Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 72 (2007) 45–58
www.elsevier.com/locate/entcs

A Higher-Order Calculus for Graph Transformation[1]

Maribel Fernández and Ian Mackie[2,3]

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

Jorge Sousa Pinto[4]

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

Abstract

This paper presents a formalism for defining higher-order systems based on the notion of graph transformation (by rewriting or interaction). The syntax is inspired by the Combinatory Reduction Systems of Klop. The rewrite rules can be used to define first-order systems, such as graph or term-graph rewriting systems, Lafont's interaction nets, the interaction systems of Asperti and Laneve, the non-deterministic nets of Alexiev, or a process calculus. They can also be used to specify higher-order systems such as hierarchical graphs and proof nets of Linear Logic, or to specify the operational semantics of graph-based languages.

Keywords: Higher-order system, graph transformation, Combinatory Reduction System, graph rewriting system

1 Introduction

Rule-based transformations of graphs have been used in many areas of computer science, including the specification and development of software systems, the definition of visual languages, the implementation of programming languages (see [5,25]). The notion of interaction, which can be seen as a particular kind of graph transformation, has been used to model concurrent systems [23], to give a semantics to (linear) logic proofs [11], as a programming discipline [17], and as an implementation technique for functional languages [3]. In each case, a syntax and an operational semantics (a calculus) has been defined, often independently.

[1] Partially supported by TMR LINEAR.

[2] Email: maribel@dcs.kcl.ac.uk

[3] Email: ian@dcs.kcl.ac.uk

[4] Email: jsp@di.uminho.pt

doi:10.1016/j.entcs.2002.09.005

In this paper we present a higher-order language that can serve to specify interaction systems as well as graph and term-graph rewriting systems. The syntax is inspired by the Combinatory Reduction Systems (CRSs) of Klop [16], and can be seen as a generalization of the equational notation for term-graph rewriting [2]. We demonstrate its use by giving several examples of application, including the definition of hierarchical graphs (where it is possible to abstract subgraphs, see [4] for more details), a first-order interaction language together with its operational semantics (all in the same language), and the specification of higher-order program transformations and optimization schemes. The latter will be defined for Lafont's interaction nets [17].

¿From a practical point of view, the higher-order syntax can be used as a tool in the design and implementation of graphical languages: it allows us to express not only graphical programs but also their operational semantics (including evaluation strategies and optimization schemes), type systems, and transformations used in the proof of meta-theoretical properties of programs. An instance of the latter kind of transformation is the *packing operator* defined by Lafont [19] to prove the universality of the interaction combinators (a specific system of interaction nets in which every interaction net can be encoded). Other packing and unpacking operations have been described in [9], and they can all be formally defined using higher-order rules in our system.

Another aspect where the higher-order syntax presents advantages is for structuring and modularizing programs defined by graphs (or nets). Hierarchical definitions are very useful in the framework of graph rewriting [4], and the same techniques can be exported to interaction nets using the higher-order syntax. In particular, the operation that combines two interaction nets to produce a new net where one or more edges have been connected together (the analogous of application in functional programming) is currently a meta-operation. We show how to internalize it using the higher-order language, and give examples where this technique is used to write modular programs. Once we have the ability to model the combination of nets, it is straightforward to express a notion of higher-order interaction nets, where a net depends on another net. As with functional programming, this technique can be used to write recursive nets: nets which depend on themselves.

Related Work. Our syntax is inspired by CRSs, but similar results can be obtained by using other higher-order systems, such as Nipkow's Higher-order Rewrite Systems [22], or Khasidashvili's Expression Reduction Systems [13]. The three formalisms are closely related [26]. CRSs have been used in previous work on interaction nets: Laneve [20] defined Interaction Systems as CRSs, and in [7] a translation function is given from interaction nets to CRSs.

Van Raamsdonk [27] defines a class of higher-order rewrite systems with a general notion of substitution and shows the encoding of several languages, including Interaction Systems and Proof Nets of linear logic. Our goal is more specific: our higher-order textual notation has been designed to represent graph-based transformations, and therefore the calculus contains specific graph-oriented features. The notation used to represent graphs in the calculus is a generalization of the equational

graph rewriting systems studied by Ariola, Klop and Blom (see for instance [2,15]).

Some of the higher-order features defined in this paper could also be defined in Generalized Interaction Nets [10], by defining an interaction language where agents can carry nets. However, the higher-order calculus gives a uniform language to define higher-order agents and higher-order rewrite rules.

Organization. Section 2 briefly reviews CRSs, graph and term-graph rewriting, and interaction nets. Section 3 introduces the syntax of our higher-order systems, and Section 4 shows how to represent graph and term-graph rewriting systems, interaction nets, interaction systems and non-deterministic nets. In Section 5 we then go on to define different sets of higher-order rules. Section 6 contains a simple example of application. We conclude the paper in Section 7.

2 Background

Combinatory Reduction Systems. CRSs [16] combine the usual first-order term rewriting systems with the presence of bound variables as in the λ-calculus. We recall the basic definitions.

Metaterms over an alphabet Σ are defined by the grammar:

$$t ::= x \mid [x]t \mid f(t_1, \ldots, t_n) \mid Z^n(t_1, \ldots, t_n)$$

where x denotes a variable, $f \in \Sigma$ is a function symbol of arity n, Z^n is a metavariable and the binary operator $[\cdot]\cdot$ denotes abstraction. *Terms* are metaterms that do not contain metavariables.

