent and by a rotational number from 0 to n (n is the arity of the agent) that indicates which port, counted clockwise from the current principal port, becomes principal (0 means that the principal port does not

move).



We write this rule $[\alpha, \beta] \to [\gamma, +i, \delta, +j]$ which means that γ replaces α, δ replaces β, the principal port of γ is the j-th clockwise port from the principal port of α and the principal port of δ is the i-th clockwise port from the principal port of β.

An interaction net that does not contain two agents connected by their principal port is *irreducible*. A net reduces to another net by applying successively zero, one or several times hard interaction rules to couples of agents connected through their principal ports. Each step substitutes the couple by the right member of the rule.

A *hard interaction system* $\mathcal{I} = (\Sigma, \mathcal{R})$ is a set of symbols Σ and a set of hard interaction rules $\mathcal{R}$ where agents in the left and right members are instances of the symbols of Σ.

A hard interaction system $\mathcal{I}$ is *deterministic* when (1) there exists at most one hard interaction rule for each couple of different agent and (2) there exists at most one hard interaction rule for the interaction of an agent with itself. In this case, the right member of this rule must be symmetric from the central point (this is necessary for a deterministic system). A hard interaction system $\mathcal{I}$ is *complete* when there is at least one rule for each couple of agent. In this paper we consider deterministic and complete systems. With these systems, we can prove that reduction is strongly confluent [3]. In fact, this property is true whenever the system is deterministic.

3 Universal hard interaction systems

Universality means that every interaction system can be *simulated* by a universal interaction system. Here, we use a very simple notion of simulation that is based on interaction net *homomorphism*.

3.1 Interaction net homomorphism

Let Σ and Σ' be two sets of symbols. An *homomorphism* Φ from Σ to Σ' is a map that associates to each symbol in Σ an interaction net of agents of Σ' with the same interface. This homomorphism is naturally extended to interaction nets of agents of Σ.

[3] A system is strongly confluent if and only if when a net reduces in one step to $\mathcal{N}$ and $\mathcal{N}'$, then $\mathcal{N}$ and $\mathcal{N}'$ reduce in on step to a common net.

3.2 Simulation

We say that an homomorphism Φ from Σ to Σ' defines a *simulation* of an interaction system $\mathcal{I} = (\Sigma, \mathcal{R})$ by another interaction system $\mathcal{I}' = (\Sigma', \mathcal{R}')$ if the reduction mechanism on interaction nets of $\mathcal{I}$ and $\mathcal{I}'$ are *compatible* by Φ [8,1]: for every interaction net $\mathcal{N}$ of Σ:

(i) $\mathcal{N}$ is irreducible if and only if $\Phi(\mathcal{N})$ is irreducible;

(ii) if $\mathcal{N}$ reduces to $\mathcal{M}$ then $\Phi(\mathcal{N})$ can reduce to $\Phi(\mathcal{M})$.

This definition brings some properties with complete and deterministic interaction systems:

(i) the translation of an interaction net composed of a unique agent must be irreducible;

(ii) this translation has at most one agent whose principal port belongs to the interface and the symbol of this interface that is connected to this agent is the same as the symbol that is connected of the principal port of the original agent;

(iii) this translation must be connexe;

(iv) an homomorphism is a simulation if (i), (ii) and (iii) are verified and if the left member $\mathcal{N}$ (composed of two agents) and the right member $\mathcal{M}$ of every rule in $\mathcal{R}$ verify $\Phi(\mathcal{N})$ reduces to $\Phi(\mathcal{M})$;

(v) the simulation relation is transitive and symmetric.

3.3 Universal hard interaction system

A hard interaction system $\mathcal{U}$ is said to be *universal* if for any hard interaction system $\mathcal{I}$, there exists a simulation $\Phi^{\mathcal{I}}$ of $\mathcal{I}$ by $\mathcal{U}$.

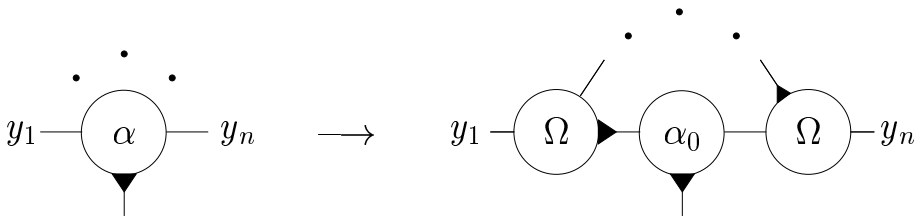
4 A universal boolean interaction system

In this section, we show how to simulate a particular hard interaction system $\mathcal{I}$ with a fixed hard interaction system.

4.1 Simulation with agents of arity 2

We can normalize the arity of agents to always be 2. In fact, we have seen that a rule may be characterized by two informations for each agent of the right member: the new name and the number of clockwise shifts, from 0 to n, where the new principal port must be set.

For $\mathcal{I} = (\Sigma, \mathcal{R})$, let $N \geq 0$ be the maximum arity of Σ. We define $\Sigma' = \{(\Omega, 2)\} \cup \{(\alpha_j, 2) \mid (\alpha, i) \in \Sigma\ ,\ j \in \{0, \ldots, N\}\}$. Let Φ_Σ the homomorphism where an agent α of arity i is transformed into an agent α_0 and i agents Ω each of arity 2:



We define $\mathcal{I}' = (\Sigma', \mathcal{R}')$, where $\mathcal{R}'$ is defined as follows. For $\mathcal{I}$, the rule between α and β results in γ in place of α with a clockwise shift of i for the principal port and δ in place of β with a clockwise shift of j for the principal port. This rule is replaced by a rule between α_0 and β_0. The right member of the rule becomes γ_i and δ_j. If $i = 0$ (resp. $j = 0$) the principal port of γ_i (resp. δ_j) is the same as the principal port of α_0 (resp. β_0). Otherwise, the principal port is the next clockwise port. For $1 \leq i \leq N$, the rule between Ω and γ_i (resp. δ_i) replaces Ω by γ_{i-1} (resp. δ_{i-1}) and γ_i (resp. δ_i) by Ω. If $i = 1$, the principal port of γ_{i-1} (resp. δ_{i-1}) is the next clockwise port. Otherwise, it is the next counter-clockwise port. For Ω, it is the next clockwise port.
$[\alpha, \beta] \to [\gamma, i, \delta, j]$ is replaced by one of the following rules:

- $[\alpha_0, \beta_0] \to [\gamma_i, 0, \delta_j, 0]$ if $i = 0$ and $j = 0$.
- $[\alpha_0, \beta_0] \to [\gamma_i, 0, \delta_j, +1]$ if $i = 0$ and $j \neq 0$.
- $[\alpha_0, \beta_0] \to [\gamma_i, +1, \delta_j, 0]$ if $i \neq 0$ and $j = 0$.
- $[\alpha_0, \beta_0] \to [\gamma_i, +1, \delta_j, +1]$ if $i \neq 0$ and $j \neq 0$.

The rules for Ω are:

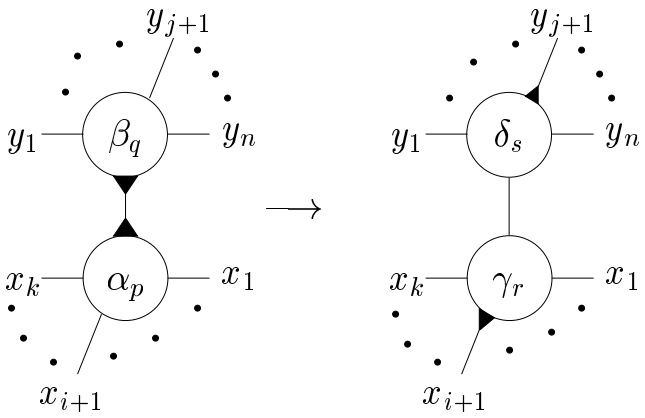
- $[\Omega, \gamma_i] \to [\gamma_{i-1}, +1, \Omega, +1]$ if $i = 1$.
- $[\Omega, \gamma_i] \to [\gamma_{i-1}, +2, \Omega, +1]$ otherwise.

Theorem 4.1 *Φ_Σ defines a simulation of $\mathcal{I}$ by $\mathcal{I}'$*

The proof is straightforward: the translation of an agent is a loop of agents which is connexe and irreducible and has only one principal port that is connected in the interface to the same symbol as the original agent. Secondly, if $\mathcal{N}$ is the left member and $\mathcal{M}$ the right member of a rule of $\mathcal{I}$, $\Phi_\Sigma(\mathcal{N})$ reduces to $\Phi_\Sigma(\mathcal{M})$ (usually in more than one step depending on the clockwise number of shifts of the principal ports of the agents between $\mathcal{N}$ and $\mathcal{M}$).

4.2 Boolean interaction system

The second step in our construction consists in the simulation of the boolean functions. For that, we use *boolean agents*. This kind of agents has a name that is composed of two informations: a boolean output state that can be either 0 or 1 and an internal state p. We note 0_p and 1_p these names. A boolean interaction rule concerning two boolean agents is a hard interaction rule $[\alpha_p, \beta_q] \to [\gamma_r, +i, \delta_s, +j]$ $(\alpha, \beta, \gamma, \delta \in \{0, 1\})$ that defines γ, r and i as functions of α_p and β (they do not dependent of q which is the internal state of β_q) and δ, s and j as functions of β_q and α (they do not dependent of p which is the internal state of α_p).



This kind of hard interaction system can be defined by a boolean function for each symbol (and not for a couple of agents as with a hard interaction rule) that we call *boolean interaction rule*: $\alpha_p[\beta] \to [\gamma_r, +i]$. This boolean rule describes a half of an interaction rule. It says that an agent α_p is transformed into an agent γ_r when it interacts with an agent with a boolean output state β. The new principal port is the i-th clockwise port from the current principal port. We call *boolean interaction systems* such hard interaction systems.

4.3 Simulation of boolean circuits

Every boolean function can be simulated by a particular boolean agent. For instance, a logical binary *NAND* (not and) gate is simulating by an agent with 3 ports (the arity of the symbols is 2). This gate reads the two inputs then gives the result on its output. After this cycle, the gate starts again to read the inputs and write the output in an endless loop. Starting with 0_a on the first input port, the agent continues with the second input port using one of the two boolean inteaction rules: $0_a[0] \to [0_b, +1]$ or $0_a[1] \to [0_c, +1]$. Then, after the interaction with the second input, the gate delivers the result on the output port using one of the four boolean inteaction rules: $0_b[0] \to [1_d, +1]$, $0_b[1] \to [1_d, +1]$, $0_c[0] \to [1_d, +1]$ or $0_c[1] \to [0_d, +1]$. Finally, the gate returns to the first input port, ready for the next cycle, using one of the four boolean inteaction rules: $0_d[0] \to [0_a, +1]$, $0_d[1] \to [0_a, +1]$, $1_d[0] \to [0_a, +1]$ or $1_d[1] \to [0_a, +1]$.

A *boolean duplicator* is also helpful. This agent has one input and two outputs. It reads the input, puts it on the first output then on the second output and starts again a new cycle. The operation are sequential like the NAND gate. Starting with 0_e on the input port, the agent goes to the first output using one of the two boolean inteaction rules: $0_e[0] \to [0_f, +1]$ or $0_e[1] \to [1_f, +1]$. Then, it switches to the second output using one of the four boolean inteaction rules: $0_f[0] \to [0_g, +1]$, $0_f[1] \to [0_g, +1]$, $1_f[0] \to [1_g, +1]$ or $1_f[1] \to [1_g, +1]$. Finally the agent returns to the input port, ready for the next cycle, using one of the four boolean inteaction rules: $0_g[0] \to [0_e, +1]$, $0_g[1] \to [0_e, +1]$, $1_g[0] \to [0_e, +1]$ or $1_g[1] \to [0_e, +1]$.

The other kinds of logical operators like *OR*, *NOT* or *AND* are also easy to simulate. In fact, every vector of boolean function with several inputs and several outputs may be simulated by a boolean agent and its boolean inteaction rules.

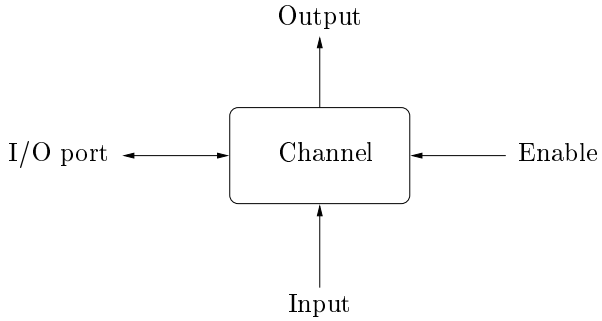
But, the *NAND* and the boolean duplicator are enough to simulate every vector of boolean functions.

Theorem 4.2 *Every (vector of) boolean function can be simulated by a boolean interaction system using the previous symbols and their rules (this system has* $5+5 = 10$ *symbols).*

Proof. In fact, every boolean function of several variables can be computed using binary *NAND* gates. Because each variable can be used more than once, we need a duplicator (the connections between duplicators and *NAND* gates must be done carefully to avoid deadlock because the inputs of *NAND* gates are tested in a certain order and the outputs of duplicators are activated in a certain order). When a variable does not appear in the boolean function, we have to "forget" its value. A very simple solution consists in the introduction of this variable x into the boolean function f using the following formula: f is replaced by f or (x and not x). Thus every variable appears at least once in f and it is not necessary to forget an outpout. □

4.4 Simulation of boolean I/O channels

To finish with the different bricks of our universal boolean interaction system, we need a boolean device that receives a validation that enables or not an I/O interaction. If the communication is enabled the channel writes the input bit to the I/O port, waits for a boolean interaction, reads the bit and copies it to the ouptut. If the communication is not enabled, the channel copies the input bit to the ouput without interacting through its I/O port.

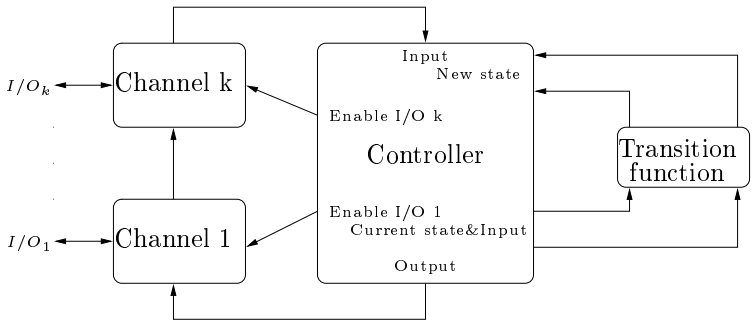


This device is simulated by a boolean agent. Starting with the state 0_h, this agent looks at the enable port. It switches to the input port using one of the two boolean interaction rules: $0_h[0] \to [0_i, +1]$ or $0_h[1] \to [0_j, +1]$. Then, it gets the input bit and following the state, puts the principal port on the I/O port (state 0_i) or on the output port (state 0_j): $0_i[0] \to [0_k, +1]$, $0_i[1] \to [1_k, +1]$, $0_j[0] \to [0_l, +2]$ or $0_j[1] \to [1_l, +2]$. If the communication is enabled (states 0_k or 1_k), the channel gives its boolean state through the I/O port and reads the boolean state of the boolean agent that is connected to this port. The channel then switches to the output port using one of the four boolean interaction rules: $0_k[0] \to [0_l, +1]$, $0_k[1] \to [1_l, +1]$, $1_k[0] \to [0_l, +1]$ or $1_k[1] \to [1_l, +1]$. Now, even if the communication is not enabled, the agent returns to the ouput port its boolean state which is either the read bit or a copy of the input bit. After that, it goes back to the enable port using one of the

rules: $0_l[0] \to [0_h, +1]$, $0_l[1] \to [0_h, +1]$, $1_l[0] \to [0_h, +1]$ or $1_l[1] \to [0_h, +1]$.

4.5 Simulation of a boolean interaction controller

A boolean interaction controller is a device that has a state, input/output boolean channels and a transitional function. The controller chooses one of its input/output channel, puts a boolean information on it, waits until it receives a boolean information from the input/output channel and, following its transitional function, changes the state. The controller repeats indefinitely these same steps.



The controller and the transition function can be simulated by a boolean interaction system using *NAND* and duplicators agents. Channels are simulated by the special boolean agent presented before. Thus, every boolean interaction controller can be simulated by a boolean interaction system that has three kind of circuits: *NAND*, duplicators and channels.

4.6 Simulation of a hard interaction system

It is relatively easy to see that every hard interaction system where the symbols are specific to a port (the principal port of an agent must be the same each time the same symbol appears on the agent) like the system that we have after the simulation by a system with agents or arity 2 can be simulated by a particular boolean interaction controller.

Theorem 4.3 *The hard interaction systems $\mathcal{I}'$ obtained by Theorem 4.1 can be simulated by a boolean interaction controller (that depends of $\mathcal{I}'$).*

Proof. We need to code the symbols of $\mathcal{I}'$ by binary numbers in a finite space. If the system has N symbols, we need $K \geq \log_2(N)$ bits. The controller can be build in such a way to operate with K bits rather than 1 (in the same spirit as we have 32-bit processors rather than 1-bit processors). The channels must exchange K bits serially (like a serial communication channel controlled by microcode). □

Corollary 4.4 *The system with* NAND *gates, duplicators and I/O channels is universal (the system has 5+5+7=17 symbols).*

5 Conclusion

We have shown that there exist universal boolean interaction systems. Our universal system has 17 symbols and is very different of Lafont's universal system. This

system is certainly not optimal in the sense that it is surely possible to find a universal boolean interaction system with less symbols (and less rules) but boolean interaction systems are a special case of hard interaction systems and a solution for universal hard interaction systems does not necessary give a solution for boolean interaction systems.

References

[1] Denis Bechet. Universal interaction systems with only two agents. In *Proccedings of the Twelve International Conference on Rewriting Techniques and Applications, Utrecht, The Netherlands, May 2001*, 2001.

[2] S. Gay. Combinators for interaction nets. In I. C. Mackie & R. Nagarajan C. L. Hankin, editor, *Proceedings of the Second Imperial College Department of Computing Workshop on Theory and Formal Methods*. Imperial College Press, 1995.

[3] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[4] G. Gonthier, M. Abadi, and J.-J. Levy. The geometry of optimal lambda reduction. In *Proceedings of the Nineteenth Annual Symposium on Principles of Programming Languages (POPL '90)*, pages 15–26, Albuquerque, New Mexico, January 1992. ACM Press.

[5] G. Gonthier, M. Abadi, and J.-J. Levy. Linear logic without boxes. In *Seventh Annual Symposium on Logic in Computer Science*, pages 223–234, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[6] Y. Lafont. Interaction nets. In *Seventeenth Annual Symposium on Principles of Programming Languages*, pages 95–108, San Francisco, California, 1990. ACM Press.

[7] Y. Lafont. From proof nets to interaction nets. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 225–247. Cambridge University Press, 1995. Proceedings of the Workshop on Linear Logic, Ithaca, New York, June 1993.

[8] Y. Lafont. Interaction combinators. *Information and Computation*, 137(1):69–101, 1997.

[9] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Seventeenth Annual Symposium on Principles of Programming Languages (POPL '90)*, pages 16–46, San Francisco, California, 1990. ACM Press.

[10] S. Lippi. Encoding left reduction in the lambda-calculus with interaction nets. *Mathematical Structure in Computer Science*, 12(6), December 2002.

[11] I. Litovsky, Y.Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. In H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation*, volume 3, pages 1–56. World Scientific, 1999.

[12] I. Mackie. *The Geometry of Implementation (an investigation into using the Geometry of Interaction for language implemetation)*. PhD thesis, Departement of Computing, Imperial College of Science, Technology and Medecine, 1994.

[13] I. Mackie. Interaction nets for linear logic. *Theoretical Computer Science*, 247:83–140, 2000.

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

Intensional properties of polygraphs

Guillaume Bonfante[1] Yves Guiraud[2]

INRIA-LORIA
615 rue du Jardin Botanique - BP 101
Villers-lès-Nancy 54602 - FRANCE

Abstract

We present polygraphic programs, a subclass of Albert Burroni's polygraphs, as a computational model, showing how these objects can be seen as first-order functional programs. We prove that the model is Turing complete. We use polygraphic interpretations, a termination proof method introduced by the second author, to characterize polygraphic programs that compute in polynomial time. We conclude with a characterization of polynomial time functions.

Keywords: Polygraphs, termination proof, complexity characterization

1 Introduction

Polygraphs are special higher-dimensional categories, introduced by Albert Burroni to provide a unified algebraic setting for rewriting [3]. For example, any term rewriting system can be translated into a polygraph which has, in case of left-linearity, exactly the same properties of termination and confluence [9,5].

Here, we study how these mathematical objects can be used as a computational model. Informally, computations generated by a polygraph are done by a net of cells which individually behave according to some local transition rules. This model is close to John von Neumann's cellular automata [15] and Yves Lafont's interaction nets [8] with notable differences: while von Neumann's automata are essentially synchronous, interaction nets and polygraphs are asynchronous; polygraphs have a much more rigid geometry than interaction nets: the underlying graphs of the formers are directed acyclic graphs, preventing the "vicious circles" of the latters.

Termgraph rewriting systems provide another model of graphical computation [14]: it is an extension of term rewriting with an additional operation, sharing, that allows for a more correct representation of actual computation. The translation of terms into polygraphs is close to the one into termgraphs and they seem to have the same properties, as

[1] Guillaume.Bonfante@loria.fr
[2] Yves.Guiraud@loria.fr, INRIA postdoctoral fellow with the support of the EADS Corporate Research Foundation.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

suggested by the first results in [7]. For example, let us consider the following term rewriting rule, used to compute the multiplication on natural numbers: $\mathtt{mult}(x, \mathtt{succ}(y)) \to \mathtt{add}(x, \mathtt{mult}(x, y))$. When applied, this rule duplicates the term corresponding to the argument x. In termgraph rewriting, one is able to share it instead, so that there is no need for extra memory space. This sharing operation can be algebraically formalized as an operation with one input and two outputs, whose semantics is a duplication operation. In polygraphs, one can have many such operations with many outputs, explicitely represented and handled.

This is a key fact in our results on implicit computational complexity: indeed, the interpretations we consider here, called *polygraphic interpretations* [5,7], can reflect the fact that two outputs of the same operation have some links between them, as we will see with the example of the list splitting function used in "divide and conquer" algorithms. This allows us to give complexity bounds where traditional polynomial interpretations [12] cannot with the method described in [4,1] or to give better bounds, as indicated here and in [7]. Moreover, the polygraphic interpretations give separated information on the spatial and on the temporal complexities of functions.

This document is an overview of ideas and results from a paper by the same authors [2], containing more comments, technical details and complete proofs. In section 2 we introduce the notion of polygraphic program in an informal way, give the corresponding semantics we consider, introduce the leading example we consider, namely the polygraphic program computing the "fusion sort" on lists, and prove that polygraphic programs form a Turing complete model of computation. In section 3, we recall the notion of polygraphic interpretation, give examples, define the notion of simple polygraphic program and prove results on termination of polygraphic programs. In section 4, we give polynomial complexity bounds for simple programs and prove that they characterize the class PTIME of functions computable in polynomial time by a Turing machine.

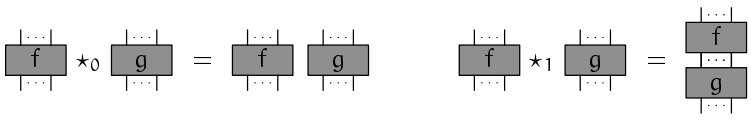
2 Polygraphs as a computational model

The general definition of polygraph can be found in documents by Albert Burroni, Yves Lafont and Franois Mtayer [3,9,13,10,11]. Here we give a rewriting-minded presentation of a special case of polygraphs, seeing them as rewriting systems on algebraic circuits.

Definition 2.1 A *monoidal 3-polygraph* is a composite object consisting of *cells*, *paths* and *compositions* organized into *dimensions*.

Dimension 1 contains elementary sorts called *1-cells* and represented by wires. Their concatenation $\star_0$ yields product types called *1-paths* and pictured as juxtaposed vertical wires. The empty product $*$ is also a 1-path, represented by the empty diagram.

Dimension 2 is made of operations called *2-cells*, with a finite number of typed inputs and outputs. They are pictured as circuit gates, with inputs at the top and outputs at the bottom. Using all the 1-cells and 2-cells as generators, one builds circuits called *2-paths*, using the following two compositions:



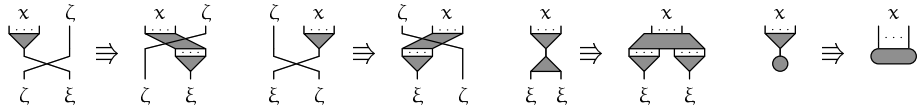
The constructions are considered *modulo* some relations, including topological deformation: one can stretch or contract wires freely, move 2-cells, provided one does not create crossings or break wires. Each 2-cell and each 2-path f has a 1-path $s_1(f)$ as input, its *1-source*, and a 1-path $t_1(f)$ as output, its *1-target*. The compact notation $f : s_1(f) \Rightarrow t_1(f)$ summarizes these facts.

Dimension 3 contains rewriting rules called *3-cells*. They always transform a 2-path into another one with the same 1-source and the same 1-target. Using all the 1-cells, 2-cells and 3-cells as generators, one can build reductions paths called *3-paths*, by application of the following three compositions, defined for F going from f to f' and G going from g to g': $F \star_0 G$ goes from $f \star_0 g$ to $f' \star_0 g'$; when $t_1(f) = s_1(g)$, then $F \star_1 G$ goes from $f \star_1 g$ to $f' \star_1 g'$; when $f' = g$, then $F \star_2 G$ goes from f to g'. These constructions are identified *modulo* some relations, given in [6], where their 3-dimensional nature was explained. The relations allow one to freely deform the constructions in a reasonable way: in particular, they identify paths that only differ by the order of application of the same 3-cells on non-overlapping parts of a 2-path. A 3-path is *elementary* when it contains exactly one 3-cell. Each 3-cell and each 3-path F has a 2-path $s_2(F)$ as left-hand side, its *2-source*, and a 2-path $t_2(F)$ as right-hand side, its *2-target*. The notation $F : s_2(F) \Rrightarrow t_2(F)$ stands for these facts.

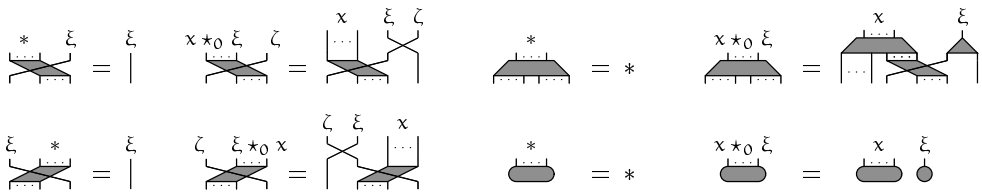
For monoidal 3-polygraphs, rewriting notions are defined in a similar way as for term rewriting systems, with terms replaced by 2-paths, reduction steps by elementary 3-paths and reduction paths by 3-paths [5]. Hence, a *normal form* in a polygraph $\mathcal{P}$ is a 2-path f which is the 2-source of no elementary 3-path. The polygraph $\mathcal{P}$ *terminates* when it does not contain infinite families $(F_n)_{n\in\mathbb{N}}$ of elementary 3-paths such that $t_2(F_n) = s_2(F_{n+1})$ for all n. Other rewriting properties, such as *confluence* or *convergence* are also defined in an intuitive way.

Definition 2.2 A *polygraphic program* is a monoidal 3-polygraph such that:

- Its 2-cells are divided into *structure 2-cells*, *constructors* and *functions*. The structure 2-cells consist of one $\times : \xi \star_0 \zeta \Rightarrow \zeta \star_0 \xi$ for each pair of 1-cells (ξ, ζ), plus one $\triangle : \xi \Rightarrow \xi \star_0 \xi$ and one $\bullet : \xi \Rightarrow *$ for each 1-cell ξ. The constructors are 2-cells such with a 1-cell as 1-target. The functions are any 2-cells.
- Its 3-cells are divided between *structure 3-cells* and *computation 3-cells*. The structure 3-cells are given, for every constructor $\bigtriangledown : x \Rightarrow \xi$ and every 1-cell ζ, by:



The 2-targets of the 3-cells of Δ^2_3 use structure 2-paths built from the structure 2-cells by using the following structural induction rules:



The computation 3-cells are 3-cells whose 2-source is of the shape $t \star_1 \varphi$, with φ a function 2-cell and t a 2-path built only with 1-cells and constructors. Furthermore, there is a finite constant that bounds the number of structure 2-cells in the 2-target of

each computation 3-cell.

- For the present study, we assume that there exists a procedure to perform each step of computation: more formally, for every 3-path $F : f \to g$ containing exactly one 3-cell, the map giving g from (f, F) is computable in polynomial time.

Example 2.3 We consider the following polygraphic program with one 1-cell, two constructors $\circ$ and $\overline{\circ}$, two functions $\curlyvee$ and $\curlywedge$ and four computation 3-cells (we do not give the structure cells):

With the constructors, one can represent the natural number n, using $\circ$ for 0 and $\overline{\circ}$ for the successor operation, yielding a 2-path t_n with zero input and one output. Furthermore, one can check that this polygraph is convergent and that, given t_m and t_n, the normal form of $(t_m \star_0 t_n) \star_1 \curlyvee$ is t_{m+n}, while the one of $(t_m \star_0 t_n) \star_1 \curlywedge$ is t_{mn}.

Hence this polygraphic program computes the addition and the multiplication on natural numbers: the 1-cells are the data types, the 2-paths $\xi \Rightarrow *$ built only from constructors are the values, while the result of the application of a function $\curlyvee$ with n inputs to well-typed values $(t_1, \dots, t_n)$ is the normal form of the 2-path $(t_1 \star_0 \cdots \star_0 t_n) \star_1 \curlyvee$. This semantical interpretation is formalized thereafter.

Definition 2.4 [Semantics] Let us fix a polygraphic program $\mathcal{P}$. If ξ is a 1-cell, a *term of type* ξ is a 2-path built only with constructors and with ξ as 1-target. A *value* or *closed term* is a term with no input. The set of values with type ξ is denoted by $\mathcal{V}(\xi)$. The *domain of computation* of $\mathcal{P}$ is the multi-sorted algebra made of the family of all the sets $\mathcal{V}(\xi)$ equipped with the operations given, for each constructor $\gamma : \xi_1 \star_0 \cdots \star_0 \xi_n \Rightarrow \xi$, by the map still denoted by γ:

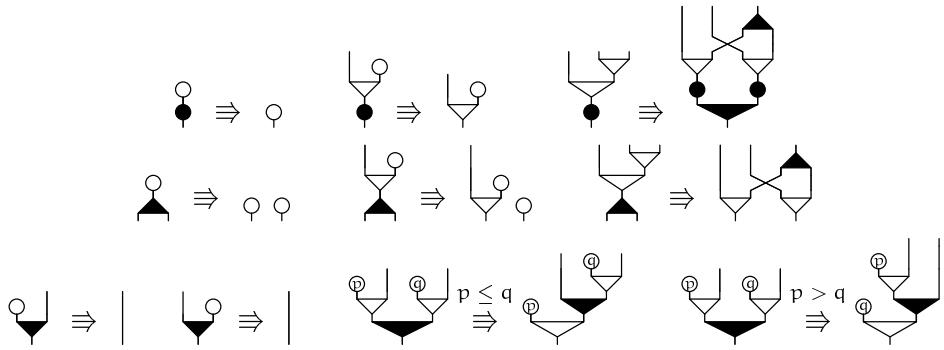
$$\begin{aligned} \gamma \ : \ \mathcal{V}(\xi_1) \times \cdots \times \mathcal{V}(\xi_n) &\to \mathcal{V}(\xi) \\ (t_1, \dots, t_n) &\mapsto (t_1 \star_0 \cdots \star_0 t_n) \star_1 \gamma. \end{aligned}$$

Let us consider a function f from $\mathcal{V}(\xi_1) \times \cdots \times \mathcal{V}(\xi_m)$ to $\mathcal{V}(\zeta_1) \times \cdots \times \mathcal{V}(\zeta_n)$. Then $\mathcal{P}$ *computes* f if there exists a 2-path, still denoted by f, from $\xi_1 \star_0 \cdots \star_0 \xi_m$ to $\zeta_1 \star_0 \cdots \star_0 \zeta_n$, such that, for every family $(t_1, \dots, t_m)$ of values in $\mathcal{V}(\xi_1) \times \cdots \times \mathcal{V}(\xi_m)$, the 2-path $(t_1 \star_0 \cdots \star_0 t_m) \star_1 f$ normalizes into the family $f(t_1, \dots, t_m)$ of values in $\mathcal{V}(\zeta_1) \times \cdots \times \mathcal{V}(\zeta_n)$.

Example 2.5 Let us consider a polygraphic program that computes, among other functions, the *fusion sort* function on lists of natural numbers. It has two 1-cells, `nat` for natural numbers and `list` for lists of natural numbers. Its other cells, apart from structure ones are:

- Constructors: one $\textcircled{n} : * \Rightarrow \texttt{nat}$ for each natural number n, plus $\circ : * \Rightarrow \texttt{list}$ for the empty list and $\curlyvee : \texttt{nat} \star_0 \texttt{list} \Rightarrow \texttt{list}$ for the list constructor.
- Functions: the main $\bullet : \texttt{list} \Rightarrow \texttt{list}$ for fusion sort, together with $\curlywedge : \texttt{list} \Rightarrow \texttt{list} \star_0 \texttt{list}$ for splitting lists and $\curlywedge : \texttt{list} \star_0 \texttt{list} \Rightarrow \texttt{list}$ for merging them.

- Computation 3-cells:

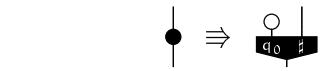


Note that the last two rules for the $\blacktriangledown$ function are not conditional: there is exactly one of them for each pair (p, q) of natural numbers, depending if $p \leq q$ or $p > q$. However, these two conditions are computable (in linear time), preventing super-Turing computations. We have chosen a simplified representation of natural numbers which considers them as being predefined, at the "hardware level", together with their predicate $\leq$. The reason for this choice is to postpone the study of modularity and of the `if-then-else` construction to subsequent work.

