Token-passing Nets for Functional Languages

José Bacelar Almeida, Jorge Sousa Pinto, Miguel Vilaça [1]

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

Abstract

Token-passing nets were proposed by Sinot as a simple mechanism for encoding evaluation strategies for the λ-calculus in interaction nets. This work extends token-passing nets to cover a typed functional language equipped with structured types and unrestricted recursion. The resulting interaction system is derived systematically from the chosen big-step operational semantics. Along the way, we actually characterize and discuss several design decisions of token-passing nets and extend them in order to achieve simpler interaction net systems with a higher degree of embedded parallelism.

Keywords: Interaction nets, reduction strategies, λ-calculus, recursion.

1 Introduction

Interaction nets [7] constitute a Turing-complete computational paradigm, where computation is purely local, and thus (strong) confluence holds.

The linear λ-calculus can be very naturally encoded in interaction nets with just two symbols. When one drops the linearity restriction however, things become more subtle: since variable substitution is implemented within the formalism, and not as an external meta-operation, copying and erasure of terms must be dealt with explicitly. Many encodings have been studied (e.g. [9,10,8]), some of which allow for a great degree of sharing of computations.

Token-passing nets [11] were proposed as a simple mechanism for encoding the most common evaluation strategies for the λ-calculus in the interaction net framework. One of the most attractive features is their simplicity, allowing computations to be easily traced in the term syntax. This makes them particularly well-suited for debugging or educational purposes.

The purpose of this paper is two-fold:

(i) To explicitly characterize token-passing nets as a class of interaction nets;

[1] Emails: {jba,jsp,jmvilaca}@di.uminho.pt. The work of the 3rd. author was funded by FCT grant SFRH / BD / 18874 / 2004.

(ii) to extend the token-passing encoding of the λ-calculus to a typed functional language with structured types. In particular, we propose a novel way of encoding arbitrary recursion with a fixpoint construct.

Sinot imposes a linearity restriction on the evaluation token, which forces evaluation to proceed sequentially. A novelty of our approach to token-passing is that sequentiality is not taken to be a defining attribute, since we believe that relaxing the sequentiality constraint gives rise to more natural (and considerably simpler) encodings for the strategies considered. This allows us to encode parallel call-by-value for our functional language.

For the sake of generality, we use the framework of Combinatory Reduction Systems to specify the syntax of our terms with binding, and we give a generic encoding of CRS terms into a class of syntactical nets. CRS syntax gives us a generic way of combining first-order term rewriting with λ-style bindings, which is very appropriate for our development in this paper.

The paper is structured as follows: Section 2 reviews basic notions of interaction nets and encodings of the λ-calculus, including the token-passing encoding. Section 3 introduces a notion of nets capable of representing the syntax of a large class of term languages with binding. Section 4 is devoted to the characterization of a general class of token-passing nets and systems, and to the description of the token-passing system for the λ-calculus. We proceed to give in Section 5 an encoding of a functional language with recursion. Section 6 concludes with some remarks, extensions, and pointers to further work.

2 Interaction Nets and λ-Calculus Encodings

An interaction net system [7] is specified by giving a set Σ of symbols, and a set $\mathcal{R}$ of interaction rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*. An occurrence of a symbol $\alpha \in \Sigma$ will be called an *agent*. If the arity of α is n, then the symbol has $n+1$ *ports*: a distinguished one called the *principal port*, and n *auxiliary ports* labelled $x_1, \ldots, x_n$. A net built on Σ is a graph (not necessarily connected) where the nodes are agents. The edges between nodes of the graph are connected to ports in the agents, such that there is at most one edge connected to every port in the net. Edges may be connected to two ports of the same agent. Principal ports of agents are depicted by an arrow. The ports where there is no edge connected are called the *free ports* of the net. The set of free ports define the *interface* of the net.

There are two special instances of a net: a wiring (a net containing no agents, only edges between free ports), and the empty net (containing no agents and no edges). The dynamics of Interaction Nets are based on the notion of *active pair*: any pair of agents (α, β) in a net, with an edge connecting together their principal ports. An *interaction rule* $((\alpha, \beta) \to N) \in \mathcal{R}$ replaces an occurrence of the active pair (α, β) by the net N. Rules must satisfy two conditions: the interfaces of the left-hand side and right-hand side are equal (this implies that the free ports are preserved during reduction), and there is at most one rule for each pair of symbols, so there is no ambiguity regarding which rule to apply.

