Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2009) 55–71
www.elsevier.com/locate/entcs

Recursive Functions with Pattern Matching in Interaction Nets

Maribel Fernández Ian Mackie Shinya Sato Matthew Walker

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

LIX, École Polytechnique, 91128 Palaiseau Cedex, France

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

Abstract

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

Keywords: pattern matching, recursion, interaction nets

1 Introduction

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

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

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

doi:10.1016/j.entcs.2009.10.017

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

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

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

Summarising, the main contributions of this paper are:

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

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

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

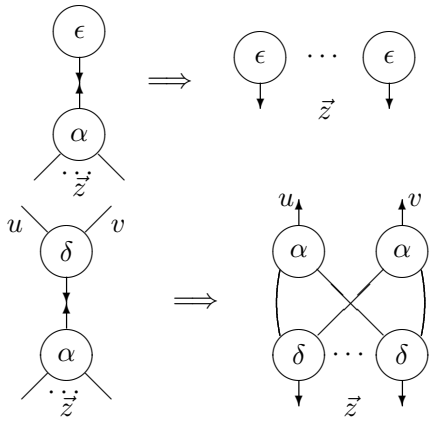
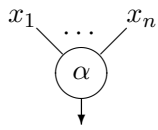


Fig. 1. Erasing and Copying

2 Background: Interaction nets

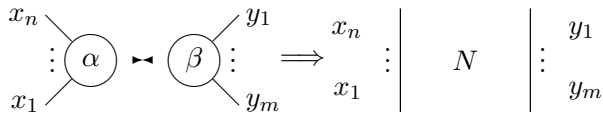
We recall the main notions from interaction nets that will be needed in the rest of the paper; for more details and examples we refer to [14].

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



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

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



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

We use the notation $\Longrightarrow$ for the one-step reduction relation and $\Longrightarrow^*$ for its transitive and reflexive closure. If a net does not contain any active pairs then it is in normal form. The key property of interaction nets, besides locality of reduction, is strong confluence.

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

3 A simple functional language

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

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

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

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

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

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

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

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

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

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

term at a given position [1]. We omit the typing rules, which are standard.

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

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

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

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

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

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

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

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

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

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

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

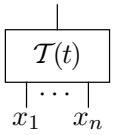


Fig. 2. Translation of a term t with $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$.

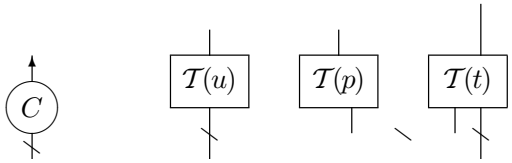


Fig. 3. Translation of constants (left) and matching constraints (right).

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

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

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

4 Implementing the language via interaction nets

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

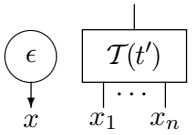
A term with free variables $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$ will be translated to a net $\mathcal{T}(t)$ with the root edge at the top, and n free edges corresponding to the free variables, as shown in Figure 2. We now define by induction the function $\mathcal{T}(\cdot)$.

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

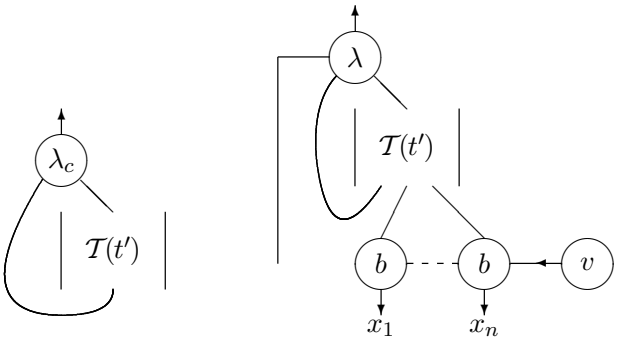
Constructor: For each constructor C we introduce an agent as shown in Figure 3 (left) [2] with the arity of the constructor matching the arity of the agent.