A *rewrite rule* is a pair $l \to r$ of closed metaterms, where l has the form $f(s_1, \ldots, s_n)$, the metavariables that occur in r occur also in l, and the metavariables Z_i^k that occur in l occur only in the form $Z_i^k(x_1, \ldots, x_k)$, where $x_1, \ldots, x_k$ are pairwise distinct variables.

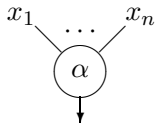
The metavariables in metaterms can be thought of as holes that must be instantiated by terms. In other words, rules act as schemes defining a reduction relation on terms. Formally, to define the rewrite relation we have to consider a notion of substitution using *substitutes* and *valuations*.

An *n-ary substitute* is an expression of the form $\lambda x_1 \ldots x_n.t$, where t is a term and $x_1, \ldots, x_n$ are different variables ($n \geq 0$). It can be applied to an n-tuple $s_1, \ldots, s_n$ of terms, and the result is the term t where $x_1, \ldots, x_n$ are simultaneously replaced by $s_1, \ldots, s_n$. A *valuation* σ is a map that assigns an n-ary substitute to each n-ary metavariable. This is extended to a mapping from metaterms to terms: given a valuation σ and a metaterm t, first we replace all metavariables in t by their images in σ and then we perform the developments of the β-redexes created by this replacement. When making a substitution, we must take care of bound variables as usual.

A *rewrite step* is defined as follows: if $l \to r$ is a rewrite rule, σ a valuation, and $C[\,]$ the usual notion of a context, then $C[l\sigma] \to C[r\sigma]$.

Interaction Nets. This is a graphical rewriting framework for programming introduced in [17]. Let Σ be a set of *agents*, each with a fixed arity which is the number of

its *auxiliary ports*, and one *principal port* (depicted by an arrow) where interaction can take place.



A *net* is an undirected graph whose vertices are agents in Σ, and whose edges join different ports in the same or in different agents. A net may be empty, or consist just of edges without agents. Ports that are not connected to other ports in the net are called *free*, and marked with edges that have a free extreme. The *interface* of a net is the (ordered) set of free extremes of edges.

Interaction rules are net rewriting rules where the left-hand side consists of two agents connected on their principal ports (this is called an *active pair*, written $\alpha \bowtie \beta$), and the right-hand side is an arbitrary net with the only constraint that it must have the same interface as the left-hand side. There is at most one rule for each pair of agents. As an example, in Fig. 1 we show Lafont's interaction combinators [19].

An interaction step on a net W replaces an active pair (i.e. the occurrence of a left-hand side of an interaction rule) by the corresponding right-hand side, plugging the edges in the interface of the right-hand side to the corresponding ports in W. We write interactions as a binary relation $W \Longrightarrow W'$, $\Longrightarrow^*$ denotes its reflexive and transitive closure.

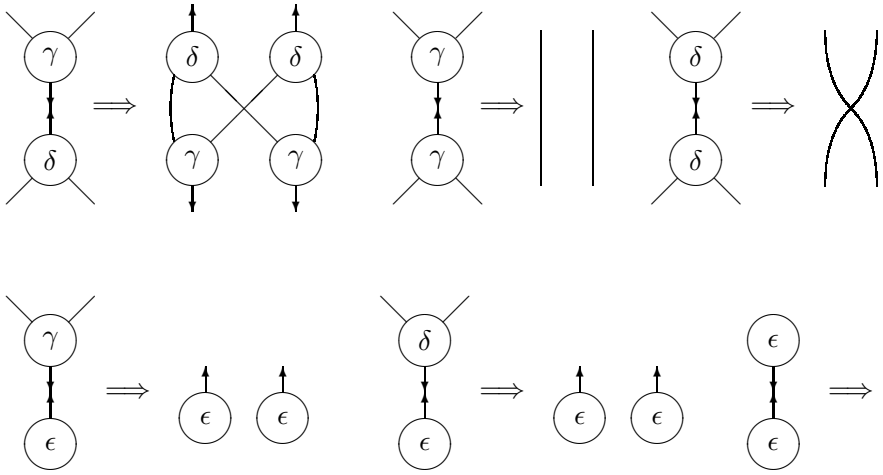


Fig. 1. Interaction Combinators

Graph and Term-Graph Rewriting. We recall the standard definition of graph-rewriting using edge-labelled hypergraphs. Let L be a *label alphabet* where each label has a fixed arity. A *hypergraph* H over L consists of a finite set V_H of nodes, a finite set E_H of hyperedges, a labelling function $lab_H : E_H \to L$ and an attachment function $att_H : E_H \to V_H^*$ such that for each hyperedge e, $|att_H(e)| = arity(lab_H(e))$. We assume that V_H and E_H are disjoint. A sequence of nodes, called *points*, may be designated as the interface of the hypergraph (where it may be glued with other

hypergraphs).

In the following we simply write graph and edge instead of hypergraph and hyperedge. Due to space limitations, we give an informal definition, taken from [12], of a simple kind of graph transformation.

A *graph transformation rule* $P \to R$ consists of a *pattern* P and a *replacement graph* R. A transformation step $G \Longrightarrow H$ using the rule $P \to R$ is defined as follows:

- Find a subgraph P' of G that is a copy of P (i.e., that matches the left-hand side of the rule);
- Check that every node in P' that is linked to an edge outside P' corresponds to a point of P (this is called the *no dangling links condition*);
- Remove P' from G up to its points, to obtain a context graph C,
- Glue a copy R' of R to C by identifying the points of P' with the corresponding points of R', obtaining H.

The relation $\Longrightarrow^*$ is the reflexive-transitive closure of $\Longrightarrow$.