If a net does not contain any active pairs then we say that it is in normal form. We use the notation $\to$ for one-step reduction and $\to^*$ for its transitive reflexive

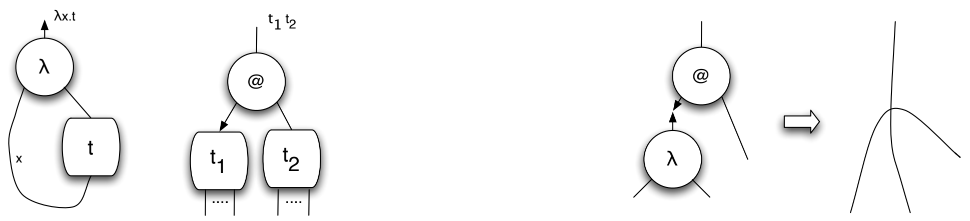


Fig. 1. Linear λ-calculus: Encoding and reduction rule

closure. Additionally, we write $N \twoheadrightarrow N'$ if there is a sequence of interaction steps $N \rightarrow^* N'$, such that N' is a net in normal form. The strong constraints on the definition of interaction rules imply that reduction is strongly commutative (the one-step diamond property holds), and thus confluence is easily obtained. Consequently, any normalizing interaction net is strongly normalizing.

Encoding the λ-calculus

The linear λ-calculus possesses a natural encoding in interaction systems. We consider two symbols λ and @ with arity 2. Principal ports for these symbols are chosen in such a way that a β-redex will correspond to an active pair in the interaction system. Figure 1 presents the translation of terms into nets. In short, variables become wires: a bound variable is connected to the corresponding binder (back-pointer in the syntax tree), and free variables are left dangling as free ports of the net. The reduction rule between these two symbols corresponds to β-reduction. It is captured by a proper rewiring of the free ports of the rule (also shown in Figure 1).

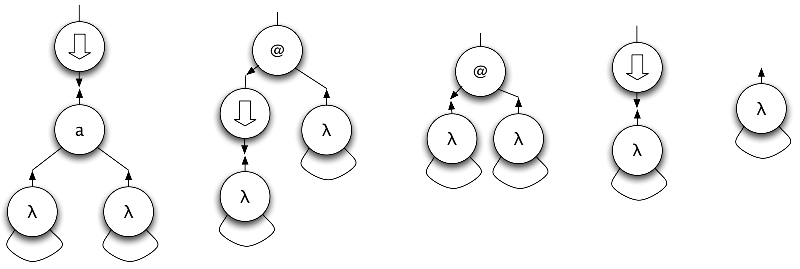
The simplicity of the encoding is certainly attractive and intuitive. However, when we leave the linearity assumption, things become considerably more subtle. In fact, interaction nets force an explicit treatment of the copying and erasing operations required by the non-linear use of variables. This is often accomplished by using specific symbols (usually denoted δ and ϵ) that implement these operations through sequences of local interactions. It is not however trivial to regulate this process in order to prevent different copying processes from interfering with each other in an unsound way.

The above mentioned difficulty motivated the proposal of complex encodings of λ-calculi in interaction nets (e.g. [2,10]). These encodings typically provide a setting where it is possible to delimit the scope of δs – the so called *boxes* (due to their close relationship to exponential boxes from Linear Logic proof-nets [4]).

The Token-passing Encoding

Token passing nets were introduced by Sinot [11] as a simple mechanism to encode the most common (rather than most efficient) strategies. The main insight is that the above mentioned difficulties may be overcome as long as duplication is restricted to nets that correspond to syntactical representations of terms: trees of agents, with principal ports at the roots.

Reduction strategies are encoded through the use of an *evaluation token* that traverses the syntactical representation of the term under consideration, in order to create active pairs and trigger computational (β) steps – at this point the nets no longer correspond to syntactical representations, so this process must be kept under control. The following figure illustrates the call-by-name evaluation of the term $(\lambda x.x)(\lambda x.x)$, using Sinot's framework. The token first enters the leftmost function, transforming the syntactic agent a into the computational @, and stops at that since $\lambda x.x$ is a canonical form. The next step is a computational (β) step, which reintroduces the token at the root of the term.



In what follows, we provide a rigorous characterization of token-passing systems: first, a description of the class of nets that act as syntactic entities, then of the admissible interaction rules. Even though our presentation diverges in several aspects from Sinot's, we believe we remain faithful to the underlying intuitions.

3 Syntactical Nets

We start with a characterization of syntactical nets, used to represent the syntax of a language of terms with binders. Syntactical nets are essentially syntax trees with backpointers that associate bound variables with their corresponding binding constructors. In the following, this class of nets will be defined as the image of a translation function from terms to interaction nets.