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

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



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

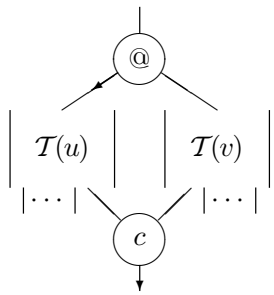


The first case, shown on the left in the above diagram, is when $\mathsf{fv}(\lambda x.t') = \emptyset$. Here we use one agent λ_c to represent a *closed abstraction*. Note that we explicitly connect the occurrence of the variable to the binding λ.

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

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

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



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

4.1 Implementing term reduction

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

4.2 Pattern Matching

The matching rules are inspired by the "simple" encoding of [10]. Assume we have just one matching constraint to solve; i.e., given a pattern p and a term t, we need to find a substitution σ such that $p\sigma = t$, if there is one (the generalisation to case structures with multiple branches will be given below). The matching algorithm is initiated by connecting the root of the pattern p with the term t (see Figure 3, right). Thus, matching against a variable is realised for free, as in the λ-calculus. Two identical constants cancel each other and the matching continues in the arguments (or results in the empty net if the constant has arity zero), as indicated in Figure 5 (upper). If the agents are not the same, then we introduce an agent fail, which represents a failure in the matching algorithm, as indicated in Figure 5 (lower). We interpret a net containing an agent fail as an overall failure, thus implementing the strict matching semantics. We do not need interaction rules for a constructor and an abstraction because the language is typed.