Theorem 2.6 *Polygraphic programs form a Turing-complete model of computation.*

Proof. Here we give a sketch of the proof, while the complete one can be found in [2]. Any Turing machine can be translated into a polygraphic program whose values are the words written in the alphabet of the machine and whose functions are the transitions steps generated by the machine transition function. More formally, the considered polygraphic program has one 1-cell, plus:

- Constructors: one $\bigcirc : 0 \Rrightarrow 1$ for the empty word plus one $\textcircled{a} : 1 \Rrightarrow 1$ for each letter a.
- Functions: one $\blacklozenge : 1 \Rrightarrow 1$ for the function to be computed plus one $\mathtt{step}_{q,a} =$  : $2 \Rrightarrow 1$ for each state q and each letter a, including the blank symbol $\sharp$.
- Computation 3-cells are given thereafter, the first rule initializing the computation, the four subsequent families replicating the transitions of the Turing machine and the final family starting the computation of the result:



both when $\delta(q, a) = (q', c, L)$

both when $\delta(q, a) = (q', c, R)$

Let us asume that the machine is in a state q, reading a letter a, with w_l and w_r the two words respectively written on the left of a, from right to left, and on the right of a, from left to right. Then, this state of the system is represented by the 2-path $(w_l \star_0 w_r) \star_1$  . It is

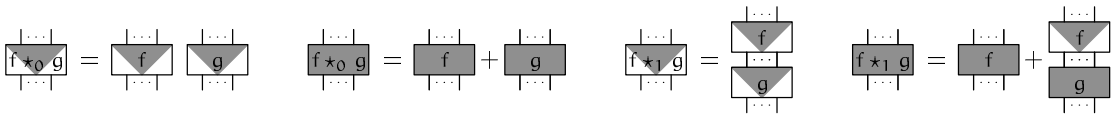
straightforward to check that each step of the Turing machine corresponds to an elementary 3-path of its polygraphic version. □

3 Polygraphic interpretations and simple programs

Intuitively, a polygraphic interpretations sees 2-paths as electrical circuits, whose components are their 2-cells. The circuits have currents plugged into their inputs, and these currents propagate into the circuits according to the "current maps" φ_* associated to each 2-cell φ. A circuit produces heat, given by the sum of the "heat maps" $[\varphi]$ of the 2-cells it is made of. In the case of polygraphic programs, we will see that current and heat maps can be used to give information respectively on the spatial size and on the temporal size of computations. Polygraphic interpretations have been introduced, in a more general version, in [5].

Definition 3.1 A *polygraphic interpretation* of a polygraphic program $\mathcal{P}$ consists into a mapping of each 2-path f with m inputs and n outputs onto two monotone maps $f_* = $  : $\mathbb{N}^m \to \mathbb{N}^n$ and $[f] = $  : $\mathbb{N}^m \to \mathbb{N}$, such that the following conditions are satisfied:

- For every 1-path x of length n, we have $x_* = \mathrm{Id}_{\mathbb{N}}^n$ and $[x] = 0$.
- For every 2-paths f and g, the following equalities hold when defined:



Given an interpretation and a 2-cell φ, we denote by φ_*^j the j^{th} component of the map φ_*. An interpretation of $\mathcal{P}$ generates a binary relation denoted by $\succ$: it is defined, on 2-paths f and g with the same 2-source and the same 2-target, by $f \succ g$ when the two inequalities $f_*(\mathbf{i}) \geq g_*(\mathbf{i})$ and $[f](\mathbf{i}) > [g](\mathbf{i})$ hold for every possible family $\mathbf{i}$ of natural numbers. An interpretation is *compatible* with a 3-cell α when $s_2(\alpha) \succ t_2(\alpha)$ and *weakly compatible* with α if $s_2(\alpha) \succeq t_2(\alpha)$.

It was proved in [5] that an interpretation is entirely determined by its values on the 2-cells of the polygraph, that the binary relation $\succ$ is a terminating strict order and that context are strictly monotone with respect to it. These are steps towards:

Theorem 3.2 ([5]) *If a polygraphic program admits an interpretation which is compatible with all of its 3-cells, then it terminates.*

Example 3.3 Let us assume that we have a current map $(\cdot)_*$ on a polygraphic program such that the following conditions hold:

- If  is a constructor with n inputs, then  $_*(i_1, \ldots, i_n) > i_1 + \cdots + i_n$.
- One structure 2-cells, we have  $_*(i,j) = (j,i)$ and  $_*(i) = (i,i)$.

We define a heat map $[\cdot]_S$ as follows:

- If  is a constructor or a function, then $\left[\right.$$\left.\right]_S = 0$.
- On structure 2-cells, we have $\left[\right.$$\left.\right]_S(i,j) = ij$, $\left[\right.$$\left.\right]_S(i) = i^2$ and $\left[\right.$  $\left.\right]_S(i) = i$.

It is proved in [2] that these values generate a polygraphic interpretation compatible with

the structure 3-cells. Hence theorem 3.2 tells us that a polygraphic program without computation 3-cell terminates.

Definition 3.4 Given a current map $(\cdot)_*$ on a polygraphic program that satisfies the conditions of example 3.3, the heat map $[\cdot]_S$ is called *structure heat* generated by $(\cdot)_*$.

Definition 3.5 We denote by $\mathbb{N}[x_1, \cdots, x_n]$ the algebra of polynomials in n variables and coefficients in $\mathbb{N}$. Let $\mathcal{P}$ be a polygraphic program. A polygraphic interpretation is *simple* when the following conditions are met:

- For any 2-cell φ with m inputs and n outputs, the maps $\sum_{j=1}^{n} \varphi_*^j$ and $[\varphi]$ are polynomials of $\mathbb{N}[x_1, \ldots, x_m]$.
- If γ is a constructor with n inputs, then $\gamma_* = \sum_{i=1}^{m} x_i + a_\gamma$ with $1 \leq a_\gamma < a$, where a is a constant depending on the program. Moreover, $[\gamma] = 0$.
- On structure 2-cells, one has  $(i,j) = (j,i)$ and  $(i) = (i,i)$. Moreover, structure cells produce no heat: $\left[\right.$$\left.\right](i) = 0$, $\left[\right.$$\left.\right](i,j) = 0$, $\left[\right.$  $\left.\right](i) = 0$.
- For every function φ with m inputs and n outputs and for every family $(i_1, \ldots, i_m)$ of natural numbers, we have $\sum_{j=1}^{n} \varphi_*^j(i_1, \ldots, i_m) \geq i_1 + \cdots + i_m$.

A polygraphic program is called *simple* when it admits a simple polygraphic interpretation which is compatible with all of its computation 3-cells.

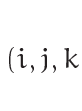
Theorem 3.6 *A simple polygraphic program terminates.*

Proof. Let $\mathcal{P}$ be a simple polygraphic program and let $(\cdot)_*$ and $[\cdot]$ be the current and heat maps of a simple interpretation, compatible with all the computation 3-cells of $\mathcal{P}$. It is a direct computation to check that such an interpretation is weakly compatible with the structure 3-cells of $\mathcal{P}$. Hence, we deduce that $\mathcal{P}$ terminates if and only if the polygraphic program $\mathcal{Q}$ does, where $\mathcal{Q}$ is built from $\mathcal{P}$ by removal of the computation 3-cells. The map $(\cdot)_*$ also satisfies the conditions to generate a structure heat map $[\cdot]_S$ proving the termination of $\mathcal{Q}$.□

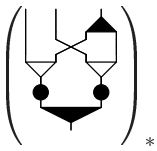
Example 3.7 Let us prove that the polygraphic program of example 2.5 is simple. Let us consider the interpretation generated by these values:

-  $_* = 1$,  $_* = 1$,  $_*(i,j) = i + j + 1$;
-  $_*(i) = i$, $_*(i) = (\lceil i/2 \rceil, \lfloor i/2 \rfloor)$,  $_*(i,j) = i + j$;
- $\left[\right.$  $\left.\right](i) = 2i^2$, $\left[\right.$  $\left.\right](i) = i$, $\left[\right.$  $\left.\right](i,j) = i + j$.

We have used the notations $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ for the rounding functions, respectively by excess and by default. This interpretation meets the conditions of definition 3.5 and, thus, is simple. Now, one has to check that it is compatible with all the computation 3-cells: we give some of the computations for the last 3-cell of the function . Let us start with $(\cdot)*$. On one hand:

$$\left(\right.\text{  }\left.\right)_*(i,j,k) = \left(\right.\text{  }\left.\right)_*\left(i, \text{}_*(j,k)\right) = \text{}_* \circ \text{}_*\left(i, \text{}_*(j,k)\right) = i+j+k+2.$$

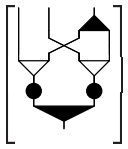
And, on the other hand:



$$(i,j,k) \;=\; i+j+\lceil k/2\rceil+\lfloor k/2\rfloor+2 \;=\; i+j+k+2.$$

Now, let us consider $[\cdot]$. For the 2-source of the 3-cell, one gets:

$$(i,j,k) \;=\; \left[\right](i+j+k+2) \;=\; 2\cdot(i+j+k+2)^2.$$

And, for its 2-target,  (i,j,k) is equal to:

$$\left[\right](k)+\left[\right](i+\lceil k/2\rceil+1)+\left[\right](j+\lfloor k/2\rfloor+1)+\left[\right](i+\lceil k/2\rceil+1,j+\lfloor k/2\rfloor+1)$$
$$=2\cdot(i+\lceil k/2\rceil+1)^2+2\cdot(j+\lfloor k/2\rfloor+1)^2+i+j+2k+2.$$

We conclude by considering two cases, depending on the parity of k.

Example 3.8 For the polygraphic program of example 2.3, the following values generate a simple interpretation which is compatible with the four computation 3-cells:

- $\;_*=1$, $\;_*(i)=i+1$, $\;_*(i)=(i,i)$, $\;_*(i,j)=i+j$, $\;_*(i,j)=ij$;
- $\left[\right]=\left[\right](i)=\left[\right](i)=\left[\right](i)=0$, $\left[\right](i,j)=i$, $\left[\right](i,j)=(i+1)j$.

4 Complexity of simple programs

Definition 4.1 Let $\mathcal{P}$ be a polygraphic program. If f is a 2-path of $\mathcal{P}$, we denote by $\|f\|$ the number of 2-cells f is made of. If F is a 3-path of $\mathcal{P}$, we denote by $\|\|F\|\|$ the number of 3-cells F is made of.

Let $\mathcal{P}$ be a simple program with a fixed interpretation made of $(\cdot)_*$ and $[\cdot]$. We want to prove that $(\cdot)_*$ is a good estimation of the size of values computed by $\mathcal{P}$, given by $\|\cdot\|$, while $[\cdot]$ is one for the size of the computations, given by $\|\|\cdot\|\|$. Once again, the complete proofs are in [2]. By induction on the size of values, we prove that $(\cdot)_*$ is an estimation of the size of values:

Lemma 4.2 *For every value* t*, the inequalities* $\|t\| \leq t_* \leq a\,\|t\|$ *hold in* $\mathbb{N}$*.*

Using the properties of the polygraphic interpretation we consider and lemma 4.2, we prove that the size of intermediate and of final values are bounded by a polynomial in the size of the initial values:

Proposition 4.3 *Let* φ *be a function with* m *inputs and* n *outputs. Let* P_φ *be the polynomial in* $\mathbb{N}[x_1,\dots,x_m]$ *defined by* $P_\varphi = \sum_{j=1}^{n} \varphi^j_*(ax_1,\dots,ax_m)$*. Let* t *be a family of values of type* $s_1(\varphi)$ *and let us assume that* $t \star_1 \varphi$ *reduces into a 2-path of the shape* $u \star_1 c$*,*

where u *has* p *outputs. Then the inequality* $\sum_{j=1}^{p} u_*^j \leq P_\varphi(\|t^1\|, \ldots, \|t^m\|)$ *holds. In particular, if* $u = \varphi(t)$*, the inequality* $\|\varphi(t)\| \leq P_\varphi(\|t^1\|, \ldots, \|t^m\|)$ *holds.*

Example 4.4 If one computes these polynomials for the simple polygraphic program of example 2.5, one sees that, for any list t, the sorted list $\bullet(t)$ and all the intermediate values computed to reach the result have their sizes bounded by the size of t:

$$P_{\bullet}(x) = \bullet_*(1 \cdot x) = x, \qquad P_{\blacktriangledown}(x, y) = \blacktriangledown_*(1 \cdot x, 1 \cdot y) = x + y,$$
$$P_{\blacktriangle}(x) = \blacktriangle_*^1(1 \cdot x) + \blacktriangle_*^2(1 \cdot x) = \lceil x/2 \rceil + \lfloor x/2 \rfloor = x.$$

For the polygraphic program of example 2.3, one gets $P_{\blacktriangledown}(x, y) = x + y$ and $P_{\blacktriangledown}(x, y) = xy$. Hence, the current maps give us information on the spatial complexity of the computation, separated from the length of computations.

Now we interest ourselves into polynomial bounds for the length of computations. We start by a technical lemma, which proves that, during a computation, the potential structure heat increase due to the application of a computation 3-cell is polynomially bounded by the size of the arguments.

Lemma 4.5 *We denote by* K *the constant bounding the number of structure 2-cells in the 2-target of every computation 3-cell. Let* φ *be a function with* m *inputs. We denote by* S_φ *the polynomial* $K \cdot P_\varphi^2$*. Let* t *be a family of values of type* $s_1(\varphi)$*, let* f *and* g *be 2-paths such that* $t \star_1 \varphi$ *reduces into* f *which itself reduces into* g *by application of a computation rule* α*. Then the following inequality holds:*

$$[f]_S + S_\varphi(\|t^1\|, \ldots, \|t^m\|) \geq [g]_S.$$

Proof. The complete, technical proof is in [2]. Here we recall the main reasoning steps. We denote by $\alpha : a \Rrightarrow b$ the computation 3-cell used to reduce f into g. We decompose f and g to make a and b appear and use the properties of current and heat maps to conclude that the inequality $[f]_S + [b]_S(i_1, \ldots, i_m) \geq [g]_S$ holds, for some natural numbers $i_1, \ldots, i_m$. Then we prove that $[b]_S(i_1, \ldots, i_m)$ is polynomially bounded by the size of t. By definition of the structure heat, $[b]_S(i_1, \ldots, i_m)$ is the sum of all the structure heats produced by the structure 2-cells b is made of. Then we use proposition 4.3 to prove that the current incoming in each input of each structure 2-cell of b is bounded by $P_\varphi(\|t^1\|, \ldots, \|t^m\|)$. Then, by definition of $[\cdot]_S$ on structure 2-cells, we conclude that the structure heat produced by each one is at most $P_\varphi^2(\|t^1\|, \ldots, \|t^m\|)$. Finally, we use the fact that b is the 2-target of a computation 3-cell to deduce that there is at most K structure 2-cells in b. □

Example 4.6 For the polygraphic program of example 2.5 we have $K = 1$, $S_{\bullet}(x) = x^2$, $S_{\blacktriangle}(x) = x^2$ and $S_{\blacktriangledown}(x, y) = (x + y)^2$. For the one of example 2.3, we have $K = 1$, $S_{\blacktriangledown}(x, y) = (x + y)^2$ and $S_{\blacktriangledown}(x, y) = x^2y^2$.

Now let us prove that the length of a computation is polynomially bounded by the size of the arguments.

Proposition 4.7 *Let* φ *be a function with* m *inputs. We define the following polynomials:*

$$Q_\varphi(x_1, \ldots, x_m) = [\varphi](ax_1, \ldots, ax_m) \qquad \textit{and} \qquad R_\varphi = Q_\varphi \cdot (1 + S_\varphi).$$

Let t *be a family of values of type* $s_1(\varphi)$*, let* F *be a* 3*-path with* 2*-source* $t \star_1 \varphi$*, made of* k *computation* 3*-cells and* l *structure* 3*-cells. Then the following inequalities hold:*

$$k \leq Q_\varphi(\|t^1\|, \ldots, \|t^m\|)) \qquad \textit{and} \qquad l \leq Q_\varphi(\|t^1\|, \ldots, \|t^m\|)) \cdot S_\varphi(\|t^1\|, \ldots, \|t^m\|)).$$

As a consequence, $|||F||| \leq R_\varphi(\|t^1\|, \ldots, \|t^m\|)$ *holds.*

Proof. We decompose F into a $\star_2$-composite of elementary computation 3-paths followed by structure 3-paths. Using the fact that the heat map we consider is strictly decreasing on computation 3-cells and weakly decreasing on structure 3-cells, we deduce that $[t \star_1 \varphi]$ is minored by k. We use the properties of $[\cdot]$ and lemma 4.2 to get the bound we seek on k. Then, we apply proposition 4.7 to each of the structure 3-paths we have isolated. We sum up the resulting inequalities and use the facts that $[t \star_1 \varphi]_S = 0$ and $[t_2(F)]_S \geq 0$ to get $k \cdot S_\varphi(\|t^1\|, \ldots, \|t^m\|) \geq l$. We deduce the inequality on l from this one and the one on k. We conclude by using the equality $|||F||| = k + l$. □

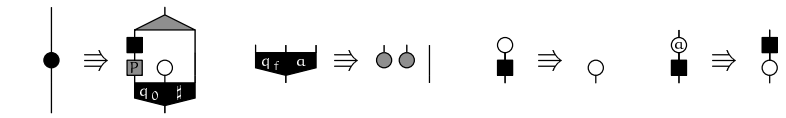
Example 4.8 For the functions of example 2.5, we have $Q_{\blacklozenge}(x) = 2x^2$, $Q_{\blacktriangle}(x) = x$ and $Q_{\blacktriangledown}(x, y) = x + y$. For example, let us fix a list t. The polynomial $Q_{\blacklozenge}$ tells us that, during the computation of the sorted list $\blacklozenge(t)$, there will be at most $\|t\|$ applications of a computation 3-cell. The polynomial $R_{\blacklozenge}$ guarantees that there is no more than $\|t\|^2\,(1+\|t\|^2)$ applications of rules. On the examples we have considered, the polynomial Q_φ gives a bound that is close to known ones but the polynomial R_φ gives a very overestimated bound. To get a better estimation, we will have to work on the structure heat increase bound S_φ.

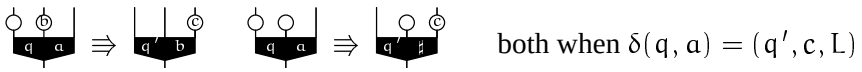
Theorem 4.9 *Functions computed by simple polygraphic programs are exactly* PTIME *functions.*

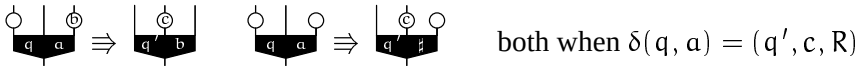
Proof. We start by proving that functions computed by simple polygraphic programs are in PTIME. Proposition 4.7 tells us that the length of any computation in such a polygraph are polynomially bounded by the size of the arguments. Furthermore, each step of computation can be done in polynomial time with respect to the size of the current 2-path: we find a redex in a directed acyclic graph with decorations then replace it by the corresponding reduce and both operations can be done in polynomial time.

Now let us prove that any PTIME function can be computed by a simple polygraphic program. The first step is to translate a Turing machine equipped with a clock into a polygraphic program. We fix a function f in PTIME, a Turing machine $\mathcal{M}$ that computes f and a polynomial P that bounds the length of the computation. We consider a copy of the polygraphic program of example 2.3 which computes addition and multiplication of natural numbers, with its 1-source denoted by $\mathtt{nat}$. Let us note that this polygraphic program computes any polynomials, including P. Then we extend it with a variant of the polygraphic Turing machine of section 2, made of a 1-cell $\mathtt{mon}$; its constructors are the empty word $\varnothing : \mathtt{mon} \Rightarrow \mathtt{mon}$ and each letter $\oplus : \mathtt{mon} \Rightarrow \mathtt{mon}$ of the alphabet of $\mathcal{M}$; its functions are the main $\blacklozenge : \mathtt{mon} \Rightarrow \mathtt{mon}$ for f, plus a size function $\blacksquare : \mathtt{mon} \Rightarrow \mathtt{nat}$, plus the modified 

$: \mathtt{nat} \star_0 \mathtt{mon} \star_0 \mathtt{mon} \Rightarrow \mathtt{mon}$ for each state q of $\mathcal{M}$ and each letter a in the alphabet of $\mathcal{M}$, including the blank symbol $\sharp$; its computation 3-cells are:







Then, one checks that this polygraphic program mimics the transition of the original Turing machine $\mathcal{M}$ and, thus, computes f. We conclude by checking that the following polygraphic interpretation, extending the one already built on natural numbers, is simple and compatible with each computation 3-cell:

- $\bigcirc_* = 1$, $\overset{a}{\bigcirc}_*(i) = i + 1$, $\blacksquare_*(i) = i$, $\underset{q\ a}{\blacksquare}_*(i, j, k) = i + j + k$, $\blacklozenge_*(i) = P_*(i) + i + 1$.
- $\left[\blacksquare\right](i) = i$, $\left[\underset{q\ a}{\blacksquare}\right](i, j, k) = i$, $\left[\blacklozenge\right](i) = [P](i) + P_*(i) + i + 1$.

□

References

[1] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion and Hlne Touzet, *Algorithms with polynomial interpretation termination proofs*, Journal of Functional Programming 11 (2001), no. 1, 33–53.

[2] Guillaume Bonfante and Yves Guiraud, *Programs as polygraphs: computability and complexity*, Submitted, 2006.

[3] Albert Burroni, *Higher-dimensional word problems with applications to equational logic*, Theoretical Computer Science 115 (1993), no. 1, 43–62.

[4] Adam Cichon and Pierre Lescanne, *Polynomial interpretations and the complexity of algorithms*, Lecture Notes in Artificial Intelligence 607 (1992), 139–147.

[5] Yves Guiraud, *Termination orders for 3-dimensional rewriting*, Journal of Pure and Applied Algebra 207(2006), no. 2, 341–371.

[6] ———, *The three dimensions of proofs*, Annals of Pure and Applied Logic 141 (2006), no. 1-2, 266–295.

[7] ———, *Polygraphs for termination of left-linear term rewriting systems*, Submitted, 2007.

[8] Yves Lafont, *Interaction nets*, Principles of Programming Languages, ACM Press, 1990, pp. 95–108.

[9] ———, *Towards an algebraic theory of boolean circuits*, Journal of Pure and Applied Algebra 184 (2003), no. 2-3, 257–310.

[10] ———, *Algebra and geometry of rewriting*, Preprint IML, 2006.

[11] Yves Lafont and Franois Mtayer, *Polygraphic resolutions and homology of monoids*, Preprint IML, 2006.

[12] Dallas Lankford, *On proving term rewriting systems are noetherian*, Tech. report, Louisiana Tech University, 1979.

[13] Franois Mtayer, *Resolutions by polygraphs*, Theory and Applications of Categories 11 (2003), 148–184.

[14] Detlef Plump, *Term graph rewriting*, Handbook of Graph Grammars and Computing by Graph Transformation 2 (1999), 3–61.

[15] John von Neumann, *Theory of self-reproducing automata*, University of Illinois Press, 1966.

Rewritings for Polarized Multiplicative and Exponential Proof Structures

Christophe Fouqueré[1,2]

LIPN-UMR7030
Université Paris 13, CNRS
Villetaneuse, France

Virgile Mogbil[1,3]

LIPN-UMR7030
Université Paris 13, CNRS
Villetaneuse, France

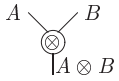
Abstract

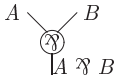
We study conditions for a concurrent construction of proof-nets in the framework of linear logic following Andreoli's works. We define specific correctness criteria for that purpose. We first study the multiplicative case and show how the correctness criterion given by Danos and decidable in linear time, may be extended to *closed modules* (i.e. validity of polarized proof structures). We then study the exponential case and give a correctness criterion by means of a contraction relation that helps to discover frontiers of exponential boxes.

Keywords: linear logic, proof-nets, logic programming, focalization.

1 Introduction

Girard in his seminal paper [9] gave a parallel syntax for multiplicative linear logic (MLL) as oriented graphs called *proof-structures*. Let us recall that a MLL formula is either an atomic formula A, a negation of an atomic formula $A^{\perp}$, or built with a binary connective $\otimes$ or $\parr$. In the original definition, a proof-structure for MLL is constructed by means of the following binary links:

$\otimes$-link: 

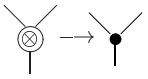
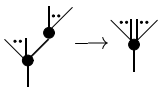
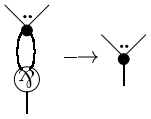
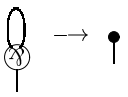
$\parr$-link: 

axiom-link: 

[1] Partially supported by ACI NIM project Géométrie du Calcul (GEOCAL), France.
[2] Email: cf@lipn.univ-paris13.fr
[3] Email: vm@lipn.univ-paris13.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

where every occurrence of formula is a premise of at most one link and is a conclusion of exactly one link. A *correctness criterion* enables one to distinguish sequentializable proof-structures (the so called *proof-nets*) from "bad" structures (that do not correspond to proofs in the sequent calculus). After Girard's long trip correctness criterion, numerous equivalent properties were found. In particular, Danos and Regnier [7] proved that *switched* proof-structures should be trees, where switching is done by deleting one of the premises of each $\parr$-link. Danos [6] showed that it is the case iff the proof structure rewrites to $\bullet$ ($\otimes$ is called a contracted node):

(1)  (2)  (3)  (4) 

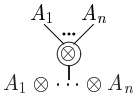
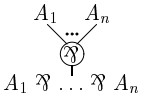
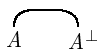
While a lot of research has been done on finding efficient correctness criteria for MLL, it still remains to study correctness criteria in case of *polarized* proof-structures in MLL, and broaden it to the exponential case. First used by Andreoli in Logic Programming [1] and also considered in Girard's works [10] and in Laurent's works about Polarized Linear Logic [13], this concept of polarization allows to consider clustered structures. Recently, polarized proof structures arose naturally in logic programming models [2,3,4]. The basic objects we consider are then proof structures with two strata we call *elementary bipolar modules*, that may be composed to get *modules*. We recall the multiplicative case in the following section (the reader may find in [8] extension to open modules). We define a correctness criterion that takes care of the parallel structure of modules, extending the Danos criterion. In section 3, we analyze how modules may be generalized to take care of exponentials.

2 The multiplicative case

We consider in this section the extension MLLu of MLL with 1 the unit of $\otimes$. Formulae F of MLLu are given by the following grammar (we allow 1 either alone or as part of a tensor):

$$\begin{aligned} F &:= 1 \mid G \\ G &:= A \mid A^{\perp} \quad \texttt{atomic formula or its negation} \\ &\quad \mid G \otimes 1 \mid 1 \otimes G \mid G \otimes G \mid G \parr G \end{aligned}$$

A binary sequent calculus for MLLu is given in Fig. 1. Let $\mathcal{PS}$ be the directed graphs where edges are labelled by formulae of MLLu and built with the following links ($n \geq 1$):

$\otimes$-link:  $\parr$-link:  axiom-link:  1-link: 

possibly with edges pending downwards. Elements of $\mathcal{PS}$ are still called *proof structures*. Formulae labelling pending edges are the *conclusions* of the proof structure, nodes with pending edges are called *conclusion nodes*. A proof structure is *sequentializable* if the sequent defined with the conclusions of the proof structure is provable

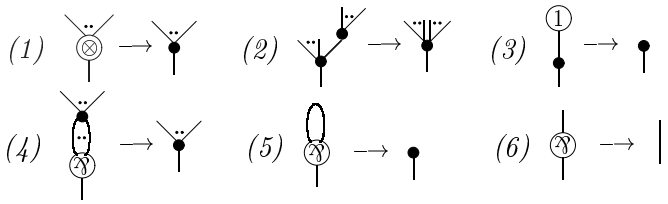
$$\frac{}{\vdash A^{\perp}, A}\ (axiom) \qquad \frac{}{\vdash 1}\ (1) \qquad \frac{\vdash \Gamma, A \quad \vdash A^{\perp}, \Delta}{\vdash \Gamma, \Delta}\ (cut)$$

$$\frac{\vdash \Gamma, G_1 \quad \vdash G_2, \Delta}{\vdash \Gamma, G_1 \otimes G_2, \Delta}\ (\otimes) \qquad \frac{\vdash G_1, G_2, \Gamma}{\vdash G_1 \parr G_2, \Gamma}\ (\parr)$$

Fig. 1. Binary sequent calculus for MLLu.

in MLLu. A sequentializable proof structure is called a *proof-net*. Labels on edges are omitted when clear from the context.

Proposition 2.1 *Let π be a proof structure of $\mathcal{PS}$, π is a proof-net (i.e. sequentializable) iff $\pi \rightarrow^* \bullet$ where $\rightarrow$ is given by the following rules:*

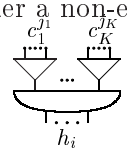


In case (4), there must exist at least one edge between the two nodes.

The proof of the proposition follows from the standard one on binary proof structures for MLL [6], and the following remarks: $\otimes$ and $\parr$ are associative and commutative, the 1-ary $\parr$ connective is by convention the identity, 1 is a unit for $\otimes$.

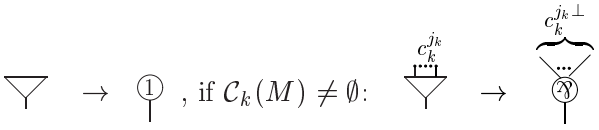
We first give the definition of an *elementary bipolar module* (EBM) and give the correspondence with proof structures. We then define a *module* as the composition of EBMs. A module is correct if the corresponding proof structure is sequentializable.

Definition 2.2 [EBM] An *EBM* M is given by a finite set $\mathcal{H}(M)$ of propositional variables (called hypotheses) h_i and a non empty finite set $\mathcal{C}(M)$ varying over k of finite sets $\mathcal{C}_k(M)$ of propositional variables (called conclusions) c_k^j. Variables are supposed pairwise distinct.[4] The set of propositional variables appearing in M is noted $v(M)$. It is denoted as a directed graph with labelled pending edges and two kinds of nodes, one *positive pole* under a non-empty finite set of *negative poles*:

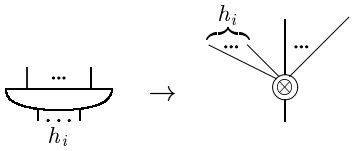


The set of pending edges of an EBM M is called the *border* $b(M)$.

The proof structure corresponding to an EBM is given by the following transformation on poles. The converse transformation requires the definition of BMs defined later.



[4] This restriction is taken for simplicity. The framework can be generalized if we consider multisets (of hypotheses and conclusions) instead of sets, and add as required a renaming mechanism: the results in this paper are still true.

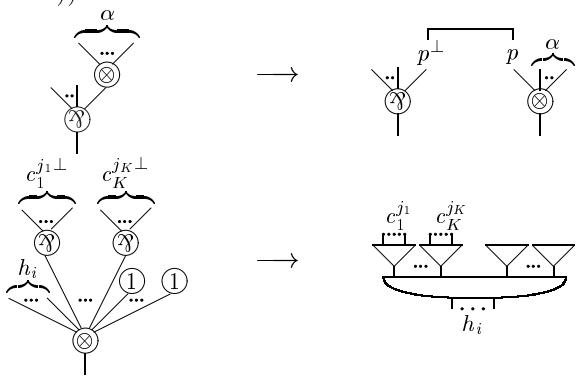


An EBM M may be equivalently defined as a (type) formula $t(M)$ in the dual language of MLLu (recall that $A \multimap B = A^{\perp} \parr B$): $t(M) = (\bigotimes_i h_i) \multimap (\parr_k (\bigotimes_{j_k} c_k^{j_k}))$, where we use the convention that $\parr_k F_k = \bigotimes_k F_k = F_1$ when the domain of k is of cardinal 1, and if the domain of i is empty, $(\bigotimes_i h_i) \multimap C = C$ and if the domain of j_k for some k is empty, $(\bigotimes_{j_k} c_k^{j_k}) = \perp$. The reader should care that this supposes a bilateral sequent calculus, although the logical reading of an EBM (or of a proof structure) is unilateral. Three kinds of EBMs are of special interest: An EBM is *initial* (resp. *final*) if its set of hypotheses is empty (resp. its set of conclusions is empty). An EBM is *transitory* if it is neither initial nor final. Initial EBMs allow to declare available resources, though final EBMs stop part of a computation by withdrawing a whole set of resources. Transitory EBMs are called definite clauses in standard logic programming.

Definition 2.3 [BM] A *bipolar module* (BM) M is defined with hypotheses $\mathcal{H}(M)$, conclusions $\mathcal{C}(M)$, and type $t(M)$, inductively in the following way:

- An EBM is a BM.
- Let M be a BM, and N be an EBM, let $I = \mathcal{C}(M) \cap \mathcal{H}(N)$, their *composition* wrt the *interface* I, $M \circ_I N$ is a BM with the multiset of hypotheses $\mathcal{H}(M) \cup (\mathcal{H}(N) - I)$, the multiset of conclusions $(\mathcal{C}(M) - I) \cup \mathcal{C}(N)$, the type $t(M) \otimes t(N)$ and variables $v(M) \cup v(N)$.

The interface will be omitted when it is clear from the context. Note that the interface may be empty. The translation from proof structures of $\mathcal{PS}$ to BMs is given by the two following rules, plus rules not explicited here due to lack of space that take care of polarity (a unary tensor node (resp. Par) is added in between if (resp. a negation of) a propositional variable is a premise of a Par node (resp. tensor)) and the constant 1:



where p is a fresh atomic formula

Considering BMs in place of proof structures for MLLu has valuable consequences in terms of simplicity of correctness criteria as one can take care of the bipole structure of BMs more directly than it is the case with a binary structure.

Definition 2.4 [Correctness (wrt sequentialization)] Let M be a BM, M is *correct* if the corresponding proof structure in $\mathcal{PS}$ is sequentializable.

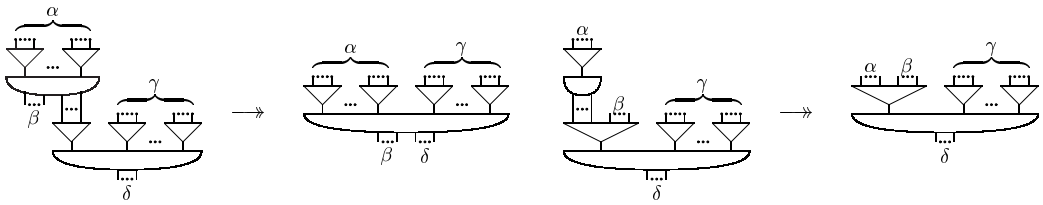


Fig. 2. Big step reduction relation.

Sequentialization means that there exists a formula C built with the connectives $\otimes$ and $\parr$, and the variables $\mathcal{C}(M)$ such that the sequent $\mathcal{H}(M), t(M) \vdash C$ is provable in Linear Logic.