A hierarchical graph is a graph where some edges contain hierarchical graphs. Formally, the class $\mathcal{H} = \bigcup_{i \geq 0} \mathcal{H}_i$ of *hierarchical graphs* consists of triples $H = (G, F, cts)$ such that G is a graph, $F \subseteq E_G$ is the set of *frame edges*, and $cts : F \to \mathcal{H}$ assigns to each frame $f \in F$ its contents $cts(f) \in \mathcal{H}$. The sets $\mathcal{H}_i$ are defined inductively as follows: $H = (G, F, cts) \in \mathcal{H}_0$ if $F = \emptyset$, and for $i > 0$, $H \in \mathcal{H}_i$ if $cts(f) \in \mathcal{H}_{i-1}$ for every $f \in F$.

The definition of graph transformation generalizes to hierarchical graphs. We refer to [12,4] for more details.

Term-graphs are particular graphs which can be seen as trees with shared subtrees. In the definition we use the notion of result (see [24]): v_0 is the *result* node of the edge e in a graph G if $att_G(e) = v_0 \dots v_n$. Let Σ be a set of function symbols with fixed arities and $\mathcal{X}$ an infinite set of variables (symbols of arity 0). A graph G over Σ is a *term-graph* if there is a node $root_G$ from which each node is reachable, G is acyclic, and each node is the result of a unique edge.

3 A Higher-Order Calculus for Graph Transformations

The syntax of our higher-order systems is inspired by CRSs. We first define an appropriate language, and then show examples of application.

Terms and Metaterms. Our language will be many-sorted. For each sort we will have a different set of meta-variables as shown:

- *Iterm* for *terms*, with metavariables $\{T_n^{\vec{A}} \mid n \geq 0\}$
- *Eq* for *equations* (pairs of terms), with metavariables $\{E_n^{\vec{A}} \mid n \geq 0\}$
- *TList* for lists of terms, with metavariables $\{I_n^{\vec{A}} \mid n \geq 0\}$
- *EqList* for lists of equations, with metavariables $\{\Delta_n^{\vec{A}} \mid n \geq 0\}$
- *Config* for configurations (representing nets or graphs), with metavariables $\{C_n^{\vec{A}} \mid$

$n \geq 0\}$

The set of sorts may contain other (user-defined) sorts. For each metavariable, say $T_n^{\vec{A}}$, $\vec{A} = A_1 \times \ldots \times A_k$ (with the A_i sorts and $k \geq 0$) is its signature, which can be determined by examining the metaterm where it occurs. Thus, as is usual for CRSs, we will omit these and furthermore $T_0, T_1, T_2, \ldots$ will often be written $T, T', T'', \ldots$ The alphabet also contains a set $\mathcal{F}$ of *function symbols* including:

- A set Σ of *agents* with fixed arities. Each agent α with arity n has the signature
$$\underbrace{\mathit{Iterm} \times \ldots \times \mathit{Iterm}}_{n} \to \mathit{Iterm}$$
- $\cdot = \cdot$ with signature $\mathit{Iterm} \times \mathit{Iterm} \to \mathit{Eq}$
- $\langle \cdot \mid \cdot \rangle$ with signature $\mathit{TList} \times \mathit{EqList} \to \mathit{Config}$ (to represent nets)
- $\{\cdot \mid \cdot\}$ with signature $\mathit{TList} \times \mathit{TList} \to \mathit{Config}$ (to represent graphs)
- tl with signature $\mathit{Iterm} \to \mathit{TList}$
- el with signature $\mathit{Eq} \to \mathit{EqList}$
- $\cdot \star \cdot$ with signature $\mathit{TList} \times \mathit{TList} \to \mathit{TList}$
- $\cdot * \cdot$ with signature $\mathit{EqList} \times \mathit{EqList} \to \mathit{EqList}$
- η with signature $\mathit{Config} \to \mathit{Config}$
- comb with signature $\mathit{Config} \times \mathit{Config} \to \mathit{Config}$

Finally, our language contains also a set $\{x, y, z, \cdots\}$ of *variables*, the *abstraction* binary operator $[\_]\_$, and the symbols '(', ')', and ','.

Definition 3.1 *Metaterms* are formed as follows, and then a *term* is a metaterm with no occurrences of metavariables.

- a variable is a metaterm of sort *Iterm*
- if t is a metaterm of sort s and x is a variable, then $[x]t$ is a metaterm of sort s
- if $t_1, \ldots, t_n$ are metaterms of sorts $A_1, \ldots, A_n$ respectively, and F is a function symbol with signature $A_1 \times \ldots \times A_n \to B$, then $F(t_1, \ldots, t_n)$ is a metaterm of sort B.
- if $t_1, \ldots, t_k$ are metaterms of sorts $A_1, \ldots, A_k$ respectively, and $M_n^{\vec{A}}$ is a metavariable of sort B, with $\vec{A} = A_1 \times \ldots \times A_k$ (and $k \geq 0$), then $M_n^{\vec{A}}(t_1, \ldots, t_k)$ is a metaterm of sort B.

A list of interaction terms is of the form $(\ldots (tl(t_1) \star tl(t_2)) \star \ldots \star tl(t_n))$ but we will simply write it as $t_1, t_2, \ldots, t_n$. A list with a single term $tl(t)$ will be written as t. The same conventions apply to lists of equations, formed with the $\cdot * \cdot$ operator.

We remark that reduction will be defined modulo associativity and commutativity of $\cdot \star \cdot$ and $\cdot * \cdot$, and also modulo commutativity of $\cdot = \cdot$. This justifies the previous notational conventions, and will give us the desired meaning for multisets of terms (used in the interface of configurations) and for multisets of equations.