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

4.3 Case structures

We now describe the encoding of case structures

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

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

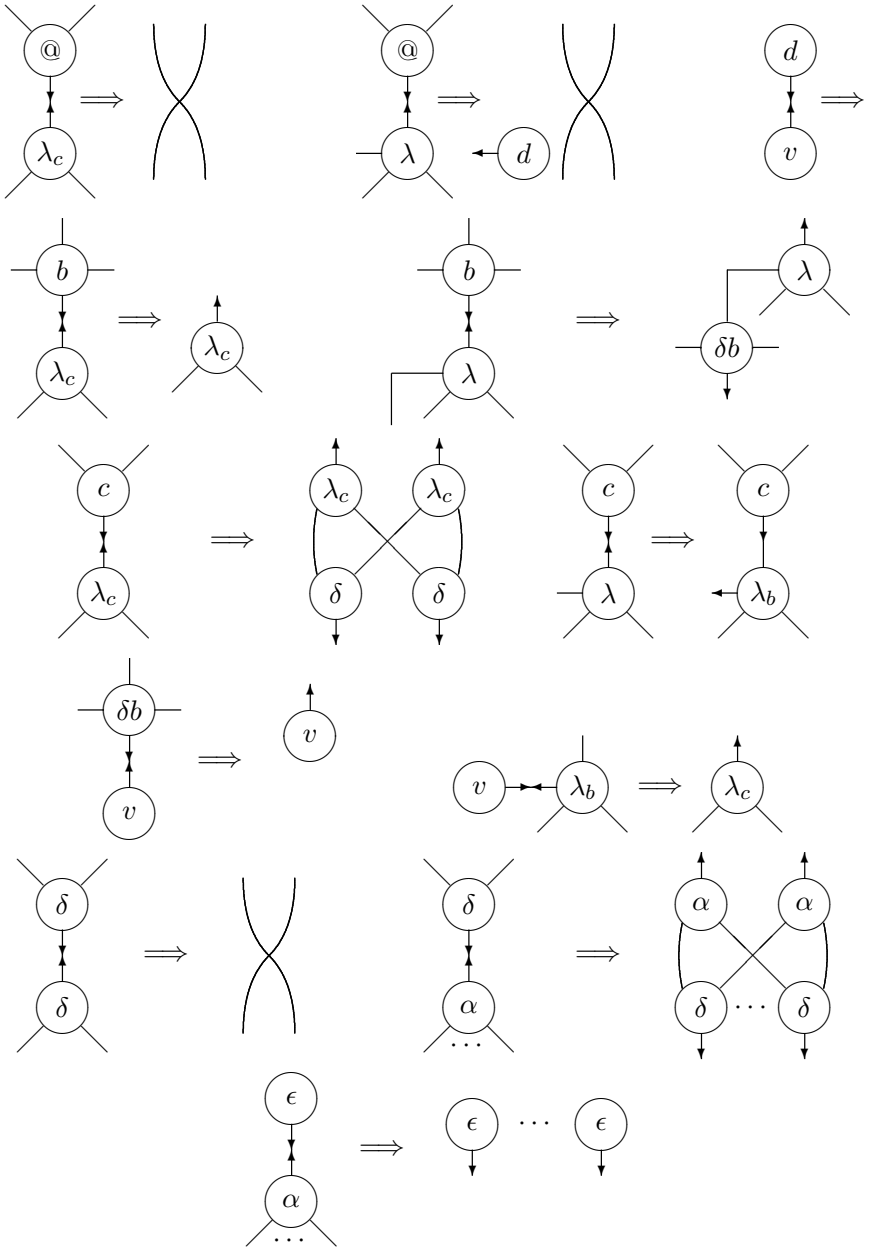


Fig. 4. Lambda calculus rules

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

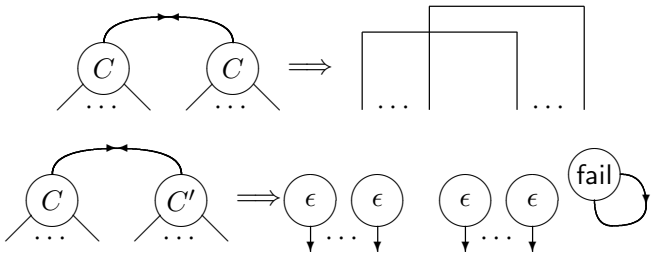


Fig. 5. Matching of constructors (success and failure where C and C' are distinct)

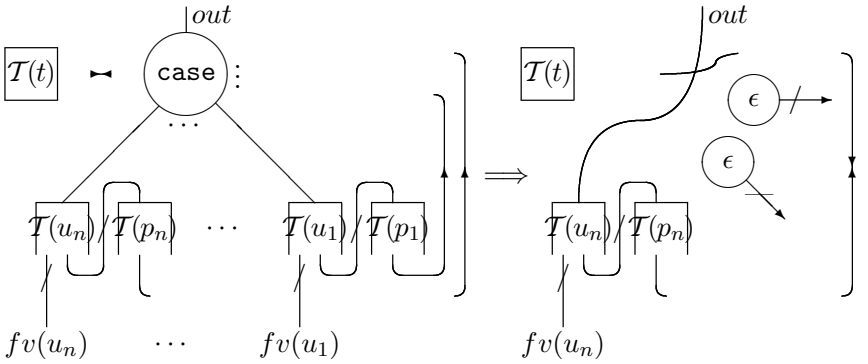


Fig. 6. A reduction where `case` allows matching to continue with p_n.

this only accounts for patterns with different root symbols; patterns with the same top-level constructors are dealt with later).

Two further modifications are needed to the encoding of structures: Firstly the free variables of all u_i need to be 'boxed'. A chain of $\mathtt{cb}_n$ agents is introduced, terminated at one end by a `v'` agent and at the other by an additional auxiliary port of `case`. Figure 7 depicts a simplified reduction sequence for `case` allowing pattern matching with $\mathcal{T}(t)$ by p_i; the agent $\mathtt{e}_i$ traverses the chain linking every free variable, y_1 to y_k, with u_i, and garbage collects everywhere else:

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

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

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

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

Proof. The interaction rules for the case agent corresponding to this particular case construct ensure that the principal port at the root of the net $\mathcal{T}(t)$ gets connected to