A closed module is a BM without any pending edges, i.e. with the sets of hypotheses and conclusions empty. Correctness of closed modules may be tested either in terms of provability in a sequent calculus or by means of correctness criteria for proof structures. In the following, we consider the correctness criteria of Danos [6] using a contraction relation and explained in the previous section, and also the one given by Danos and Regnier [7] that uses switchings: let π be a proof structure with binary links and $S(\pi)$ the set of (switched) graphs obtained from π by removing exactly one premise edge for each $\parr$ link, π is a proof net iff each graph in $S(\pi)$ is acyclic and connected. One generalizes this definition to n-ary connectives by introducing generalized switches: each n-ary $\parr$ connective induces n switched graphs. One still can define switched proof-structures and a criterion generalizing Danos-Regnier correctness criterion on $\mathcal{PS}$: a proof structure π is a proof net iff the graphs in $S(\pi)$ are acyclic and connected. A closed module M is *DR-correct* if the proof structure M^* associated to M is a proof net wrt the previous criterion. We abusively refer to the module M instead of the corresponding proof structure M^* in the following, speaking of for instance switched module instead of switched proof structure. We immediately have the following proposition as a corollary of the Danos and Regnier criterion theorem:

Proposition 2.5 *Let M be a closed module, M is correct iff M is DR-correct.*

We give below a (big step) reduction relation that takes care of the *focalization* property. Though a Danos-like relation would reduce each step one variable, our formulation uses as a whole the structure of a module thanks to focalization. The focalization property states that a sequent is provable iff there exists a proof such that decomposition of the positive stratum of formulae is done in one step. Considering bipolar modules, it means that one may define a reduction relation such that each step reduces one positive-negative pair of nodes.

Proposition 2.6 (Stability) *Let M and N be two closed modules such that $M \twoheadrightarrow N$, M is correct iff N is correct (see Fig. 2).*

Proof. One can define a function from the switched structures of the module on the left of the relation onto the switched structures associated to the module on the right such that a switched structure from the left is acyclic (resp. connected) iff the corresponding switched structure from the right is acyclic (resp. connected). □

Theorem 2.7 (correctness) *A closed module M is correct iff $M \twoheadrightarrow^* \bigtriangledown$.*

Proof.

- Suppose $M \twoheadrightarrow^* \bigtriangledown$. As $\bigtriangledown$ is correct, by prop. 2.6, we deduce that M is correct.
- Suppose M is correct. Let N be a normal form of M wrt $\twoheadrightarrow$, then by proposition 2.6, N is correct. Let us define a partial relation on negative poles of N: let m and n be two negative poles, $m < n$ if $\exists p$ a positive pole such that m is linked to the bottom of p and n is linked to the top of p. We consider the transitive closure of this relation. We prove a contradiction if N is in normal form, correct and different from $\bigtriangledown$:
  - either there is no maximal negative pole. Let us suppose $\exists m$ such that $m < m$. Then there exists one cycle containing m in the module alternating positive and negative poles. We can then define a switching function on the module (choosing the correct links for negative poles) such that the switched module has a cycle. Hence contradiction with the fact that N is correct.
  - or let m be a maximal negative pole and p the corresponding positive pole.
    If p has other negative poles, N is not in normal form as we can omit the maximal negative pole by neutrality.
    If p has no other negative poles and no incoming link then N is either equal to $\bigtriangledown$ or not connected hence not correct.
    If p has no other negative poles and each incoming negative pole has at least one link going to another positive pole, then one can define a switching function using for each of these negative poles one of the links that does not go to p: the switched module is not connected. Hence contradiction with the fact that N is correct.
    If p has no other negative poles and there exists one incoming negative pole with the whole set of links going to p, the first rule applies: N is not in normal form.

□

Note that this proof extensively uses the bipolar nature of modules. Moreover, the proof may have been given considering minimal poles in place of maximal poles, and for each proof only one of the two reduction rules is sufficient and necessary! Finally, the same technique Guerrini [11] used for Danos criterion may be applied here to get a linear algorithm. We detailed in another paper the extension of the technique presented before to open modules as it is a necessary step towards the specification of a logic programming language based on bipolar modules [8].

3 Dealing with exponentials

3.1 Multiplicative exponential linear logic (MELL)

Adding exponentials to the language obviously increases its expressivity: it allows for representing reusable resources. In linear logic, the 'of course' modality ! has this main property: $!A \multimap A \otimes \cdots \otimes A$. Technically, three operations are necessary: *contraction*, *dereliction* and *weakening*. The first operation states that $!A$ is duplicable. Dereliction allows to consider the classical formula $!A$ as the linear one A. The last operation states that $!A$ may be forgotten. The dual modality 'why not' ?

may be interpreted in the following way: $?A^{\perp}$ waits for the 'classical' resource $!A$. This *promotion* operation is more complex than the other operations: in terms of proofnets, correctness is assured if a 'box' in the proof net characterizes the context (and this context has to be correct by itself). Entries of such a box are given by one ! and a set of ?.

3.1.1 From MELLu to ?-EBMs.
The translation from formulae of MELL to modules is not as easy as it is without exponentials. We consider an extension MELLu of MELL with the neutral element 1 for $\otimes$, a formula F of MELLu is given by the following grammar:

$$F := 1 \mid G$$
$$G := A \mid A^{\perp} \mid G \otimes 1 \mid 1 \otimes G \mid G \otimes G \mid G \parr G \mid ?G \mid !G$$

Converting from formulae to modules requires the use of polarization and focalization. Focalization allows to consider n-ary connectives. Formulae are polarized negatively or positively according to their main connectives, considering conveniently that variables $A, B, \dots$ are positive whereas their negations $A^{\perp}, B^{\perp}, \dots$ are negative. A precise study of the exponential connectives leads to the acknowledgment that exponential connectives change the polarity of formulae: if A is a positive formula, $?A$ is negative whereas $!A^{\perp}$ is positive. Hence exponential connectives may be split into two parts: $!A^{\perp} = \downarrow\sharp A^{\perp}$ and $?A = \uparrow\flat A$. The shift connectives $\downarrow$ and $\uparrow$ do the changing of polarities. The introduction of shift connectives may be generalized also to the linear case whenever there is a change of polarity. The two modalities $\flat$ and $\sharp$ express exponentiality.

We consider a slightly different version of a polarized system as it was designed by Boudes [5] or Laurent [13]: the system $\mathrm{LL}_{\mathtt{pol}}$ given by Laurent takes care of multiplicative as well as additive connectives where atomic formulae are always exponentialized. Following our motivations, our language ${}_n\mathrm{MELL}_{\mathtt{pol}}$ is restricted to the multiplicative case for simplicity and atomic formulae may be linear or exponential. Finally we use n-ary connectives and the decomposition of exponentials is explicit. The grammar for ${}_n\mathrm{MELL}_{\mathtt{pol}}$ is given in the following way where the set of formulae is explicitly split into positive $(P, \dots)$ and negative $(N, \dots)$ formulae (A is a positive atomic formula):

$$\begin{cases} P := \bigotimes_{i\in I} \rho_i \mid \flat(\bigotimes_{i\in I} \rho_i) \\ \rho := A \qquad\quad \mid \downarrow N \end{cases} \qquad \begin{cases} N := \parr_{k\in K} \nu_k \mid \sharp(\parr_{k\in K} \nu_k) \\ \nu := A^{\perp} \qquad\quad \mid \uparrow P \end{cases}$$

We keep as convention that a 1-ary tensor is the identity and a 0-ary tensor is the tensor unit $\mathbf{1}$. Moreover, one can remark that *defining* $\mathbf{1}$ as $\downarrow\sharp\top$, where $\top$ is the neutral for the additive connective $\&$, is coherent with our setting and may be useful when extending our framework to additives. Nevertheless, in the following, the standard rule for $\mathbf{1}$ is implicitly added to the calculi. One can define a n-ary focalized sequent calculus (A is an atomic formula) as in Fig. 3. Sequents contain a distinguished place between $\vdash$ and $;$, they are in one of the two following forms:

$$\frac{}{\vdash\ ;\ A^\perp, A, \flat\Xi}\ (axiom)\quad \frac{}{\vdash 1, \flat\Xi}\ (1)\quad \frac{\vdash\ ;\ \Gamma, A, \flat\Xi \quad \vdash\ ;\ A^\perp, \Delta, \flat\Xi}{\vdash\ ;\ \Gamma, \Delta, \flat\Xi}\ (cut)$$

$$\frac{\ldots \quad \vdash N_i\ ;\ \Gamma_i, \flat\Xi \quad \ldots \quad \vdash\ ;\ A_j, \Delta_j, \flat\Xi \quad \ldots}{\vdash\ ;\ \otimes_{i\in I} \downarrow N_i \otimes_{j\in J} A_j, \Gamma_1, \ldots, \Gamma_{|I|}, \Delta_1, \ldots, \Delta_{|J|}, \flat\Xi}\ (\otimes)$$

$$\frac{\ldots \quad \vdash N_i\ ;\ \flat(\otimes_{i\in I} \downarrow N_i \otimes_{j\in J} A_j), \Gamma_i, \flat\Xi \quad \ldots \quad \vdash\ ;\ \flat(\otimes_{i\in I} \downarrow N_i \otimes_{j\in J} A_j), A_j, \Delta_j, \flat\Xi \quad \ldots}{\vdash\ ;\ \flat(\otimes_{i\in I} \downarrow N_i \otimes_{j\in J} A_j), \Gamma_1, \ldots, \Gamma_{|I|}, \Delta_1, \ldots, \Delta_{|J|}, \flat\Xi}\ (\flat\otimes)$$

$$\frac{\vdash\ ;\ P_1, \ldots, P_{|I|}, A_1^\perp, \ldots, A_{|J|}^\perp, \Gamma}{\vdash \parr_{i\in I} \uparrow P_i \parr_{j\in J} A_j^\perp\ ;\ \Gamma}\ (\parr)\quad \frac{\vdash\ ;\ P_1, \ldots, P_{|I|}, A_1^\perp, \ldots, A_{|J|}^\perp, \flat\Gamma}{\vdash \sharp(\parr_{i\in I} \uparrow P_i \parr_{j\in J} A_j^\perp)\ ;\ \flat\Gamma}\ (\sharp\,\parr)$$

Fig. 3. n-ary sequent calculus for ${}_n\mathrm{MELL}_{\mathtt{pol}}$ (0-ary tensor is $\mathbf{1}$).

$\vdash\ ;\ \Gamma$ or $\vdash N\ ;\ \Gamma$ where N is a negative non atomic formula and Γ is a multiset of positive formulae or atomic negative formulae. The sequent calculus is designed such that, beginning with the distinguished place empty, search for proofs consists of repeating the decomposition of a positive formula followed by the decomposition of negative formulae (necessarily subformulae of the positive formula just decomposed), until applying axioms. Note that exponential rules are as possible integrated to linear rules to quotient the search space (e.g. the axiom rule includes $(\flat w)$, $(\flat\otimes)$ manages $(\flat c)$). The following translation $(-)^-$ from MELLu to ${}_n\mathrm{MELL}_{\mathtt{pol}}$ is such that if F is a MELLu formula, $\vdash_{\mathtt{MELLu}} F$ is provable iff $\vdash_{{}_n\mathtt{MELL}_{\mathtt{pol}}} F^-$; is provable:

$\mathbf{1}^+ = \mathbf{1}$	$A^+ = A$	$(F_1 \otimes F_2)^+ = F_1^+ \otimes F_2^+$	$(!F)^+ = \downarrow\sharp F^-$	$F^+ = \downarrow F^-\,\mathtt{otherwise}$
	$A^{\perp -} = A^\perp$	$(F_1 \parr F_2)^- = F_1^- \parr F_2^-$	$(?F)^- = \uparrow\flat F^+$	$F^- = \uparrow F^+\,\mathtt{otherwise}$

The final step to get modules consists in flattening ${}_n\mathrm{MELL}_{\mathtt{pol}}$ formulae. Bipolar modules were previously obtained by adding atomic formulae between two strata (say from negative to positive): let P_1, P_2 be positive formulae, N a negative formula, $\vdash P_1 \otimes (N \parr P_2)$ is provable iff $\vdash P_1 \otimes (N \parr Z^\perp), Z \otimes P_2$ is provable, where Z is a fresh (positive) atomic formula. However this principle cannot be fully applied when exponentials occur: try to flatten the (provable) sequent $\vdash A^\perp \parr \uparrow\flat(B \otimes C), A \otimes \downarrow\sharp(B^\perp \parr C^\perp)$. This can be overcome by allowing exponential atomic formulae in the language. These exponential atomic formulae are noted with $\sharp$ or $\flat$ superscripts: $Z^\sharp$ and $Z^\flat$ are respectively defined as $\downarrow\sharp\uparrow Z$ and $\uparrow\flat\downarrow Z^\perp$. We then consider the translation $(-)^\circ$: let $\mathcal{C}$ be a non-empty context (negative or positive), Z is a fresh atomic formula

$$\mathcal{C}[\uparrow \otimes_{i\in I} \rho_i]^\circ = \mathcal{C}[Z^\perp]^\circ, [Z \otimes_{i\in I} \rho_i]^\circ$$
$$\mathcal{C}[\uparrow\flat \otimes_{i\in I} \rho_i]^\circ = \mathcal{C}[Z^\flat]^\circ, [\flat(Z^\sharp \otimes_{i\in I} \rho_i)]^\circ$$

otherwise (i.e. empty context) $P^\circ = P, N^\circ = \downarrow N$. We still have if F is a MELLu formula, $\vdash_{\mathtt{MELLu}} F$ is provable iff $\vdash_{{}_n\mathtt{MELL}_{\mathtt{pol}}}; F^\circ$ is provable. We consider now drawings of the following kind we call ?-EBM:

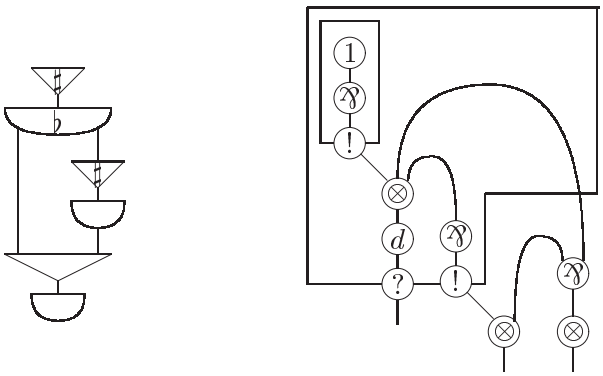
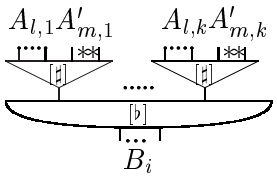
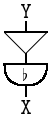


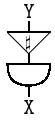
Fig. 4. ?-EBM and proofnets



Positive and negative poles may now be labelled: a ?-EBM is reusable when ♭ labels its positive part, ♯ labels a promoted variable, brackets mean optional. $*$ labels an exponential atomic negative conclusion of a ?-EBM and we refer to $*$-edge in that case. Roughly, the correspondence between places of exponentials in formulae and labelled elements is the following one:

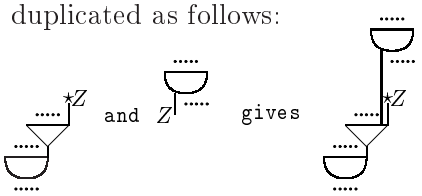
$!(X \multimap Y)$ is drawn with the positive pole labelled ♭: 

$X \multimap !Y$ is drawn with a $*$-edge: 

$X \multimap ?Y$ is drawn with the negative pole labelled ♯: 

The type of a ?-EBM generalizes the type given for an EBM (brackets mean optional): $\mathbb{C} = [!](\bigotimes_{i\in I}\ B_i \multimap \parr_{k\in K}\ [?]\ (\bigotimes_{l\in L}\ A_{l,k} \bigotimes_{m\in M}\ Z^{\sharp}_{m,k}))$. Such a type (*clause* in logic programming terminology) could be interpreted as: $\mathbb{C}$ is a reusable clause iff ! is explicit. The application of a clause is allowed if the B_i are available, then one of the conclusions is fired, a conclusion being a multiset of atomic formulae $A_{l,k}$ or exponential, i.e. reusable, atomic formulae $Z^{\sharp}_{m,k}$. If the ? modality is present, the multiset of conclusions is required to be reusable as a whole: not only these conclusions cannot be used with a linear clause but such a clause cannot use linear hypotheses. For example, consider the set of clauses $\{1 \multimap A \otimes B, B \multimap ?C, !(A \otimes C) \multimap \bot\}$. The corresponding module we get is drawn in Fig.4 on the left. The figure on the right is the corresponding proof-structure (see [9,12] for definitions of proof structures with boxes, extended here to n-ary connectives). The traversal of the box without the use of a ♭-node shows that the sequent is not provable (a dereliction should have been applied), i.e. the ?-EBM on the left is not correct.

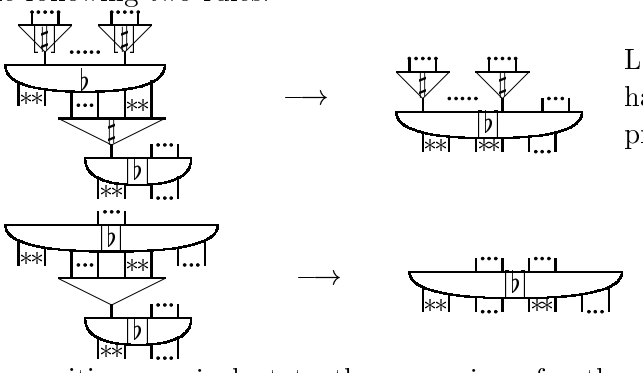
3.1.2 From ?-EBMs to modules.
Definitions given in section 2 for EBMs, that is to say composition and correctness of modules, cannot be straightfully extended to the exponential case. Obviously, composition should satisfy identification of variables occurring on links, noticing that $*$-edges can only be linked to $*$-edges. However, contraction needs a special attention. For the following, we consider explicit contraction: ?-EBMs with positive nodes labelled $\flat$, and $*$-edges are duplicated if necessary mimicking the property $!A \multimap !A \otimes A$, hence the degree of edges is always 1. The definition of composition given in section 2 is then adapted consequently for ?-EBMs labelled $\flat$ and $*$-edges. For example, $*$-edges are duplicated as follows:



It is then possible to define the type $t(M)$ of a module M as the formula given as the Par of the formulae occurring as ?-EBMs taking care of possible contractions. Moreover, it is possible to recover a proof-structure M^* (with, as usual, contraction, weakening and dereliction nodes) from a given module M. Finally, a module is *correct* if M^* is a proofnet.

3.2 ?-EBMs and corresponding correctness criteria

Extending the language with exponentials yields a major difficulty due to the promotion rule, as it is inherently contextual. Note that allowing $\flat$ in the language (and exclude $\sharp$) is sufficient to embed the framework of the previous sections in a programming language: one can consider a program as a set of (exponential, reusable) EBMs along with a multiset of (linear, usable once) EBMs. This system already extends classical logic programming in a straightforward way and correctness of modules is tested with the same reduction relation given in previous section, after deleting $*$-edges (application of the weakening rule) and by considering that normal forms may contain ?-EBMs. We consider for the full language the reduction system given by the following two rules:



Label $\flat$ is put on right hand side if option is present on left part

Label $\flat$ is put on right hand side if the two options are present on left part

Propositions equivalent to the ones given for the multiplicative case may be proved. Obviously, if M is a closed correct module in this fragment then the module $forget(M)$ built from M forgetting exponentials (omitting labels and replacing $*$-

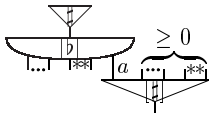
edges by normal edges) is a closed correct BM. We must also characterize normal forms. We add to the reduction system two rules correponding to neutrality of **1** and weakening of $\flat$:

Proposition 3.1 (Stability) *Let M and N be two closed modules such that $M \longrightarrow N$. The module M is correct iff N is correct.*

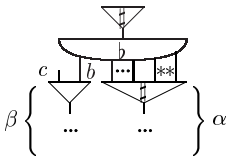
Proof. One can define a function from left switched module onto right switched module such that the relation and its inverse are stable wrt acyclicity, connectedness. □

Theorem 3.2 *A closed module M is correct iff $M \longrightarrow^* \nabla$ or $M \longrightarrow^* \nabla_{\flat}$.*

Proof. The proof used for the linear case is adapted here. As the reduction rules are stable wrt correctness, it remains to prove that a correct non-terminal closed module M can always be reduced. We consider the same relation as in proof of Th.2.7. If maximal negative poles do not exist then there exists at least one cycle in the $forget(M)$ module alternating positive and negative poles. We can then define a switching function on this module (choosing the correct links for negative poles) such that this switched module has a cycle. Hence contradiction. So let us consider one of the maximal negative poles, and the corresponding positive pole. We remark that such a negative pole has no outcoming links (the module is closed and the negative pole is maximal). If the positive pole has other negative poles, we can omit the maximal negative pole by neutrality. Otherwise, let us study the incoming negative poles: (1) If there is no such incoming link, then M is the terminal module. (2) If each incoming negative pole has at least one link a going to another positive pole as in the following figure:



then one can define a switching function using for each of these negative poles one of the link that does not go to the positive pole we considered first. Hence the $forget(M)$ switched module is not connected (there are no outgoing links). Hence contradiction. (3) Else there exists at least one incoming negative pole α with the whole set of links associated to the positive pole: the reduction rules apply and we are finished or this positive pole is linearly linked with b to a negative pole β. Such β is not $\sharp$-marked otherwise it corresponds to a proof-structure with an exponential box with two principal ports, hence contradiction. The reduction rules apply to β (and then to α) or these exists a link c from β to another positive pole as in the following figure:



then one can define a switching function using the c link but not b: the corresponding switched proof-structure contains an unconnected component in the exponential box induced by the ($\sharp$-marked) α negative. Hence contradiction. This holds because the α links are all linear or none are linear. (4) Finally, there exists at least one incoming negative pole α with the whole set of links associated to the positive pole itself not linearly linked: the reduction rules apply. □

Corollary 3.3 *If F is a provable formula then there exists a correct (closed) module M such that $t(M) = F$.*

4 Conclusion

We first adapt the classical rewriting criterion of Danos to the n-ary bipolar case for testing the correctness of closed modules. We show in particular that polarization greatly simplifies the rewriting procedure. We extend our results to the exponential case. In particular, we give a local criterion for testing correctness of modules in presence of exponentials. Note that current criteria presupposes that 'boxes' are already given, although our reduction relation helps to discover it. These results may be useful in designing concurrent logic programming languages, in the style suggested by Andreoli in recent papers, as it extends his works by removing constraints on programming objects.

References

[1] Andreoli, J.-M., *Logic programming with focusing proofs in linear logic.*, J. Log. Comput. **2** (1992), pp. 297–347.

[2] Andreoli, J.-M., *Focussing and proof construction*, Annals of Pure and Applied Logic **107** (2001), pp. 131–163.

[3] Andreoli, J.-M., *Focussing proof-net construction as a middleware paradigm*, in: A. Voronkov, editor, *CADE*, Lecture Notes in Computer Science **2392** (2002), pp. 501–516.

[4] Andreoli, J.-M. and L. Mazaré, *Concurrent construction of proof-nets*, in: M. Baaz and J. A. Makowsky, editors, *CSL*, Lecture Notes in Computer Science **2803** (2003), pp. 29–42.

[5] Boudes, P., *Projecting games on hypercoherences.*, in: J. Díaz, J. Karhumäki, A. Lepistö and D. Sannella, editors, *ICALP*, Lecture Notes in Computer Science **3142** (2004), pp. 257–268.

[6] Danos, V., "Une application de la logique linéaire à l'étude des processus de normalisation (principalement de λ-calcul)," Ph.D. thesis, Université Denis Diderot, Paris 7 (1990).

[7] Danos, V. and L. Regnier, *The structure of multiplicatives*, Archive for Mathematical Logic **28** (1989), pp. 181–203.

[8] Fouqueré, C. and V. Mogbil, *Rewritings in polarized (partial) proof structures*, in: F. L. Paola Bruscoli and J. Stewart, editors, *1st Workshop on Structures and Deductions*, Technical Report **ISSN 1430-211X** (2005), pp. 95–109.

[9] Girard, J.-Y., *Linear logic*, Theoretical Computer Science **50** (1987), pp. 1–102.

[10] Girard, J.-Y., *On the unity of logic.*, Ann. Pure Appl. Logic **59** (1993), pp. 201–217.

[11] Guerrini, S., *Correctness of multiplicative proof nets is linear*, in: *Logic in Computer Science*, 1999, pp. 454–463.
URL citeseer.ist.psu.edu/guerrini99correctness.html

[12] Lafont, Y., *From proof-nets to interaction nets*, in: J.-Y. Girard, Y. Lafont and L. Regnier, editors, *Advances in Linear Logic*, London Mathematical Society Lecture Note **222**, Cambridge University Press, 1995 pp. 225–247, proceedings of the Workshop on Linear Logic, Ithaca, NewYork, June 1993.

[13] Laurent, O., *Syntax vs. semantics: a polarized approach*, Theoretical Computer Science **343** (2005), pp. 177–206.

Deduction Graphs with Universal Quantification

Herman Geuvers, Iris Loeb[1]

Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands

Abstract

Deduction Graphs are meant to generalise both Gentzen-Prawitz style natural deductions and Fitch style flag deductions. They have the structure of acyclic directed graphs with boxes. In [2] we have investigated the deduction graphs for minimal proposition logic. This paper studies the extension with first-order universal quantification, showing the robustness of the concept of deduction graphs.

Keywords: Natural deduction, universal quantification, cut-elimination.

1 Introduction

In this paper we extend deduction graphs, DGs, of [2], with first-order universal quantification. In [2] we have presented deduction graphs for minimal propositional logic (only implication) as a formalism for "natural deduction with sharing". The natural deductions become acyclic directed graphs with *boxes* to delimit the scope of local assumptions. The boxes are used in the $\rightarrow$-introduction rule. Figure 1 presents an example of a deduction graph that represents a deduction of B (node 9) from the hypotheses $A \rightarrow A \rightarrow B$ and $(A \rightarrow B) \rightarrow A$ (nodes 3 and 7).

The arrow represents (inverse) derivability, so e.g. node 9 (B) is derived from nodes 6 ($A \rightarrow B$) and 8 (A). Similarly node 6 ($A \rightarrow B$) is derived from 5 (B) while discharging the "free" nodes (i.e. cancelling the assumptions) 1 and 2 (A). Deduction graphs are singled out from a larger set of graph-structures, the so called *closed box directed graphs*, cbdg, which basically are labelled directed graphs with *boxes*, where a box is a collection of nodes, $\mathcal{B}$. Each box $\mathcal{B}$ corresponds to a node, the *box node* of $\mathcal{B}$. In a cbdg it is required that two boxes are disjoint or one is contained in the other; there is only one outgoing edge from a box node and that edge points into the box itself; apart from the edge from the box node, there are no edges pointing into a box.

[1] Email: H.Geuvers@cs.ru.nl, I.Loeb@cs.ru.nl

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

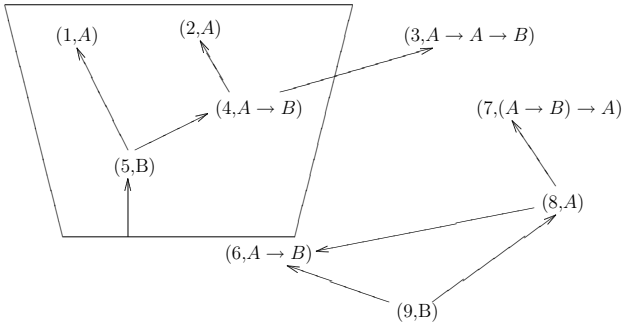


Fig. 1. Deduction graph in the implicational fragment

To make all this precise, we repeat some definitions of [2].

Definition 1.1 A *closed box directed graph* is a triple $\langle X, G, (\mathcal{B}_i)_{i\in I}\rangle$ where X is a set of *labels*, G is a directed graph where all nodes have a label in X and $(\mathcal{B}_i)_{i\in I}$ is a collection of sets of nodes of G, the *boxes*. Each box $\mathcal{B}_i$ corresponds to a node, the *box node* of $\mathcal{B}_i$. Moreover, the boxes $(\mathcal{B}_i)_{i\in I}$ should satisfy the following properties.

(i) (Non-overlap) Two boxes are disjoint or one is contained in the other: $\forall i, j \in I(\mathcal{B}_i \cap \mathcal{B}_j = \emptyset \vee \mathcal{B}_i \subset \mathcal{B}_j \vee \mathcal{B}_j \subset \mathcal{B}_i)$,

(ii) (box node edge) There is only one outgoing edge from a box node and that points into the box itself (i.e. to a node in the box),

(iii) (No edges into a box) Apart from the edge from the box node, there are no edges pointing into a box.

Definition 1.2 Let G be a closed box directed graph. A *box-topological ordering* of G is a linear ordering $<$ of the nodes of G, such that for all nodes n_0, n_1 of G:

(i) If $n_0 \longrightarrow n_1$, then $n_1 < n_0$.

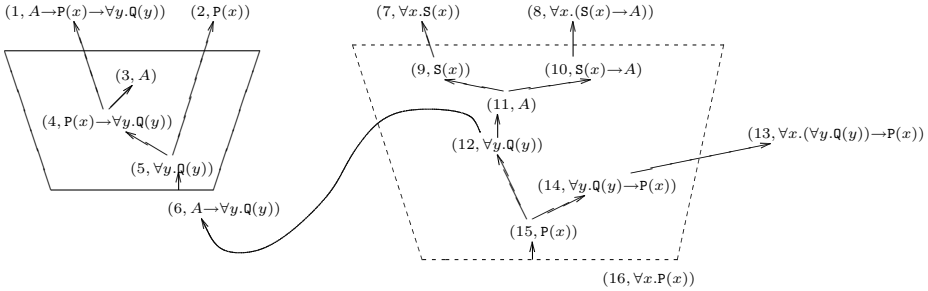
(ii) If n_0 is the box node of a box containing n_1, then $n_1 < n_0$.

Definition 1.3 Let $\langle G, (\mathcal{B}_i)_{i\in I}\rangle$, be a closed box directed graph and let n_0 and n_1 be nodes in this graph.

- Node n_1 is *in scope of* n_0 if n_0 is in all boxes that n_1 is in. In a formula: $\forall i \in I(n_1 \in \mathcal{B}_i \Rightarrow n_0 \in \mathcal{B}_i)$. (So the nodes in scope of n_0 are the nodes that are in 'wider' boxes.)
- The nodes n_0 and n_1 are *at the same depth*, when n_0 is in scope of n_1, and n_1 is in scope of n_0. Node n_0 is *at a greater depth* than n_1, when n_1 is in scope of n_0, but n_0 is not in scope of n_1.
- Node n_1 is a *top-level node* if n_1 is not contained in any box.
- The *free nodes* are the top-level nodes that have no outgoing edges.

Originally, boxes were meant to border the scope of a local assumption, but now we also use boxes to border the scope of a quantifier: When we do a $\forall$-introduction, we create a box with box node $\forall x.\varphi$. To carry this extension through we have to consider how to deal with the side condition on the $\forall$-introduction rule, which

is stated in Gentzen-Prawitz style natural deduction as follows: "the *eigenvariable* does not occur free in any of the non-discharged assumptions". (The eigenvariable is the quantified variable x in the introduction of $\forall x.\varphi$.) In $\mathtt{DG_{US}}$ we want to represent this by a more "local" side condition. A first idea would be to require that *there is no edge pointing out of the box to a formula in which the eigenvariable occurs free*, like usually done in Fitch deductions. (So when we introduce $\forall x.\varphi$, the box we create should not have edges pointing out to a node ψ with $x \in \mathrm{FV}(\psi)$.) However, this would cause severe problems in the cut-elimination procedure, as the following graph shows. The ∀-box has been depicted with a dashed line.



There is a hidden →-cut in node 12: The implication has been introduced in node 6 and is then immediately eliminated using node 11 to derive node 12. The cut is hidden because nodes 6 and 12 are not at the same *depth*. So we first have to do an *incorporation* step, moving the box with box node 6 into the box with box node 16. [2]

The eigenvariable of the ∀-box is x. If we would do an incorporation directly, there would be arrows from inside the ∀-box to the nodes 1 and 2 outside the box, in which x occurs free. This is forbidden. We therefore first have to do a renaming of the eigenvariable, like shown in Fig. 2.

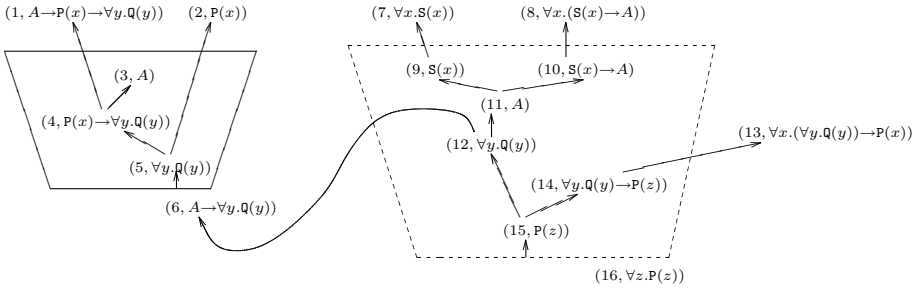


Fig. 2. Renaming of eigenvariable.

This renaming is not so trivial because it not only involves nodes inside the box but also the x in node 16. But when we rename x in node 16, we also have to rename it in nodes that refer to 16, and propagate that through the graph. This could thus involve any node of G, eventually even nodes 1 and 2. Renaming is hence not just complicated, but it might a priori not even solve the problem.

As this looks like Gentzen-Prawitz style natural deduction, why doesn't the

[2] This is explained in detail in [2]. We now just remark that eliminating the cut directly includes adding an edge from 3 to 11. This does not yield a DG because the edge would be pointing into box, so we have to incorporate first.

necessity to rename variables occur in that formalism? There is no sharing in the example graph, so we can present the deduction faithfully as a tree in the following way (where $[A]^1$ denotes the discharging (or cancelling) of hypothesis A at the the application of the logical rule referred to by 1):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A]^1 \quad A{\to}\mathtt{P}(x){\to}\forall y.\mathtt{Q}(y)}{\mathtt{P}(x){\to}\forall y.\mathtt{Q}(y)}}{\mathtt{P}(x)} \quad \dfrac{\dfrac{\forall x.\mathtt{S}(x)}{\mathtt{S}(x)} \quad \dfrac{\forall x.(\mathtt{S}(x){\to}A)}{\mathtt{S}(x){\to}A}}{A}}{\forall y.\mathtt{Q}(y)}\,1}{A{\to}\forall y.\mathtt{Q}(y)} \quad \dfrac{\forall x.(\forall y.\mathtt{Q}(y){\to}\mathtt{P}(x))}{\forall y.\mathtt{Q}(y){\to}\mathtt{P}(x)}}{\forall y.\mathtt{Q}(y)}}{\mathtt{P}(x)}}{\forall x.\mathtt{P}(x)}$$

But this is not a correct Gentzen-Prawitz style natural deduction, as the variable x occurs free in the non-discharged assumption $A{\to}P(x){\to}\forall y.Q(y)$ when it gets bound in the $\forall$-I rule introducing $\forall x.P(x)$. Apparently the $\forall$-introduction rule in Gentzen-Prawitz style natural deduction is strict enough to prevent the need for renaming variables during $\forall$-cut-elimination.