Notation 3.2 *$[x_1][x_2]\ldots[x_n]C$ will be written as $[x_1, x_2, \ldots, x_n]C$, and we write a*

term like $\eta[x_1, x_2, \ldots, x_n]C$ simply as $\eta x_1 \ldots x_n.C$. We will also use the notation $\vec{x}$ for a list of variables $x_1, x_2, \ldots, x_n$; $\vec{x}\vec{y}$ denotes concatenation of lists $\vec{x}$ and $\vec{y}$. We will in particular abstract variables in configurations, to represent internal links.

Rewrite Rules. Metaterms are used as the left- and right-hand side of rewrite rules. The metavariables contained in them indicate places where terms can be substituted, as in CRSs.

Definition 3.3 A rewrite rule is a pair $l \to r$ of closed metaterms such that the metavariables in r occur also in l.

Note that left-hand sides of rules may contain patterns of the form $Z(t)$ where Z is a meta-variable and t a term. This will allow us to write contextual rules (such as the optimization rules in Section 5). The Context-sensitive Reduction Systems defined in [14] allow this use of metavariables.

The definition of rewrite step given for CRSs still applies in this generalized setting, but since we have some associative and commutative symbols in the alphabet we need to use pattern-matching modulo. More precisely: if $l \to r$ is a rewrite rule, σ a valuation, $C[\,]$ a context, and $t = l\sigma$ modulo associativity and commutativity of $\cdot \star \cdot$ and $\cdot * \cdot$, and modulo commutativity of $\cdot = \cdot$, then $C[t] \to C[r\sigma]$.

4 Representing First-Order Systems

Here we show that graph and term-graph rewriting systems, interaction nets and other first-order systems of interaction can be represented in our language.

Graph Rewriting. A hypergraph $H = (V_H, E_H, lab_H, att_H)$ will be represented by a term of the form $\eta\vec{x}.\{s \mid t\}$ of sort *Config* (a configuration for short), where s represents the points (interface) of the graph, and t is a list containing a term $\alpha(x_1, \ldots, x_n)$ for each hyperedge $e \in E_H$ such that $lab_H(e) = \alpha$, and $att_H(e) = x_1, \ldots, x_n$. In other words, nodes are represented by variables and edges by agents (named as the label). The purpose of the binder η at the head of the configuration is to hide all the nodes x_i that are not in the interface.

The representation of graph transformation rules is straightforward: Let $P \to R$ be a rule, where the pattern graph P is represented by the configuration $\eta\vec{x}.\{\vec{y} \mid l\}$ and the replacement graph R is represented by the configuration $\eta\vec{x}.\{\vec{y'} \mid r\}$. We assume that $\vec{y}$, $\vec{y'}$ are lists of variables, defining a mapping between the points of P and R; the mapping is not necessarily injective since some points of P might be identified in R. Then in the calculus we write a rule:

$$\eta\vec{z}\vec{x}.\{I(\vec{Y}) \mid l^*, Z(\vec{Y}, \vec{z})\} \to \eta\vec{z'}.\{I(\vec{Y'}) \mid r^*, Z(\vec{Y'}, \vec{z'})\}$$

where the metavariables in $\vec{Y'}$ occur in $\vec{Y}$, l^* and r^* are obtained from l and r by replacing each free variable y by a metavariable Y (this is necessary because rules are pairs of closed metaterms), $\vec{x}$ contains the internal nodes of the pattern l, $\vec{z}$ contains the internal nodes of the rest of the graph, and $\vec{z'}$ contains the internal nodes of r and of the rest of the graph. Note that the condition *no dangling links* can be easily checked: $\vec{x}$ must be different from $\vec{z}$.

Term-Graph Rewriting. Since term-graphs are particular cases of graphs, we could use the previous representation. But we could also give a more direct encoding, inspired by the equational encoding of term-graphs (see for instance [15]): A term-graph $G = (V_G, E_G, lab_G, att_G)$ with root $root_G$ will be represented by a configuration of the form $\eta\vec{x}.\langle root_G \mid t\rangle$ of sort *Config* where t contains an equation $x = f(x_1, \ldots, x_n)$ for each edge $e \in E_G$ such that $lab_G(e) = f$ and $att_G(e) = x, x_1, \ldots, x_n$, and $\vec{x}$ contains all the variables that occur twice in t (i.e. the internal nodes).

Interaction Nets. An interaction net will be represented by a term $\eta\vec{x}.\langle s \mid t\rangle$ of sort *Config* (a configuration for short). To obtain the configuration representing a given net, we proceed as in [8]: A term $\alpha(t_1, \ldots, t_n)$ of sort *Iterm* built out of agents in Σ and variables represents a tree, with the free principal port of α at the root and all the principal ports of the agents in $t_1, \ldots, t_n$ facing in the same direction. To represent active pairs (a connection between two principal ports) we use equations. Therefore, any interaction net can be represented by a list t of equations (a term of sort *EqList*) and a list s of free variables (the interface of the net). All the other variables, representing internal edges, are bound by η. A configuration $\eta\vec{x}.\langle s \mid t\rangle$ must satisfy three constraints: every variable occurs twice in $\langle s \mid t\rangle$; each variable occurs at most once in the interface; and all the variables that do not occur in the interface are explicitly bound by η.

Note that α-conversion applies to variables bound in configurations, both for nets and graphs. In nets these typically correspond to edges, whereas in hypergraphs they correspond to nodes. Free variables occurring in the interface can be used for structuring programs, as will be shown in Section 5.

