# From functional programs to interaction nets via the Rewriting Calculus

New evaluation strategies for functional languages

Horatiu Cirstea[a], Germain Faure[a], Maribel Fernández[b], Ian Mackie[b,c,1] and François-Régis Sinot[c,1]

[a] *LORIA, BP 239 54506 Vandœuvre-lès-Nancy Cedex, France*

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

[c] *LIX, École Polytechnique, 91128 Palaiseau Cedex, France*

**Abstract**

We use the $\rho$-calculus as an intermediate language to compile functional languages with pattern-matching features, and give an interaction net encoding of the $\rho$-terms arising from the compilation. This encoding gives rise to new strategies of evaluation, where pattern-matching and 'traditional' $\beta$-reduction can proceed in parallel without overheads.

**Keywords:** pattern-matching, interaction nets, rewriting calculus

## 1 Introduction

The $\lambda$-calculus is usually put forward as the abstract computational model underlying functional programming, and graph rewriting or environment machines are used to describe evaluation strategies and to derive concrete implementations (see for instance [26]). However, modern functional programming languages have pattern-matching features which cannot be directly expressed in the $\lambda$-calculus. To palliate this problem, pattern-calculi [23,22,4,6,8,13] have been introduced. The $\rho$-calculus [6,8] is a pattern calculus combining the expressiveness of pure functional calculi and algebraic term rewriting. It is an extension of the $\lambda$-calculus where we can abstract on patterns, not just on variables: *abstractions* are written $(p \twoheadrightarrow t)$

$^\star$ Partially funded by the Alliance France-UK project "Implementation Techniques for the Rho Calculus".

[1] Projet Logical, Pôle Commun de Recherche en Informatique du plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

where $p$ is a pattern and $t$ is the body. The rule describing the dynamics of application introduces a *matching constraint*:

$$(\rho) \quad (p \twoheadrightarrow t)\ u \to [p \ll u]t$$

and the $(\sigma)$ rule solves this constraint and applies the matching solution $\sigma_{p \ll u}$ to $t$.

$$(\sigma) \quad [p \ll u]t \to \sigma_{p \ll u}(t)$$

The $\rho$-calculus is parametric in the matching theory: we can use syntactic matching or any arbitrary matching theory, even without unique principal solutions. In the latter case, we can use a *structure* to deal with the multiple solutions.

As an intermediate language for the compilation of functional languages, the $\rho$-calculus has several advantages: patterns are an integral part of the framework, which allows us to reason about pattern-matching and to study the interaction between pattern-matching and $\beta$-reduction at an abstract level; and the $\rho$-calculus can be used to model not only functional behaviour but also imperative features [17], object-oriented features [7], etc.

In this paper we exploit the first point above: we use the $\rho$-calculus as an intermediate language to compile functional languages with pattern-matching features, and adapt the evaluation strategies developed for the $\rho$-calculus to the specific constraints arising from typed functional programs. We then use interaction nets to define and implement the evaluation strategies. This methodology gives rise to new, efficient strategies of evaluation for functional languages, which we describe below.

In [11] we defined two alternative encodings of the $\rho$-calculus in interaction nets [15]. Interaction nets are graph rewrite systems which have been used for the implementation of efficient reduction strategies for the $\lambda$-calculus [12,1,20]. Since interactions are local and strongly confluent, they can take place in any order, even in parallel (see [24]), which makes interaction nets well-suited for the implementation of programming languages and rewriting systems [10]. The first encoding of the $\rho$-calculus in interaction nets given in [11] is simple and exploits the implicit parallelism of rules $(\rho)$ and $(\sigma)$: a term $t$ with a matching constraint (generated by an application of $\rho$) can be applied to another term (again using $\rho$) while the matching constraint is being solved. However, this *simple encoding* can only model a *strict* semantics (see [8]) where a $\rho$-calculus term with a blocked matching evaluates to $\perp$ (fail). The second encoding of [11], which introduces a matching agent and will be called the *explicit encoding*, can implement either a strict or a non-strict semantics, but it looses parallelism.

In the case of *typed* functional languages with pattern-matching, the $\rho$-terms arising from the compilation of programs do not remain blocked. More precisely, a matching failure may occur only if the definitions by pattern-matching are non-exhaustive. We will show that, in this case, a combination of the simple interaction net encoding and the explicit encoding provides an implementation where pattern-matching and 'traditional' $\beta$-reduction can proceed in parallel, without additional overheads. For example, if we have a function with two branches (patterns), say `cons x nil` and `nil`, and the argument is a `cons`, this will compile into a net which, after selecting the `cons` branch, will check that the nested `nil` pattern matches while the substitution for `x` is being performed. We give more examples in Section 4. This is the main contribution of this paper: indeed, the compilation of functional programs in the $\rho$-calculus, and the subsequent interaction net encoding, uncover a new strategy of evaluation which naturally exploits the implicit parallelism of the $\rho$ and $\sigma$ rules.

This paper is organised as follows: after giving some background (Section 2), in Section 3 we define a minimalistic functional language, and give a compilation into the $\rho$-calculus. Section 4 shows an interaction net encoding for this intermediate language. We conclude in Section 5.