Our solution is to use two sets of variables: free variables, Var and bound variables BVar and to rename the free variable with a fresh bound variable when doing the $\forall$-introduction. Furthermore, we require that the eigenvariable is unique for that box (i.e. it does not occur anywhere outside the box). A further discussion of the choice of syntax can be found in Section 3.1.

In Section 2 we give the definition of deduction graphs with universal quantification, called $\mathtt{DG_U s}$, starting from definitions for terms and formulas of first-order predicate logic. The process of cut-elimination is discussed in Section 3, followed by strong normalisation in Section 4. Finally, Section 5 compares $\mathtt{DG_U s}$ with developments in proof nets.

2 Definition

Different from the language of first-order predicate logic for Gentzen-Prawitz style natural deduction [1,6], we define the language Pred of first-order predicate logic with universal quantification and equality for deduction graphs to have two kinds of variables. The first kind, Var denoted by $u, v, w, \ldots$, are meant to be used as *free* variables. The second kind, BVar, denoted by $x, y, z, \ldots$, will only be used *bound*. The same idea is often used for the language of first-order predicate logic for Fitch style flag deductions [7].

We now define the terms, basic formulas, formulas and axioms of Pred.

Definition 2.1 (i) The set of *terms* of Pred, Term is defined as follows.

$$\mathsf{Term} ::= \mathsf{Var} \mid \mathtt{F}(\mathsf{Term}, \ldots, \mathsf{Term})$$

where $\mathtt{F}$ is a function symbol with a fixed arity and the length of the sequence of terms following it should be equal to the arity of $\mathtt{F}$.

(ii) The set of *formulas* of Pred, Form, is defined as follows.

$$\mathsf{Form} ::= \mathsf{R}(\mathsf{Term}, \ldots, \mathsf{Term}) \mid \mathsf{Term} = \mathsf{Term} \mid \mathsf{Form} \to \mathsf{Form} \mid \forall x.\mathsf{Form}[x/u]$$

where R is a relation symbol with a fixed arity and the length of the sequence of terms following it should be equal to the arity of R; x ranges over BVar and u over Var. So, in the $\forall x.\mathsf{Form}[x/u]$ case, when we introduce the $\forall$, we also replace a free variable (u) by a bound one (x).

We adopt the following convention for the brackets in formula: we omit brackets around $\to$-formulas by letting $\to$ associate to the right; $\forall$ binds stronger than $\to$; outer brackets are not written, nor are any other brackets that do not contribute to our understanding of the formula.

So, for example, $\forall x.\varphi \to \psi \to \xi \equiv ((\forall x.\varphi) \to (\psi \to \xi))$. Note, however, that $\forall x.\mathtt{P}(x) \to \mathtt{Q}(x)$ can formally only be understood as $(\forall x.(\mathtt{P}(x) \to \mathtt{Q}(x)))$, because $((\forall x.\mathtt{P}(x)) \to \mathtt{Q}(x))$ is not a formula. In these cases we will write the inner brackets under the quantifier explicitly anyway, for the convention would otherwise lead us to misinterpret the formula.

Because Pred deviates from the language of first-order predicate logic for Gentzen-Prawitz natural deduction, this also means that the $\forall$-introduction for deduction graphs cannot be similar to the $\forall$-introduction in the Gentzen-Prawitz formalism.

The $\forall$-introduction in Gentzen-Prawitz style natural deduction is as follows:

$$\dfrac{\dfrac{D}{\varphi}}{\forall x.\varphi}$$

Where x may not be free in the non-discharged assumptions of D. This means that x might be free in φ, although it is bound in $\forall x.\varphi$.

In deduction graphs we will introduce a fresh (bound) variable in the $\forall$-introduction step. The advantage is then, that a deduction graph is still well-formed, when we rename only free variables. We will use this later, in the process of cut-elimination.

Definition 2.2 The collection of *deduction graphs for first-order universal quantification*, $\mathtt{DG_U}$ is the set of closed box directed graphs over $\mathbb{N} \times \mathsf{Pred}$ inductively defined as follows:

Axiom A single node (n, A) is a deduction graph,

$\to$-E If G is a deduction graph containing two nodes $(n, A \to B)$ and (m, A) at the top level, then the graph $G' := G$ with
- a new node (p, B) at the top level
- an edge $(p, B) \longrightarrow (n, A \to B)$,
- an edge $(p, B) \longrightarrow (m, A)$,

is a deduction graph.

$\to$-I If G is a deduction graph containing a node (j, B) with no ingoing edges and a finite set of free nodes with label A, $(n_1, A), \ldots, (n_k, A)$, all at the top level, then

the graph $G' := G$ with
· A box $\mathcal{B}$ with box node $(n, A{\to}B)$, containing the nodes (j, B) and (n_1, A), $\ldots, (n_k, A)$ and no other nodes that were free in G,
· An edge from the box node $(n, A{\to}B)$ to (j, B)
is a deduction graph under the proviso that it is a closed box directed graph.

Repeat If G is a deduction graph containing a node (n, A) at the top level, the graph $G' := G$ with
· a new node (m, A) at the top level,
· an edge $(m, A) \longrightarrow (n, A)$
is a deduction graph.

$\forall$-I If G is a $\mathtt{DG_U}$ containing a node (j, φ) with no ingoing edges at top-level for some formula φ of Pred, then the graph $G' := G$ with
· A box $\mathcal{B}$ with box node $(n, \forall x.\varphi[x/u])$, not containing any nodes without outgoing edges, where we call u the *eigenvariable* of $\mathcal{B}$ if u occurs in φ,
· An edge from the box node $(n, \forall x.\varphi[x/u])$ to (j, φ)
is a $\mathtt{DG_U}$ under the proviso that: G' is a well-formed closed box directed graph and u does not occur in the label of any node that is not in $\mathcal{B}$.

$\forall$-E If G is a $\mathtt{DG_U}$ with a node $(n, \forall x.\varphi)$ at top-level for some formula φ of Pred, then the graph $G' := G$ with
· a node $(p, \varphi[t/x])$ where none of the variables of t is the eigenvariable of any box of G,
· an edge from $(p, \varphi[t/x])$ to $(n, \forall x.\varphi)$
is a $\mathtt{DG_U}$.

Join$_\mathtt{U}$ If G and G' are two $\mathtt{DG_U}$s then $G'' = G \cup G'$ is a $\mathtt{DG_U}$ under proviso that the eigenvariables of G and the eigenvariables of G' are disjoint.

So the rules for $\mathtt{DG_U}$ are the ones for $\mathtt{DG}$ with the $\forall$-I and $\forall$-E rules added and the Join rule slightly modified.

Example 2.3 Let $\mathtt{P}$ and $\mathtt{Q}$ be unary predicate symbols of Pred. Figure 3 shows an example of an $\mathtt{DG_U}$. The graph is constructed by adding the nodes in their numerical order: first nodes 1 and 2 by Axiom, then 3 and 4 by $\forall$-E, then 5 by $\to$-E, then 6 by $\forall$-I and then 7 by $\to$-I.

Lemma 2.4 *Let G be a $\mathtt{DG_U}$. Then for every variable u:*

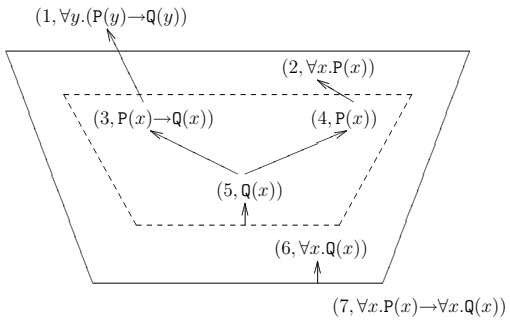


Fig. 3. An example of a $\mathtt{DG_U}$.

(i) *u occurs as eigenvariable of a box of G at most once;*

(ii) *If u is an eigenvariable of a box $\mathcal{B}$ of G, it does not occur in a label of a node outside $\mathcal{B}$.*

We formulate a criterion to check relatively easily whether a given closed box directed graph is a $\mathtt{DG_U}$ (Lemma 2.5). As an important notion for $\mathtt{DG_U}$s is the eigenvariable of a box, we need a similar notion for general closed box directed graphs. So, we call a variable u a *box-variable of* $\mathcal{B}$, if u does not occur in the label of the box node of $\mathcal{B}$, but it does occur in the label of the node that the box node points to. Remark that for $\mathtt{DG_U}$s the notion of eigenvariable and the notion of box-variable coincides. We also recall from [2] the notion of a *box-topological ordering*: $>$ is a box-topological ordering of G if it is a linear ordering of the nodes of G, such that $n \longrightarrow m \Rightarrow n > m$ and if $\mathcal{B}$ has box node n and $m \in \mathcal{B}$, then $n > m$.

Lemma 2.5 *A finite closed box directed graph G is a $\mathtt{DG_U}$ if and only if the following hold*

(i) *If u is a box-variable of a box $\mathcal{B}$ of G, it does not occur in a label of a node outside $\mathcal{B}$.*

(ii) *There is a box-topological ordering $>$ of G.*

(iii) *Every node n of G is of one of the following six types:*

A *It has no outgoing edges.*

$\rightarrow$-E *It has label B and has exactly two outgoing edges: one to a node $(m, A{\rightarrow}B)$ and one to a node (p, A), both within the scope of n.*

$\rightarrow$-I *It is a box node of a box $\mathcal{B}$ with label $A{\rightarrow}B$ and has exactly one outgoing edge, which is to a node (j, B) inside the box $\mathcal{B}$ (and not in any deeper boxes) with no other ingoing edges. All nodes inside the box without outgoing edges have label A.*

R *It has label A and has exactly one outgoing edge, which is to a node (m, A) that is within the scope of n.*

$\forall$-E *It has label $\varphi[t/x]$ for some formula φ, some term t and some variable x, and n has exactly one outgoing edge to a node $(m, \forall x.\varphi)$ within the scope of n.*

$\forall$-I *It is a box node of a box $\mathcal{B}$ with label $\forall x.\varphi$ for some variable x and some formula φ, and has exactly one outgoing edge, which is to a node $(j, \varphi[u/x])$ inside the box $\mathcal{B}$ (and not in any deeper boxes). Node $(j, \varphi[u/x])$ has no other ingoing edges and there are no nodes without outgoing edges in $\mathcal{B}$.*

Proof. $\Rightarrow$:By induction on the definition of deduction graph. $\Leftarrow$:By induction on the number of nodes of G, distinguishing according to the type of (one of) the maximal node (in the box-topological ordering) of G. □

3 Cut-elimination

We now also encounter a "detour" in a proof, when a $\forall$-introduction is immediately followed by a $\forall$-elimination. Definition 3.3 describes the elimination of a safe $\forall$-cut.

Not all $\forall$-cuts are safe, so it might be necessary to apply some transformations to make them safe. These transformations are the same ones as for $\rightarrow$-cuts: repeat-elimination, unsharing, and incorporation. The only difference with the transfor-

mations on DGs is, that unsharing has become a little more involved, because of the eigenvariable requirement.

Definition 3.1 A $\forall$-*cut* in a $\mathtt{DG_U}$ G is a subgraph of G consisting of:

- a box node $(n, \forall x.\varphi)$,
- a node $(p, \varphi[t/x])$,
- a sequence of R-nodes $(s_0, \forall x.\varphi), \ldots, (s_i, \forall x.\varphi)$,
- wdges $(p, \varphi[t/x]) \longrightarrow (s_i, \forall x.\varphi) \longrightarrow \ldots \longrightarrow (s_0, \forall x.\varphi) \longrightarrow (n, \forall x.\varphi)$.

We call the node $(n, \forall x.\varphi)$ the *major premiss* and we call the node $(p, \varphi[t/x])$ the *conclusion*.

Similarly, in a $\rightarrow$-cut, we call $(n, A{\rightarrow}B)$ the major premiss and the node (p, B) the conclusion.

Definition 3.2 Let $\mathcal{B}$ be the box associated to box node n. A $(\forall/\rightarrow)$-cut in a $\mathtt{DG_U}$ G is *safe* if the following requirements hold:

- there is an edge from the conclusion to the major premiss and that is the only edge to the major premiss;
- the major premiss and the conclusion are at the same depth (relative to the box structure);

Definition 3.3 The process of *eliminating a safe* $\forall$-*cut* is the following operation on $\mathtt{DG_U}$s (see Figure 4):

- change the labels ψ of the nodes in the box of n, to $\psi[t/u]$;
- remove the box and box node $(n, \forall x.\varphi[x/u])$;
- add an edge from $(p, \varphi[t/x])$ to $(j, \varphi[t/u])$ (the node that n pointed to).

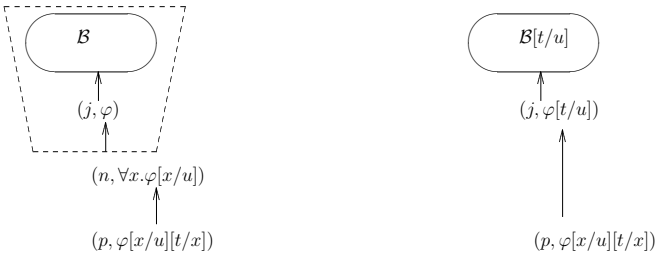


Fig. 4. Schematic presentation of a safe $\forall$-cut elimination.

Lemma 3.4 *If G is a $\mathtt{DG_U}$ with safe $\forall$-cut c and G' is obtained from G by eliminating c, then G' is also a $\mathtt{DG_U}$.*

Proof. By Lemma 2.5. □

We can generalise repeat-elimination, unsharing, and incorporation without much ado. Because after unsharing we still want every eigenvariable to occur just once, this step now includes the renaming of eigenvariables of copied boxes.

Definition 3.5 Let G be a $\mathtt{DG_U}$ with a cut with major premiss (n, φ) and conclusion (p, ψ). Suppose G contains a node (n_0, φ), an R-node (n_1, φ) and edges $n_1 \longrightarrow n_0$ and $p \longrightarrow n_1$. The *repeat-elimination* at n_0, n_1, p is obtained by:

- When an edge points to n_1, redirect it to n_0;
- Remove n_1.

Lemma 3.6 *For G a $\mathtt{DG_U}$ with a cut with major premiss (n, φ) and conclusion (p, ψ). Suppose G contains a node (n_0, φ), an R-node (n_1, φ) and edges $n_1 \longrightarrow n_0$ and $p \longrightarrow n_1$, the repeat-elimination of at n_0, n_1, p is also a $\mathtt{DG_U}$.*

Proof. By Lemma 2.5. □

Definition 3.7 Let G be a $\mathtt{DG_U}$ with a $\forall$-box $\mathcal{B}$ with eigenvariable u. Let v be a fresh variable. Then the *renaming of u by v* is the graph G in which the labels ψ of the nodes of $\mathcal{B}$ have been replaced by $\psi[v/u]$.

Lemma 3.8 *Let G be a $\mathtt{DG_U}$ with a $\forall$-box $\mathcal{B}$ with eigenvariable u. Let v be a fresh variable. Then the renaming of u by v is a $\mathtt{DG_U}$.*

Proof. By Lemma 2.5. □

Definition 3.9 Let G be a $\mathtt{DG_U}$ with a cut c with major premiss n. Suppose n is a box node of a box $\mathcal{B}$ and has $k \geq 2$ ingoing edges, from $p_1, \ldots, p_k$. Then the *unsharing of G at nodes $n, p_1, \ldots p_n$* is obtained by:

- making a box $\mathcal{B}'$ that contains a copy of all nodes and edges of $\mathcal{B}$,
- copy all outgoing edges of $\mathcal{B}$ to $\mathcal{B}'$ (thus if we had $q \longrightarrow m$ with $q \in \mathcal{B}$, $q' \in \mathcal{B}'$ and $m \notin \mathcal{B}$, then we add $q' \longrightarrow m$, where q' is the copy of $q \in \mathcal{B}'$,
- letting $p_2, \ldots, p_k$ point to n' (the box node of $\mathcal{B}'$) instead of n;
- renaming the eigenvariable of $\mathcal{B}'$ and of all boxes contained in $\mathcal{B}'$.

Lemma 3.10 *Let G be a $\mathtt{DG_U}$ with a cut c with major premiss n. Suppose n is a box node of a box $\mathcal{B}$ and has $k \geq 2$ ingoing edges, from $p_1, \ldots, p_k$. Then the unsharing of G at nodes $n, p_1, \ldots p_n$ is a $\mathtt{DG_U}$.*

Proof. By Lemma 2.5. □

Definition 3.11
We have a *depth-conflict* in the $\mathtt{DG_U}$ G, if G contains a cut with major premiss n and conclusion p at a greater depth, such that there is an arrow from p to n and that is the only arrow to n. In that case the *incorporation* of G at n, p is obtained by moving $\mathcal{B}_n$, i.e. the box of n, into the box at the lowest depth that includes p but excludes n.

Lemma 3.12 *Suppose G is a $\mathtt{DG_U}$ with a depth conflict. Then the incorporation at the major premiss and the conclusion is a $\mathtt{DG_U}$.*

Proof. By case analysis on the incorporating box. Then by Lemma 2.5. □

Definition 3.13 Given a $\mathtt{DG_U}$ G with a cut c, the *process of $\rightarrow/\forall$-cut elimination* is the following;

(i) (Repeat elimination) As long as there is no edge from the conclusion to the major premiss, perform the appropriate repeat-elimination as described in Definition 3.5;

(ii) (Unsharing) If there is an edge from the conclusion to the major premiss, but this is not the only edge to the major premiss, perform an appropriate unsharing step, as defined in Definition 3.9;

(iii) (Incorporation) As long as the conclusion is at a greater depth than the major premiss, perform the appropriate incorporation step, as described in Definition 3.11.

(iv) (Eliminating a safe cut) If c is safe, perform either the safe $\to$-cut-elimination step, or the safe $\forall$-cut-elimination step, as defined in Definition 3.3.

3.1 Discussion

We have made some choices in the definition of $\mathtt{DG_U s}$ that facilitate the process of cut-elimination. Except for the choice of the language, which has already been discussed in the Introduction, these are:

(i) We deviate from the side-condition for the $\forall$-introduction rule as normally used in Fitch-style flag deduction, as discussed in the Introduction.

(ii) We require the uniqueness of the eigenvariables.

Suppose we would adopt the Fitch-style side-condition for the $\forall$-introduction rule, then this results in having to do an additional renaming in the incorporation step in some cases.

If we would abandon the requirement of unique eigenvariables and adopt the Fitch-style side-condition, this would move renaming from the unsharing step to the incorporation step.

4 Strong Normalisation

To obtain strong normalisation for cut-elimination on $\mathtt{DG_U s}$, we extend the λ-calculus with tupling as defined in [2], and prove strong normalisation for it. Then a reduction preserving translation from $\mathtt{DG_U s}$ to this calculus is defined.

The strong normalisation result we thus get is relatively weak: It is assumed that first one cut is made safe and is eliminated, before handling another cut.

For Gentzen-Prawitz natural deduction, strong normalisation for cut-elimination can be proven by (1) defining a $\to$-cut preserving translation to the $\to$-fragment and (2) showing that an infinite $\forall$-cut reduction is impossible. That might also work for the $\mathtt{DG_U}$ case, but (2) is now problematic, because a $\forall$-cut contraction may involve unsharing and then other $\forall$-cuts may be copied. We therefore opt for a direct proof of strong normalisation for cut-elimination for $\mathtt{DG_U s}$.

Definition 4.1 The typed *expressions* $\mathsf{T}_{\langle\rangle}$ and *types* of the $\lambda{\to}\langle\rangle$*-calculus* for first order predicate logic with universal quantification are defined as follows.

(i) For $\varphi \in \mathsf{Form}$, all variables x^{φ} are of type φ.

(ii) If T is of type $\varphi{\to}\psi$ and S is of type φ, (TS) is of type ψ.

(iii) If T is of type φ, then $\lambda x^{\psi}.T$ is of type $\psi{\to}\varphi$.

(iv) If T is of type $\forall x.\varphi$ and t is a term, then (Tt) is of type $\varphi[t/x]$.

(v) If T is of type φ, then $\lambda y.T[u := y]$ is of type $\forall y.\varphi[y/u]$.

(vi) If $T_1, \ldots, T_n$ are of types $\varphi_1, \ldots \varphi_n$ respectively, $\langle T_1, \ldots, T_n\rangle$ is an expression of type φ_1.

Definition 4.2 The reduction rules for the expressions are as follows:

$$\begin{aligned}
(\lambda x^{\sigma}.M)N &\longrightarrow_{\bar\beta} \langle M, N\rangle \text{ if } x \notin \mathrm{FV}(M)\\
(\lambda x^{\sigma}.M)N &\longrightarrow_{\bar\beta} M[x := N] \text{ if } x \in \mathrm{FV}(M)\\
(\lambda y.M)t &\longrightarrow_{\bar\beta} M[y := t]\\
\langle M, P_1, \ldots, P_k\rangle N &\longrightarrow_{\bar\beta} \langle MN, P_1, \ldots, P_k\rangle\\
\langle M, P_1, \ldots, P_k\rangle t &\longrightarrow_{\bar\beta} \langle Mt, P_1, \ldots, P_k\rangle\\
N\langle M, P_1, \ldots, P_k\rangle &\longrightarrow_{\bar\beta} \langle NM, P_1, \ldots P_k\rangle\\
\langle \ldots, \langle M, P_1, \ldots, P_k\rangle, \ldots\rangle &\longrightarrow_{\bar\beta} \langle \ldots, M, P_1, \ldots, P_k, \ldots\rangle
\end{aligned}$$

As can be observed from the typing and the reduction rules, the $N_1, \ldots, N_k$ in $\langle M, N_1, \ldots, N_k\rangle$ act as a kind of 'garbage'. The order of the terms in $N_1, \ldots, N_k$ is irrelevant and we therefore consider terms *modulo* permutation of these vectors, which we will write as $\equiv_p$.

Definition 4.3 Given a deduction graph G and a node n in G, we define the λ-term $\langle\![G, n]\!\rangle$ as follows (by induction on the number of nodes of G).

A If (n, A) has no outgoing edges, $\langle\![G, n]\!\rangle := x_n^A$,

$\to\mathsf{E}$ If $(n, B) \longrightarrow (m, A{\to}B)$, and $(n, B) \longrightarrow (p, A)$, define $\langle\![G, n]\!\rangle := \langle\![G, m]\!\rangle\, \langle\![G, p]\!\rangle$.

R If $(n, A) \longrightarrow (m, A)$, define $\langle\![G, n]\!\rangle := \langle\![G, m]\!\rangle$

$\to\mathsf{I}$ If $(n, A{\to}B)$ is a box node with $(n, A{\to}B) \longrightarrow (j, B)$, the free nodes of the box are $n_1, \ldots, n_k$ and the nodes without incoming edges inside the box are $m_1, \ldots, m_p$, then

$$\langle\![G, n]\!\rangle := \lambda x^A.\langle\langle\![G, j]\!\rangle, \langle\![G, m_1]\!\rangle, \ldots, \langle\![G, m_p]\!\rangle\rangle[x_{n_1} := x, \ldots, x_{n_k} := x].$$

$\forall\mathsf{E}$ If $(n, \varphi[t/y]) \longrightarrow (p, \forall y.\varphi)$, define $\langle\![G, n]\!\rangle := \langle\![G, p]\!\rangle t$.

$\forall\mathsf{I}$ If $(n, \forall y.\varphi[y/u]) \dashrightarrow (j, \varphi)$ and the nodes without incoming edges are $m_1, \ldots, m_p$, then

$$\langle\![G, n]\!\rangle := \lambda y.\langle\langle\![G, j]\!\rangle, \langle\![G, m_1]\!\rangle, \ldots, \langle\![G, m_p]\!\rangle\rangle[u := y]$$

The interpretation of the deduction graph G, $\langle\![G]\!\rangle$, is defined as $\langle\langle\![G, r_1]\!\rangle, \ldots, \langle\![G, r_l]\!\rangle\rangle$, where $r_1, \ldots, r_l$ are the top-level nodes without incoming edges in the deduction graph G.

Definition 4.4 A $\lambda{\to}\langle\rangle$ *context* is given by the following abstract syntax $K[-]$.

$$K[-] := [-] \mid \mathsf{T}_{\langle\rangle} K[-] \mid K[-]\mathsf{T}_{\langle\rangle}$$

So a $\lambda{\to}\langle\rangle$ *context* is a $\lambda{\to}\langle\rangle$-term consisting only of applications (no abstractions) with one open place. The following is immediate by induction on $K[-]$.

Lemma 4.5 *For all $\lambda{\to}\langle\rangle$ contexts $K[-]$ and $\lambda{\to}\langle\rangle$-terms $M, N_1, \ldots, N_k$*

$$K[\langle M, N_1, \ldots, N_k\rangle] \twoheadrightarrow_{\bar{\beta}} \langle K[M], N_1, \ldots, N_k\rangle.$$

Lemma 4.6 ($\forall$ Cut-elimination is $\bar{\beta}$-reduction in $\lambda{\to}\langle\rangle$)
If G' is obtained from G by a $\forall$-cut-elimination, then $(\!|G|\!) \twoheadrightarrow^{+}_{\bar{\beta}} (\!|G'|\!)$.

Proof. By induction on the structure of G. □

Theorem 4.7 *The process of cut-elimination is terminating for* $\mathtt{DG_U}$*s.*

Proof. Suppose it is not terminating. Then by Lemma 4.6, we have an infinite reduction in $\lambda{\to}\langle\rangle$, but $\lambda{\to}\langle\rangle$ is strongly normalising (see [2]). □

5 Connection with Proof Nets

In [3] we have seen a correspondence between a variant of DGs and proof nets of MELL. We remarked that there are some superficial similarities between the two: both have boxes and both enable sharing (contraction). Using this, we were able to define a translation from these deduction graphs to proof nets that preserves reduction.

In the way they handle quantification proof nets also seem fairly close to deduction graphs. In the early days [4] boxes were used to delimit the scope of a quantification. Later (see for example [8]), this was put aside and replaced by global correctness criteria. It seems plausible that in deduction graphs too we could omit boxes for this use. We have not done this as deduction graphs serve another purpose than proof nets, and leaving out the $\forall$-boxes would make the deduction graphs less perspicuous. This discrepancy in the handling of quantification does not seem to jeopardise the aim to extend the translation given in [3]: Because $(\forall x.\varphi)^* = !(\forall x.\varphi^*)$ (where $()^*$ is Girard's translation), it is the *exponential* box that should act like the $\forall$-boxes in deduction graphs anyway.

The main difficulty in both proof nets and deduction graphs is that during cut-elimination it is in some cases necessary to do a renaming. In anticipation to this, we have changed the $\forall$-introduction rule for deduction graphs and we have used two kinds of variables: one kind for bound uses, and one for free uses (see also [7]).

In [5], Girard discusses proof nets of MLL with quantifiers. Note that, as these proof nets do not include the exponential rules, this results in a simpler system. His approach is similar to ours. He replaces some free variables by constants, which reminds of our solution with two kinds of variables. He also insists on uniqueness of the eigenvariable. About renaming he says:

> In practice, it would be crazy to rename bound variables (...).

Luckily, as there is no copying going on in the cut-elimination of MLL, renaming is nowhere necessary.

This changes when we shift our attention to MELL proof nets with quantification. The most complete study of this can be found in [8], and although it handles

only second order quantification explicitly, it is generally assumed [4] [8], that first order quantifiers do not provide additional difficulties.

Here another approach has been taken. Instead of discriminating between different kinds of variables, an equivalence relation on the formulas is defined, making two formulas the same when one can be obtained from the other by renaming bound variables. Deviating from [8], this line might be pursued as follows: [3]

(i) Define formulas;

(ii) Define proof-structures;

(iii) Define the equivalence relation on formulas;

(iv) Extend the equivalence relation on proof-structures.

Once this has been done, it needs to be shown that after cut-elimination on a proof-net, one gets a proof-structure that is equivalent to a proof net.

This plan has two difficulties, the first being the exact definition of equivalence relation on proof structures. Just saying that two proof structures are equivalent, when they have the same structure and when formulas at the same places are equivalent, would not suffice. In addition, it should also consider renaming of *free* variables in formulas that will get bound somewhere else in the structure.

Secondly, it could be rather complicated to find an equivalent proof-net after cut-elimination. This problem is very similar to the ones discussed in the Introduction. It is not at all clear how this renaming can be done for example after c-b-reduction (copying a box).

Another way out would be to extend the idea used in [5], similarly to deduction graphs: Change the $\forall$-rule and work with two kinds of variables. This might very well work.

Whence proof nets with quantifiers are defined properly and completely, it seems likely that we can define a reduction-preserving translation from $\mathtt{DG_U s}$ to them.

6 Acknowledgements

We thank the referees for their useful comments.

References

[1] Gerhard Gentzen. Untersuchungen über das logische Schliessen. In M.E. Szabo, editor, *Collected Papers of Gerhard Gentzen*. North-Holland Publishing Company, 1969.

[2] Herman Geuvers and Iris Loeb. Natural deduction via graphs: Formal definition and computation rules. To appear in MSCS.

[3] Herman Geuvers and Iris Loeb. From deduction graphs to proof nets: Boxes and sharing in the graphical presentation of deductions. In Rastislav Královič and Pawel Urzyczyn, editors, *MFCS*, volume 4162 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2006.

[4] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[5] Jean-Yves Girard. Quantifiers in linear logic II. In Giovanna Corsi and Giovanni Sambin, editors, *Atti della Congresso: Nuovi Problemi della Logica e della Filosofia della Scienza, Vol. 2*, pages 79–89. Bologna: CLUEB, 1991.

[3] Personal communication Lorenzo Tortora de Falco

[6] Dag Prawitz. *Natural Deduction: a proof-theoretical study.* Stockholm: Almqvist och Wiksell, 1965.

[7] Richmond H. Thomason. *Symbolic Logic: an Introduction.* New York: Macmillan, 1970.

[8] Lorenzo Tortora de Falco. *Réseaux, cohérence et expériences obsessionelles.* PhD thesis, Université Paris VII- Denis-Diderot, 2000.

An Algebra for Directed Bigraphs*

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

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

Abstract

In this paper, we study the algebraic structure of *directed bigraphs*, a bigraphical model of computations with locations, connections and resources previously introduced by the authors as a unifying generalization of other variants of bigraphs. We give a sound and complete axiomatization of the (pre)category of directed bigraphs. Moreover, we use this axiomatization for encoding the λ-calculus, both in call-by-name and call-by-value variants, showing in this way the expressive power of directed bigraphs.

Keywords: Bigraphical models, categorical meta-models for Concurrency, λ-calculus.

1 Introduction

Bigraphical reactive systems (BRSs) are an emerging graphical meta-model of computation introduced by Milner [7,8] in which both *locality* and *connectivity* are central notions. The key structure of BRSs are *bigraphs*, which are composed by two orthogonal graph structures: a hierarchical *place graph* describing locations, and a *link (hyper-)graph* describing connections. The reaction rules, representing the dynamics of the BRS, may change both these structures. Several process calculi for Concurrency can be represented in bigraphs, such as CCS, Ambients, and (using a mild generalization called *binding bigraphs*), also the π-calculus and the λ-calculus. An important feature of bigraphs is that they support a very general construction, based on the notion of *relative pushout* (RPO) [5], which allows to turn reaction rules into labelled transition systems.

However, Milner's definition of bigraphs is not the only possible one. Sassone and Sobociński have given in [11] an alternative definition, derived from a general categorical construction, the "input-linear cospan" over a particular 2-category of place-link graphs. Also this variant enjoys a general construction of RPOs. Interestingly, Milner's and Sassone-Sobociński's variants do not coincide; in fact, these two categories and their respective RPO constructions do not generalize each other.

* Work supported by Italian MIUR project 2005015824 Art.

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

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

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

In previous work [4,3], we have presented *directed bigraphs*, a generalization of both these kinds of bigraphs. Intuitively, the idea of directed bigraphs is to notice that names are not resources on their own, but only a way for denoting (abstract) resources (i.e., edges). A system can "ask" for external resources thorugh the names on its interfaces. Thus, we can identify a "resource request flow" starting from control ports, going through names and terminating in edges. This information is represented in the new notion of *directed link graph*, which replaces the previous notion of link graphs. We have given RPO constructions for this model, generalizing and unifying the constructions independently given by Jensen-Milner and Sassone-Sobociński in their respective variants. Moreover, the very same construction can be used for calculating relative pullbacks as well.

In this paper, we continue this line of investigation. We study the algebraic structure of directed bigraphs, giving a sound and complete axiomatization of this (pre)category. Moreover, we use this axiomatization for encoding the λ-calculus, both in call-by-name and call-by-value variants. Notably, we do not need to introduce further extensions (such as binding signatures) to this end; thus, directed bigraphs turn out to be more expressive than the two variants previously proposed.

Synopsis In Section 2 we briefly recall the main definitions about the precategory $'$DBig of directed bigraphs, and the category DBig of abstract directed bigraphs. In Section 3 we analyze the algebraic structure of the precategory $'$DBig; this analysis is then carried on to the category DBig in Section 4. In Section 5 we put directed bigraphs at work, giving the encodings of λ-calculus. Conclusions are in Section 6.

2 Directed bigraphs

In this section we recall the definition and some properties of directed bigraphs; for details, we refer to [4,3]. Following Milner's approach, we work in *precategories;* see [6, §3] for an introduction to the theory of supported monoidal precategories. [3]

Let $\mathcal{K}$ be a given signature of controls, and $ar : \mathcal{K} \to \omega$ the arity function.