Interaction Systems. Laneve [20] defines Interaction Systems as a subclass of CRSs representing intuitionistic interaction nets. The syntax defined above allows us to write the rules of an interaction system directly as rewrite rules.

Non-deterministic Interaction Nets and Process Calculi. Alexiev [1] defined a generalization of Lafont's nets in which agents can have multiple principal ports. A textual calculus for these nets was defined in [6], in which an agent α of arity n with m principal ports is represented by a term of the form $(l_1, \ldots, l_m)\alpha(t_1, \ldots, t_n)$. This can be transformed into a term in our system, for instance by defining a function symbol of arity $n + m$ associated to α. The (finitary) π-calculus can be encoded in Alexiev's system (using agents with multiple principal ports to simulate the non-deterministic communication between processes [1]) therefore we can give a system of rules defining the interaction and communication between concurrent processes.

5 Higher-Order Rewriting: Applications

Operational Semantics of Interaction Nets. The operational semantics of a program in an interaction net system $(\Sigma, \mathcal{R})$ is given by a set of computation rules on configurations in [8]. The computation rules can be specified in our calculus as higher-order rules:

Interaction: For each interaction rule in $\mathcal{R}$, we will have a rewrite rule in our system. The interaction rule for the agents α and β of arities n and m respectively will be written:

$$\eta\vec{x}.\langle I \mid \alpha(T_1(\vec{x}),\ldots,T_n(\vec{x})) = \beta(T'_1(\vec{x}),\ldots,T'_m(\vec{x})), \Delta(\vec{x})\rangle \longrightarrow$$
$$\eta\vec{x}\vec{x'}.\langle I \mid T_1(\vec{x}) = t_1,\ldots,T_n(\vec{x}) = t_n, T'_1(\vec{x}) = s_1,\ldots,T'_m(\vec{x}) = s_m, \Delta(\vec{x})\rangle$$

where $t_1,\ldots,t_n,s_1,\ldots,s_m$ encode the right-hand side of the graphical interaction rule for α and β and the bound variables in the vector $\vec{x'}$ represent edges in the right-hand side of the (graphical) interaction rule.

Indirection: $\eta\vec{x}z.\langle I \mid z = T(\vec{x}), \Delta(\vec{x},z)\rangle \longrightarrow \eta\vec{x}.\langle I \mid \Delta(\vec{x},T(\vec{x}))\rangle$

The Indirection rule is a "bureaucratic rule" in the sense that it does not correspond to any modification of the underlying (graphical) net.

As an example, we give the Interaction rewrite rule $\delta \bowtie \epsilon$ in Fig. 1:

$$\eta\vec{x}.\langle I \mid \delta(T(\vec{x}),T'(\vec{x})) = \epsilon, \Delta(\vec{x})\rangle \longrightarrow \eta\vec{x}.\langle I \mid T(\vec{x}) = \epsilon, T'(\vec{x}) = \epsilon, \Delta(\vec{x})\rangle$$

Specifying Strategies. Most functional language evaluators stop at weak head normal form. The corresponding notion for interaction nets is called interface normal form [8]. The idea is to reduce the net until all the free ports are either principal ports, or will never become principal ports. We can specify a lazy reduction strategy that computes interface normal forms by modifying the computation rules, so that equations are reduced only if their reduction can affect the terms associated to interface variables. For instance, Interaction will only be applied to active pairs that are directly connected to the interface of the net. This can be easily expressed by rewriting equations that contain a variable occurring in the interface of the configuration. An interaction is then specified in two steps: the first selects the equation and the second one performs the actual interaction.

$$\eta\vec{x}.\langle Z, I \mid E(Z,\vec{x}), \Delta(\vec{x})\rangle \longrightarrow \mathsf{inter}([\vec{x}](E(Z,\vec{x}),\langle Z, I \mid \Delta(\vec{x})\rangle))$$
$$\mathsf{inter}([\vec{x}](\alpha(T_1(\vec{x}),\ldots,T_n(\vec{x})) = \beta(T'_1(\vec{x}),\ldots,T'_m(\vec{x})),\langle Z, I \mid \Delta(\vec{x})\rangle)) \longrightarrow$$
$$\eta\vec{x}\vec{x'}.\langle Z, I \mid T_1(\vec{x}) = t_1,\ldots,T_n(\vec{x}) = t_n, T'_1(\vec{x}) = s_1,\ldots,T'_m(\vec{x}) = s_m, \Delta(\vec{x})\rangle$$

where Z is a metavariable of sort *Iterm* to be instantiated by a variable (in the interface of a configuration) in a rewrite step, and the terms $t_1,\ldots,t_n$, $s_1,\ldots,s_m$ represent the right hand side of the rule for $\alpha \bowtie \beta$ as before.

Indirection is only performed when the equations involved contain variables occurring in the interface. Again this condition can be expressed directly in the higher-order syntax:

$$\eta\vec{x}z.\langle Y, I \mid E(z,\vec{x},Y), z = T(\vec{x}), \Delta(\vec{x})\rangle \longrightarrow \eta\vec{x}.\langle Y, I \mid E(T(\vec{x}),\vec{x},Y), \Delta(\vec{x})\rangle$$
$$\eta\vec{x}z.\langle Y, I \mid E(z,\vec{x}), z = T(Y,\vec{x}), \Delta(\vec{x})\rangle \longrightarrow \eta\vec{x}.\langle Y, I \mid E(T(Y,\vec{x}),\vec{x}), \Delta(\vec{x})\rangle$$

Property 5.1 *The interaction net configurations that are irreducible in this system are interface normal forms.*

It is therefore easy for language-designers and compiler-writers to define and compare different strategies of evaluation using the calculus. In the case of interaction nets all that needs to be done is to change two rewrite rules.