## 2 Background

### *2.1 The $\rho$-calculus*

We assume familiarity with the $\lambda$-calculus [2], and start with a short presentation of the $\rho$-calculus; for more details see [6,8,3]. We write $x, y, \ldots$ for variables and $f, g, \ldots$ for constants. The set of *$\rho$-terms* (or just terms, ranged over by $t, u, v$) $\mathcal{T}$ is defined by:

$$t, u ::= x \mid f \mid p \twoheadrightarrow t \mid [p \ll u]t \mid (t\ u) \mid \langle t, u\rangle$$

where $\mathcal{P}$ is an arbitrary subset of $\mathcal{T}$ ($p \in \mathcal{P}$ are called *patterns*); $p \twoheadrightarrow t$ is a *generalised abstraction* (it can be seen either as a $\lambda$-abstraction on a pattern $p$ instead of a single variable, or as a standard term rewriting rule); $[p \ll u]t$ is a *delayed matching constraint* denoting a matching problem $p \ll u$ whose solutions (if any) will be applied to $t$; $(t\ u)$ denotes an *application* (we omit brackets whenever possible, and associate to the left); and finally, $\langle t, u\rangle$ is called a *structure*. Terms are always considered modulo $\alpha$-conversion (later this will be realised for free in interaction nets).

As usual substitutions are mappings from variables to terms, with finite domain, written $\{x_1 := t_1, \ldots, x_n := t_n\}$. We write substitutions postfix: $t\sigma$ denotes the term obtained by applying the substitution $\sigma$ to $t$.

The $\rho$-calculus is parameterised by the set $\mathcal{P}$ of patterns. Here we use *linear* (i.e., each variable occurs at most once) *algebraic patterns*: $p ::= x \mid f\ p_1 \ldots p_n$.

**Example 2.1** The boolean function `null` that tests if its argument is the empty list can be defined in the $\rho$-calculus as follows:

$$\texttt{null} = l \twoheadrightarrow (\langle Nil \twoheadrightarrow True,\ Cons\, x\, y \twoheadrightarrow False\rangle\ l)$$

The following reduction rules give the dynamics of the calculus. We write the reduction $\to_i$ (for implicit) or simply $\to$ when there is no risk of confusion:

$$\begin{array}{lrcl} (\rho) & (p \twoheadrightarrow t)\ u & \to & [p \ll u]t \\ (\sigma) & [p \ll u]t & \to & t\sigma_{p \ll u} \\ (\delta) & \langle t, u\rangle\ v & \to & \langle t\ v, u\ v\rangle \end{array}$$

The rule $(\sigma)$ asks for an *external* matching algorithm to find a solution of the matching of $p$ with $u$, and applies the corresponding substitution to $t$. In this paper we assume linear syntactic matching; under this assumption the calculus is confluent [8].

## 2.2 *Explicit $\rho$-calculus with Structures*

In order to implement the $\rho$-calculus we need to make explicit the specification of the matching algorithm. We recall the explicit $\rho$-calculus of [11] (see also [5]), and extend it with rules to customise structures. Substitution will remain implicit, but we introduce an explicit application symbol $\bullet$ in patterns.

We write reduction in the explicit $\rho$-calculus $\to_x$ (for explicit) or simply $\to$ when there is no risk of confusion.

The rule $(\rho)$ remains unchanged. We can decompose the rule $(\sigma)$ into a finite set of local rules:

$$\begin{array}{lrcl}
(a_c) & f\ t & \to & f \bullet t \\
(a_a) & (t \bullet u)\ v & \to & (t \bullet u) \bullet v \\
(\sigma_a) & [(p \bullet r) \ll (u \bullet v)]t & \to & [p \ll u][r \ll v]t \\
(\sigma_c) & [f \ll f]t & \to & t \\
(\sigma_v) & [x \ll u]t & \to & t\{x := u\}
\end{array}$$

A matching problem $(p \ll u)$ may have no solution; this is called a *blocked matching*. We add rules to detect failure (i.e., a clash):

$$\begin{array}{lrcll}
(\bot_1) & [f \ll g]t & \to & \bot & \text{if } f \neq g \\
(\bot_2) & [f \ll (u \bullet v)]t & \to & \bot & \\
(\bot_3) & [f \ll (p \twoheadrightarrow u)]t & \to & \bot & \\
(\bot_4) & [(u \bullet v) \ll f]t & \to & \bot & \\
(\bot_5) & [(u \bullet v) \ll (p \twoheadrightarrow s)]t & \to & \bot &
\end{array}$$

and rules to propagate $\bot$. There are mainly two options:

(i) Strict Semantics: (*strict*) $C[\bot] \quad \to \quad \bot$ for any context $C[\cdot]$
 This rule corresponds to an exception-like semantics of matching failure, as in ML (e.g., even if the argument of an application is not used by the function, the result is $\bot$). In this semantics, a higher priority is given to this rule than to any other applicable rule (i.e., this rule is tried before the others).