Definition 2.1 *A* polarized interface X *is a pair of disjoint sets of names* $X = (X^-, X^+)$*; the two components are called* downward *and* upward *faces, respectively.*

A directed link graph $A : X \to Y$ *is* $A = (V, E, ctrl, link)$ *where* X *and* Y *are the* inner *and* outer *interfaces,* V *is the set of* nodes*,* E *is the set of* edges*,* $ctrl : V \to \mathcal{K}$ *is the* control map*, and* $link : \mathsf{Pnt}(A) \to \mathsf{Lnk}(A)$ *is the* link map*, where the* ports*, the* points *and the* links *of* A *are defined as follows:*

$$\mathsf{Prt}(A) \triangleq \sum_{v \in V} ar(ctrl(v)) \qquad \mathsf{Pnt}(A) \triangleq X^+ \uplus Y^- \uplus \mathsf{Prt}(A) \qquad \mathsf{Lnk}(A) \triangleq X^- \uplus Y^+ \uplus E$$

The link map cannot connect downward and upward names of the same interface, i.e., the following condition must hold: $(link(X^+) \cap X^-) \cup (link(Y^-) \cap Y^+) = \emptyset$.

Directed link graphs are graphically depicted much like ordinary link graphs, with the difference that edges are explicit objects and points and names are associated to edges (or other names) by (simple) directed arcs. This notation makes

[3] We prefer precategories to 2-categories, because their concreteness allows for more direct definitions.

explicit the "resource request flow": ports and names in the interfaces can be associated either to locally defined resources (i.e., a local edge) or to resources available from outside the system (i.e., via an outward name).

Definition 2.2 (′DLG) *The precategory of* directed link graphs *has polarized interfaces as objects, and directed link graphs as morphisms.*

Given two directed link graphs $A_i = (V_i, E_i, ctrl_i, link_i) : X_i \to X_{i+1}$ $(i = 0, 1)$, *the composition* $A_1 \circ A_0 : X_0 \to X_2$ *is defined when the two link graphs have disjoint nodes and edges. In this case,* $A_1 \circ A_0 \triangleq (V, E, ctrl, link)$, *where* $V \triangleq V_0 \uplus V_1$, $ctrl \triangleq ctrl_0 \uplus ctrl_1$, $E \triangleq E_0 \uplus E_1$ *and* $link : X_0^+ \uplus X_2^- \uplus P \to E \uplus X_0^- \uplus X_2^+$ *is defined as follows (where* $P = \mathsf{Prt}(A_0) \uplus \mathsf{Prt}(A_1)$*):*

$$link(p) \triangleq \begin{cases} link_0(p) & \text{if } p \in X_0^+ \uplus \mathsf{Prt}(A_0) \text{ and } link_0(p) \in E_0 \uplus X_0^- \\ link_1(x) & \text{if } p \in X_0^+ \uplus \mathsf{Prt}(A_0) \text{ and } link_0(p) = x \in X_1^+ \\ link_1(p) & \text{if } p \in X_2^- \uplus \mathsf{Prt}(A_1) \text{ and } link_1(p) \in E_1 \uplus X_2^+ \\ link_0(x) & \text{if } p \in X_2^- \uplus \mathsf{Prt}(A_1) \text{ and } link_1(p) = x \in X_1^-. \end{cases}$$

The identity link graph of X *is* $id_X \triangleq (\emptyset, \emptyset, \emptyset_{\mathcal{K}}, Id_{X^- \uplus X^+}) : X \to X$.

Definition 2.3 *The* support *of* $A = (V, E, ctrl, link)$ *is the set* $|A| \triangleq V \oplus E$.

Definition 2.4 (idle, lean, open, closed, peer) *Let* $A : X \to Y$ *be a link graph.*

A link $l \in \mathsf{Lnk}(A)$ *is* idle *if it is not in the image of the link map (i.e.,* $l \notin link(\mathsf{Pnt}(A))$*). The link graph* A *is* lean *if there are no idle links.*

A link l *is* open *if it is an inner downward name or an outer upward name (i.e.,* $l \in X^- \cup Y^+$*); it is* closed *if it is an edge.*

A point p *is* open *if* $link(p)$ *is an open link; otherwise it is closed. Two points* p_1, p_2 *are* peer *if they are mapped to the same link, that is* $link(p_1) = link(p_2)$.

Proposition 2.5 *A link graph* $A : X \to Y$ *is epi iff there are no peer names in* Y^- *and no idle names in* Y^+. *Dually,* A *is mono iff there are no idle names in* X^- *and no peer names in* X^+.

A is an isomorphism iff it has no nodes, no edges, and its link map can be decomposed in two bijections $link^+ : X^+ \to Y^+$, $link^- : Y^- \to X^-$.

Definition 2.6 *The* tensor product $\otimes$ *in* ′DLG *is defined as follows. Given two objects* X, Y, *if these are pairwise disjoint then* $X \otimes Y \triangleq (X^- \uplus Y^-, X^+ \uplus Y^+)$. *Given two link graphs* $A_i = (V_i, E_i, ctrl_i, link_i) : X_i \to Y_i$ $(i = 0, 1)$, *if the tensor products of the interfaces are defined and the sets of nodes and edges are pairwise disjoint then the tensor product* $A_0 \otimes A_1 : X_0 \otimes X_1 \to Y_0 \otimes Y_1$ *is defined as* $A_0 \otimes A_1 \triangleq (V_0 \uplus V_1, E_0 \uplus E_1, ctrl_0 \uplus ctrl_1, link_0 \uplus link_1)$.

Finally, we can define the *directed bigraphs* as the composition of standard place graphs (see [6, §7] for definitions) and directed link graphs.

Definition 2.7 *A* (bigraphical) interface I *is composed by a* width *(a finite ordinal, denoted by* $width(I)$*) and by a polarized interface of link graphs (i.e., a pair of finite sets of names). A* directed bigraph *with signature* $\mathcal{K}$ *is* $G = (V, E, ctrl, prnt, link) : I \to J$, *where* $I = \langle m, X \rangle$ *and* $J = \langle n, Y \rangle$ *are its inner and outer interfaces respectively;* V *and* E *are the sets of nodes and edges respectively, and* $prnt$, $ctrl$ *and* $link$

are the parent, control and link maps, such that $G^P \triangleq (V, ctrl, prnt) : m \to n$ *is a place graph and* $G^L \triangleq (V, E, ctrl, link) : X \to Y$ *is a directed link graph.*

We denote G as combination of G^P and G^L by $G = \langle G^P, G^L \rangle$. In this notation, a place graph and a (directed) link graph can be put together iff they have the same sets of nodes and edges.

Definition 2.8 (′DBig) *The precategory* ′DBig *of directed bigraph with signature* $\mathcal{K}$ *has interfaces* $I = \langle m, X \rangle$ *as objects and directed bigraphs* $G = \langle G^P, G^L \rangle : I \to J$ *as morphisms. If* $H : J \to K$ *is another directed bigraph with sets of nodes and edges disjoint from* V *and* E *respectively, then their composition is defined by composing their components, i.e.:* $H \circ G \triangleq \langle H^P \circ G^P, H^L \circ G^L \rangle : I \to K$..

The identity directed bigraph of $I = \langle m, X \rangle$ *is* $\langle id_m, Id_{X^- \uplus X^+} \rangle : I \to I$.

Proposition 2.9 *A directed bigraph* G *in* ′DBig *is epi (respectively mono) iff its two components* G^P *and* G^L *are epi (respectively mono).*

The isomorphisms in ′DBig *are all the combinations* $\iota = \langle \iota^P, \iota^L \rangle$ *of an isomorphism in* ′PLG *and an isomorphism in* ′DLG.

Definition 2.10 *The* tensor product $\otimes$ *in* ′DBig *is defined as follows. Given* $I = \langle m, X \rangle$ *and* $J = \langle n, Y \rangle$*, where* X *and* Y *are pairwise disjoint, then* $\langle m, X \rangle \otimes \langle n, Y \rangle \triangleq \langle m + n, (X^- \uplus Y^-, X^+ \uplus Y^+) \rangle$.

The tensor product of $G_i : I_i \to J_i$ *is defined as* $G_0 \otimes G_1 \triangleq \langle G_0^P \otimes G_1^P, G_0^L \otimes G_1^L \rangle : I_0 \otimes I_1 \to J_0 \otimes J_1$*, when the tensor products of the interfaces are defined and the sets of nodes and edges are pairwise disjoint.*

Remarkably, directed link graphs (and bigraphs) have relative pushouts (RPOs) and pullbacks (RPBs), which can be obtained by a general construction, subsuming both Milner's and Sassone-Sobociński's variants. We refer the reader to [4,3].

Actually, in many situations we do not want to distinguish bigraphs differing only on the identity of nodes and edges. To this end, we introduce the category DBig of *abstract directed bigraphs*. The category DBig is constructed from ′DBig forgetting the identity of nodes and edges and any idle edge. More precisely, abstract bigraphs are concrete bigraphs taken up-to an equivalence $\eqslantless$ (see [6] for details).

Definition 2.11 (abstract directed bigraphs) *Two concrete directed bigraphs* G *and* H *are* lean-support equivalent*, written* $G \eqslantless H$*, if they are support equivalent after removing any idle edges.*

The category DBig *of abstract directed bigraphs has the same objects as* ′DBig*, and its arrows are lean-support equivalence classes of directed bigraphs. We denote by* $\mathcal{A} :$ ′DBig $\to$ DBig *the associated quotient functor.*

We remark that DBig is a category (and not only a precategory); moreover, $\mathcal{A}$ enjoys several important properties which we omit here due to lack of space; see [6].

3 Algebraic structure of ′DBig

We begin this section introducing some useful notations.

Remark 3.1 *An interface* $\langle 0, (X^-, X^+) \rangle$ *is abbreviated as* (X^-, X^+)*; a singleton set* $\{x\}$ *as* x*; and* $\langle m, (\emptyset, \emptyset) \rangle$ *as* m*. The interfaces* $(\emptyset, \emptyset)$ *and* 0 *denote the same*

interface, the origin ϵ. Hence the identity id_ϵ can be expressed as ϵ, $(\emptyset,\emptyset)$ or 0.

A bigraph $A : (\emptyset, X^+) \to (\emptyset, Y^+)$ is defined by a (not necessarily surjective) function $\sigma : X^+ \to Y^+$, called substitution*, if it has no nodes and no edges and the link map is σ; analogously a bigraph $A : (X^-, \emptyset) \to (Y^-, \emptyset)$ is defined by a (not necessarily surjective) function $\delta : Y^- \to X^-$, called* fusion*, if it has no nodes and no edges and the link map is δ. With abuse of notation, we write σ and δ to mean their corresponding bigraphs.*

Let $\vec{x}, \vec{y}$ be two vectors of the same length; we write $(y_0/x_0, y_1/x_1, \dots)$ or $\triangle^{\vec{y}}_{\vec{x}}$, where all the x_i are distinct, for the surjective map $x_i \mapsto y_i$; similarly, we write $(y_0/x_0, y_1/x_1, \dots)$ or $\nabla^{\vec{y}}_{\vec{x}}$, where all y_i are distinct, for the surjective map $y_i \mapsto x_i$.

We denote by $\triangle^X : (\emptyset,\emptyset) \to (\emptyset, X)$ the bigraph defined by the empty substitution $\sigma : \emptyset \to X$, in the same way we denote $\nabla_X : (X, \emptyset) \to (\emptyset,\emptyset)$ for the bigraph defined by the empty fusion $\delta : \emptyset \to X$.

Note that each substitution σ can be expressed in a unique way as $\sigma = \tau \otimes \triangle^X$, where τ is a surjective substitution; while each fusion δ can be expressed in a unique way as $\delta = \zeta \otimes \nabla_X$, where ζ is a surjective fusion. We denote the renamings by α, i.e. the bijective substitution or bijective fusion.

Finally, we introduce the closure bigraphs*. The* closure $\blacktriangledown^x_y : (\emptyset, y) \to (x, \emptyset)$ *has no nodes, a unique edge e and the link map is $link(x) = e = link(y)$. Two other types of closures are obtained by composing the closure $\blacktriangledown^x_y$ and $\triangle^x$ or ∇_y respectively:*

- *the* up-closure $\blacktriangle^y : (\emptyset, y) \to (\emptyset,\emptyset)$ *has no nodes, one edge e and $link(y) = e$;*
- *the* down-closure $\blacktriangledown_x : (\emptyset,\emptyset) \to (x, \emptyset)$ *has no nodes, one edge e and $link(x) = e$.*

Definition 3.2 (wirings) *A* wiring *is a bigraph whose interfaces have zero width (and hence has no nodes). The wirings ω are generated by the composition or tensor product of three base elements: the substitutions $\sigma : (\emptyset, X^+) \to (\emptyset, Y^+)$; the fusions $\delta : (Y^-, \emptyset) \to (X^-, \emptyset)$; and the closures $\blacktriangledown^x_y : (\emptyset, y) \to (x, \emptyset)$.*

Definition 3.3 (prime bigraph) *An interface is* prime *if it has width equal to 1. Often we abbreviate a prime interface $\langle 1, (X^-, X^+)\rangle$ with $\langle (X^-, X^+)\rangle$, in particular $\langle(\emptyset,\emptyset)\rangle = 1$. A prime bigraph $P : \langle m, (Y^-, \emptyset)\rangle \to \langle(\emptyset, X^+)\rangle$ has no upward inner names and no downward outer names, and has a prime outer interface.*

An important prime bigraph is $merge_m : m \to 1$, it has no nodes and it maps m sites to an unique root. A bigraph $G : n \to \langle m, (X^-, X^+)\rangle$ without inner names, it can be simply converted in a prime bigraph as follows: $(merge_m \otimes id_{(X^-,X^+)}) \circ G$.

Definition 3.4 (discrete bigraph) *A bigraph D is* discrete *if it has no edges and the link map is a bijection. That means all points are open, no pair of points is a peer and no link is idle.*

The discreteness is well-behaved, and preserved by composition and tensor products. It is easy to see that discrete bigraphs form a monoidal sub-precategory of $'$DBig.

Definition 3.5 (ion, atom and molecule) *For any non atomic control K with arity k and a pair of sequence $\vec{x}^-$ and $\vec{x}^+$ of distinct names, whose overall length is k, we define the discrete* ion *$K(v)^{\vec{x}^+}_{\vec{x}^-} : \langle(\vec{x}^-, \emptyset)\rangle \to \langle(\emptyset, \vec{x}^+)\rangle$ as the bigraph with a unique K-node v, whose ports are separately linked to $\vec{x}^-$ or to $\vec{x}^+$. We omit v when it can be understood.*

For a prime discrete bigraph P with outer names (Y^-, Y^+) the composite $(K^{\vec{x}^+}_{\vec{x}^-} \otimes id_{(Y^-,Y^+)}) \circ P$ is a discrete molecule*. If K is atomic, we define the discrete* atom *$K^{\vec{x}^+}_{\vec{x}^-} : (\vec{x}^-, \emptyset) \to \langle (\emptyset, \vec{x}^+) \rangle$; it resembles an ion, but has no site.*

An arbitrary (non-discrete) ion, molecule or atom is formed by the composition of $\omega \otimes id_1$ with a discrete one. Often we omit $\cdots \otimes id_I$ in the compositions, when there is no ambiguity; for example we write $merge_m \circ G$ to mean $(merge_m \otimes id_{(X^-,X^+)}) \circ G$ and $K^{\vec{x}^+}_{\vec{x}^-} \circ P$ to mean $(K^{\vec{x}^+}_{\vec{x}^-} \otimes id_{(Y^-,Y^+)}) \circ P$. Note that every atom and every molecule are prime, furthermore an atom is also ground, but a molecule is not necessarily ground, since it may have sites.

Now, we define some variants of the tensor product, whose can allow the sharing of names. Process calculi often have a parallel product $P \mid Q$, that allows the processes P and Q to share names. In directed bigraphs, this sharing can involve inner downward names and/or outer upword names, as described by the following definitions.

Definition 3.6 (sharing products) *The* outer sharing product*,* inner sharing product *and* sharing product *of two link graphs $A_i : X_i \to Y_i$ $(i = 0, 1)$ are defined as follows:*

$$(X^-, X^+) \curlywedge (Y^-, Y^+) \triangleq (X^- \uplus Y^-, X^+ \cup Y^+)$$
$$(X^-, X^+) \curlyvee (Y^-, Y^+) \triangleq (X^- \cup Y^-, X^+ \uplus Y^+)$$

$$A_0 \curlywedge A_1 \triangleq (V_0 \uplus V_1, E_0 \uplus E_1, ctrl_0 \uplus ctrl_1, link_0 \uplus link_1) : X_0 \otimes X_1 \to Y_0 \curlywedge Y_1$$
$$A_0 \curlyvee A_1 \triangleq (V_0 \uplus V_1, E_0 \uplus E_1, ctrl_0 \uplus ctrl_1, link_0 \uplus link_1) : X_0 \curlyvee X_1 \to Y_0 \otimes Y_1$$
$$A_0 \parallel A_1 \triangleq (V_0 \uplus V_1, E_0 \uplus E_1, ctrl_0 \uplus ctrl_1, link_0 \uplus link_1) : X_0 \curlyvee X_1 \to Y_0 \curlywedge Y_1$$

defined when their interfaces are defined and A_i have disjoint node and edge sets.

The outer sharing product*,* inner sharing product *and* sharing product *of two bigraphs $G_i : I_i \to J_i$ are defined by extending the corresponding products on their link graphs with the tensor product on widths and place graphs:*

$$\langle m, X \rangle \curlywedge \langle n, Y \rangle \triangleq \langle n + m, X \curlywedge Y \rangle \qquad \langle m, X \rangle \curlyvee \langle n, Y \rangle \triangleq \langle n + m, X \curlyvee Y \rangle$$
$$G_0 \curlywedge G_1 \triangleq \langle G_0^P \otimes G_1^P, G_0^L \curlywedge G_1^L \rangle : I_0 \otimes I_1 \to J_0 \curlywedge J_1$$
$$G_0 \curlyvee G_1 \triangleq \langle G_0^P \otimes G_1^P, G_0^L \curlyvee G_1^L \rangle : I_0 \curlyvee I_1 \to J_0 \otimes J_1$$
$$G_0 \parallel G_1 \triangleq \langle G_0^P \otimes G_1^P, G_0^L \parallel G_1^L \rangle : I_0 \curlyvee I_1 \to J_0 \curlywedge J_1.$$

defined when their interfaces are defined and G_i have disjoint node and edge sets.

It is simple to verify that $\curlywedge$, $\curlyvee$ and $\parallel$ are associative, with unit ϵ.

Another way of constructing a sharing product of two bigraphs G_0, G_1 is to disjoin the names of G_0 and G_1, then take the tensor product of the two bigraphs and finally merge the name again:

Proposition 3.7 *Let G_0 and G_1 be bigraphs with disjoint node and edge sets. Then*

$$G_0 \curlywedge G_1 = \sigma(G_0 \otimes \tau G_1 \zeta) \qquad G_0 \curlyvee G_1 = (G_0 \otimes \tau G_1 \zeta)\delta \qquad G_0 \parallel G_1 = \sigma(G_0 \otimes \tau G_1 \zeta)\delta$$

where the substitution σ and τ are defined in the following way: if z_i $(i \in n)$ are the upward outer names shared by G_0 and G_1, and w_i are fresh names in bijection with the z_i, then $\tau(z_i) = w_i$ and $\sigma(w_i) = \sigma(z_i) = z_i$ $(i \in n)$. The substitution δ and ζ are defined in a very similar way, but acting on the downward inner names.

Definition 3.8 (prime products) *The* prime outer sharing product *and* prime sharing product *of two bigraphs $G_i : I_i \to J_i$ are defined as follows:*

$$\langle m, (X^-, X^+)\rangle \barwedge \langle n, (Y^-, Y^+)\rangle \triangleq \langle (X^- \uplus Y^-, X^+ \cup Y^+)\rangle$$

$$G_0 \barwedge G_1 \triangleq merge_{(width(J_0)+width(J_1))} \circ (G_0 \parallel G_1) : I_0 \otimes I_1 \to J_0 \barwedge J_1$$
$$G_0 \mid G_1 \triangleq merge_{(width(J_0)+width(J_1))} \circ (G_0 \parallel G_1) : I_0 \bigvee I_1 \to J_0 \barwedge J_1.$$

defined when their interfaces are defined and G_i have disjoint node and edge sets.

It is easy to show that $\barwedge$ and $\mid$ are associative, with unit 1 when applied to prime bigraphs. Note that for a wiring ω and a prime bigraph P, we have $\omega \barwedge P = \omega \parallel P$ and $\omega \mid P = \omega \parallel P$, because in this case these products have the same meaning.

Now, we can describe *discrete bigraphs,* which are the complement of wirings:

Theorem 3.9 (discrete normal form) (i) *Every bigraph G can be expressed uniquely (up to iso) as: $G = (\omega \otimes id_n) \circ D \circ (\omega' \otimes id_m)$, where D is a discrete bigraph and ω, ω' are two wirings satisfying the following conditions:*

- *in ω, if two outer downward names are peer, then their target is an edge;*
- *in ω' there are no edges, and no two inner upward names are peer (i.e., on inner upward names ω' is a renaming, but outer downward names can be peer).*

(ii) *Every discrete bigraph $D : \langle m, (X^-, X^+)\rangle \to \langle n, (Y^-, Y^+)\rangle$ may be factored uniquely (up to iso) on the domain of each factor D_i, as:*

$$D = \alpha \otimes ((D_0 \otimes \cdots \otimes D_{n-1}) \circ (\pi \otimes id_{dom(\vec{D})}))$$

with α a renaming, each D_i prime and discrete, and π a permutation.

Proof. For the first part, consider a bigraph $G : \langle n, (X^-, X^+)\rangle \to \langle m, (Y^-, Y^+)\rangle$. We divide G in three parts: a discrete $D : \langle n, (Z^-, Z^+)\rangle \to \langle m, (W^-, W^+)\rangle$ and two wirings $\omega : (W^-, W^+) \to (Y^-, Y^+)$ and $\omega' : (X^-, X^+) \to (Z^-, Z^+)$ satisfying the previous conditions. We proceed by cases:

$p \in P$, $link_G(p) = e \in E$: we add a fresh name $w_e \in W^+$ and define $link_D(p) = w_e$ and $link_\omega(w_e) = e$;

$p \in P$, $link_G(p) = y \in Y^+$: we add a fresh name $w_y \in W^+$ and define $link_D(p) = w_y$ and $link_\omega(w_y) = y$;

$p \in P$, $link_G(p) = x \in X^-$: this case is analogous to the previous one;

$y \in Y^-$, $link_G(y) = e \in E$: we define $link_\omega(y) = e$;

$x \in X^+$, $link_G(y) = e \in E$: we add a fresh name $z_e \in Z^+$, a fresh name $w_e \in W^+$ and define $link_{\omega'}(x) = z_e$, $link_D(z_e) = w_e$, $link_\omega(w_e) = e$;

$y \in Y^-$, $link_G(y) = x \in X^-$: we add a fresh name $w_x \in W^-$, a fresh name $z_x \in Z^-$ and define $link_\omega(y) = w_x$, $link_D(w_x) = z_x$ and $link_{\omega'}(z_x) = x$;

$x \in X^+$, $link_G(x) = y \in Y^+$: this case is analogous to the previous one; it is sufficient to invert the direction of links and swap the rule of ω with ω'.

Note that there are no idle names in Z^-, Z^+, W^- and W^+, so those sets are formed only by the fresh names defined in this proof. Furthermore, the three conditions above holds because we create a fresh name every time we need one.

The proof of the second part is easy. Since the outer interface of D has width n, we can decompose D in n discrete and prime parts, obtaining $D_0 \otimes \cdots \otimes D_{n-1}$. The renaming α describe the connections between the inner interface and the outer one. Finally the permutation π gives the right sequence of the sites, so we can take the tensor product of D_i $(i = 0, \ldots .n-1)$ in any order. □

We call this unique factorization *discrete normal form* (DNF). The DNF applies to abstract bigraphs as well, and indeed it will play an important part in the complete axiomatization of DBig, as we will discuss in the next section.

Note that a renaming is discrete but not prime (since it has zero width); this is why the factorization in Theorem 3.9(ii) has such a factor. This unique factorization depends on the fact that the prime bigraphs have no upward inner names and downward outer names. In the special case that D is ground, the factorization in Theorem 3.9(ii) is simply $D = d_0 \otimes \cdots \otimes d_{n-1}$, that is a product of discrete and prime ground bigraphs.

4 Algebraic structure of DBig

In this section we describe a sound and complete axiomatization for directed abstract bigraphs. Furthermore we give a normal form for discrete bigraphs, that is useful to prove the completeness of the axiomatization.

First we introduce the *algebraic signature*, that is a set of elementary bigraphs able to define any other bigraph (Figure 1).

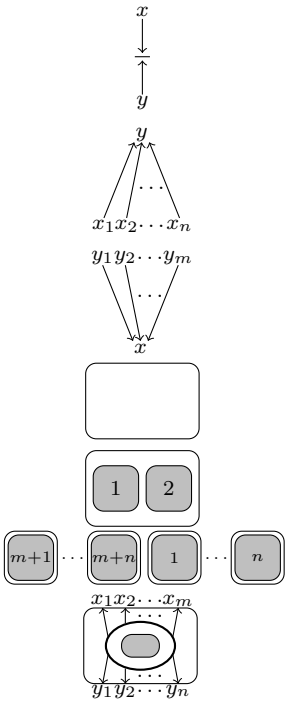
We have to show that all bigraphs can be constructed from these elementary ones by composition and tensor product. Before giving a formal result, we provide an intutive explanation of the meaning of these elementary bigraphs.

- The first three bigraphs build up all wirings, i.e. all the link graphs having no nodes. Indeed, all substitutions (fusions, resp.) can be obtained as tensor products of elementary substitutions $\triangle_X^y$ (fusions ∇_x^Y, resp.); the tensor products of singleton substitutions $\triangle_x^y$ and/or singleton fusions ∇_y^x give all renamings. The composition and the tensor product of substitutions, fusions and closures give all wirings.
- The next three bigraphs define all placings, i.e. all place graphs having no nodes; for example $merge_m : m \to 1$, merging m sites in a unique root, are defined as:

$$merge_0 \triangleq 1 \qquad merge_{m+1} \triangleq merge \circ (id_1 \otimes merge_m).$$

  Notice that $merge_1 = id$ and $merge_2 = merge$, and that all permutations $\pi : m \to m$ are constructed by composition and tensor from the $\gamma_{m,n}$.
- Finally, for expressing any direct bigraph we need to add only the discrete ions $K_{\vec{x}^-}^{\vec{x}^+}$. In particular, we can express any discrete atoms as $K_{\vec{x}^-}^{\vec{x}^+} \circ 1$.

The following proposition shows that every bigraph can be expressed in a normal form, called (again) *discrete normal form* (DNF). We will use D, Q and N to denote primes, discrete prime bigraphs, and the discrete molecules respectively.



$\blacktriangledown_y^x:(\emptyset, y) \to (x, \emptyset)$ closure

$\triangle_X^y:(\emptyset, X) \to (\emptyset, y)$ substitution

$\nabla_x^Y:(x, \emptyset) \to (Y, \emptyset)$ fusion

$1:\epsilon \to 1$ a barren root

$merge:2 \to 1$ mapping 2 sites in 1 root

$\gamma_{m,n}:m+n \to n+m$ swapping m with n

$K_{\vec{y}}^{\vec{x}}:\langle(\vec{y}, \emptyset)\rangle \to \langle(\emptyset, \vec{x})\rangle$ a discrete ion

Fig. 1. Elementary Bigraphs

Proposition 4.1 (discrete normal form) *In* DBig *every bigraph G, discrete D, discrete and prime Q and discrete molecule N can be described by an expression of the respective following form:*

$$G = (\omega \otimes id_n) \circ D \circ (\omega' \otimes id_m) \quad (1)$$

where ω, ω' satisfy the conditions given in Theorem 3.9(i);

$$D = \alpha \otimes ((Q_0 \otimes \cdots \otimes Q_{n-1}) \circ (\pi \otimes id_{dom(\vec{Q})})) \quad (2)$$

$$Q = (merge_{n+p} \otimes id_{\emptyset, Y^+}) \circ (id_n \otimes N_0 \otimes \cdots \otimes N_{p-1}) \circ (\pi \otimes id_{(Y^-, \emptyset)}) \quad (3)$$

$$N = (K_{\vec{x}^-}^{\vec{x}^+} \otimes id_{\emptyset, Y^+}) \circ Q. \quad (4)$$

Furthermore, the expression is unique up to isomorphisms on the parts.

Proof. The proof is quite similar to the proof of Theorem 3.9. □

We can use these equations for normalizing any bigraph G as follows; first, we apply equations (1), (2) to G once, obtaining an expression containing discrete and prime bigraphs $Q_0, \ldots, Q_{n-1}$. These are decomposed further using equations (3), (4) repeatedly: each Q_i is decomposed into an expression containing molecules $N_{i,0}, \ldots, N_{i,p_i-1}$, each of which is decomposed in turn into an ion containing another discrete and prime bigraph $Q'_{i,j}$. The last two steps are repeated recursively until the ions are atoms. Note that the unit 1 is a special case of Q when $n = p = 0$.

In Figure 2 we give a set of axioms which we prove to be sound and complete.

Each of these equations holds only when both sides are defined; in particular, recall that the tensor product of two bigraphs is defined only if the name sets are disjoint. It is important to notice also that for ions only the renaming axiom is

Categorical Axioms

$$A \circ id = A = id \circ A \qquad A \circ (B \circ C) = (A \circ B) \circ C$$
$$A \otimes id_\epsilon = A = id_\epsilon \otimes A \qquad A \otimes (B \otimes C) = (A \otimes B) \otimes C$$
$$\gamma_{I,\epsilon} = id_I \qquad \gamma_{J,I} \circ \gamma_{I,J} = id_{I \otimes J}$$
$$(A_1 \otimes B_1) \circ (A_0 \otimes B_0) = (A_1 \circ A_0) \otimes (B_1 \circ B_0)$$
$$\gamma_{I,K} \circ (A \otimes B) = (B \otimes A) \circ \gamma_{H,J} \qquad (\text{where } A : H \to I, B : J \to K)$$
$$\gamma_{I \otimes J,K} = (\gamma_{I,K} \otimes id_J) \circ (id_I \otimes \gamma_{J,K})$$

Link Axioms

$$\blacktriangledown^x_y \circ \triangle^y_z = \blacktriangledown^x_z \qquad \nabla^z_x \circ \blacktriangledown^x_y = \blacktriangledown^z_y \qquad \nabla_x \circ \blacktriangledown^x_y \circ \triangle^y = id_\epsilon$$
$$\triangle^z_{(Y \uplus y)} \circ (id_{(\emptyset,Y)} \otimes \triangle^y_X) = \triangle^z_{(Y \uplus X)} \qquad (id_{(Y,\emptyset)} \otimes \nabla^X_y) \circ \nabla^{(Y \uplus y)}_z = \nabla^{(X \uplus Y)}_z$$

Place Axioms

$$merge \circ (1 \otimes id_1) = id_1 \qquad merge \circ \gamma_{1,1} = merge$$
$$merge \circ (merge \otimes id_1) = merge \circ (id_1 \otimes merge)$$

Node Axioms

$$(id_1 \otimes \alpha) \circ K^{\vec{x}^+}_{\vec{x}^-} = K^{\alpha(\vec{x}^+)}_{\vec{x}^-} \qquad K^{\vec{x}^+}_{\vec{x}^-} \circ (id_1 \otimes \alpha) = K^{\vec{x}^+}_{\alpha(\vec{x}^-)}$$

Fig. 2. Axiomatization for the abstract directed bigraphs.

needed (because the names are treated positionally).

Theorem 4.2 (Completeness of the axiomatization) *Let us consider two expressions E_0, E_1 constructed from the elementary bigraphs by composition and tensor product. Then, E_0 and E_1 denote the same bigraph in* DBig *if and only if the equation $E_0 = E_1$ can be proved by the axioms in Figure 2.*

Proof. The proof is similar to that of [6, Theorem 10.2]. The "if" direction is simple to prove, since it requires to check that each axiom is valid. The "only if" direction is in two steps. First, we prove by induction on the structure of expressions, that the equality between an expression and its DNF is derivable from the axioms. Next, since DNFs are taken up to iso, we have to show that the equality between isomorphic DNFs is provable from the axioms. This is proved by showing that the axioms can prove the isomorphisms of the components of a DNF, which are ions, discrete and prime bigraphs, and discrete bigraphs. □

5 Application: the λ-calculus

In this section we describe an encoding of both the call-by-name and the call-by-value λ-calculus. Recall that the set Λ of λ-terms are the terms up-to α-equivalence generated by the following grammar:

$$M, N ::= x \mid \lambda x.M \mid MN.$$

A *value* is either a λ-abstraction or a variable; values are ranged over by V.

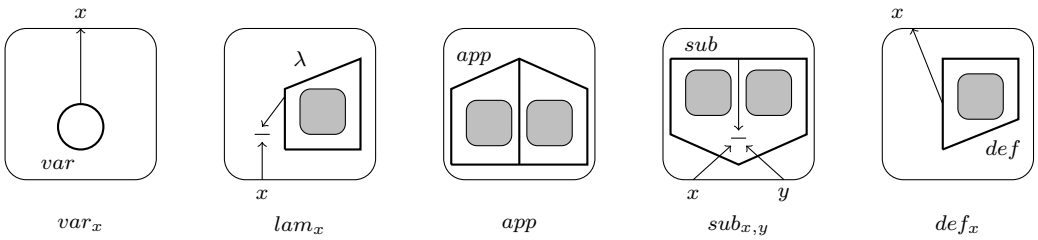


Fig. 3. The signature for the λ-calculus.

The call-by-name reduction semantics is defined by the following rules

$$(\lambda x.M)N \to M[N/x] \quad (\beta) \qquad \frac{M \to M'}{MN \to M'N} \qquad \frac{N \to N'}{MN \to MN'}$$

while the call-by-value reduction semantics is defined by the following rules

$$(\lambda x.M)V \to M[V/x] \quad (\beta_v) \qquad \frac{M \to M'}{MN \to M'N} \qquad \frac{N \to N'}{MN \to MN'}$$

In Figure 3 we give a signature for representing the λ-calculus "with single substitutions", that is where a substitution is performed once for each variable occurrence. This signature resembles Milner's encoding using binding bigraphs, but in directed bigraphs we do not need to introduce further binding structures.

We can define a translator operator $[\![\cdot]\!] : \Lambda \to \text{DBig}$ as follows:

$$[\![x]\!] = var_x \qquad [\![\lambda x.M]\!] = lam_x \circ ([\![M]\!] \jmath \triangle^x) \qquad [\![MN]\!] = app \circ ([\![M]\!] \jmath [\![N]\!])$$

Intuitively, a λ-term M is represented by a ground bigraph $[\![M]\!] : \epsilon \to \langle (\emptyset, X^+) \rangle$ whose place hierarchy reflects the syntactic tree of M and the outer upwards names X^+ are the free variables of M. Each λ-expression is represented by a control and a local resource which is bound to a upward name in the inner interface.