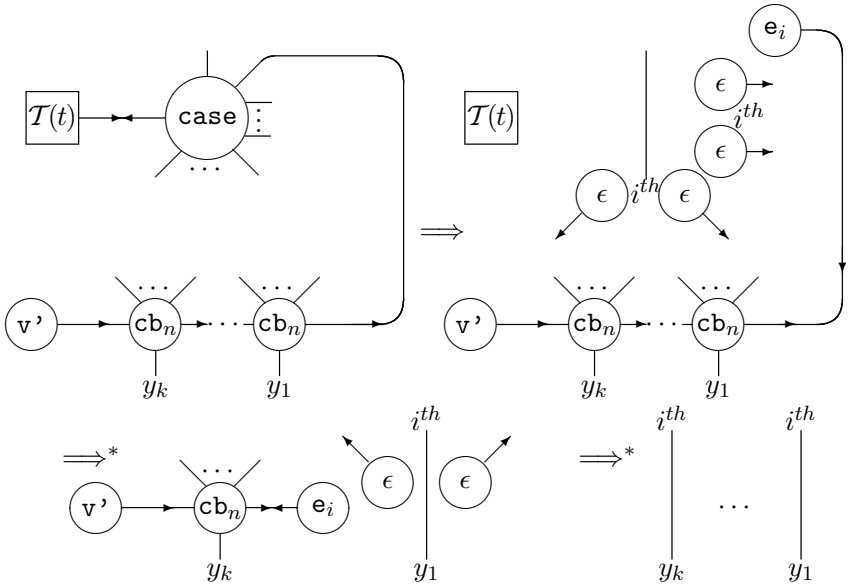


Fig. 7. Chaining of free variables within the structure with $k = max\{|\bigcup_i fv(u_i)|\}$. Note that if $y_m \notin fv(u_i)$ then it is connected to an ϵ-agent.

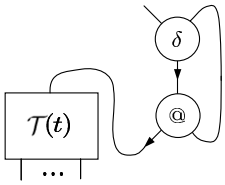
the principal port at the root of $\mathcal{T}(p_i)$, as depicted in Figure 6. Since the matching algorithm we are using is correct [10], the interactions in this subnet will generate the matching substitution $\mathcal{T}(\sigma)$. Finally, the interactions between the boxing agents $\mathtt{cb}_n$ and $\mathtt{e}_i$ connect the $\mathcal{T}(\sigma)$ to $\mathcal{T}(u_i)$. □

4.4 Recursion

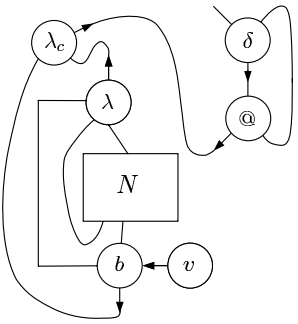
There is a standard way to encode recursion in interaction nets for the λ-calculus, which consists of building a cyclic structure which explicitly "ties the knot". The idea corresponds exactly to an encoding of recursion in graph reduction [20] and was adapted to interaction nets in [16]. For example,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

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

(1) Translation of Yt:

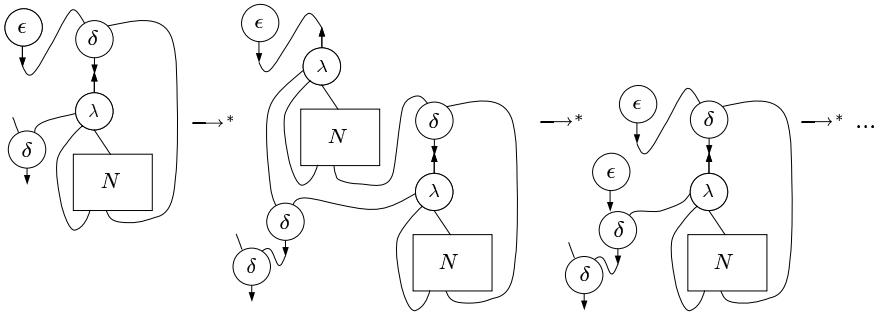


(2) Translation of the recursive function `length`:



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

Fig. 8. Recursion using cycles



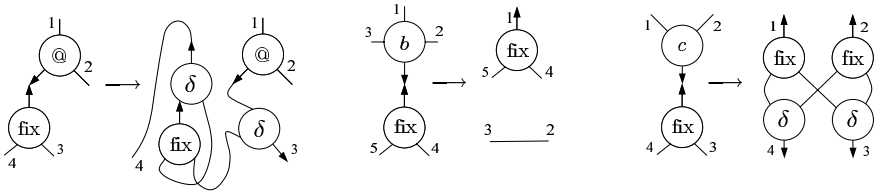
The standard solution to this problem relies on the introduction of a reduction strategy, called *connected reduction* or reduction to interface normal form (INF for short, see [9]), which restricts reductions to active pairs connected to the interface of the net (in this way, non-terminating reduction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