(ii) Non-Strict Semantics: The rule *(strict)* defined above can be weakened to a particular class $\mathcal{C}$ of strict contexts (for instance, $\mathcal{C} = \{([]\ t), t \in \mathcal{T}\}$):

$$(\textit{non-strict}) \quad C[\bot] \quad \to \quad \bot \qquad \text{for any } C[\cdot] \in \mathcal{C}$$

We now turn our attention to structures. Since we will focus on $\rho$-terms arising from functional programs, structures will only be created by the compilation of a function defined by cases. Hence, structures will have the form $\langle p_1 \multimap t_1, \ldots, p_n \multimap t_n\rangle$. Using ($\delta$) and ($\rho$), an application of such structure to an argument $u$ produces $\langle [p_1 \ll u]t_1, \ldots, [p_n \ll u]t_n\rangle$ where only one branch will succeed. In our equational theory for structures $\bot$ should be a neutral element. This is achieved by the rules:

$$
\begin{array}{lrcl}
(\mathit{stk}) & \langle t_1, \ldots, t_{i-1}, \bot, t_{i+1}, \ldots, t_n\rangle & \to & \langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\rangle \; 1 \leq i \leq n \\
(\mathit{singleton}) & \langle t \rangle & \to & t
\end{array}
$$

The rule (stk) was used previously (see [27,9]) to encode term rewriting systems in the $\rho$-calculus. We could be more specific and force evaluation from left to right for instance, but we prefer not to fix the strategy of evaluation yet.

Notice that a naive implementation of ($\delta$) would copy the argument $u$, which is inefficient. Since our use of structures will be limited to the compilation of case constructs in typed programs, we will actually be able to use the information provided by the type system to avoid copying the argument, thus optimising the reduction of structures (see Section 4.4).

We finally give an example of reduction in the $\rho$-calculus.

**Example 2.2** Following Example 2.1 and assuming that the constants $Z$ and $S$ are used to represent Peano integers, we show that the $\rho$-term $\texttt{null}(\mathit{Cons}\ Z\ \mathit{Nil})$ reduces to $\mathit{False}$ as expected:

$$
\begin{array}{l}
\texttt{null}\ (\mathit{Cons}\ Z\ \mathit{Nil}) \\
\quad = (l \multimap \langle \mathit{Nil} \multimap \mathit{True},\ \mathit{Cons}\, x\, y \multimap \mathit{False}\rangle\ l)(\mathit{Cons}\ Z\ \mathit{Nil}) \\
\quad \to_{\rho} [l \ll (\mathit{Cons}\ Z\ \mathit{Nil})]\langle \mathit{Nil} \multimap \mathit{True},\ \mathit{Cons}\, x\, y \multimap \mathit{False}\rangle l \\
\quad \to_{\sigma_v} \langle \mathit{Nil} \multimap \mathit{True},\ \mathit{Cons}\, x\, y \multimap \mathit{False}\rangle\ (\mathit{Cons}\ Z\ \mathit{Nil}) \\
\quad \to_{\delta} \langle (\mathit{Nil} \multimap \mathit{True})(\mathit{Cons}\ Z\ \mathit{Nil}), (\mathit{Cons}\, x\, y \multimap \mathit{False})(\mathit{Cons}\ Z\ \mathit{Nil})\rangle \\
\quad \to_{\rho}^{2} \langle [\mathit{Nil} \ll (\mathit{Cons}\ Z\ \mathit{Nil})]\mathit{True}, [\mathit{Cons}\, x\, y \ll (\mathit{Cons}\ Z\ \mathit{Nil})]\mathit{False}\rangle \\
\quad \to_{a_c,a_a}^{*} \langle [\mathit{Nil} \ll ((\mathit{Cons} \bullet Z) \bullet \mathit{Nil})]\mathit{True}, [((\mathit{Cons} \bullet x) \bullet y) \ll ((\mathit{Cons} \bullet Z) \bullet \mathit{Nil})]\mathit{False}\rangle \\
\quad \to_{\bot_2} \langle \bot, [((\mathit{Cons} \bullet x) \bullet y) \ll ((\mathit{Cons} \bullet Z) \bullet \mathit{Nil})]\mathit{False}\rangle \\
\quad \to_{\mathit{stk}} \langle [((\mathit{Cons} \bullet x) \bullet y) \ll ((\mathit{Cons} \bullet Z) \bullet \mathit{Nil})]\mathit{False}\rangle \\
\quad \to_{\mathit{singleton}} [((\mathit{Cons} \bullet x) \bullet y) \ll ((\mathit{Cons} \bullet Z) \bullet \mathit{Nil})]\mathit{False} \\
\quad \to_{\sigma_c,\sigma_a}^{*} [x \ll Z][y \ll \mathit{Nil}]\mathit{False} \\
\quad \to_{\sigma_v}^{*} \mathit{False}
\end{array}
$$

**Example 2.3** [Fixpoints] A fixpoint operator is a term $Y$ such that for all terms $t$, $Y\,t \to^{*} t\,(Y\,t)$. It is easy to check that the following terms are fixpoint operators (the second has the advantage of being well-typed [27]):

- $Y_T = (y \multimap x \multimap x\,(y\,y\,x))\,(y \multimap x \multimap x\,(y\,y\,x))$
- $Y_{\mathit{rec}} = x \multimap ((z \multimap z\,(\mathit{rec}\,z))\,(\mathit{rec}\,f \multimap (x\,(f\,(\mathit{rec}\,f)))))$ where $\mathit{rec}$ is a constant.

## 3 From a functional language to the $\rho$-calculus

### 3.1 The Language

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

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

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

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

A `case` branch of the form $(p_i \leadsto \cdot)$ acts as a binder i.e., $\mathsf{fv}(p_i \leadsto u_i) = \mathsf{fv}(u_i) \setminus \mathsf{fv}(p_i)$ where $\mathsf{fv}(u_i)$ denotes the set of free variables of $u_i$.