Proposition 5.1 *Let M, N be two λ-terms; then, $M \equiv_\alpha N$ iff $[\![M]\!] = [\![N]\!]$.*

Let us now see how we can represent the two semantics of the λ-calculus. For the call-by-name semantics, we define the controls *lam* and *def* as passive, *sub* and *app* as active. The reaction rules are given in Figure 4.

For the call-by-value λ-calculus, we have to replace the App_{cbn} rule with two rules $App_{cbv\text{-}var}$ and $App_{cbv\text{-}lam}$ (Figure 5) corresponding to the two cases of values where the application can be performed.

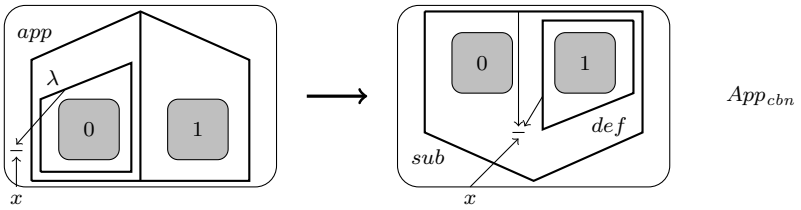
For both variants, we can prove the following result:

Proposition 5.2 *Let M, M' be two λ-terms.*

(i) *If $M \to M'$ then $[\![M]\!] \to^* [\![M']\!]$;*
(ii) *If $[\![M]\!] \to^* [\![M']\!]$ then $M \to^* M'$.*

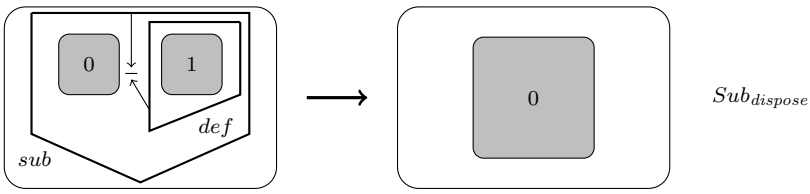
Proof. By induction on the lenght of traces.

(i) The application of β (or β_v) is encoded by applying App_{cbn} (or one of $App_{cbv\text{-}var}$ and $App_{cbv\text{-}lam}$) on the correct sub-bigraph, i.e. the one which encodes the right



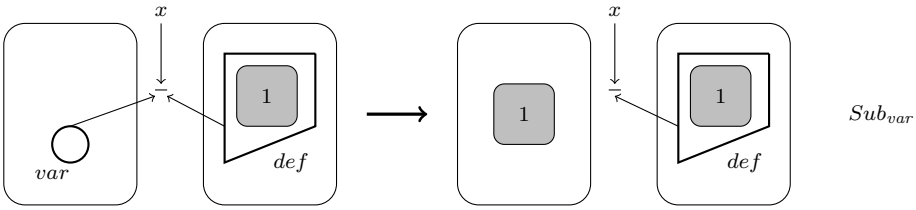
App_{cbn}

$$app \circ (lam_x \between id_1) \longrightarrow sub_{x,y} \circ (id_1 \between def_y)$$



$Sub_{dispose}$

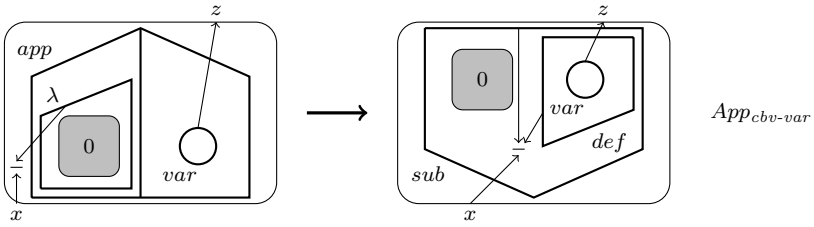
$$sub_{x,y} \circ (id_1 \between \triangle^x \between def_y) \longrightarrow id_1$$



Sub_{var}

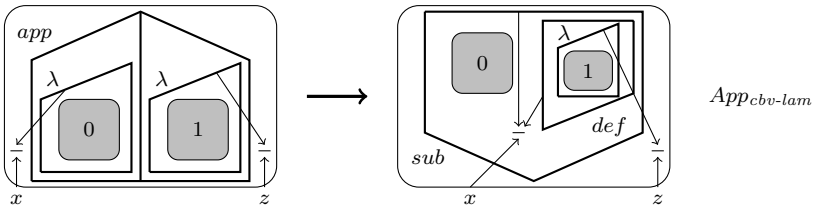
$$⧗_y^x \circ (var_y \between def_y) \longrightarrow ⧗_y^x \circ (id_1 \between def_y)$$

Fig. 4. Reactions for the call-by-name λ-calculus.



$App_{cbv\text{-}var}$

$$app \circ (lam_x \between var_z) \longrightarrow sub_{x,y} \circ (id_1 \between (def_y \circ var_z))$$



$App_{cbv\text{-}lam}$

$$app \circ (lam_x \between lam_z) \longrightarrow sub_{x,y} \circ (id_1 \between (def_y \circ lam_z))$$

Fig. 5. Reactions for the call-by-value λ-calculus.

side of the rule. Next we use Sub_{var} for every occurrence of x in M, finally we apply $Sub_{dispose}$ to eliminate the unnecessary controls sub and def.

(ii) First of all note that, by definition of $[\![\cdot]\!]$, $[\![M']\!]$ has no sub or def controls. If $[\![M]\!] \rightarrow^* [\![M']\!]$, in the trace there are one or more application of App_{cbn} (or $App_{cbv\text{-}var}$ and $App_{cbv\text{-}lam}$), so we use the β (or β_v) rule on the corresponding λ-subterm. We can ignore the Sub_{var} and $Sub_{dispose}$ rules because the substitutions in λ-calculus are performed instantaneously. □

6 Conclusions

In this paper we have given a sound and complete axiomatization of the precategory of directed bigraphs, a bigraphical model which subsumes and generalizes both Milner's and Sassone-Sobociński variants. We have used this axiomatization for encoding the λ-calculus, both in call-by-name and call-by-value variants. It is interesting to notice that no further extensions (such as binding signatures) are needed.

We plan to use this axiomatization for representing other calculi, in particular calculi with resources, locations, etc., which can be represented by edges. Interesting candidates could be the Fusion calculus [9] and the ν-calculus [10]; it will be interesting to see which kind of wide transition systems we would obtain.

The new discrete normal form, and associated composition operations, presented in this paper can be useful in view of possible applications and extensions of logics and matching tools for bigraphs, in the line of [1,2]. Another future work is to give a 2-categorical definitions of directed link graphs.

References

[1] Birkedal, L., T. C. Damgaard, A. Glenstrup and R. Milner, *Matching of bigraphs*, in: *Proceedings of Graph Transformation for Verification and Concurrency 2006*, 2007.

[2] Conforti, G., D. Macedonio and V. Sassone, *Spatial logics for bigraphs*, in: *Proc. ICALP*, Lecture Notes in Computer Science **3580** (2005), pp. 766–778.

[3] Grohmann, D. and M. Miculan, *Directed bigraphs: theory and applications*, Technical Report UDMI/12/2006/RR, University of Udine (2006). http://www.dimi.uniud.it/miculan/Papers/.

[4] Grohmann, D. and M. Miculan, *Directed bigraphs*, in: *Proc. XXIII MFPS*, ENTCS **173** (2007), pp. 121–137.

[5] Jensen, O. H. and R. Milner, *Bigraphs and transitions*, in: *Proc. POPL*, 2003, pp. 38–49.

[6] Jensen, O. H. and R. Milner, *Bigraphs and mobile processes (revised)*, Technical report UCAM-CL-TR-580, Computer Laboratory, University of Cambridge (2004).

[7] Milner, R., *Bigraphical reactive systems*, in: K. G. Larsen and M. Nielsen, editors, *Proc. 12th CONCUR*, Lecture Notes in Computer Science **2154** (2001), pp. 16–35.

[8] Milner, R., *Pure bigraphs: Structure and dynamics*, Inf. Comput. **204** (2006), pp. 60–122.

[9] Parrow, J. and B. Victor, *The fusion calculus: Expressiveness and symmetry in mobile processes*, in: *Proc. LICS'98*, IEEE (1998), pp. 176–185. http://www.docs.uu.se/~victor/tr/fusion.shtml

[10] Pitts, A. and I. Stark, *Observable properties of higher order functions that dynamically create local names, or what's new?*, in: *Proc. MFCS*, Lecture Notes in Computer Science **711** (1993), pp. 122–141.

[11] Sassone, V. and P. Sobociński, *Reactive systems over cospans*, in: *Proc. LICS* (2005), pp. 311–320.

Interaction Nets with nested pattern matching

Shinya Sato[a,1] and Abubakar Hassan[b,2]

[a] *Faculty of Econoinformatics*
Himeji Dokkyo University
7-2-1 Kamiohno, Himeji-shi, Hyogo 670-8524, JAPAN

[b] *Department of Computer Science*
King's College London
Strand, London WC2R 2LS, UK

Abstract

Reduction rules in Interaction Nets are constrained to pattern match exactly one argument at a time. Consequently, a programmer has to introduce auxiliary rules to perform more sophisticated matches. We propose an extension of Interaction Nets which facilitate nested pattern matching on interaction rules. We then define a practical compilation scheme from extended rules to pure interaction rules. We achieve a system that provides convenient ways to express Interaction Net programs without defining auxiliary rules.

Keywords: Interaction nets, pattern matching, programming language design.

1 Introduction

Interaction Nets [5] can be considered as a graphical–or visual–programming language. Programs are expressed as graphs, and computation is graph reduction. From another perspective, Interaction Nets are also a low level implementation language: we can define systems of Interaction Nets that are instructions for the target of compilation schemes of other programming languages. For instance, Interaction Nets have been used for the implementation of optimal reduction [4,6] and other efficient implementations of the λ-calculus [8]. In addition, there has been various implementations of Interaction Nets [7,9]. Despite that we can already program in Interaction Nets (they are Turing complete), they still remain far from being used as a programming language. Drawing an analogy with functional programming, we only have the pure λ-calculus that is without syntactic sugar, constants, data-structures, etc.

[1] shinya@himeji-du.ac.jp
[2] abubakar.hassan@kcl.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

In this paper we take step towards developing a richer language based on Interaction Nets. Interaction Nets have a very primitive notion of pattern matching since only two agents can interact at a time. Consequently, many auxiliary agents and rules are needed to implement more sophisticated matches. These auxiliaries are implementation details and should be generated automatically other than by the programmer. To achieve this, we extend Interaction Nets to allow rules with nested patterns to be defined. We then give a compilation scheme from extended to ordinary interaction rules.

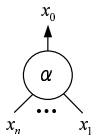
There has been several works that extend Interaction Nets in some way (see Section 6.2). Sinot and Mackie's Macros for Interaction Nets [10] are quite close to what we present in this paper. They allow pattern matching on more than one argument by relaxing the restriction of one principal port per agent. The main difference with our work is that their system does not allow nested pattern matching. Our system facilitates nested/deep pattern matching of agents.

The rest of this paper is organised as follows: In the next section we give a brief introduction to Interaction Nets. In Section 3 we motivate our work through an example. We give the proposed extensions in Section 4, followed by the compilation schemes in Section 5. In Section 6 we discuss some implementation issues. Finally, we conclude the paper in Section 7

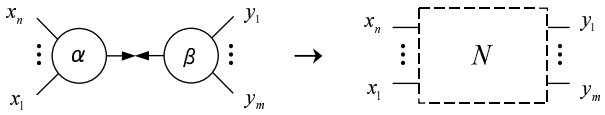
2 Interaction Nets

We review the basic notions of Interaction Nets. See [5] for a more detailed presentation. Interaction Nets are specified by the following data:

- A set Σ of *symbols*. Elements of Σ serve as *agent* (node) labels. Each symbol has an associated arity *ar* that determines the number of its *auxiliary ports*. If $ar(\alpha \in \Sigma) = n$, then α has $n+1$ *ports:* n auxiliary ports and a distinguished one called the *principal port*.

  We use the textual notation $x_0 - \alpha(x_1, ..., x_n)$ to represent an agent α where x_0 is the principal port and $x_1, ..., x_n$ are its auxiliary ports.
- A *net* built on Σ is an undirected graph with agents at the vertices. The edges of the net connect agents together at the ports such that there is only one edge at every port. A port which is not connected is called a *free port*.
- Two agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected via their principal ports form an *active pair* (analogous to a redex). An interaction rule $((\alpha, \beta) \to N) \in \mathcal{R}$ replaces the pair (α, β) by the net N. All the free ports are preserved during reduction, and there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.



We represent this rule textually as $\alpha(x_1, ..., x_n) \bowtie \beta(y_1, ..., y_m) \rightarrow N$. The order of writing the active agents in this textual form is not important. The same rule can be written as $\beta(y_1, ..., y_m) \bowtie \alpha(x_1, ..., x_n) \rightarrow N$. We use the notation $N_1 \Rightarrow N_2$ for the one step reduction and $\Rightarrow^*$ for its transitive and reflexive closure.

Interaction Nets have the following property [5]:

- **Strong Confluence**: Let N be a net. If $N \Rightarrow N_1$ and $N \Rightarrow N_2$ with $N_1 \neq N_2$, then there is a net N_3 such that $N_1 \Rightarrow N_3$ and $N_2 \Rightarrow N_3$.

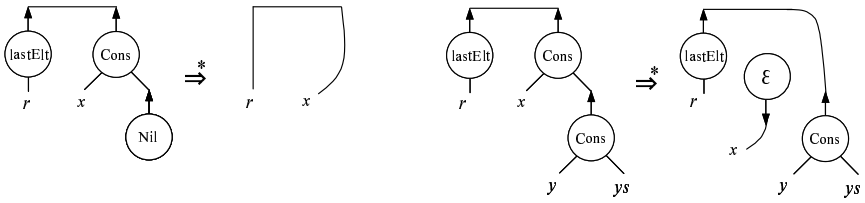
3 Motivations

In this section, we motivate our work by investigating how we can translate a function with pattern matching into Interaction Nets.

Example 3.1 Our example is the following definition of a function that returns the last element of a list:

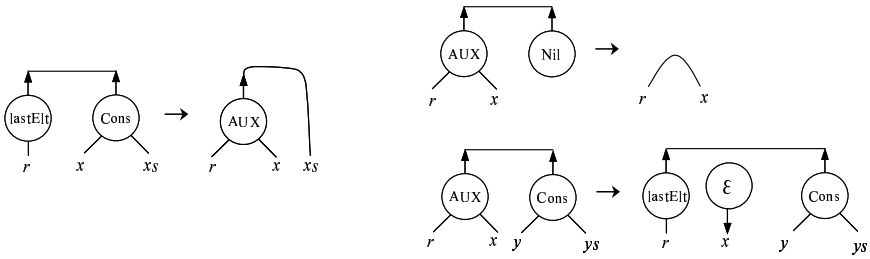
```
fun lastElt [x] = x
  | lastElt (x::xs) = lastElt xs;
```

If we consider a functional programming language as an orthogonal term rewriting system, we can translate programs into Interaction Nets [3]. In this way, if we take both the name of the function and the first argument as agents, we can represent the above function as interaction rules:



However, these rules are not valid in Interaction Nets as the left hand side (LHS) of a rule must be a net with exactly two agents (active pair).

Therefore, to encode this example in interaction nets, we have to introduce auxiliary agents and rules:



This set of rules will compute and return the last element of a list. We argue that the introduction of the auxiliary agents to the system is not satisfactory from

a programmers perspective. Programmers want to write simpler programs rather than more complicated ones. To solve this problem, we extend the definition of rules to facilitate nested pattern matching.

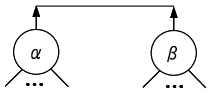
4 Interaction rules for nested patterns (INP)

4.1 An extension of the definition of interaction rules

In this section we present our framework INP that extends ordinary interaction rules (ORN) so that we can perform rewritings between nested agents. The main difference from ORN is that we allow the left hand side of a rule to contain more than two agents. The definition of agents and nets remain the same as for ORN.

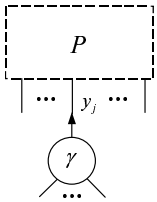
Definition 4.1 A *nested active pair* P is inductively defined as follows:

Base: Every active pair in ORN is a nested active pair



represented textually as: $\langle \alpha(x_1, ..., x_n) \bowtie \beta(y_1, ..., y_m) \rangle$.

Step: A net obtained as a result of connecting the principal port of some agent to a free port in a nested active pair P is also a nested active pair.



We represent this nested active pair textually as $\langle P,\ y_j - \gamma(z_1, ..., z_l) \rangle$.

Definition 4.2 An interaction rule in INP is given by $P \to N$ where P is a nested active pair. All the free ports are preserved during reduction, and there is at most one rule with P in any given system.

Proposition 4.3 *ORN* $\subset$ *INP.*

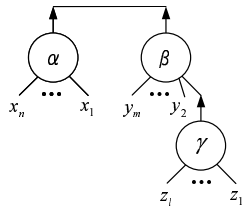
Proof. All rules $P \to N$ where P contains just two agents (active pair) are valid ORN rules. These active pairs fall into the base definition of nested active pairs. □

We aim to extend ORN in a conservative way and retain the property of strong confluence. For this purpose, we introduce a condition that restricts the formation of the set of interaction rules in INP.

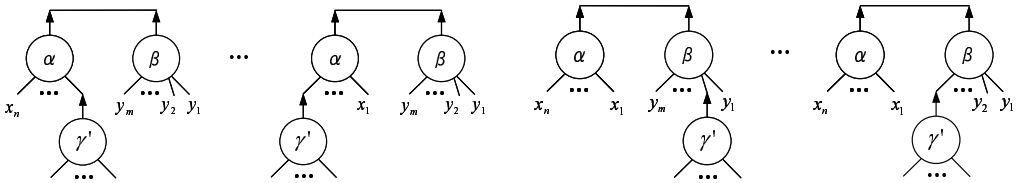
Definition 4.4 A set of nested active pairs $\mathcal{P}$ is *sequential* if and only if, when $\langle P,\ y_j - \gamma(z_1, ..., z_l) \rangle \in \mathcal{P}$, then

- for the nested pair P, $P \in \mathcal{P}$ and,
- for all free ports y in P except the y_j and for all agents α, $\langle P,\ y - \alpha(w_1, ..., w_n) \rangle \notin \mathcal{P}$.

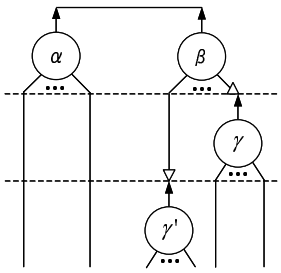
As an example, consider the following nested active pair P in a sequential set $\mathcal{P}$:



represented textually as $\langle \alpha(x_1, ..., x_n) \bowtie \beta(y_1, ..., y_m),\ y_1 - \gamma(z_1, ..., z_l) \rangle$. Then we can not have any other nested active pair (α, β) such that the port y_1 is free. Thus, the following definitions violet the condition of the set $\mathcal{P}$:



For clarity, we draw lines and triangles on auxiliary ports that connect to nested agents. As an example, we represent a nested active pair $\langle P,\ y_m - \gamma'(w_1, ..., w_k) \rangle$ graphically as follows:

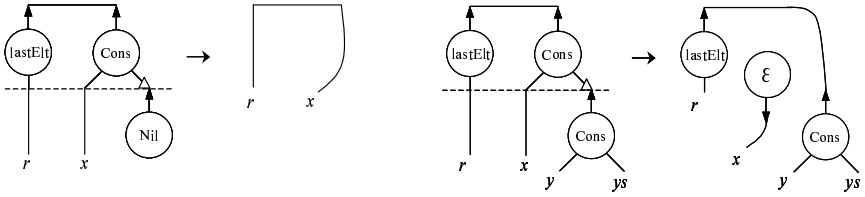


Note that this nested active pair belongs to the set $\mathcal{P}$ because $P \in \mathcal{P}$.

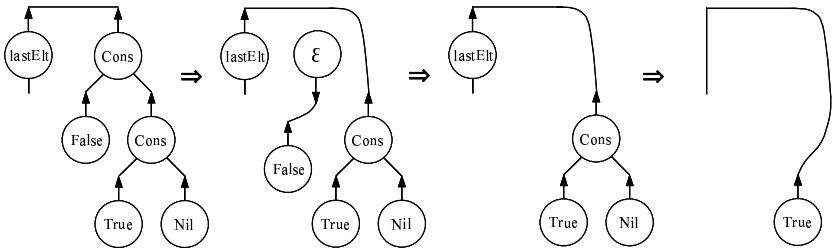
Definition 4.5 A set of rules $\mathcal{R}$ in INP is *well-formed* if and only if,

- there is a sequential set which contains every nested active pair of the LHS in $\mathcal{R}$,
- for every rule $P \to N$ in $\mathcal{R}$, there is no interaction rule $P' \to N'$ in $\mathcal{R}$ such that P' is a subnet of P.

Example 4.6 The rule set in Example 3.1 is well-formed



and the following computation can be performed:

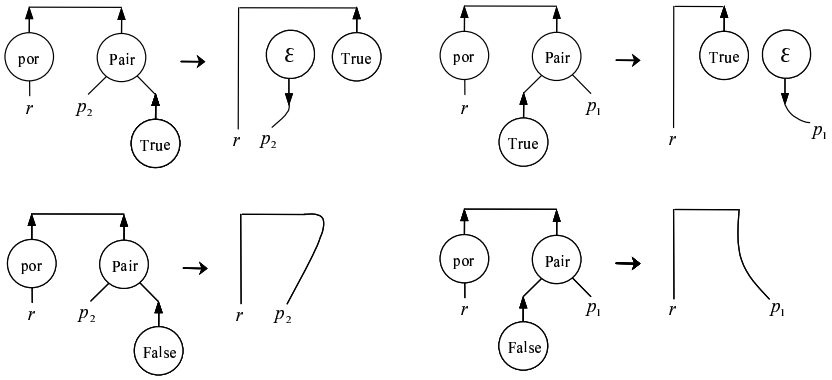


In the above example, the rewriting is strongly confluent because there is no *critical pair*. We loose this property if there are more than two rules that can be applied to the same net.

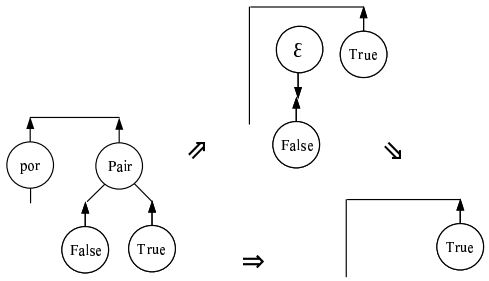
Example 4.7 We can encode the following definition of the parallel-or function `por`:

$$\begin{aligned}
\texttt{por}(\texttt{True}, y) &= \texttt{True} \\
\texttt{por}(y, \texttt{True}) &= \texttt{True} \\
\texttt{por}(\texttt{False}, y) &= y \\
\texttt{por}(y, \texttt{False}) &= y
\end{aligned}$$

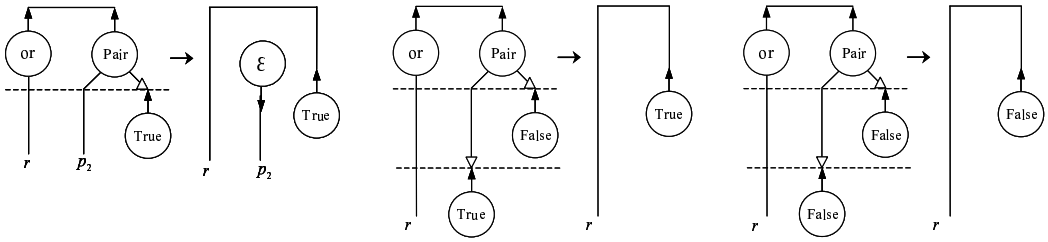
as a set of INP rules:



However, this is not a well-formed set of rules because there is no sequential set which contains both $\langle \texttt{por}(x) \bowtie \texttt{Pair}(y_1, y_2),\ y_1 - \texttt{True} \rangle$ and $\langle \texttt{por}(x) \bowtie \texttt{Pair}(y_1, y_2),\ y_2 - \texttt{True} \rangle$. Therefore, the reduction is not strongly confluent (but still confluent in this example).



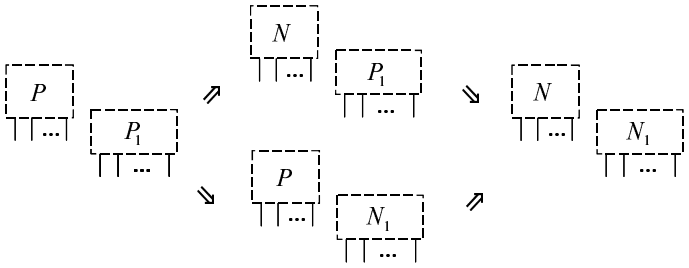
On the other hand, the following rule set of the *or* function is well-defined:



Proposition 4.8 (Strong Confluence) *If a given rule set $\mathcal{R}$ in INP is well-formed, then the reduction in $\mathcal{R}$ is strongly confluent.*

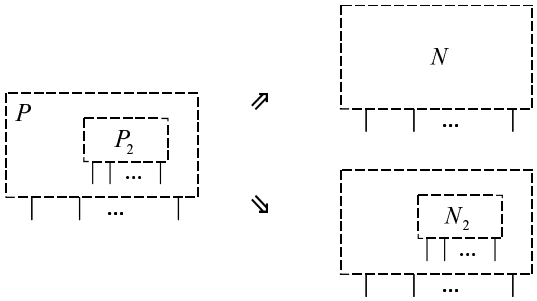
Proof. Assume that $P \to N \in \mathcal{R}$. There are two cases where critical pairs can arise for a net which contains P:

case 1: there is no overlap between rules. We assume that there is a rule $P_1 \to N_1 \in \mathcal{R}$ where P_1 does not overlap with P. In this case, the reduction is strongly confluent:



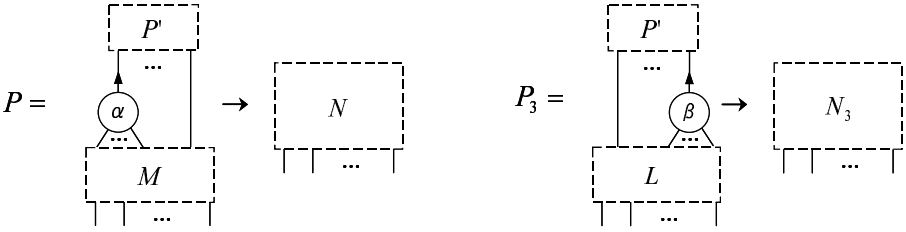
case 2: there are overlaps between rules.

case 2.1: We assume that there is a rule $P_2 \to N_2 \in \mathcal{R}$ where P_2 is a subnet of P.



This case can not arise if $\mathcal{R}$ is well formed. Therefore $P_2 \to N_2 \notin \mathcal{R}$

case 2.2: We assume that there is a rule $P_3 \to N_3 \in \mathcal{R}$ where P_3 contains the subnet of P'.



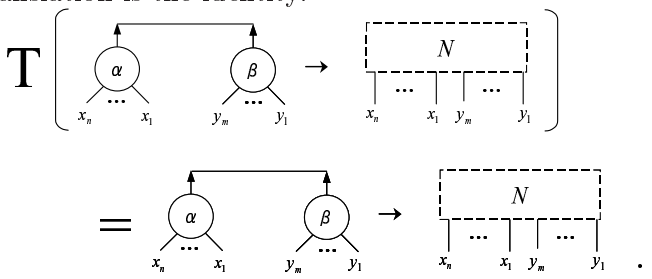
There is no sequential set which contains both P and P_3, therefore $P_3 \to N_3 \notin \mathcal{R}$.

□

5 Translation

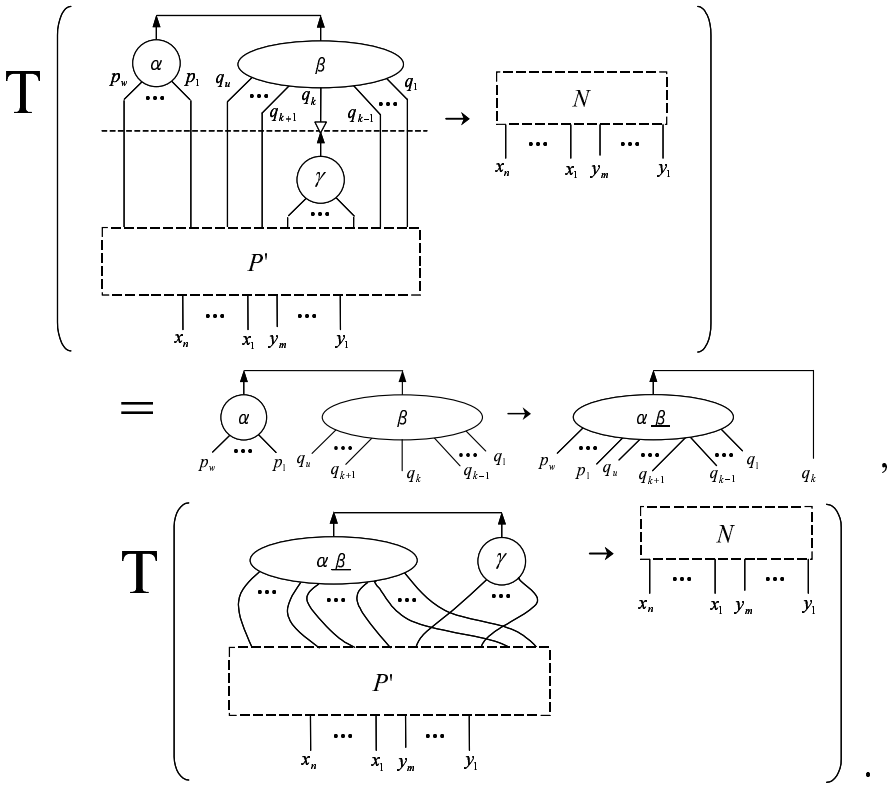
In this section, we define the translation function **T** from interaction rules with nested active pairs to interaction rules with only active pairs:

- If a nested active agent contains an active pair of just two agents, then the translation is the identity:

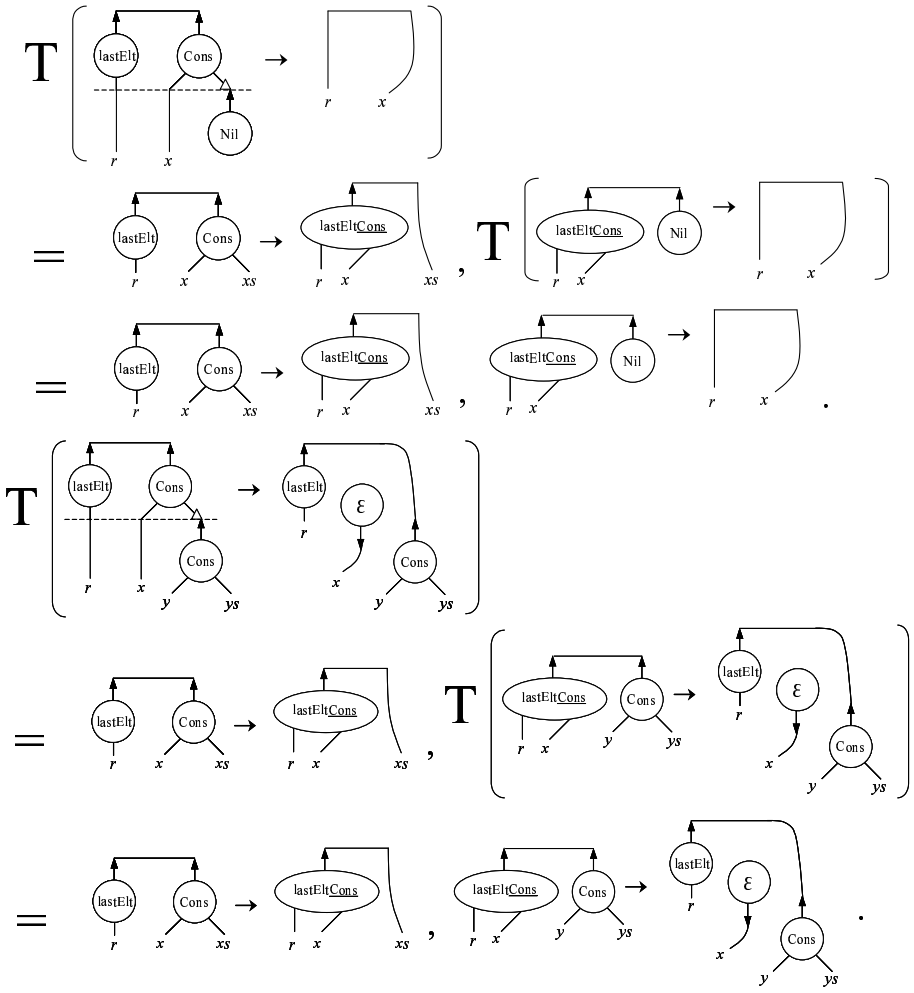


- The translation of a rule $P \to N$ where
  $P = \langle \alpha(p_1, ..., p_w) \bowtie \beta(q_1, .., q_k, .., q_u),\ q_k - \gamma(z_1, ..., z_l),\ \mathbf{a} \rangle$ where $\mathbf{a}$ is a sequence of agents, generates the following rules:
  - $\alpha(p_1, ..., p_w) \bowtie \beta(q_1, .., q_k, .., q_u) \to q_k - \alpha\underline{\beta}(q_1, .., q_{k-1}, q_{k+1}, .., q_u, p_1, ..., p_w)$ where $\alpha\underline{\beta}$ is a new agent named from a concatenation of the LHS nested active pair agents. Since q_k is connected to the principal port of γ, an active pair $(\alpha\underline{\beta}, \gamma)$ will be formed.
  - $\langle \alpha\underline{\beta}(q_1, .., q_{k-1}, q_{k+1}, .., q_u, p_1, ..., p_w) \bowtie \gamma(z_1, ..., z_l),\ \mathbf{a} \rangle \to N$. This rule is recursively translated to obtain a rule with just an active pair.

    Graphically, this translation is given by:



Example 5.1 We give the translation of the function in Example 3.1 that computes and returns the last element of a list.