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

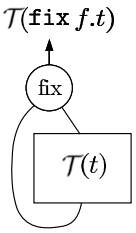
First, recall that recursive functions in the functional language are defined using the syntax $\texttt{fix}\ f.t$, where we can assume $\texttt{fv}(t) = \{f\}$ in the case of programs (i.e., closed terms). We have the following reduction: $(\texttt{fix}\ f.t)u \rightarrow^* (t\{f := \texttt{fix}\ f.t\})u$,

which we implement by introducing the following binary agent `fix`:

and the following interaction rules:



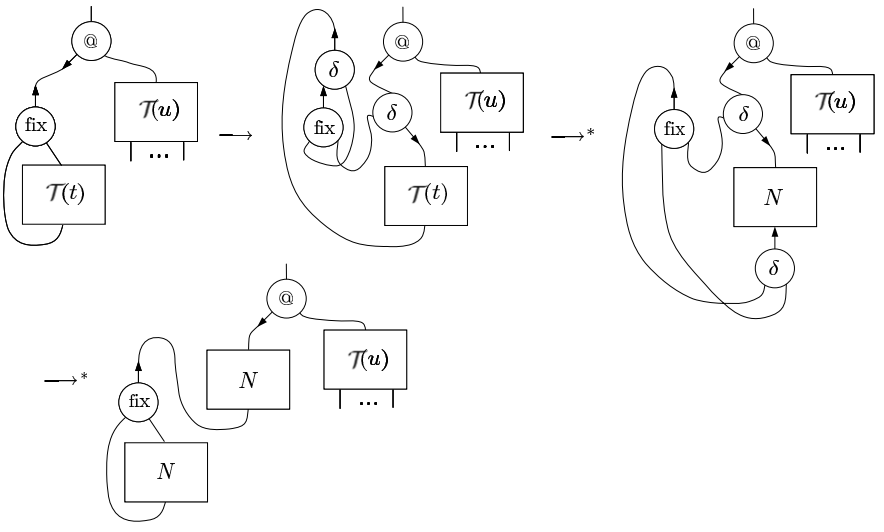
The translation of `fix` $f.t$, $\mathcal{T}(\texttt{fix}\ f.t)$, is shown below.



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

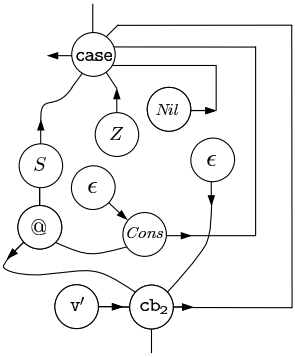
Proof. We assume $\mathcal{T}(t)$ has a normal form N which contains no cycles of principal ports. Then, from $\mathcal{T}((\texttt{fix}\ f.t)u)$ we can perform the following reduction:

$\mathcal{T}((\texttt{fix}\ f.t)u)$



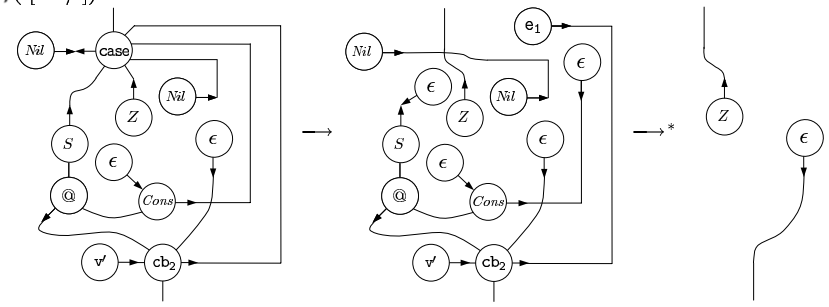
The resulting net can be obtained from $\mathcal{T}((t\{f := \texttt{fix}\ f.t\})u)$ by reducing $\mathcal{T}(t)$ to a normal form N. □