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

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

**Example 3.1** In the sequel we will use the following datatypes for numbers and lists with elements of type $\alpha$:

$$\begin{aligned} Int &= Z \mid S(Int) \\ List\ \alpha &= Nil \mid Cons(\alpha, List\ \alpha) \end{aligned}$$

As usual, the type system ensures that in a case construct $\texttt{case}\ t\ \texttt{of}\ (p_i \leadsto u_i)_{i \in I}$ all the branches have the same type and $t$ has the same type as the patterns $p_i$ (for all $i \in I$), that is, some datatype $DT$. We do not assume that the cases are exhaustive, but we do assume they are non-overlapping for simplicity. We use a strict matching semantics, as in ML (i.e., an application of a function to an argument that is not covered by the case definition will produce a runtime error). We omit the typing rules, which are standard.

The dynamics of the language is given by the following reduction rules (reduction is denoted by $\to_f$ or simply $\to$) where $\{x := u\}$ denotes the substitution of $x$ by $u$.

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

Since the rewrite rules are left-linear and non-overlapping (that is, they define an orthogonal system [14]), the language is confluent. It is easy to see that it is not terminating, due to the presence of the fixpoint operator `fix`.

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

**Example 3.2** (i) Assuming that $Nil$ with arity 0, and $Cons$ with arity 2, are used to define the datatype $List$ as in Example 3.1, and that $True$ and $False$ are the boolean constants, we can define the boolean function `null` by pattern-matching as follows:

$\mathtt{null} \triangleq \mathtt{fn}\, l.\mathtt{case}\; l \;\mathtt{of}\; (Nil \rightsquigarrow True, Cons(x,y) \rightsquigarrow False)$

(ii) Assuming that $Z$ with arity 0, and $S$ with arity 1 are used to define the datatype $Int$ as in Example 3.1, the recursive function `length` can be defined by pattern-matching as follows:

$\mathtt{length} \triangleq \mathtt{fix}(\mathtt{fn}\, len.\mathtt{fn}\, l.\mathtt{case}\; l \;\mathtt{of}\; (Nil \rightsquigarrow Z, Cons(x,y) \rightsquigarrow S(len\; y)))$

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

$$\begin{aligned} \mathtt{let}\;\; x = t \;\;\mathtt{in}\;\; u &\triangleq (\mathtt{fn}\, x.u)t \\ \mathtt{letrec}\;\; f = t \;\;\mathtt{in}\;\; u &\triangleq \mathtt{let}\;\; f = \mathtt{fix}(\mathtt{fn}\, f.t) \;\;\mathtt{in}\;\; u \end{aligned}$$

We can also define mutually recursive definitions as follows:

$$\begin{aligned} &\mathtt{letrec}\;\; f = u \;\;\mathtt{and}\;\; g = v \;\;\mathtt{in}\;\; w \triangleq \\ &\qquad \mathtt{letrec}\;\; h = \mathtt{fn}\, g.(\mathtt{let}\;\; f = h\; g \;\;\mathtt{in}\;\; u) \;\;\mathtt{in} \\ &\qquad\qquad \mathtt{letrec}\;\; g = (\mathtt{let}\;\; f = h\; g \;\;\mathtt{in}\;\; v) \;\;\mathtt{in} \\ &\qquad\qquad\qquad \mathtt{let}\;\; f = h\; g \;\;\mathtt{in}\;\; w \end{aligned}$$

## *3.2 Compilation*

The following compilation function, defined by induction on terms, translates terms in the typed functional language into the $\rho$-calculus:

$$\begin{aligned} [\![x]\!] &= x \\ [\![\mathtt{fn}\, x.t]\!] &= (x \twoheadrightarrow [\![t]\!]) \\ [\![t\; u]\!] &= [\![t]\!]\; [\![u]\!] \\ [\![C(t_1,\dots,t_n)]\!] &= C\; [\![t_1]\!] \dots [\![t_n]\!] \\ [\![\mathtt{case}\; t \;\mathtt{of}\; (p_1 \rightsquigarrow u_1,\dots,p_n \rightsquigarrow u_n)]\!] &= \langle [\![p_1]\!] \twoheadrightarrow [\![u_1]\!], \dots, [\![p_n]\!] \twoheadrightarrow [\![u_n]\!] \rangle\; [\![t]\!] \\ [\![\mathtt{fix}(\mathtt{fn}\, f.t)]\!] &= Y [\![\mathtt{fn}\, f.t]\!] \end{aligned}$$

where $Y$ is a fixpoint operator of the explicit $\rho$-calculus (see Example 2.3). We leave $Y$ abstract because it is an implementation choice. In particular, this will enable us to use a more efficient translation into interaction nets.

**Example 3.3** We can check that the compilation of the function `null` defined in Example 3.2 gives the function `null` in the $\rho$-calculus as given in Example 2.1.

Note that case constructs are compiled into structures applied to an argument, and can be reduced using the $\delta$ rule. The interaction net encoding will ensure that $[\![t]\!]$ is not copied, and moreover it will allow matching to be carried in parallel with other reductions, if possible.

We define the compilation of $\sigma = \{x_1 := u_1, \ldots, x_n := u_n\}$ to be the substitution $[\![\sigma]\!] = \{x_1 := [\![u_1]\!], \ldots, x_n := [\![u_n]\!]\}$.

We now state some soundness invariants.