Lemma 5.2 *Let $\mathcal{R}$ be a well-formed rule set in* INP *and $R_1, R_2 \in \mathcal{R}$. Then, a rule set $\mathbf{T}[R_1] \cup \mathbf{T}[R_2]$ contains no rule such that $P \to N_1$ and $P \to N_2$ where $N_1 \neq N_2$.*

Proof. Let $R_1 = P_1 \to M_1$ and $R_2 = P_2 \to M_2$.

case 1: the active pairs in P_1 and P_2 are different. In this case, distinct names are introduced by $\mathbf{T}$ for those active pairs respectively. Therefore, every LHS of the rules generated by recursively applying $\mathbf{T}$ also have distinct active pairs.

case 2: the active pairs in P_1 and P_2 are the same. Because both P_1 and P_2 belong to the same sequential set, then P_1 and P_2 have a same sequence of agents succeeding from the active pair. Therefore, in the set obtained from this sequence by using $\mathbf{T}$, there is no rule such that $P \to M_1$ and $P \to M_2$. For the remaining agents, it turns out that there is no such rule by applying case 1.

□

Proposition 5.3 *Let $\mathcal{R}$ be a well-formed rule set in* INP*. The set $\bigcup \mathbf{T}[R]$ where $R \in \mathcal{R}$ is a correct rule set in* ORN.

Proof. From the definition of $\mathbf{T}$, it is clear that every LHS of rules obtained by using $\mathbf{T}$ contains only an active pair. Moreover, by Lemma 5.2, there is no rule $P \to N_1$ and $P \to N_2$ in the resulting rule set. □

Proposition 5.4 (Conservativity) *Let $\mathcal{R}$ be a well-formed set of rules in* INP*. If $P \to N \in \mathcal{R}$, then $P \Rightarrow^* N$ by using the rules obtained by the translation $\mathbf{T}[P \to N]$.*

Proof. If P is just an active pair, then we can perform $P \Rightarrow N$ because $\mathbf{T}[P \to N] = P \to N$.

If $P = \langle \alpha(\mathbf{x}) \bowtie \beta(\mathbf{y}, y),\ y - \gamma(\mathbf{z}),\ \mathbf{a} \rangle$ where $\mathbf{x}, \mathbf{y}, \mathbf{z}$ are sequences of auxiliary ports and $\mathbf{a}$ is a sequence of agents, then

$$\mathbf{T}[P \to N] = \alpha(\mathbf{x}) \bowtie \beta(\mathbf{y}, y) \to \alpha\underline{\beta}(\mathbf{x}, \mathbf{y}) - y,\ \mathbf{T}[\, \langle \alpha\underline{\beta}(\mathbf{x}, \mathbf{y}) \bowtie \gamma(\mathbf{z}),\ \mathbf{a} \rangle \to N].$$

By using the first rule,

$$\alpha(\mathbf{x}) \bowtie \beta(\mathbf{y}, y),\ y - \gamma(\mathbf{z}),\ \mathbf{a} \ \Rightarrow\ \alpha\underline{\beta}(\mathbf{x}, \mathbf{y}) - \gamma(\mathbf{z}),\ \mathbf{a}.$$

Applying recursively this operation to the rule $\langle \alpha\underline{\beta}(\mathbf{x}, \mathbf{y}) \bowtie \gamma(\mathbf{z}),\ \mathbf{a} \rangle \to N$ and the nested agent pair $\alpha\underline{\beta}(\mathbf{x}, \mathbf{y}) \bowtie \gamma(\mathbf{z})$, we will perform $P \Rightarrow^* N$. □

6 Discussion

6.1 Implementation

In this section we briefly discuss implementation issues of INP. There are two approaches to implement INP. One is to translate into ORN rules then use existing evaluators of Interaction Nets. The other is to implement them directly. Here we look at this second option, and show how the main tasks of performing computation in this framwork can be achieved. Our aim here is to show that a direct implemention of INP can be done quite easily. We describe a simple method of achieving this.

The main tasks of an Interaction Net evaluator are to locate the next active pair to reduce, find the matching rule, and apply it to the active pair.

Locating the next active pair can be done locally during rewrite; while rewiring the ports, we check if an active pair is formed then push it into a stack. Reduction will then pop the active pairs from the stack and find the matching rule to apply.

We can store rules in a hash table with a key formed from an ordered concatenation of the (LHS) active pair names. Since INP rules can have more than one active pair of the same agents, we maintain a list such that each key maps onto a list of rules that share the same active pair names. We iterate through the list to find a rule that matches the structure of the active pair to be reduced.

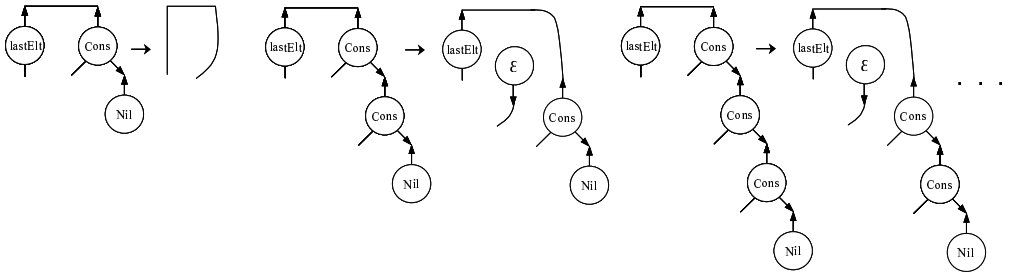
Although ORN will find the matching rule in constant time (each key will only map to one rule) the total number of interactions I performed in ORN: $I(\text{ORN}) > I(\text{INP})$ for a system with nested agents, and $I(\text{ORN}) = I(\text{INP})$ if there is no nested agents. This comes from the fact that ORN introduces extra auxiliary rules for pattern matching.

If we define the cost of computation to be the number of interactions performed, then INP provides an efficient model. However, without empirical studies we are not able to say which system is efficient in terms of execution speed.

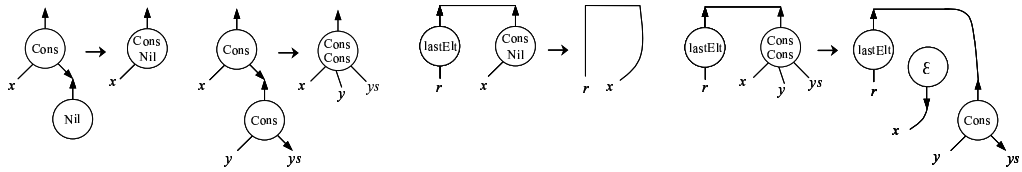
6.2 Related Works

In this section, we discuss other approaches to nested pattern matching by using methods that have been proposed as extensions of Interaction Nets.

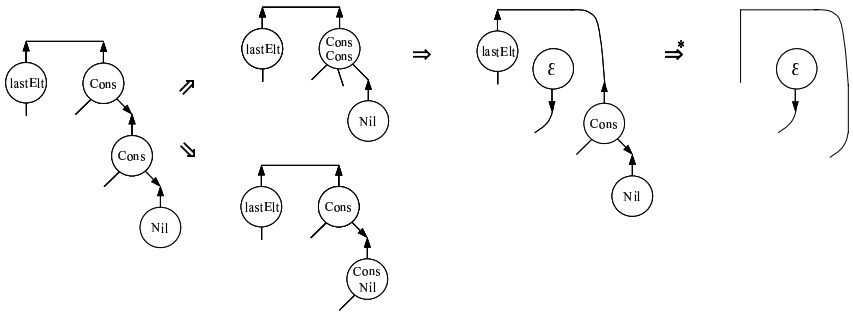
Pattern matching on more than one argument: Sinot and Mackie [10] introduced *Macros* for Interaction Nets and they allow pattern matching on more than one argument by relaxing the restriction of one principal port per agent. Their system requires all principal ports of an agent in the LHS net of a rule to be connected to principal ports of other agents for the purpose of holding the property of strong confluence. Therefore, this system is useful as a conservative extension. However, we can hardly encode the function `lastElt` as it requires nested pattern matching. This is because in the case that the `Cons` agent has two principal ports, we have to write all cases as follows:



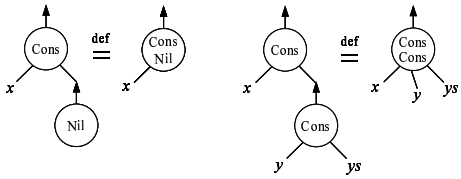
Alexiv's *interaction nets with multiple principal ports* (IMNPP) [1] is also useful for this purpose because this system also allows more than one principal port per agent. However, interactions are still performed only on an active pair. Therefore, in the case of nested pattern matching, we have to introduce auxiliary agents and rules as in Section 3. As another solution, we can introduce rules between `Cons` and `Nil`:



These cause computation between the list structures even if it is not needed.



Computation for nets: Bachet [2] proposed computation for nets on interaction rules as *abbreviations*, where nets are captured as an agent and reductions of the agent are realized by the rules corresponding to the computation of the net. As an example of applying this method to nested pattern matching, we consider our example function `lastElt`. One solution is to define the agent `lastElt` by using other agents that have already been defined. It is not simple to find a good combination with those agents. As another solution, we introduce abbreviations for list structures:



However, we have to define rules between `lastElt` and `Cons` for the case that those abbreviations are unfolded, therefore we have to introduce auxiliary agents in the end.

7 Conclusion

We have shown how to extend Interaction Nets to facilitate nested pattern matching without introducing auxiliary rules. This provides a convenient and a more natural way of expressing Interaction Net programs. We see this extension as a positive step towards using Interaction Nets as a practical programming language.

References

[1] Alexiev, V., "Non-deterministic interaction nets," Ph.D. thesis (1999), adviser-Jia You.

[2] Bechet, D., *Partial evaluation of interaction nets*, in: M. Billaud, P. Castéran, M. M. Corsini, K. Musumbu and A. Rauzyand, editors, *Proceedings of the Second Workshop on Static Analysis WSA'92*, Bigre Journal **81-82**, 1992, pp. 331–338.

[3] Fernández, M. and I. Mackie, *From term rewriting to generalised interaction nets*, in: H. Kuchen and S. D. Swierstra, editors, *Proceedings of the 8th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, Lecture Notes in Computer Science **1140** (1996), pp. 319–333.

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

[5] Lafont, Y., *Interaction nets*, in: *Seventeenth Annual Symposium on Principles of Programming Languages* (1990), pp. 95–108.

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

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

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

[9] Pinto, J. S., *Parallel evaluation of interaction nets with mpine.*, in: A. Middeldorp, editor, *RTA*, Lecture Notes in Computer Science **2051** (2001), pp. 353–356.

[10] Sinot, F.-R. and I. Mackie, *Macros for interaction nets: A conservative extension of interaction nets.*, Electr. Notes Theor. Comput. Sci. **127** (2005), pp. 153–169.

Sub-λ-calculi, classified

François-Régis Sinot[1]

Universidade do Porto (DCC & LIACC)
Rua do Campo Alegre 1021–1055
4169–007 Porto, Portugal

Abstract

When sharing is studied in the λ-calculus, some sub-calculi often pop up, for instance λI or the linear λ-calculus. In this note, we propose a definition and a complete classification of a large class of such sub-calculi.

Keywords: λ-calculus, sharing, linearity

1 Introduction

Sharing is an important but difficult issue, in particular in the λ-calculus. Sometimes, in order to simplify the problem or tackle it in a more focused way, attention is restricted to a particular subsystem of the λ-calculus. Such subsystems include λI, where erasing is forbidden, or the linear λ-calculus, where all terms are linear [1]. These subsystems are defined by imposing some restrictions on the number of occurrences of bound variables. In this note, we propose to generalise this idea and study in a systematic way such sub-calculi.

2 Sub-calculi

We assume basic knowledge of the λ-calculus, see [1] for more details.

Definition 2.1 We define the number of free occurrences of a variable x in a λ-term

[1] Email: frs@dcc.fc.up.pt

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

t, written $|t|_x$, as follows:

$$\begin{aligned} |x|_x &= 1 \\ |y|_x &= 0 \\ |t\,u|_x &= |t|_x + |u|_x \\ |\lambda x.t|_x &= 0 \\ |\lambda y.t|_x &= |t|_x \end{aligned}$$

Remark that $|t|_x$ is well-defined on α-equivalence classes (i.e. if $t =_\alpha u$ then $|t|_x = |u|_x$ for any variable x).

Our purpose is to study subsystems of the λ-calculus. One possible way to characterise such subsystems in the most general fashion would be to have a predicate $\mathcal{P}$ on λ-terms such that t belongs to the subsystem if and only if $\mathcal{P}(t)$ holds. This representation is too general to be interesting: there is little hope to obtain a nice characterisation theorem in such a general setting. We thus focus our attention on the following, more restricted class of subsystems.

Definition 2.2 If $\mathcal{P}$ is a predicate on natural numbers, we define the set of $\lambda_\mathcal{P}$-terms as follows:

$$\begin{aligned} t, u ::= {} & x \mid t\,u \\ & \mid \lambda x.t \qquad\qquad \text{if } \mathcal{P}(|t|_x) \end{aligned}$$

In addition, we may or may not impose that $\mathcal{P}(|t|_x)$ holds for every free variable x of an open term t. If we do, we say that we are under the *strict convention*; if we do not, we say that we are under the *relaxed convention*. We also define the $\lambda_\mathcal{P}$-calculus as the set of $\lambda_\mathcal{P}$-terms equipped with β-reduction $\to_\beta$.

The strict convention entails some unpleasant syntactic accidents, as will be shown later. These accidents would disappear if we added (unconstrained) constants. However, the pure λ-calculus view of constants is exactly as free variables, which should thus be unconstrained. Consequently, we always assume the relaxed convention, unless otherwise stated.

With such a definition, it is natural to wonder how well $\mathcal{P}$ characterises $\lambda_\mathcal{P}$. From now on, operations on propositions are always implicitly lifted to predicates, which means that, for instance, if $\mathcal{P}$ and $\mathcal{Q}$ are predicates, $\mathcal{P} \wedge \mathcal{Q}$ is the predicate defined by, for all n, $(\mathcal{P} \wedge \mathcal{Q})(n) = \mathcal{P}(n) \wedge \mathcal{Q}(n)$. Moreover, $\mathcal{P}$, $\mathcal{Q}$, implicitly denote predicates on natural numbers. We also allow the definition of predicates by partial application of infix binary predicates, in a Haskell-like style, e.g. (≥ 3) is the predicate defined by, for all n, $(\geq 3)(n) = (n \geq 3)$.

Proposition 2.3 $(\lambda_\mathcal{P} \subseteq \lambda_\mathcal{Q}) \iff (\mathcal{P} \Rightarrow \mathcal{Q})$.

Proof.

$\Rightarrow$ Assume $n \geq 1$ and $\mathcal{P}(n)$. We can build the term $t = \lambda x.\underbrace{x \ldots x}_{n} \in \lambda_\mathcal{P} \subseteq \lambda_\mathcal{Q}$. Then $t \in \lambda_\mathcal{Q}$, hence $\mathcal{Q}(n)$, by definition of $\lambda_\mathcal{Q}$. For the case $n = 0$, we use

instead $t = \lambda x.z$ where z is a free variable (remember that we assume the relaxed convention).

$\Leftarrow$ Assume $t \in \lambda_{\mathcal{P}}$ and let $\lambda x.u$ be a subterm of t. By definition of $\lambda_{\mathcal{P}}$, $\mathcal{P}(|u|_x)$ holds, and so does $\mathcal{Q}(|u|_x)$. Since this holds for every sub-abstraction of t, we may conclude $t \in \lambda_{\mathcal{Q}}$.

□

Remark 2.4 The left-to-right implication is false under the strict convention without constants, for instance $\lambda_{\bot} = \lambda_{(=0)} = \varnothing$.

In particular, an immediate corollary of Proposition 2.3 is that $\lambda_{\mathcal{P}} = \lambda_{\mathcal{Q}}$ if and only if $\mathcal{P} \Leftrightarrow \mathcal{Q}$. In other words, $\mathcal{P}$ exactly characterises $\lambda_{\mathcal{P}}$.

3 Stability

Definition 3.1 A set of λ-terms S is said to be stable if it is closed under β-reduction, i.e. if whenever $t \in S$ and $t \to_\beta u$, then $u \in S$. Moreover, we also say that $\mathcal{P}$ is stable if $\lambda_{\mathcal{P}}$ is stable.

The notion of $\lambda_{\mathcal{P}}$-calculus only makes sense when the set of $\lambda_{\mathcal{P}}$-terms is stable. Fortunately, we can characterise this in a slightly more operational way.

Lemma 3.2 *$\mathcal{P}$ is stable if and only if*

$$\forall m \geq 0, n \geq 0, 0 \leq k \leq n. \mathcal{P}(m) \wedge \mathcal{P}(n) \implies \mathcal{P}(n + k \cdot m - k).$$

Proof. First remark that, with the notations of the lemma, $n + k \cdot m - k \geq 0$, because $n - k \geq 0$ and $k \cdot m \geq 0$. Let's consider an arbitrary β-reduction under an arbitrary binder (if the reduction is not under a binder, this is irrelevant to the kind of conditions we have):

$$\lambda y.C[(\lambda x.t)\, u] \to_\beta \lambda y.C[t\{x := u\}].$$

Let us write $m = |t|_x$, $n = |C[(\lambda x.t)\, u]|_y$ and $k = |u|_y$. We thus have $|C[t]|_y = n - k$ and $|C[t\{x := u\}]|_y = n + k \cdot m - k$. The reduct thus belongs to $\lambda_{\mathcal{P}}$ if and only if $\mathcal{P}(n + k \cdot m - k)$ holds for all n and k (corresponding to every choice of outer binder λy). Indeed, $\mathcal{P}$ is stable if and only if $\mathcal{P}(n + k \cdot m - k)$ holds whenever $\mathcal{P}(m)$ and $\mathcal{P}(n)$ hold. □

Theorem 3.3 *$\mathcal{P}$ is stable (i.e. the $\lambda_{\mathcal{P}}$-calculus is well-defined) if and only if*

$$\forall m \geq 0, n \geq 1. \mathcal{P}(m) \wedge \mathcal{P}(n) \implies \mathcal{P}(m + n - 1).$$

Proof. Using Lemma 3.2 and the fact that the other implication is trivial, we assume that $\forall m \geq 0, n \geq 1. \mathcal{P}(m) \wedge \mathcal{P}(n) \implies \mathcal{P}(m + n - 1)$ and we only have to show that $\mathcal{P}(n + k \cdot m - k)$ holds if $m \geq 0$, $n \geq 0$, $0 \leq k \leq n$, $\mathcal{P}(m)$ and $\mathcal{P}(n)$. If $n = 0$, this is trivially true, because $k = 0$, thus $n + k \cdot m - k = 0$, and $\mathcal{P}(0) = \mathcal{P}(n)$ holds by hypothesis. We may thus assume $n \geq 1$, and we prove the statement by induction on k. If $k = 0$, it is true because $n + k \cdot m - k = n$ and $\mathcal{P}(n)$ holds

by hypothesis. Let $0 \leq k \leq n-1$ and assume $\mathcal{P}(n + k \cdot m - k)$ holds. Then, $\mathcal{P}(n+(k+1)\cdot m-(k+1)) = \mathcal{P}(m+(n+k\cdot m-k)-1)$ holds using the assumption, the induction hypothesis, and the fact that $n + k \cdot m - k \geq 1$ because $k \leq n-1$ and $k \cdot m \geq 0$. We indeed conclude that the statement holds for all k such that $0 \leq k \leq n$. □

4 Examples

Example 4.1 We use Theorem 3.3 to recover some well-known stability results.

- $\lambda_{\top}$ (aka. the λ-calculus) is stable;
- $\lambda_{(=1)}$ (aka. the linear λ-calculus) is stable;
- $\lambda_{(\geq 1)}$ (aka. λI) is stable;
- $\lambda_{(\leq 1)}$ (aka. the affine λ-calculus) is stable.

Proof. We only show the proof for $\lambda_{(\geq 1)}$. Assume $m \geq 1$ and $n \geq 1$, then $m+n-1 \geq 1+1-1 = 1$. Using Theorem 3.3, we conclude that (≥ 1) is stable. The proof shows that 1 plays a special role in this framework. □

Example 4.2 There are also some less usual sub-calculi, with a more questionable computational content.

- $\lambda_{\bot}$ (where there is no λ-abstraction) is stable (under the strict convention, this calculus is empty);
- $\lambda_{(=0)}$ (where there is no occurrence of bound variables) is stable (under the strict convention, this calculus is empty);
- $\lambda_{(\geq 2)}$ is stable;
- more generally, if $b \geq 1$, $\lambda_{(\geq b)}$ is stable;
- however, if $b \geq 2$, $\lambda_{(\leq b)}$ is not stable.

Proof. The first two sub-calculi are degenerated, which is evidenced by the fact that the condition in Theorem 3.3 is true because the premises of the implication can never be satisfied. Let $b \geq 1$, we verify that $\lambda_{(\geq b)}$ is stable. Let $m \geq b$ and $n \geq b$, then $m + n - 1 \geq 2 \cdot b - 1 \geq b$, since $b - 1 \geq 0$. Thus $\lambda_{(\geq b)}$ is stable. □

Using Theorem 3.3, we give some non-trivial sub-calculi (or non-sub-calculi) of the λ-calculus (of course only those of the form $\lambda_{\mathcal{P}}$ for some $\mathcal{P}$).

Example 4.3 Let $\mathsf{odd}(n) = (\exists k \geq 0. n = 1 + 2 \cdot k)$, then odd is stable. The "odd calculus" λ_{odd} is a simple, non-trivial stable sub-calculus.

Proof. Assume $\mathsf{odd}(m)$ and $\mathsf{odd}(n)$. Then, there exist $k, k' \geq 0$ such that $m = 1+2\cdot k$ and $n = 1+2\cdot k'$. Then $m+n-1 = (1+2\cdot k)+(1+2\cdot k')-1 = 1+2\cdot(k+k')$ with $k + k' \geq 0$, and indeed $\mathsf{odd}(m + n - 1)$ holds. □

Remark 4.4 The "even calculus" λ_{even} defined by $\mathsf{even}(n) = (\exists k. n = 2 \cdot k)$ is not stable (we are therefore reluctant to call it a calculus). This can be seen as a consequence of Theorem 3.3 or directly: $\lambda y.(\lambda x.x\, x)\, y\, y \to_{\beta} \lambda y.y\, y\, y$.

In fact, the previous example can be generalised to the following large class of stable sub-calculi.

Example 4.5 Let $q \geq 1$ and $\mathsf{mult}_q(n) = (\exists k \geq 0. n = 1 + k \cdot q)$, then $\lambda_{\mathsf{mult}_q}$ is stable.

Proof. Similarly, using $(1 + k \cdot q) + (1 + k' \cdot q) - 1 = 1 + (k + k') \cdot q$. □

We will see in Section 6 that essentially all stable $\lambda_{\mathcal{P}}$-calculi can be decomposed in calculi of this form.

5 Syntactic Properties

Theorem 5.1 *If $\mathcal{P}$ is stable, the $\lambda_{\mathcal{P}}$-calculus is confluent.*

Proof. Assume $u_1 \,{}^*_\beta\!\leftarrow t \rightarrow^*_\beta u_2$ in the $\lambda_{\mathcal{P}}$-calculus. Then there exists a λ-term v such that $u_1 \rightarrow^*_\beta v \,{}^*_\beta\!\leftarrow u_2$, by confluence of the λ-calculus, and v is a $\lambda_{\mathcal{P}}$-term by stability of $\mathcal{P}$. □

Theorem 5.2 *If $\mathcal{P}$ is stable, the $\lambda_{\mathcal{P}}$-calculus is strongly normalising if and only if $\mathcal{P}(n)$ does not hold for any $n \geq 2$.*

Proof. If $\mathcal{P}(n)$ does not hold for any $n \geq 2$, the $\lambda_{\mathcal{P}}$-calculus is a subsystem of $\lambda_{(\leq 1)}$, i.e. the affine λ-calculus, which is strongly normalising. Conversely, assume that $\mathcal{P}(n)$ holds for some $n \geq 2$. Then we can build the non-normalising $\lambda_{\mathcal{P}}$-term $(\lambda x. \underbrace{x \ldots x}_{n})\,(\lambda x. \underbrace{x \ldots x}_{n})$. □

6 Classification

With Theorem 3.3 in hand, we characterise further the sub-calculi of the λ-calculus (of the form $\lambda_{\mathcal{P}}$ for some $\mathcal{P}$).

Proposition 6.1 *If $\mathcal{P}$ and $\mathcal{Q}$ are stable, then $\mathcal{P} \wedge \mathcal{Q}$ is stable.*

Proof. Straightforward, even without Theorem 3.3. □

Remark 6.2 If $\mathcal{P}$ and $\mathcal{Q}$ are stable, $\mathcal{P} \vee \mathcal{Q}$ is not necessarily stable.

Proof. Let $\mathcal{P}(n) = (\exists k \geq 0. n = 1 + 2 \cdot k)$ and $\mathcal{Q}(n) = (\exists k \geq 0. n = 1 + 3 \cdot k)$. According to Example 4.5, $\mathcal{P}$ and $\mathcal{Q}$ are stable. $(\mathcal{P} \vee \mathcal{Q})(3)$ holds since $\mathcal{P}(3)$ holds, $(\mathcal{P} \vee \mathcal{Q})(4)$ holds since $\mathcal{Q}(4)$ holds, but $(\mathcal{P} \vee \mathcal{Q})(3 + 4 - 1) = (\mathcal{P} \vee \mathcal{Q})(6)$ does not hold since neither $\mathcal{P}(6)$ or $\mathcal{Q}(6)$ holds. In other words, $\mathcal{P} \vee \mathcal{Q}$ is not stable. □

Proposition 6.3 *If $\mathcal{P}$ is stable and $\mathcal{P}(2)$ holds then $\mathcal{P}(n)$ holds for all $n \geq 2$.*

Proof. By induction on n. $\mathcal{P}(2)$ holds by hypothesis. Assume $\mathcal{P}(n)$ holds, then using Theorem 3.3, $\mathcal{P}(n + 2 - 1) = \mathcal{P}(n + 1)$ also holds. □

Proposition 6.4 *If $\mathcal{P}$ is stable and if $\mathcal{P}(0)$ and $\mathcal{P}(n)$ hold for some $n \geq 2$, then $\mathcal{P}(n)$ holds for all $n \geq 0$. In other words, we get the full λ-calculus.*

Proof. We show that $\mathcal{P}(k)$ holds for $0 \leq k \leq n$ by reverse induction on k. $\mathcal{P}(n)$ holds by hypothesis. Let $1 \leq k \leq n$ and assume that $\mathcal{P}(k)$ holds. Then $\mathcal{P}(k - 1) =$

$\mathcal{P}(0+k-1)$ holds using Theorem 3.3, stability of $\mathcal{P}$, the induction hypothesis, and the facts that $k \geq 1$ and $\mathcal{P}(0)$ holds. In particular, $\mathcal{P}(2)$ holds and we use Proposition 6.3. □

As evidenced in Remark 6.2, disjunction is not a well-behaved operation with respect to stability. However, the following proposition exhibits the particular behaviour of 1, and tends to show that, to some extent, the choice of $\mathcal{P}(1)$ is not relevant for the stability of $\mathcal{P}$.

Proposition 6.5 (i) *if $\mathcal{P}$ is stable, then $\mathcal{P} \vee (=1)$ is stable;*

(ii) *if $\mathcal{P} \vee (=1)$ is stable and either $\mathcal{P}(0)$ or $\mathcal{P}(2)$ does not hold, then $\mathcal{P}$ is stable.*

Proof.

(i) Assume $\mathcal{P}$ is stable, $(\mathcal{P} \vee (=1))(m)$ and $(\mathcal{P} \vee (=1))(n)$. If $m = 1$, then $m+n-1 = n$ and $(\mathcal{P} \vee (=1))(m+n-1)$ holds; and similarly if $n = 1$. Otherwise, both $\mathcal{P}(m)$ and $\mathcal{P}(n)$ hold, and $(\mathcal{P} \vee (=1))(m+n-1)$ indeed holds.

(ii) Assume $\mathcal{P} \vee (=1)$ is stable, $\mathcal{P}(m)$ and $\mathcal{P}(n)$ hold. Then $(\mathcal{P} \vee (=1))(m+n-1)$ holds. Either $\mathcal{P}(m+n-1)$ holds and we are done, or $m+n-1 = 1$, hence $m = n = 1$ and $\mathcal{P}(1)$ holds, because the case $m = 0$ and $n = 2$ is excluded.

□

Lemma 6.6 *If $\mathcal{P}$ is stable and there exists $n \geq 2$ such that $\mathcal{P}(n)$ holds, then there exists $q \geq 1$ such that $\mathcal{P}(1+k\cdot q)$ holds for every $k \geq 1$.*

Proof. With the notations of the lemma, let $q = n-1$. We prove by induction on $k \geq 1$ that $\mathcal{P}(1+k\cdot q)$ holds. This is true for $k = 1$. Assume $\mathcal{P}(1+k\cdot q)$ holds, $n+(1+k\cdot q)-1 = 1+(k+1)\cdot q$ and $\mathcal{P}(1+(k+1)\cdot q)$ holds, using Theorem 3.3.□

We now have everything in hand to exhibit a complete classification of the $\lambda_{\mathcal{P}}$-calculi.

Theorem 6.7 *$\mathcal{P}$ is stable if and only of one of the following holds for all n:*

(i) $\mathcal{P}(n) \Leftrightarrow \bot$;

(ii) $\mathcal{P}(n) \Leftrightarrow \top$;

(iii) $\mathcal{P}(n) \Leftrightarrow (n = 0)$;

(iv) $\mathcal{P}(n) \Leftrightarrow (n = 0 \vee n = 1)$;

(v) *there exist $0 \leq p \leq \omega$ and $1 \leq q_1 < \ldots < q_p$ pairwise non divisible such that:* $\mathcal{P}(n) \Leftrightarrow (\exists k_1, \ldots, k_p \geq 0. n = 1 + \sum_{1\leq i\leq p} k_i \cdot q_i)$;

(vi) *there exist $1 \leq p \leq \omega$ and $1 \leq q_1 < \ldots < q_p$ pairwise non divisible such that:* $\mathcal{P}(n) \Leftrightarrow (\exists k_1, \ldots, k_p \geq 0, 1 \leq j \leq p. k_j \geq 1 \wedge n = 1 + \sum_{1\leq i\leq p} k_i \cdot q_i)$.

Moreover, this decomposition is unique.

Proof. If one of the cases (i–iv) holds, it has already been noted in Section 4 that $\mathcal{P}$ is stable. If (v) or (vi) holds, this is a consequence of Theorem 3.3, similar to Example 4.5. Conversely, suppose $\mathcal{P}$ is stable. We distinguish cases according to whether or not $\mathcal{P}(0)$ holds.

- If $\mathcal{P}(0)$ holds, does there exist $n \geq 2$ such that $\mathcal{P}(n)$ holds ?
  - If there is such a n, we are in case (ii), thanks to Proposition 6.4.
  - If not, we are indeed in case (iii) or (iv).
- If $\mathcal{P}(0)$ does not hold, we look at $\mathcal{P}(1)$.
  - If $\mathcal{P}(1)$ holds, we prove by induction on $p \geq 0$ that there exist $1 \leq q_1 < \ldots < q_p$ pairwise non divisible such that $(\exists k_1, \ldots, k_p \geq 0. n = 1 + \sum_{1 \leq i \leq p} k_i \cdot q_i) \Rightarrow \mathcal{P}(n)$. This is true for $p = 0$. Assume this is true for some p, and consider the smallest n not equal to $(1 + \sum_{1 \leq i \leq p} k_i \cdot q_i)$ for some $k_1, \ldots, k_p \geq 0$ such that $\mathcal{P}(n)$ holds. There are two cases. If there is no such n, that means that the condition is verified and $\mathcal{P}$ is fully described. Otherwise, let $q_{p+1} = n - 1$. Indeed, by construction, $q_{p+1} > q_p$ and none of $q_1, \ldots, q_p$ is a divisor of q_{p+1}. Thanks to Theorem 3.3 and in a similar way to Lemma 6.6, for all $k_{p+1} \geq 0$, $\mathcal{P}(1 + k_{p+1} \cdot q_{p+1})$. Then, using again Theorem 3.3, the statement holds for $p + 1$. If the process stops, the equivalence is clear. If it does not, let's write $\mathcal{P}_p(n) = (\exists k_1, \ldots, k_p \geq 0. n = 1 + \sum_{1 \leq i \leq p} k_i \cdot q_i)$. For all p, $\mathcal{P}_p \Rightarrow \mathcal{P}_{p+1} \Rightarrow \mathcal{P}$ where the first implication is strict. The sequence $(\mathcal{P}_p)_p$ is strictly increasing and bounded, it thus has a limit $\mathcal{P}_\omega$. There is no n such that $\mathcal{P}(n)$ but not $\mathcal{P}_\omega(n)$, because this would contradict the construction. We conclude $\mathcal{P} \Leftrightarrow \mathcal{P}_\omega$.
  - If $\mathcal{P}(1)$ does not hold, let's consider the smallest $n \geq 2$ such that $\mathcal{P}(n)$ holds. If there is no such n, we are in case (i). Otherwise, we can proceed as in the previous case, starting at $p = 1$, with $q_1 = n - 1$, and obtain case (vi).

Unicity is clear: the different cases do not overlap, and in cases (v) or (vi), the non-pairwise divisibility of $q_1, \ldots, q_p$ ensures that there is no redundancy. □

Remark 6.8 Theorem 6.7 gives a complete classification of the stable $\lambda_{\mathcal{P}}$-calculi in terms of equality, but this is not necessarily the "best" description. For instance, we have seen that (≥ 3) is stable, but its description using Theorem 6.7 is case (vi) with $p = \omega$ and q_i is the i-th prime number. In particular, it is not a finite description.

7 Conclusion

We have defined and given a complete characterisation of a class of subsystems of the λ-calculus taking into consideration the number of occurrences of variables, which is a crucial issue for sharing. We recover well-known calculi such as λI or the linear λ-calculus, but we also discover unconventional calculi whose interest as a computational model remain to study. Moreover, our characterisation is very algebraic and may lead to a better understanding of the λ-calculus and its subsystems.

Acknowledgement

This article has benefited from comments by several anonymous referees.

References

[1] Barendregt, H. P., "The Lambda Calculus: Its Syntax and Semantics," Studies in Logic and the Foundations of Mathematics **103**, North-Holland Publishing Company, 1984, second, revised edition.