For concreteness, we adopt a term syntax inspired by Combinatory Reduction Systems (CRS) [6], which introduces an abstraction $[x]t$ denoting the binding of variable x in t. Taking the pure lambda-calculus as an example, we consider a binary constructor @ that makes no binding in its arguments, and a unary constructor λ that binds a variable in its argument. For instance, the lambda-term $\lambda x.xx$ is represented as $\lambda([x]@(x,x))$. Note however that we fix not only the arity of term constructors but also the number of binders in each of its arguments – a limitation usually not adopted in CRS. We use the standard definitions of free and bound variables, α-equivalence and substitution ($FV(t)$ denotes the set of free variables of t, $t\{t'/x\}$ denotes the substitution of free occurrences of x in t by t').

We start by defining the symbols used by the translation of terms to interaction nets. To each term constructor s with arity n we associate a symbol s with arity $n + \sum_{i=1}^{n} b_i$, where b_i is the number of variables bound in the i-th argument of s.

Given a term t, the translation constructs a syntatical net $\mathcal{T}(t)$ whose interface consists of a set of ports corresponding to the free variables of t (at the bottom of the net), one port corresponding to the root of the term (the rightmost port at the top), and a set of ports corresponding to variables that are not bound at the current constructor level, but have been abstracted by some constructor at an upper level. Thus the net $\mathcal{T}([x]@(x,x))$ has one port on top for the bound variable x, and

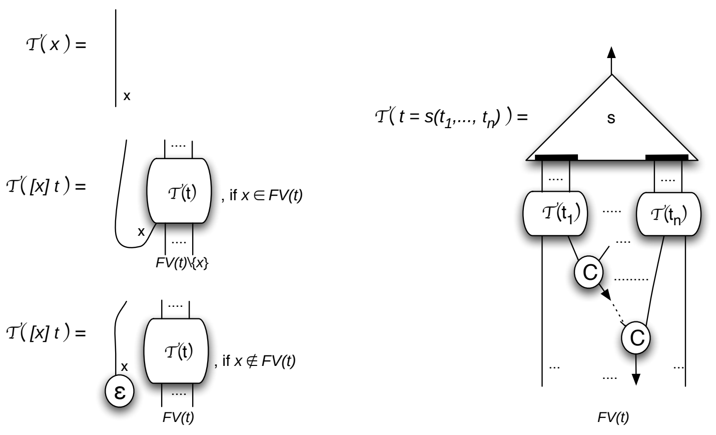


Fig. 2. Term translation

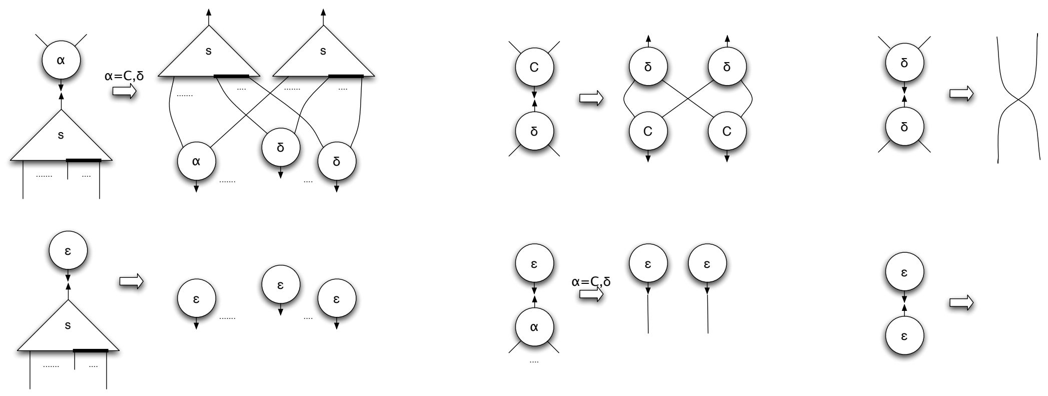


Fig. 3. Rules for symbols ϵ, c and δ

another representing the root of the term. The net $\mathcal{T}(\lambda([x]@(x,x)))$ has a single port, since the abstraction is now an argument of a constructor.