**Proposition 3.4** (i) *For all terms $t$ and all substitutions $\sigma$, $[\![t\sigma]\!] = [\![t]\!][\![\sigma]\!]$.*

(ii) *For all patterns $p$ and all terms $u$:*
- *The matching problem $p \ll t$ has a solution iff the matching problem $[\![p]\!] \ll [\![t]\!]$ has a solution.*
- *The substitution $\sigma$ is a solution of the matching problem $p \ll t$ iff the substitution $[\![\sigma]\!]$ is a solution of the matching problem $[\![p]\!] \ll [\![t]\!]$.*

(iii) *For all terms $t$ and $u$, if $t \to_f u$, then $[\![t]\!] \to_x^* [\![u]\!]$.*

One can notice that in the $\rho$-calculus the granularity of the reduction is finer than in the chosen functional language and thus, the intermediate $\rho$-terms obtained during the reduction of the translation of a program $t$ do not necessarily correspond to a program. More precisely, for a reduction $[\![t]\!] \to_x u$ we cannot always exhibit a term $u'$ such that $t \to_f u'$ and $[\![u']\!] = u$. Nevertheless, if the reduction of the term $u$ continues then the term $[\![u']\!]$ is eventually reached.

**Lemma 3.5** *For all programs $t$ and for all terms $u$ such that $[\![t]\!] \to_x u$ there exists a program $v$ such that $u \to_x^* [\![v]\!]$ and $t \to_f^* v$.*

The following proposition is a corollary of the previous results.

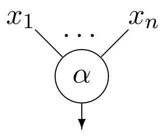
**Proposition 3.6 (Correctness)** *Let $t$ be a program and $v$ be a normal form, then $t \to_f^* v$ iff $[\![t]\!] \to_x^* [\![v]\!]$.*

In the following section we give an interaction net implementation for the functional language defined above, which defines a strategy of evaluation based on the encodings of the $\rho$-calculus presented in [11] and the coding of datatypes discussed in [21]. Although we focus on implementation in this paper, the intermediate $\rho$-calculus compilation has also interesting applications for programming language design (for instance one could study the properties of a more general language including non-linear patterns, or non-syntactic matching theories) and could also be used to study program transformations (in the same way as, for instance, explicit substitution calculi) and to prove correctness of program optimisations.

## 4 Interaction Net Encoding

### *4.1 Preliminaries*

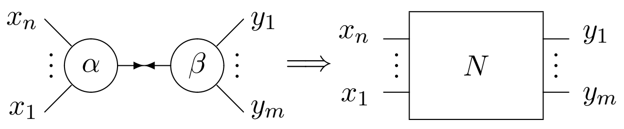
A system of interaction nets is specified by a set $\Sigma$ of symbols with fixed arities, and a set $\mathcal{R}$ of interaction rules. An occurrence of a symbol $\alpha \in \Sigma$ is called an *agent*. If the arity of $\alpha$ is $n$, then the agent has $n+1$ *ports*: a *principal port* depicted by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



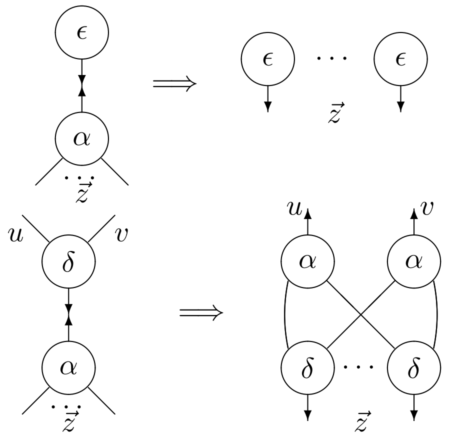
Intuitively, a net $N$ is a graph (not necessarily connected) with agents at the vertices and each edge connecting at most 2 ports. The ports that are not connected to another agent are *free*. There are two special instances of a net: a wiring (no agents) and the empty net; the extremes of wirings are also called free ports. The interface of a net is its set of free ports.

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

The following diagram shows the format of interaction rules ($N$ can be any net built from $\Sigma$).



We show as an example the interaction rules of two ubiquitous agents, namely the *erase* ($\epsilon$), of arity 0, which deletes everything it interacts with, and the *duplicator* ($\delta$), of arity 2, which copies everything. These are represented by the following diagrams, where $\alpha$ is any node. We refer to [15] for more details and examples.



We use the notation $\Longrightarrow$ for the one-step reduction relation and $\Longrightarrow^*$ for its transitive and reflexive closure. If a net does not contain any active pairs then it is in normal form. The key property of interaction nets, besides locality of reduction, is strong confluence. There are several implementations of interaction nets, see for instance [16] and [25]; the latter has been designed to take advantage of additional processors, thus giving a parallel implementation of interaction nets.

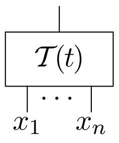


Fig. 1. Translation of a $\rho$-term $t$ with $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$.

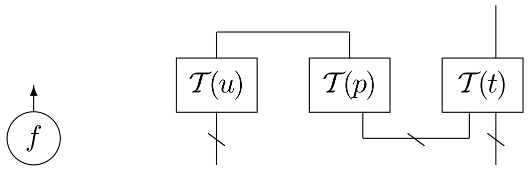


Fig. 2. Translation of constants (left) and matching constraints (right).