Modeling and Verifying Graph Transformations in Proof Assistants

Martin Strecker[1]

IRIT
Université Paul Sabatier
118 route de Narbonne
F-31062 Toulouse

Abstract

This paper takes first steps towards a formalization of graph transformations in a general setting of interactive theorem provers, which will form the basis for proofs of correctness of graph transformation systems. Whereas graph rewriting is usually performed by mapping a pattern graph into a source graph by means of a graph morphism and then carrying out operations on the image node and edge set, this article generalises the notion of pattern graph to path expressions, which are formulae in a fragment of first-order logic. We examine the correspondence with traditional graph rewriting and show that this interpretation is beneficial when formally reasoning about model transformations with the aid of proof assistants.

Keywords: Graph Transformations, Theorem Proving

1 Introduction

Graph rewriting examines which structural changes are engendered when applying rewrite rules to a graph. There is no unique approach to graph rewriting - one may cite algebraic [Bar03] and categorical [CMR+96,EHK+97] formalisms.

The discipline has accumulated an impressive amount of results on properties of rewrite systems (such as confluence and termination) resulting from specific rule formats [Plu99]. Recently, there is a growing practical interest in graph rewriting in the context of model driven engineering, where a software or hardware artifact is represented graphically and can be refined or refactored by the application of graph rewriting rules. Several graph rewriting tools are available. They emanate from foundational work and are usually equipped with some analyses of rule properties [Tae03,KS06,Agr04], or take a more pragmatic view (ATL [BBDV03] and Kermeta [MFV+05]).

In spite of a large body of work on graph transformations, the question of verification of transformations "in general" is far from settled. The foundational work

[1] Email: strecker@irit.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

of [Cou90] aims at a logical characterization of graph transformations, where effective verification of structural properties is not a primary concern. Usually, however, graph transformation systems are perceived as extensions of term rewriting systems, so much of the effort has gone into investigating specific properties such as confluence and termination [Plu99], which does not necessarily allow to determine whether a graph has a certain shape after transformation. These questions may be answered for graph replacement systems having a restricted structure [FM97], for properties expressed in specialized logics such as monadic second order logic [KS93] or type systems [BCE+05]. There are automated approaches based on model checking [Var04], which however can only handle graphs with an a priori bounded number of elements. [RD06] presents techniques for dealing with specific structural properties such as multiplicities.

However, in some circumstances, it is useful to resort to a more general setting, in order to express stronger properties or to overcome limitations of a restricted rule format. This gives us the same kind of advantage a program logic may have over a static analysis for determining the correctness of an imperative program – and it suffers from the same drawbacks, notably a sometimes heavy user intervention to carry out interactive proofs.

The verification of structural properties will be the main focus of this paper. The work reported here has grown out of an effort to formalise model transformations in interactive proof assistants. A first attempt [SG06], aiming at formalising traditional graph rewriting as sketched above, required complex reasoning about graph morphisms. It has turned out that replacing the pattern graph by formulae over graph structure (which we will call *path formulae* in the following) yields much more manageable proof obligations. At the same time, path formulae are more expressive than pattern graphs and have therefore an interest in their own, independently from concerns about formal verification.

Path formulae can be understood as formulae over a fragment of first order logic (possibly including transitive closure), which are interpreted over graphs. Determining whether a graph satisfies a path formula is decidable, which is indispensable for effectively applying a transformation rule to a given graph. On the downside, validity of path formulae may not be decidable, so that interactive proofs become necessary.

The paper is structured as follows: In Section 2, we informally introduce generalised graph transformations. The formal model is presented in Section 3. In Section 4, we show how we can recover the traditional model of graph rewriting. We take a glimpse at how to reason about graph transformations in a proof assistant in Section 5 before concluding with an outlook on future work.

2 Example Transformations

To set the stage, we describe two toy transformations: a transformation duplicating a graph, and another one implementing a simple garbage collector.

The purpose of the *graph duplication* transformation is to generate a new graph consisting of two exact copies of the original graph. We assume that the original graph has nodes of type `Node`, with edges of type `E` between them. For the purposes

of transformation, we need nodes of type `Orig`, supposed to mark the nodes of the original graph during transformation, and edge types `Or` (between `Orig` and `Node`) and `Cp` (between a node and its original).

Duplication proceeds in several steps: First, we mark all nodes of the original graph with `Orig` nodes. We then create a duplicate node for each original, memorising the relation between the original and the clone with a `Cp` edge. We can similarly reproduce the edges of the original graph in the copy. All that remains to be done now is to erase the auxiliary marking.

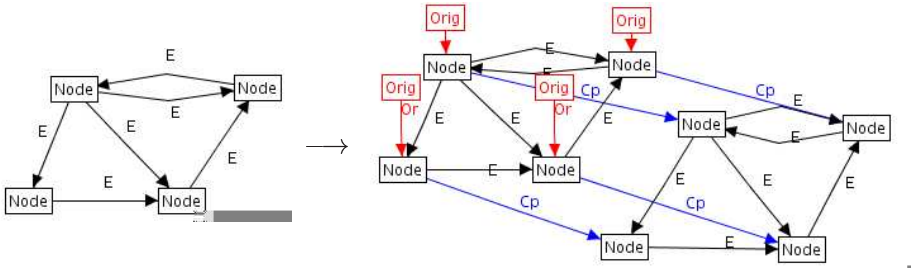


Fig. 1. Duplicating a graph

An example graph and the result of its transformation, just before deletion of the `Cp` edges and the markers, is shown in Figure 1. This is a screen shot of graphs produced by the AGG tool [Tae03], based on a categorical approach, which allows to conveniently model this kind of transformation (a more detailed comparison follows in Section 4).

How do we formalise the marking phase, i.e. the first step of our transformation? In our setting, a transformation rule is composed of two elements: an application condition and an action part. The application condition, a path formula F expressing if and where a rule can be applied, says that the rule can operate on any node n of type `Node` which is not already marked by some node m of type `Orig`:

$$F(n) \equiv Node(n) \wedge \neg \exists m.\ \ (Orig(m) \wedge m \xrightarrow{Or} n)$$

Here, $m \xrightarrow{Or} n$ represents an `Or` edge between m and n.

The action part (not shown here) expresses what we do if F is satisfied for a node n: We generate a new node, say m', having type `Orig`, and we create an `Or`-edge (m', n). We will come back to this example in Section 3.3.

Of course, a single transformation step of this kind will not suffice to mark all nodes of a graph. Rather, we have to iterate the rule until no further application is possible, i.e. until F is false for all nodes of the graph. We will briefly look at this question in Section 5.

The *garbage collector* is an example of a transformation that is not directly expressible in traditional graph rewriting approaches. We assume to have a number of `Root` objects and a number of `Node` objects. `Root` objects are linked to `Nodes` through `rn` edges, `Nodes` are linked among themselves through `nn` edges. Any `Node` not accessible from a `Root` is considered as garbage.

The predicate $G(n)$ saying that node n is garbage can be written as the path formula

$$G(n) \equiv \neg \exists r\ n'.\ \ r \xrightarrow{rn} n' \wedge n' \xrightarrow{nn}{}^{*} n$$

where $\xrightarrow{rn}$ is an `rn` edge (and similarly for `nn`), and the "star" is transitive closure.

$G(n)$ is the application condition of a rule `collect`, whose action part just says that n should be deleted (in doing so, all adjacent edges disappear as well).

In the case of $G(n)$, we have chosen not to make the typing information explicit in the rule itself. In fact, it can be deduced from general typing predicates, expressible as path formulae, that could form the "background theory" of the application. For example, the typing of the `rn` edge is stated as

$$\forall r\ n.\ \ (r\ \xrightarrow{rn}\ n) \longrightarrow Root(r) \wedge Node(n)$$

3 Formal Model

In this section, we formally present the basic notions of our graph rewriting approach, notably graphs, graph transformations and morphisms and some well-formedness conditions we have to impose to ensure consistency of the model. Since our development has been carried out in the Isabelle proof assistant [NPW02], we will use Isabelle's syntax, which we will explain wherever needed.

3.1 Graphs

Our purpose is not to formalize any particular approach to graph rewriting, such as the one based on category theory. Our model is set-theoretic. Roughly, graphs are composed of a finite set of nodes, a finite set of edges and a typing of the nodes.

In order to create new nodes during graph rewriting, we have to have an infinite supply of fresh nodes. We have therefore chosen to take the natural numbers as the base type of our nodes. The edges are sets of pairs of nodes, indexed by an *edge type* $'et$, such as `Cp` and `E` in the introductory example. This precludes to have more than one edge of a given edge type between two nodes. However, under this definition, one can more easily use standard relational operators like composition and transitive closure, which comes handy when defining the semantics of path expressions further below. A *node typing* assigns a node type $'nt$ (such as `Root` and `Node`) to each node of the graph. Altogether, this gives the following definition of the type of graphs:

```
record ('nt, 'et) graph =
  nodes :: nat set
  edges :: 'et ⇒ (nat * nat) set
  nodetp :: nat ⇒ 'nt option
```

(An option type $T\ option$ has a distinguished value *None*, representing undefinedness, and defined values *Some* t for t and element of T.)

In a minimalistic model, node typing is inessential, but it is useful for describing some structural aspects of graphs. However, we have excluded more complex node attributes that would be required for formalising the semantics of an artifact. They could be easily added by providing a mapping in the spirit of *nodetp* from the node set to an attribute domain.

Finiteness of the node set is expressed by a *structural well-formedness* predicate, just as the containment of the endpoints of edges in the node set and well-definedness of node typing:

```
struct-wf-gr :: ('nt, 'et) graph ⇒ bool
struct-wf-gr gr ==
  (finite (nodes gr)) ∧
  (∀ et. (Field (edges gr et)) ⊆ (nodes gr)) ∧
  dom (nodetp gr) = (nodes gr)
```

Here, *dom* is the domain of a mapping, *Field* the union of the domain and range of a relation. Access to a component of a record, such as *nodes*, is written in functional notation.

3.2 Path expressions

The application of graph transformations to a graph is subject to an applicability condition. Traditionally, this applicability condition is given in the form of a *pattern graph* which is mapped, via a graph morphism, into a source graph to which the transformation will be applied.

In a first attempt [SG06], we have faithfully coded this approach, but it has turned out that the formulae resulting from this graph mapping require considerable massaging for being usable any further. We try to circumvent this problem by replacing the pattern graph by a predicate on (source) graphs, which at the same time opens up the possibility of expressing more general properties (we come back to this in Section 4).

However, we have to take care not to use too complex predicates: The least we can expect from a graph rewriting engine is to be able to decide whether a predicate is satisfied for a particular graph and thus, whether a rule is applicable to this graph. Differently said, the model checking problem for the class of predicates should be decidable, even though entailment need not be, see Section 5.

In the following, we present a logic of path formulae, which we have found useful for expressing interesting properties (see the discussion in Section 4). However, there is no intrinsic reason to adopt precisely the language constructors we have selected, and the decidability of the logic, as well as the complexity of model checking, is greatly influenced by this choice. Similar notions can be found in [YRS+06,KS93,Ren03]

To have a fine control over the logic of predicates on graphs, we deeply embed it into Isabelle's higher order logic. We start by defining node set expressions (representing sets of nodes) and path expressions (representing endpoints of paths):

```
datatype 'nt nodeset
  = All-set              — set of all nodes of graph
  | Type-set 'nt         — set of all nodes of given type
  | Singleton-set nat    — singleton containing constant

datatype ('nt, 'et) path
  = Empty-pth                                  — empty path
  | Edge-pth 'et                               — edge with given edge type
  | InvEdge-pth 'et                            — inverse edge
  | Seq-pth ('nt, 'et) path ('nt, 'et) path    — sequential composition
  | Alt-pth ('nt, 'et) path ('nt, 'et) path    — alternative
  | Clos-pth ('nt, 'et) path                   — transitive closure
```

Based on this, we define path formulae, which are constructed from two base cases (set and path formulae, for node set and path expressions, respectively), and the usual Boolean connectives and quantifiers:

```
datatype ('nt, 'et) path-form
  = S-form 'nt nodeset nat          — set formula
```

```
| P-form ('nt, 'et) path nat nat  — path formula
| Neg-form ('nt, 'et) path-form  — negation
| Conj-form ('nt, 'et) path-form ('nt, 'et) path-form  — conjunction
| All-form ('nt, 'et) path-form  — universal quantification
```

With the above, other connectives and the existential quantifier *Ex-form* can be defined as abbreviation. Universal quantification does not use a named, but rather a positional representation of variables (de Bruijn indices, [dB72]). Thus, variables are not identifiers, but just numbers.

In our informal notation of Section 2, we have written *S-form* (*Type-set T*) *n* simply as $T(n)$ and *P-form* (*Edge-pth e*) *n n'* as $n \xrightarrow{e} n'$. For instance, the application condition $\neg\exists r\ n'.\ r \xrightarrow{rn} n' \wedge n' \xrightarrow{nn}{}^{*} n$ of the garbage collector example of Section 2 becomes:

```
Neg-form (Ex-form (Ex-form
  (Conj-form
    (P-form (Edge-pth rn) 1 0)
    (P-form (Clos-pth (Edge-pth nn)) 0 2))))
```

The semantics of expressions respectively formulae is defined by means of functions *nodeset-interp*, *path-interp* respectively *path-form-interp* that interpret the expressions respectively formulae under a variable interpretation $I : nat \Rightarrow nat$ in a graph *gr*.

```
consts
  nodeset-interp :: [nat ⇒ nat, ('nt, 'et) graph, 'nt nodeset] ⇒ nat set
primrec
  nodeset-interp I gr All-set = nodes gr
  nodeset-interp I gr (Type-set t) = {n. nodetp gr n = Some t}
  nodeset-interp I gr (Singleton-set n) = {I n}

consts
  path-interp :: [nat ⇒ nat, ('nt, 'et) graph, ('nt, 'et) path] ⇒ (nat * nat) set

primrec
  path-interp I gr Empty-pth = diag UNIV
  path-interp I gr (Edge-pth e) = edges gr e
  path-interp I gr (InvEdge-pth e) = (edges gr e)^-1
  path-interp I gr (Seq-pth p p') = (path-interp I gr p) O (path-interp I gr p')
  path-interp I gr (Alt-pth p p') = (path-interp I gr p) ∪ (path-interp I gr p')
  path-interp I gr (Clos-pth p) = (path-interp I gr p)^*

consts
  path-form-interp :: [nat ⇒ nat, ('nt, 'et) graph, ('nt, 'et) path-form] ⇒ bool
primrec
  path-form-interp I gr (P-form p n n') = ((I n, I n') ∈ path-interp I gr p)
  path-form-interp I gr (S-form s n) = (I n ∈ nodeset-interp I gr s)
  path-form-interp I gr (Neg-form pf) = (¬ (path-form-interp I gr pf))
  path-form-interp I gr (Conj-form pf pf') =
     ((path-form-interp I gr pf) ∧ (path-form-interp I gr pf'))
  path-form-interp I gr (All-form pf) =
      (∀ x. x ∈ nodes gr ⟶
          path-form-interp ((I o (λ x. x - 1))(0:=x)) gr pf)
```

In the above, *UNIV* is the set of all elements (of the given type), *diag* the diagonal of a set (the relation (e, e)), the converse of a relation R is written $R\hat{}-1$, and O is relation composition and $\circ$ function composition.

Model checking of node set and path expressions, i.e. checking that a graph *gr* satisfies a node set or path expression, reposes on well-known graph algorithms. Universal quantification is relativised to the node set of the graph, which is finite by well-formedness of graphs. Therefore, checking a universal formula only has to examine a finite number of elements.

3.3 Graph Transformations

Roughly speaking, a graph transformation rule should specify under which condition the transformation is applicable, and what to do when applying the transformation at a position in a source graph to obtain a target graph.

The applicability condition is just given by a path formula, as outlined in the previous section. Note that this path formula may contain free variables, for example n in $G(n)$ of Section 2, which can be understood as references to nodes in the source graph. Of course, in its coding as path formula, the free variables are numbers.

It is these numbers that we refer to when specifying the action: we say which nodes are to be deleted respectively freshly generated (*ndel* resp. *ngen*) and which edges are deleted resp. generated (*edel* resp. *egen*). Furthermore, we have to know how to type the newly generated nodes. Altogether, graph transformations have the form:

```
record ('nt, 'et) graphtrans =
  — applicability condition
  appcond :: ('nt, 'et) path-form
  — mapping of nodes
  ndel  :: nat set   — deleted nodes
  ngen  :: nat set   — generated nodes
  — mapping of edges
  edel  :: 'et ⇒ (nat * nat) set   — deleted edges, indexed by type
  egen  :: 'et ⇒ (nat * nat) set   — generated edges, indexed by type
  — typing of generated nodes
  ngentp :: nat ⇒ 'nt option
```

For example, the marking rule of Section 2 can now be expressed by the transformation:

```
mark :: (nodetp, edgetp) graphtrans
mark ==
(| appcond = mark-F 0,
  ndel = {},
  ngen = {1},
  edel = λ et. {},
  egen = (λ et. {})(Or:={(1,0)}),
  ngentp = [1 ↦ Orig]
|)
```

Here, *mark-F* is the coding of the application condition. The application position of the rule is node 0. No nodes and edges are deleted, a node numbered 1 is generated and an `Or` edge is added between node 1 and 0. (The syntax for update of function f at x with value y is $f(x{:=}y)$.)

For graph transformations to make sense, the references to nodes to be deleted have to be among the references to nodes in the applicability condition (thus, to the free variables of the applicability condition), whereas references to generated nodes should not occur in the applicability condition. We only generate a finite number of nodes in each transformation step, and to all of these nodes we assign a type. Similar constraints hold for deleted and generated edges. To summarise, structural well-formedness of a graph transformation is expressed by the following predicate:

```
struct-wf-gt :: ('nt, 'et) graphtrans ⇒ bool
struct-wf-gt gt ==
  (ndel gt) ⊆ (fv-path-form (appcond gt)) ∧
  finite (ngen gt) ∧ (fv-path-form (appcond gt)) ∩ (ngen gt) = {} ∧
  dom (ngentp gt) = (ngen gt) ∧
  (∀ et. Field (edel gt et)  ⊆  (fv-path-form (appcond gt))) ∧
  (∀ et. Field (egen gt et)  ⊆  ((fv-path-form (appcond gt)) − (ndel gt)) ∪ (ngen gt))
```

3.4 Applying Graph Transformations

We now come to the application of a graph transformation to a source graph at a particular position. In graph rewriting, matching a pattern graph to a source graph (and thus determining the application position) is traditionally achieved with the aid of a graph morphism. We adopt the same terminology and define

types *graphmorph* = (*nat* ⇒ *nat option*)

with the understanding that the node references occurring in a graph transformation rule are mapped to the nodes in a source graph. For the "garbage collection" example, such a situation is depicted in Figure 2.

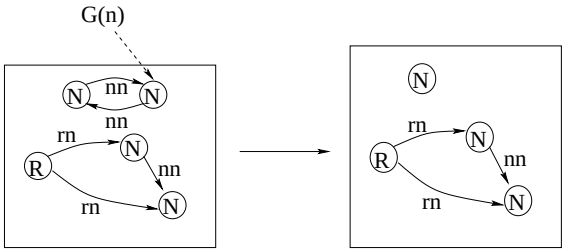


Fig. 2. Application of a graph morphism in a graph

We now have to spell out in detail how the target graph is composed, provided we apply a graph transformation *gt* to a graph *gr* using a morphism *gm*. Quite simply, the nodes to be deleted are just the ones in the image of the morphism under the *ndel*-set.

It is more difficult to express which nodes are generated. The choice could be, non-deterministically, any node set having the same cardinality as the *ngen*-set and having no nodes in common with the nodes of the source graph. We have adopted a deterministic solution: The nodes freshly allocated are numbered $m+1$ through $m+k$, where m is the maximal number present in the node set of graph *gr* and k is the cardinality of the *ngen*-set. All this is hidden in the definition of *gt-gen-nodes*. However, we only exploit the property that the fresh nodes do not occur in the original graph, and that there is a bijection b between the ngen-nodes and the fresh nodes.

The latter property is needed for determining the type of the generated nodes. How do we compute it, for a fresh node n? We map n back into the graph transformation *gt*, where we can look up its type. Thus, roughly, the type of n is $(ngentp\ gt)(b^{-1}(n))$.

The morphism on nodes induces a morphism on edges. From the *edel*- and *egen*-sets, we can thus determine the edges in the source graph which are candidates for deletion and for insertion. We want to avoid dangling edges that result when nodes are requested to be deleted, but not their adjacent edges. Therefore, the edges that survive are those whose nodes are among the nodes of the target graph. A similar restriction applies to the typing of the target nodes.

With these explanations, the exact definition should be understandable:

```
apply-graphtrans ::
    [('nt, 'et) graphtrans, graphmorph, ('nt, 'et) graph] ⇒ ('nt, 'et) graph
apply-graphtrans gt gm gr ==
  let del-nodes = ran (gm |` (ndel gt)) in
```

```
let gen-nodes = gt-gen-nodes gr gt in
let morph-gen = separ-map (ngen gt) (nodes gr) in
let morph-c = gm ++ morph-gen in
let nds = ((nodes gr) − del-nodes) ∪ gen-nodes in
let del-edges = (λ et. (induced-emorph gm) ‘ (edel gt et)) in
let gen-edges = (λ et. (induced-emorph morph-c) ‘ (egen gt et)) in
let tp-ngen = ((ngentp gt) ∘m (inv-m morph-gen)) in
(| nodes = nds,
  edges = λ et. (restrict-rel ((edges gr et − del-edges et) ∪ gen-edges et) nds),
  nodetp = (restrict-map ((nodetp gr) ++ tp-ngen) nds)
|)
```

In the above, f ‘ S is the image of set S under function f, and m |‘ S restricts map m to S. In *m1* ++ *m2*, map *m2* overrides *m1*, and $\circ_m$ is the composition of maps.

3.5 Applicability of Graph Transformations

What we have called “graph morphisms” in Section 3.4 is essential for determining whether a transformation is applicable, and if yes, where to apply it. It should be emphasised again that “graph morphism” is a slight misnomer, because we do not map graphs into graphs, as in traditional graph rewriting. Rather, we want to verify that the applicability condition of a transformation rule is true.

The following predicate states that a graph morphism *gm* satisfies a path formula *pfs* in a graph *grt*:

```
applicable-gm :: [graphmorph, ('nt, 'et) path-form, ('nt, 'et) graph] ⇒ bool
applicable-gm gm pfs grt ==
  (dom gm = fv-path-form pfs) ∧ (ran gm ⊆ nodes grt) ∧
   path-form-interp (the o gm) grt pfs
```

The domain of the graph morphism has to be the set of free variables of the path formula, and its range has to be a subset of the nodes of the graph. Most importantly, the path formula has to be satisfied in the graph when interpreting its free variables by the graph morphism in the given graph. (*the* is the left inverse of *Some*, thus *the* (*Some* x) = x).

In most of our reasoning, we want to abstract away from particular graph morphisms and just say that a transformation is applicable in a graph:

```
applicable-transfo :: [('nt, 'et) graphtrans, ('nt, 'et) graph] ⇒ bool
applicable-transfo gt gr == ∃ gm. applicable-gm gm (appcond gt) gr
```

Now, applying a graph transformation to a graph amounts to selecting an arbitrary graph morphism and applying it to the graph:

```
apply-transfo :: [('nt, 'et) graphtrans, ('nt, 'et) graph] ⇒ ('nt, 'et) graph
apply-transfo gt gr ==
  apply-graphtrans gt (SOME gm. (applicable-gm gm (appcond gt) gr)) gr
```

Here, *SOME* is Hilbert’s choice operator which could be replaced by a constructive choice based, for example, on a node ordering.

3.6 Properties of Graph Transformations

We can now state a major result: application of well-formed graph transformations to well-formed graphs yields again well-formed graphs:

```
struct-wf-gr gr ∧ struct-wf-gt gt ⟶ struct-wf-gr (apply-graphtrans gt gm gr)
```

This can be construed as a generic invariant of graph transformations that need not be reproved for each transformation rule when reasoning about graph transfor-

mation programs (see Section 5). Note that the structural well-formedness of the resulting graph depends on the well-formedness of the graph transformation gt, but is valid for arbitrary graph morphisms gm.

In [SG06], we have shown that for traditional graph rewriting, we can similarly ensure preservation of well-typing. In our current setting, we can express more general typing properties than those examined in [SG06], for example cardinality constraints, so that "typing" in full generality becomes undecidable. We are currently exploring fragments of our path logic that permit sufficiently interesting typing properties to expressed and preservation of typing to be proved.

4 Correspondence with Graph Rewriting

In the following, we will argue that transformations expressible in traditional graph rewriting approaches can be coded in our system. It is therefore possible to "compile" traditional graph rewriting rules to expressions involving our path formulae. It is then possible to use the techniques described in Section 5 as a verification backend.

In the rules of the AGG system [Tae03], for example, there are positive and negative applicability conditions, and each such condition is a graph that has to occur, respectively must not occur, in the graph where the rule is applied. As seen in Section 2, we can code positively occurring graphs by a conjunction of node set and path constraints, more precisely

- a node set constraint $T(n)$ for every node n of type T in the graph
- a path constraint $n \xrightarrow{e} n'$ for each edge e in the graph.

As mentioned before, we do not allow multiple edges of the same edge type between a pair of nodes. We do not see that as a major drawback – if necessary, edges can be "reified" by introducing a node representing the edge.

For negative applicability conditions, we proceed in an analogous manner, with the difference that the nodes of the graph are asserted not to exist. Thus, for an edge e occurring in a negative applicability graph, we have a path formula $\neg\exists n\ n'.n \xrightarrow{e} n'$.

The GREAT language [AKK+05] includes, among others, cardinality constraints. It is thus possible to specify that a node n must (or must not) have k outgoing e-edges. Cardinality constraints are not present as primitive constructs in our language, but they can be coded by a schema like

$$C_k(n) \equiv \exists x_1 \ldots x_k.\ \ n \xrightarrow{e} x_1 \wedge \ldots n \xrightarrow{e} x_k \wedge distinct(x_1, \ldots x_k)$$

where $distinct(x_1, \ldots x_k)$ is the conjunction $\neg(x_i = x_j)$, for $i, j \in \{1, \ldots, k\}, i \neq j$.

The fact that the graph morphisms between a pattern and a source graph is injective is usually an external notion in traditional graph rewriting. In a similar spirit as the above formula, we can internalise this notion and express that the nodes a rule is applied to are distinct.

5 Reasoning about Graph Transformations

As mentioned in Section 2, it is not sufficient to apply a transformation rule once. Rather, one has to apply a rule repeatedly, or several rules have to be applied in a specific order. Most graph rewriting tools permit to iterate rule application, often by dividing the tool set into "layers". The need for exerting finer control on graph transformations has been recognised, among others, by the developers of the GReAT language, who develop a graphical language including conditional and loop constructs [AKK+05].

We are currently developing a simple language for writing graph transformation programs and reasoning about them. The language is not sufficiently polished to present details, so we just give a sketch and describe how we might treat the "marking" example of Section 2.

The language is composed of statements *stmt*, among which we only mention *Do* and *Loop*. An operational semantics describes how a state is modified by these constructs. We distinguish between success and failure states. In our case, a "state" is just a graph with a "success" or "failure" tag. The meaning of the mentioned constructs is then:

- *Do b f* checks whether condition b is satisfied in the current state s. If this is the case, function f is applied to s to produce a success state s'. Otherwise, s is returned as a failure state.
- *Loop c* applies statement c indefinitely often, until winding up in a failure state, which is the result of the loop.

Let us introduce the following abbreviation:

```
App :: ('nt, 'et) graphtrans ⇒ ('nt, 'et) graph stmt
App gt == Do (λ s. applicable-transfo gt (outcome-val s))
             (λ s. apply-transfo gt (outcome-val s))
```

Here, *outcome-val* discards the success / failure tag of a state. Consequently, *App* applies a graph transformation, if possible, and returns the current state as failure state otherwise.

The marking phase of the introductory example can now be written as the program *Loop* (*App mark*), where we use the definition *mark* of Section 3.3. The entire graph duplication transformation consists of a sequence of such loops, each with a different rule.

The language comes equipped with a Hoare-style program logic. We write $W \vdash \{P\}\ c\ \{Q\}$ to express that statement c establishes the postcondition Q provided the precondition P and some invariant well-formedness conditions W hold. W is typically the predicate *struct-wf-gr* that we have shown to be invariant under application of graph transformations in Section 3.6. Furthermore, the statement c usually contains annotations corresponding to loop invariants.

Suppose we want to show, for our example program, that all nodes of type `Node` are correctly marked, i.e. have exactly one incoming `Or` edge, provided that in the outset, these nodes had zero or one incoming `Or` edges. Let us first define *nset* as the set of nodes in a graph having a given node type:

```
nset :: [('nt, 'et) graph, 'nt] ⇒ nat set
nset gr nt == {n ∈ nodes gr. (nodetp gr n) = Some nt}
```

We can now state the precondition:

$$\forall x{\in}nset\ gr\ Node.\ \ card\ ((edges\ gr\ Or)^{-1}\ ``\ \{x\}) \leq 1$$

(here, $R\ ``\ S$ is the image of a set S under a relation R, and *card* the cardinality of a set). The postcondition is similar, with the inequality replaced by an equality.

The verification condition generator leaves us essentially with two goals: showing that the loop invariant is preserved if the rule *mark* is applicable, and showing that the postcondition is satisfied if the rule is not applicable. We just look at the latter case.

So assume that $\neg$ *applicable-transfo mark gr*. According to the definition of *applicable-transfo*, this is equivalent to $\forall\, gm.\ \neg$ *applicable-gm gm* (*appcond mark*) *gr*, which contains an annoying second-order quantifier over a graph morphism *gm*.

However, when looking at the definition of *applicable-gm*, we realise that the domain of *gm* is finite - it is just the set of free variables of the application condition of *mark*. We now apply repeatedly the following lemma:

lemma *dom-reduce-insert*:
$(dom\ gm' = insert\ a\ A) =$
$(\exists\ \ b\ gm''.\ gm' = gm''(a{\mapsto}b) \wedge gm'\ a = Some\ b \wedge dom\ gm'' = A)$

which gradually reduces the domain of the morphism gm' and instead introduces a first-order quantifier b, so that we are eventually left with the hypothesis

$$\forall n.\ n \in nodes\ gr \longrightarrow nodetp\ gr\ n = Some\ Node \longrightarrow (\exists x.\ nodetp\ gr\ x = Some\ Orig \wedge (x, n) \in edges\ gr\ Or)$$

which naturally describes the non-applicability of the rule and eventually permits to prove the required cardinality property.

6 Conclusions

In this paper, we have presented first steps towards the verification, in an interactive proof assistant, of structural properties established by graph rewriting systems. At the same time, the path formulae we have introduced give an alternative view on applicability conditions for graph rewriting rules, that may profitably be used in graph rewriting systems.

Our path formulae are very expressive, which has the downside of leading, in general, to undecidable verification problems. As we want to reduce the amount of human proof effort as much as possible, we intend to address this topic in future work, by developing specialized analyses for fragments of our logic. In fact, our path formulae resemble path expressions used in shape analysis for pointer programs [YRS+06,KS93], other subsets have been identified in the context of description logics [GM05]. A detailed comparison of these approaches still has to be done.

Acknowledgement

This work has been strongly influenced by suggestions from Jean-Paul Bodeveix and Mamoun Filali and discussions with Louis Féraud, Ralph Matthes, Marc Pantel, Maxime Rebout and Sergei Soloviev. Mathieu Giorgino has elaborated several example transformations.

References

[Agr04] Aditya Agrawal. *A Formal Graph-Transformation Based Language for Model-to-Model Transformations*. PhD thesis, Vanderbilt University, August 2004.

[AKK+05] A. Agrawal, G. Karsai, Z. Kalmar, S. Neema, F. Shi, and A. Vizhanyo. The design of a language for model transformations. *Journal of Software and System Modeling*, 2005.

[Bar03] Erik Barendsen. *Term Rewriting Systems*, chapter Term Graph Rewriting. Cambridge University Press, 2003.

[BBDV03] Jean Bézivin, Erwan Breton, Grégoire Dupé, and Patrick Valduriez. The ATL Transformation-based Model Management Framwork. Technical report, IRIN, September 2003.

[BCE+05] Paolo Baldan, Andrea Corradini, Javier Esparza, Tobias Heindel, Barbara König, and Vitali Kozioura. Verifying red-black trees. In *Proc. of COSMICAH '05*, 2005. Proceedings available as report RR-05-04 (Queen Mary, University of London).

[CMR+96] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Loewe. Algebraic approaches to graph transformation, part I: Basic concepts and double pushout approach. Technical Report TR-96-17, Dipartimento di Informatica, March 21 1996.

[Cou90] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 193–242. Elsevier, 1990.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392, 1972.

[EHK+97] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation - part II: Single pushout approach and comparison with double pushout approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 247–312. World Scientific, 1997.

[FM97] P. Fradet and D. Le Métayer. Shape types. In *Proc. of Principles of Programming Languages*, Paris, France, Jan. 1997. ACM Press.

[GM05] Lilia Georgieva and Patrick Maier. Description logics for shape analysis. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 321–330, Koblenz, Germany, September 2005. IEEE Computer Society, IEEE.

[KS93] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *POPL*, pages 196–205, 1993.

[KS06] A. Königs and A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. In R. Heckel, editor, *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 113–150, Amsterdam, 2006. Elsevier Science Publ.

[MFV+05] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *Proc. Model Transformations In Practice Workshop*, 2005.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer Verlag, 2002.

[Plu99] Detlef Plump. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools, chapter Term Graph Rewriting. World Scientific, 1999.

[RD06] Arend Rensink and Dino Distefano. Abstract graph transformation. *Electr. Notes Theor. Comput. Sci*, 157(1):39–59, 2006.

[Ren03] Arend Rensink. Towards model checking graph grammars. In *Proc. Workshop on Automated Verification of Critical Systems (AVoCS)*, 2003.

[SG06] Martin Strecker and Mathieu Giorgino. Towards a formalisation of graph transformations in proof assistants. In *Proc. AVOCS'06*, September 2006.

[Tae03] Gabriele Taentzer. AGG: A graph transformation environment for system modeling and validation. In *Proc. Tool Exihibition at Formal Methods 2003*, September 2003.

[Var04] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004.

[YRS+06] Greta Yorsh, Alexander Moshe Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. In Luca Aceto and Anna Ingólfsdóttir, editors, *FoSSaCS*, volume 3921 of *Lecture Notes in Computer Science*, pages 94–110. Springer, 2006.