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

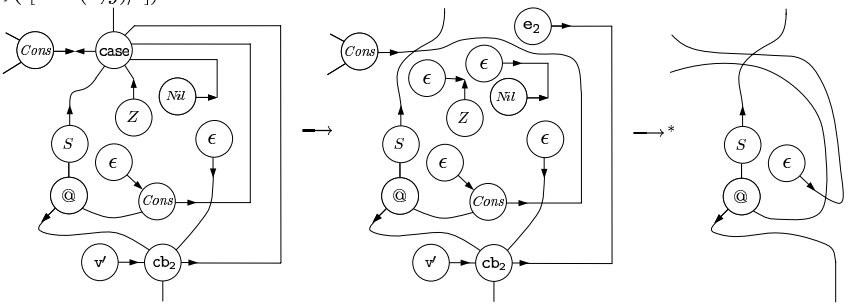


For $\mathcal{T}(t\{l := Nil\})$ and $\mathcal{T}(t\{l := Cons(x,y)\})$, we can perform the following reductions:

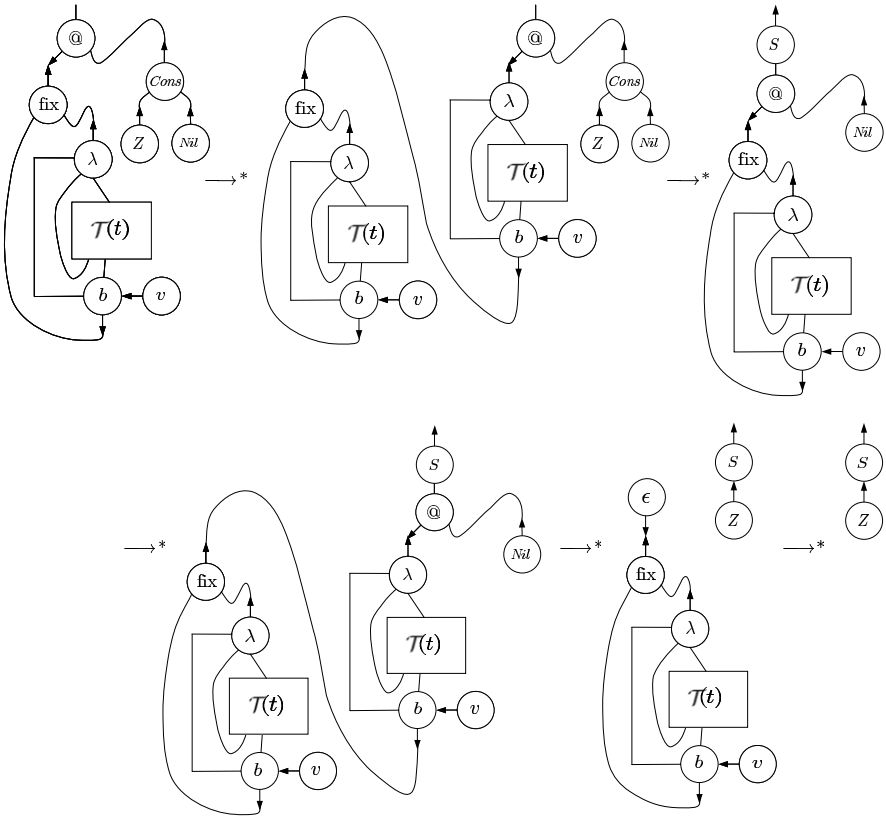
$\mathcal{T}(t[Nil/l])$



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



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



Operationally, the agent fix plays the role of a "controller", it allows us to generate precisely the number of recursive calls that are needed to evaluate a recursive function on a given argument. The alternative approach using a "cycle" is based on the idea that we can implement recursion by generating a potentially infinite sequence of calls (via the cyclic net), relying on some external mechanism (e.g. a strategy) to stop the reduction process when the result has been found. Using the agent fix, we avoid global operations, but the price to pay is the extra interactions of the agent fix. This is a constant number only for each recursive call and thus does not affect the performance of the compiler. With the cyclic approach, there would be an overhead for *every* rewrite step.

5 Conclusion

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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