### *4.2 Implementing the Language*

We will assume that the problems of binding and substitution can be solved as in any off-the-shelf interaction net encoding of the $\lambda$-calculus (see for instance [18,19]), and concentrate on the encoding of the explicit matching and structure rules given in Section 2.2. This methodology is justified by the fact that the terms $p \twoheadrightarrow t$ and $x \twoheadrightarrow [p \ll x]t$ are extensionally equivalent, so that we can safely precompile terms in order to abstract only on variables, as in the $\lambda$-calculus, and have explicit matching constraints from the beginning. Also, there is a standard, efficient way to encode recursion in interaction nets for the $\lambda$-calculus, which consists of building a cyclic structure which explicitly "ties the knot". The idea corresponds exactly to an encoding of recursion in graph reduction (see Peyton Jones [23] for instance), and was adapted to interaction nets in [18]. We use this for the encoding of $Y$ (see [20] for details).

We now define by induction a function $\mathcal{T}(\cdot)$ to translate the $\rho$-terms arising from the compilation of functional programs into interaction nets, and we give the interaction rules that will be used to evaluate them. As in the *simple* interaction net encoding of the $\rho$-calculus described in [11], a $\rho$-term $t$ with free variables $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$ will be translated to a net $\mathcal{T}(t)$ with the root edge at the top, and $n$ free edges corresponding to the free variables, as shown in Figure 1.

The translation function $\mathcal{T}(\cdot)$ is defined by induction as follows:

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

**Constant:** For each constant $f$ we introduce an agent as shown in Figure 2 (left).

**Matching Constraint:** A term of the form $[p \ll u]t$ is encoded as shown in Figure 2 (right)[2] which can be interpreted as the substitution in $t$ of the (possible) solution of the matching (the left subnet corresponds to the matching problem $p \ll u$).

**Structure:** We will discuss the encoding of structures at the end of the section.

**Abstraction:** We assume that terms have been precompiled to abstract only on variables, as described above; hence we can reuse the abstraction of the $\lambda$-calculus.

**Application:** Similarly for application, we introduce an agent @ with its principal

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

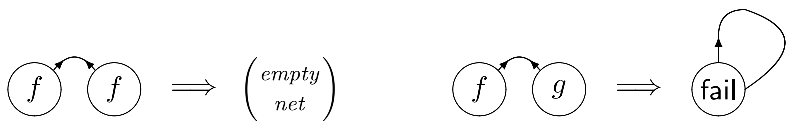


Fig. 3. Matching of constants (success and failure)

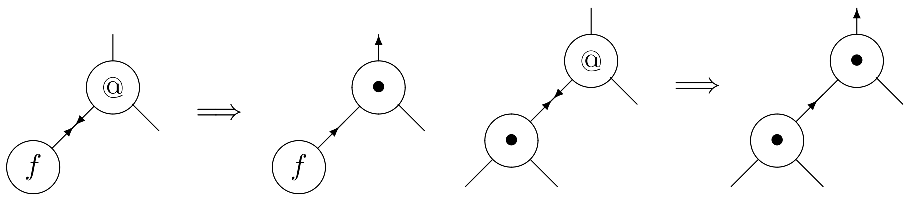


Fig. 4. Rules to transform patterns

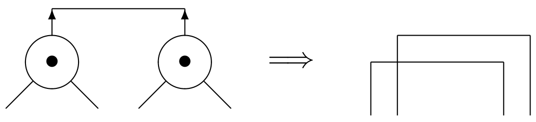


Fig. 5. Matching applications

port oriented towards the left subterm, so that interaction with an abstraction is possible. To implement the rule ($\rho$), we define an interaction rule between abstraction and application as in the $\lambda$-calculus (see for instance [19]).

### *4.3 Matching Rules*

The matching rules are inspired by the "simple" encoding of [11]. Assume we have just one matching constraint to solve (the general case of a structure with multiple branches will be treated below). The matching algorithm is initiated by connecting the root of a pattern with the term to match. Thus, the rule ($\sigma_v$) (matching against a variable) is realised for free, as in the $\lambda$-calculus. To simulate ($\sigma_a$) and ($\perp_1$), constants will interact: Two identical constants cancel each other to give the empty net, as indicated in Figure 3 (left). If the agents are not the same, then we introduce an agent fail, which represents a failure in the matching algorithm, as indicated in Figure 3 (right). We interpret a net containing an agent fail as an overall failure, thus implementing the strict matching semantics.

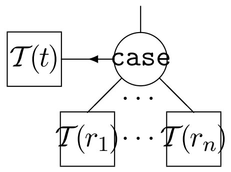
We need rules to convert a usual application (@) into a pattern application ($\bullet$) when it is part of an algebraic pattern (or term), these are shown in Figure 4; and a rule to match applications, which is given in Figure 5, as well as interaction rules corresponding to the rules ($\perp_2$) and ($\perp_4$) which we omit. We do not need interaction rules corresponding to ($\perp_3$) and ($\perp_5$) since the language is typed.

We refer to [11] for a detailed description and correctness proofs for matching constraints. In particular, in [11] it is shown that with this encoding of matching we can only implement a strict $\rho$-calculus semantics, but, on the positive side, it allows us to obtain a strategy of evaluation with a good potential for parallelism. This is because matching interactions involving the constraint associated to an abstraction can take place in parallel with a traditional $\beta/\rho$ reduction involving the same abstraction, without introducing any 'administrative' agents (i.e., no overheads). We use this feature in the encoding of functional programs below, to derive an evaluation strategy with the same potential for parallelism. We give examples at the end of the section.