Modularity and Dependence: Combining Nets. The comb function is used for building configurations modularly by composing two smaller configurations.

$$\mathsf{comb}(\eta\vec{x}.\langle Z_1,\ldots,Z_n,I \mid \Delta(Z_1,\ldots,Z_n,\vec{x})\rangle, \eta\vec{x'}.\langle Z_1,\ldots,Z_n,I' \mid \Delta'(Z_1,\ldots,Z_n,\vec{x'})\rangle) \longrightarrow \eta\vec{x}\vec{x'}z_1\ldots z_n.\langle I,I' \mid \Delta(z_1\ldots z_n,\vec{x}),\Delta'(z_1\ldots z_n,\vec{x'})\rangle$$

All the free variables of the same name occurring in the interface of both arguments of comb are pairwise connected together. Since these variables disappear from the interface, they must necessarily be bound in the resulting configuration (variables $z_1,\ldots,z_n$). Note that in this rule the variable convention is used to avoid name clashes: if the same variable is bound in both arguments of comb, α-conversion is used to change that variable in one of the terms before the rule is applied.

One of the main uses of combine that we foresee is as a programming tool for an interaction net programming language. We give an example of this in Section 6, where we show how we can write names for nets, and combine them together to build larger programs. This very same feature also allows us to express a notion of higher-order nets: nets depending on nets. Additional features, such as *rule templates*, can be seen as specific instances of this idea. For example the rules for $\gamma \bowtie \epsilon$ and $\delta \bowtie \epsilon$ shown in Fig. 1 are the same modulo the names of the agents. We can write them in a compact way:

$$\epsilon = X(T_1,T_2) \longrightarrow \epsilon = T_1, \epsilon = T_2$$

Net Transformation: Optimization Rules. In certain contexts (such as Asperti's safe operators for the optimal reduction of λ-terms [3], or that of garbage-collection of non-terminating or deadlocked nets) interaction net reduction can be greatly improved by admitting rules which fall outside the strict scope of interaction. For example, the following rules involve only two agents and the interface is preserved. The difference with respect to interaction rules is that in a redex an auxiliary port of an agent is connected to the principal port of the other.

Eq$^{\delta\epsilon}$: $\delta(\epsilon,T) \longrightarrow T$ and $\delta(T,\epsilon) \longrightarrow T$

Eq$^{\gamma\epsilon}$: $\Delta(\gamma(\epsilon,T)) \longrightarrow \Delta(\epsilon), T = \epsilon$

The first is an optimization rule: when an erasing agent is connected to an auxiliary port of a duplicator we can replace both agents by an edge. The second rule allows to garbage-collect non-terminating nets. Note that since Δ is a metavariable (we need to modify the context where $\gamma(\epsilon,t)$ occurs) this rule is not allowed in CRSs, but it could be replaced by a set of CRS's rules.

Other Meta-operations: Packing Nets. In different contexts it has been necessary to define global operations on nets. One typical problem is that of copying a net by using a duplicator agent δ. Since active pairs cannot be copied and the δ agent does not duplicate itself (see Fig. 1), some transformation is required to produce a *packed* net which can be copied (we refer the reader to [19,9] for details). These packing operations can be formally defined using rules in our system. As an example we show the δ-extraction operation used to prove the universality of the interaction combinators in [19]. It removes all occurrences of the δ agent, collects together

in three sequences their n principal ports and their left and right auxiliary ports, and uses three *multiplexing* nets of arity n to bundle these sequences into three edges, which are added to the interface of the packed net. Unpacking proceeds by connecting a δ agent to these three ports. We define $\mathsf{extract}_\delta$ and extr_δ with signature *Config* $\to$ *Config*:

δ-**Extraction**$_1$ (y is a fresh variable)

$$\mathsf{extract}_\delta(\eta\vec{x}.\langle I \mid \Delta(\delta(T'(\vec{x}), T''(\vec{x})), \vec{x})\rangle) \longrightarrow \mathsf{extr}_\delta(\eta\vec{x}y.\langle \mathsf{pp}(y), \mathsf{a}(T'(\vec{x})), \mathsf{b}(T''(\vec{x})), I \mid \Delta(y, \vec{x})\rangle)$$

δ-**Extraction**$_2$ (y is a fresh variable)

$$\mathsf{extr}_\delta(\eta\vec{x}.\langle \mathsf{pp}(T_\delta(\vec{x})), \mathsf{a}(T'_\delta(\vec{x})), \mathsf{b}(T''_\delta(\vec{x})), I \mid \Delta(\delta(T'(\vec{x}), T''(\vec{x})), \vec{x})\rangle) \longrightarrow \mathsf{extr}_\delta(\eta\vec{x}y.\langle \mathsf{pp}(\gamma(T_\delta(\vec{x}), y)), \mathsf{a}(\gamma(T'(\vec{x}), T'_\delta(\vec{x}))), \mathsf{b}(\gamma(T''(\vec{x}), T''_\delta(\vec{x}))), I \mid \Delta(y, \vec{x})\rangle)$$

δ-**Extraction**$_3$

$$\mathsf{extr}_\delta(\eta\vec{x}.\langle \mathsf{pp}(T_\delta(\vec{x})), \mathsf{a}(T'_\delta(\vec{x})), \mathsf{b}(T''_\delta(\vec{x})), I \mid \Delta(\vec{x})\rangle) \longrightarrow \eta\vec{x}.\langle x_\delta, x'_\delta, x''_\delta, I \mid \Delta(\vec{x}), x_\delta = T_\delta(\vec{x}), x'_\delta = T'_\delta(\vec{x}), x''_\delta = T''_\delta(\vec{x})\rangle$$

Rules δ-**Extraction**$_1$ and δ-**Extraction**$_2$ extract (step-by-step) occurrences of the δ agent from the list of equations Δ. The first rule builds multiplexing nets of arity one (edges), and the second rule uses γ agents to build bigger multiplexers.