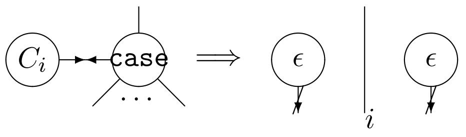
### *4.4 Structures*

We now describe the encoding of structures and the rule ($\delta$). First remark that structures only arise from the compilation of `case` constructs, more precisely, structures can only occur in subterms of the form: $\langle l_1 \twoheadrightarrow r_1, \ldots, l_n \twoheadrightarrow r_n \rangle\ t$. The goal is to avoid making multiple copies of $t$ in the implementation of ($\delta$), and to permit matching to proceed in parallel with functional computation, whenever possible. For these reasons, we will not treat these terms as standard applications. Instead, for each structure $\langle l_1 \twoheadrightarrow r_1, \ldots, l_n \twoheadrightarrow r_n \rangle\ t$ (with $n > 1$; if $n = 1$ we can treat it as an abstraction) occurring in the compilation of a program we will introduce an agent `case` as explained below, where we build a net that minimises the number of selections necessary. To keep the diagrams simple, we show the compilation in stages.

First, we consider the case when each $l_i$ is a different constant $C_i$. We can then encode the structure using a simple case agent as follows:



with the following collection of rules which select the appropriate branch of the case, and erase all other options using $\epsilon$ agents.



The top auxiliary port of a `case` agent represents the output; the interaction rule above selects the branch $i$ corresponding to the constructor $C_i$ and connects it to the output port (all other branches are erased). It is a straightforward exercise to verify that this indeed mimics the corresponding reduction rule. Note that we are assuming that all patterns are disjoint (non-overlapping) but they may be non-exhaustive. Note also that garbage collection (using $\epsilon$) is explicit in interaction nets, since interaction rules must preserve the interface of the net.

Next we deal with deeper patterns, including variables. To give the idea we consider the case where there is just one pattern of depth greater than 1 (i.e., the root is an application), for instance: $\langle C_1 \twoheadrightarrow r_1, C_2\ x\ y \twoheadrightarrow r_2 \rangle\ t$. The compilation and interaction rules are in Figure 6.

Again the top auxiliary port of `case` is the output; the first rule above corresponds to a pattern of depth 1 (as before). Note that in the second interaction rule, the right hand side has a wire to connect the net $\mathcal{T}(r_2)$ to the output port of the `case` agent. In the compilation, the nets $\mathcal{T}(y)$ and $\mathcal{T}(C_2\ x)$ are there precisely to complete the pattern matching, even though the branch would have already been selected. Any resulting substitutions generated are connected to the free variables of $\mathcal{T}(r_2)$.

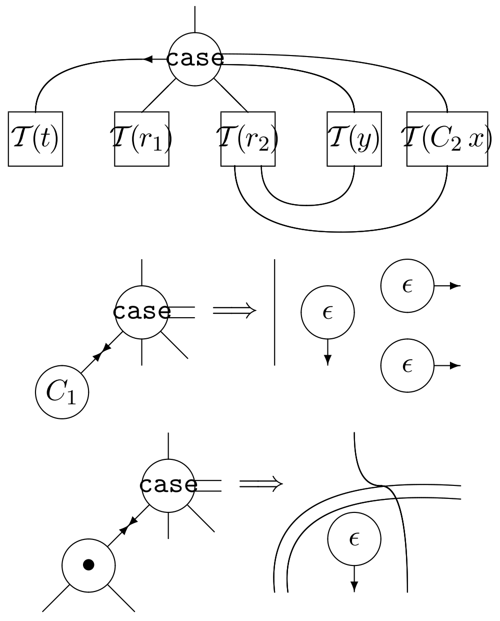
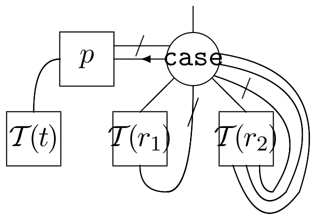


Fig. 6. Compilation and rules

The extension to the case where there are more constant patterns is straightforward.

Next we examine the case when there are more than one application cases to consider. Again, to keep the diagrams simple, we will concentrate on this aspect, and ignore the patterns of depth one that were given previously (extra ports in the `case` agent would be needed). Consider the example: $\langle C_1x \twoheadrightarrow r_1, C_2\ y\ z \twoheadrightarrow r_2\rangle\ t$. Both patterns have an application at the root, so we cannot use a case to distinguish them. However, we can identify where the patterns disagree, and consume that part before using a case agent, as above. Once the common prefix has been consumed, then we are left with a situation which is exactly as explained in the previous case (i.e., constant and an application).

The following is the compilation, where the net $p$ is the common prefix, with principal ports pointing towards $\mathcal{T}(t)$ (so the rules in Subsection 4.3 will apply). In the diagram below we assume that there is nothing else to the pattern, as this situation has already been dealt with previously.



The interaction rules are now identical to the ones previously given for the case agent, except that in addition we must connect the additional bindings to the correct branch.

This completes the encoding of structures, which requires the combination of the above features. In addition, when the terms $r_i$ have common free variables we must use extra agents to allow these variables to be shared. The compilation for such a feature is standard and will be omitted here; we refer the reader to [21] for details.

Pattern matching is slightly more efficient if we use an alternative encoding for patterns, where a constructor of arity $n$ in the functional language is represented by an agent of arity $n$ (instead of a 0-ary constant). This has the advantage of avoiding interactions between case agents and the algebraic application agent.

In Section 4.5 below we give an example showing how the encoding of the $\rho$-calculus used here allows us to exploit the implicit parallelism between matching and functional computations.

### *4.5 The Parallel Strategy at Work*

To illustrate the potential for parallelism, we give an example using a variant of the Ackermann function on coloured trees, which is based on the datatype:

$$Tree = Nil \mid Red(Int, Tree, Tree) \mid Black(Int, Tree, Tree)$$

Let `ack` be the Ackermann function. The function `ackt` takes two trees and computes a new tree where the nodes contain integers obtained by applying `ack` to the corresponding nodes of the arguments, but only when the trees have the same alternating colours. It is defined in our functional language as shown in Figure 7.

The compilation of this program in the $\rho$-calculus and subsequent encoding in interaction nets produces a net with an active pair between the agent case representing the first case in the program and the agent *Red*.

After the interaction between the first case agent and *Red*, the actual value of $x_1$ gets connected to the multiplication agent in the first branch of the case, so that we can start computing $2 * x_1$ in parallel with the rest of the matching. Then, after the interaction between the second case agent and *Red*, we also get the value of $x_2$ connected to the net representing the Ackermann function and we can then compute in parallel the value of $\texttt{ack}(2 * x_1, x_2)$ while the rest of the pattern (i.e., Black, Black) is checked.

## 5 Conclusion

We have proposed to use the $\rho$-calculus as an alternative foundation for functional programming languages, and provided a compilation of a simple functional programming language into the $\rho$-calculus. This calculus is better adapted than the $\lambda$-calculus for representing features, specifically pattern matching, of functional languages. One of the main features of our compilation is that we can experiment with different pattern-matching algorithms and matching strategies, in a modular way. We have thus a powerful formalism for programming language design, and for reasoning about functional program implementation. Using this as an intermediate language, we have demonstrated that we can compile, also in a modular way, into interaction nets, and obtain new strategies of evaluation of programs with pattern-matching. Since the translation into interaction nets is modular, the strategy specified here can be combined with any $\beta$-reduction strategy, including an optimal one.

```
$\texttt{ackt} \triangleq \texttt{fix}(\texttt{fn}\, ackt.\texttt{fn}\, t_1.\texttt{fn}\, t_2.\texttt{case}\, t_1\, \texttt{of}$
        (
        $Red(x_1, Black(y_1, t_L, t_R), Black(z_1, s_L, s_R)) \leadsto$
          $\texttt{case}\, t_2\, \texttt{of}$
            (
            $Red(x_2, Black(y_2, t'_L, t'_R), Black(z_2, s'_L, s'_R)) \leadsto$
                $Red(\texttt{ack}(2 * x_1, x_2),$
                    $ackt(Black(y_1, t_L, t_R), Black(y_2, t'_L, t'_R)),$
                    $ackt(Black(z_1, s_L, s_R), Black(z_2, s'_L, s'_R)))$
            )
        $Black(x_1, Red(y_1, t_L, t_R), Red(z_1, s_L, s_R)) \leadsto$
          $\texttt{case}\, t_2\, \texttt{of}$
            (
            $Black(x_2, Red(y_2, t'_L, t'_R), Red(z_2, s'_L, s'_R)) \leadsto$
                $Black(\texttt{ack}(2 + x_1, x_2),$
                    $ackt(Red(y_1, t_L, t_R), Red(y_2, t'_L, t'_R)),$
                    $ackt(Red(z_1, s_L, s_R), Red(z_2, s'_L, s'_R)))$
            )
        )
```

Fig. 7. Ackermann

The interaction net encoding, although derived from a strategy of evaluation in the $\rho$-calculus, could of course be defined directly on the functional programs, without the intermediate compilation. We hope that the compilation into the $\rho$-calculus will allow us to transfer other results into the functional language (e.g., extensions to accommodate imperative features).

## Acknowledgement

We thank Gilles Dowek for useful discussions on the subject of this paper.

## References

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

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[3] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure patterns type systems. In *Principles of Programming Languages - POPL2003, New Orleans, USA*. ACM, Jan. 2003.

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

[5] H. Cirstea, G. Faure, and C. Kirchner. A rho-calculus of explicit constraint application. In *Proceedings of the 5th workshop on rewriting logic and applications*. Electronic Notes in Theoretical Computer Science, 2004.

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

[7] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In A. Middeldorp, editor, *Proceedings of RTA'2001*, Lecture Notes in Computer Science, Utrecht (The Netherlands), May 2001. Springer-Verlag.

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

[9] H. Cirstea, L. Liquori, and B. Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. In *Post-proceedings of TYPES*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

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

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

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

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

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

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

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

[17] L. Liquori. iRho: The Software (system demonstration). In *Proceedings of Developments in Computational Models (DCM'05), Lisbon, Portugal, 2005*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

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

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

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

[21] I. Mackie. An interaction net implementation of additive and multiplicative structures. *Journal of Logic and Computation*, 15(2):219–237, April 2005.

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

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

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

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

[26] R. Plasmeijer and M. Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[27] B. Wack. *Typage et déduction dans le calcul de réécriture.* PhD thesis, Université Henri Poincaré - Nancy I, 2005.