The rule δ-**Extraction**$_3$ is only used when δ-**Extraction**$_2$ no longer applies (a strategy would force this). Its role is to remove the extr_δ operator and to introduce (free) names in the interface for the multiplexing nets. It is easy to see that the *unpacking net* consists of a single δ agent:

Property 5.2 *Let* $N_{u\delta}$ *be* $\eta\langle x_\delta, x'_\delta, x''_\delta \mid x_\delta = \delta(x'_\delta, x''_\delta)\rangle$. *For every interaction net configuration* c, $\mathsf{comb}(N_{u\delta}, \mathsf{extract}_\delta(c)) \longrightarrow^* c$.

Proof Nets of Linear Logic. The encodings of proof nets in interaction nets that can be found in the literature are of two kinds: either boxes (which are nets containing nets) are defined by agents which contain a proof net as label, therefore we need an infinite set of agents in the system (see for instance [18]), or a first-order encoding of binders is used to model the box and its contents, and this can be done with a finite number of agents (see for instance [21]). In the first case, the Dereliction Cut Elimination step is performed in one rewrite step, using an infinite rewrite system, whereas in the second case it is performed in several steps using a finite system.

Using a higher-order syntax we can model this Cut Elimination step with a single rule and a finite number of agents. We would write this as:

$$\mathsf{d}(Y) = \mathsf{box}(\eta\vec{x}.\langle Z, I \mid \Delta(\vec{x})\rangle, I) \longrightarrow \eta\vec{x}.\langle Y, I \mid \Delta(\vec{x}), Y = Z\rangle$$

where $\mathsf{d}, \mathsf{box} \in \Sigma$ and $\mathsf{d} : \mathit{Iterm} \to \mathit{Iterm}$, $\mathsf{box} : \mathit{Config} \times \mathit{TList} \to \mathit{Iterm}$.

Hierarchical Graph Rewriting Systems. We show the representation of a hierarchical graph $H = (G, F, cts)$ in the higher-order system. For this, it is sufficient to add to the alphabet a function symbol $f : \mathit{Config} \times \mathit{Iterm} \times \ldots \times \mathit{Iterm} \to \mathit{Iterm}$ to represent frames in F. More precisely, frames are represented by terms that have an argument of type *Config* carrying a graph. We write configurations as in the case of standard

graphs: each standard edge is represented as shown in Section 4 and each frame f such that $cts(f) = H'$ is represented by a term $f(\eta\vec{x}.\{\vec{y} \mid T(\vec{x})\}, z_1, \ldots, z_n)$ where $\eta\vec{x}.\{\vec{y} \mid T(\vec{x})\}$ is the configuration associated to H'. For some applications, we may require that the interface of the configuration H' coincide with $z_1, \ldots, z_n$ (the attachment nodes of the frame).

As an example, we give the specification of the flattening operation that transforms a hierarchical graph H into a standard graph $\mathsf{flat}(H)$, by gluing $cts_H(f)$ to $att_H(f)$ for each frame in H (recursively). To simplify the definition we assume that $att_H(f)$ coincides with the interface of $cts_H(f)$.

$$\mathsf{flat}(\eta\vec{x}.\{I \mid f(\eta\vec{z}.\{\vec{x} \mid T(\vec{x},\vec{z})\},\vec{x}), \Delta(\vec{x})\}) \longrightarrow \mathsf{flat}(\eta\vec{x}\vec{z}.\{I \mid T(\vec{x},\vec{z}), \Delta(\vec{x})\})$$

The rule eliminates one frame at the time, replacing it by its contents, until no more frames remain.

6 An Example

We show a simple system of interaction, and use the modularity features of the higher-order calculus to simplify the writing of nets. We can encode integers (not uniquely though) by a pair of natural numbers $z = (p, q)$ which we interpret as the difference $p - q$. This is encoded into interaction nets as shown in Fig. 2 (left). The

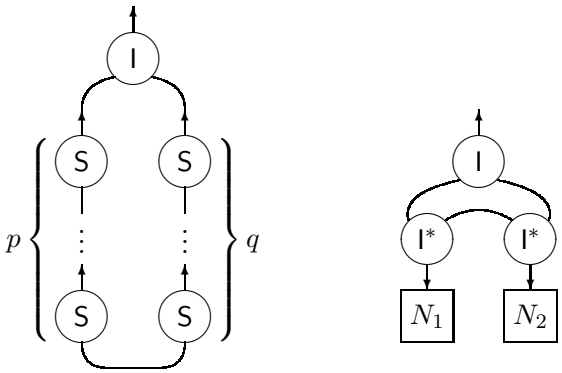
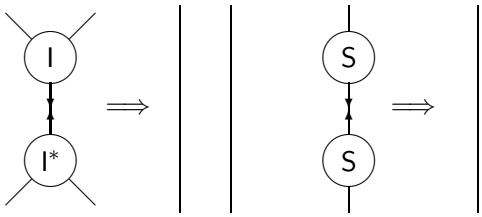


Fig. 2. Encoding integers (left) and addition (right)

agent S (of arity 1) is interpreted as *successor*. The representation of an integer simply takes two chains of length p and q, and connects them together as shown, using the agent I (of arity 2). Although this representation is not unique, we can talk about *canonical forms* when $p = 0$ or $q = 0$. If N_1 and N_2 are the net representations of $z_1 = (p_1 - q_1)$ and $z_2 = (p_2 - q_2)$ respectively, then we can use the configuration to encode addition shown in Fig. 2 (right). The following are the only rules of this

system, where the agent I^* is used to implement addition in constant time:



In our calculus, we can write this system in the following way:

$$\mathsf{I}(T_1, T_2) = \mathsf{I}^*(T_3, T_4) \longrightarrow T_1 = T_4, T_2 = T_3$$

$$\mathsf{S}(T_1) = \mathsf{S}(T_2) \longrightarrow T_1 = T_2$$

$$\mathsf{Add}(X, Y, Z) \longrightarrow \eta abc.\langle X, Y, Z \mid X = \mathsf{I}(a, b), Y = \mathsf{I}^*(b, c), Z = \mathsf{I}^*(c, a)\rangle$$

We can then define numbers as follows:

$$\mathsf{Two}(X) \longrightarrow \eta a.\langle X \mid X = \mathsf{I}(\mathsf{S}(\mathsf{S}(a)), a)\rangle$$

and use them to build a program in a modular way:

$$\mathsf{comb}(\mathsf{comb}(\mathsf{Add}(a, b, c), \mathsf{Two}(b)), \mathsf{Two}(c))$$

where a represents the result of the addition.

7 Conclusions

We have shown a higher-order rewrite framework which can express several systems of graph reduction. The power of the framework can be seen for the particular case of interaction nets, where we can write a program together with its evaluator, all in the same language. We see two main uses of this framework. First, as a tool for the design and implementation of graphical languages: the language, its semantics and metaoperators, can all be defined using the same language. Second, as a tool for adding structure to graphical programs: the higher-order features can be used to write hierarchical systems, and to name and reuse different components of the program.

Some further work that we foresee includes the development of a programming environment for interaction nets. Since interaction nets are also used as an implementation language for functional languages, this will allow for fast prototyping of functional compilers, facilitating the definition and comparison of strategies of evaluation and optimization techniques.

References

[1] V. Alexiev. *Non-deterministic Interaction Nets*. PhD thesis, University of Alberta, 1999.

[2] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3–4):207–240, 1996.

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

[4] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *Journal of Computer and System Science (to appear)*, 2002.

[5] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Applications, Languages and Tools*. World Scientific, Singapore, 1999.

[6] M. Fernández and L. Khalil. Interaction nets with McCarthy's amb. In *Proc. Express'02*, *Electronic Notes in Theoretical Computer Science*, 2002.

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

[8] M. Fernández and I. Mackie. A calculus for interaction nets. In *Proc. Int. Conf. Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 170–187. Springer, 1999.

[9] M. Fernández and I. Mackie. Packing interaction nets: Applications to linear logic and the lambda calculus. In *Proc. WAIT'01*, volume 30 of *Anales JAIIO*, pages 91–107, 2001.

[10] M. Fernández, I. Mackie, and J. S. Pinto. Combining interaction nets with externally defined programs. In *Electronic proceedings of the APPIA-GULP-PRODE Joint Conference on Declarative Programming*, Portugal, 2001.

[11] J.-Y. Girard. Towards a geometry of interaction. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 69–108. American Mathematical Society, 1989.

[12] B. Hoffmann. From graph transformation to rule-based programming with diagrams. In M. Nagl, A. Schrr, and M. Mnch, editors, *Applications of Graph Transformation with Industrial Relevance (AGTIVE'99) - Selected Papers*, pages 165–180. Springer, 2000. Lecture Notes in Computer Science Vol. 1779.

[13] Z. Khasidashvili. Expression reduction systems. In *Proceedings of I.Vekua Institute of Applied Mathematics*, volume 36, pages 200–220, Tbisili, 1990.

[14] Z. Khasidashvili and V. van Oostrom. Context-sensitive Conditional Reduction Systems. *Electronic Notes in Theoretical Computer Science, Volume 2, Proc. SEGRAGRA'95*, 1995.

[15] J. Klop. Term graph rewriting. In *Higher-Order Algebra, Logic, and Term Rewriting, Second International Workshop, HOA'95, Selected Papers*, number 1074 in LNCS, Paderborn, Germany, 1995.

[16] J.-W. Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. Mathematischen Centrum, 413 Kruislaan, Amsterdam, 1980.

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

[18] Y. Lafont. From proof nets to interaction nets. In *Advances in Linear Logic*, number 222 in London Mathematical Society Lecture Note Series, pages 225–247. Cambridge University Press, 1995.

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

[20] C. Laneve. *Optimality and Concurrency in Interaction Systems*. PhD thesis, Dipartmento di Informatica, Università degli Studi di Pisa, 1993.

[21] I. Mackie. Interaction nets for linear logic. *Theoretical Computer Science*, 247(1):83–140, 2000.

[22] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.

[23] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[24] D. Plump. Term Graph Rewriting. In *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools, Chapter 1*, eds. H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg. World Scientific, 1999.

[25] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*. World Scientific, Singapore, 1997.

[26] V. van Oostrom and F. van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In *Proc. HOA'93*, volume 816 of *LNCS*, Amsterdam, 1993.

[27] F. van Raamsdonk. Confluence and Normalization for Higher-Order Rewriting. PhD thesis, Vrije Universiteit, Amsterdam, 1996.