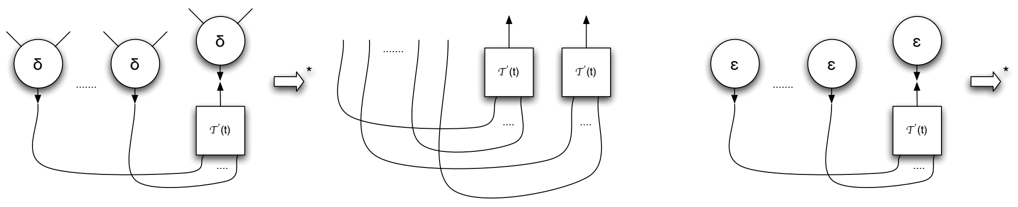
The translation is defined in Figure 2. Agents corresponding to occurrences of s are depicted as triangles, with their principal port on top. For the sake of clarity, bindings are represented explicitly by black bars that group together the auxiliary ports corresponding to each argument of s. As an example, the first net in figure 4 corresponds to $\mathcal{T}((\lambda x.x)\,((\lambda x.x)\,(\lambda x.x)))$.

Note that, to account for the non-linearity in terms, two new symbols are considered: ϵ (with arity 0), for discarding unused bound variables; and c (with arity 2) for sharing free variables between arguments of term constructors (glued together with spines of c agents). All principal ports are connected to auxiliary ports (except the root port of the net, which is free), which means that the constructed nets are in normal form.

Definition 3.1 A *syntactical net* is a net N such that $N = \mathcal{T}(t)$ for some term t.

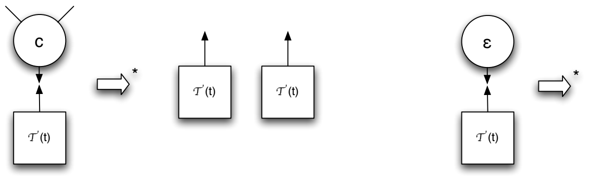
The dynamics of symbols ϵ and c is given in Figure 3, corresponding to management operations on syntactical nets: ϵ is used for erasing and c for copying. δ is a variant of c used for copying inside binders. The following properties justify the claim that syntactical nets are amenable to being manipulated by ϵ, δ and c.

Lemma 3.2 *For every open term t, the following reduction sequences exists:*



Proof. By induction on the structure of t. □

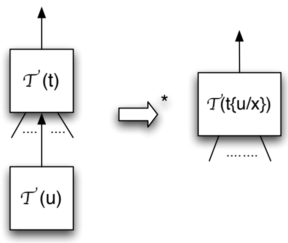
Proposition 3.3 *For every closed term t, the following reduction sequences exists:*



Proof. By induction on the structure of t and the previous result. □

The justification of the "substitution as rewiring" slogan may now be established.

Corollary 3.4 *For every term t and closed term u we have the following.*



Proof. Direct from the definition of substitution and the previous result. □

Note that the previous proposition does not hold if u is an open term, due to a mismatch between δ and c agents that occurs when open abstractions are copied.

4 Token-passing Nets

Having established a systematic way of representing terms with binding as interaction nets, we now turn our attention to the implementation of reduction strategies in the net framework. In what follows we outline how a call-by-value semantics can be implemented using the token-passing style, and exemplify this for the case of the λ-calculus. We then give a rigorous characterization of token-passing nets.

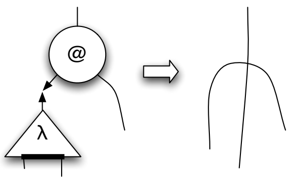
The redexes that can be simulated in the net framework in a straightforward way are quite restricted: redexes in the term language must correspond to active pairs between two term constructor agents at the level of nets. Let s be a syntactic symbol corresponding to the outermost term constructor in some redex. Active pairs can be created by introducing a new symbol $\hat{s}$ for each such s, and then replacing in a controlled way occurrences of s by $\hat{s}$, with their principal ports facing downwards.

Symbols $\hat{s}$ will be called *computation symbols*; their occurrences will be represented graphically as circles.

Let us take the pure λ-calculus again as an example. In CRS syntax, the β-reduction rule is written as:

$$@(\lambda([x]Z(x)), u) \to Z(u)$$

We must now introduce the computation symbol $\hat{\mathsf{a}}$ (denoted @). The appropriate interaction rule is of course very similar to the rule in Figure 1, involving the symbols (syntactic) λ and (computation) @.



Observe that computation agents are ephemeral since they are not re-introduced in the right-hand side of computation rules between constructor agents.

We now turn our attention to the evaluation process. To specify an evaluation order at the term level, the so-called big-step style of operational semantics (or natural semantics [5]) is particularly well-suited, since it is essentially syntax-directed. The following rules specify the call-by-value reduction strategy for the λ-calculus (the symbol $\Rightarrow$ denotes the *evaluates-to* relation).

$$\frac{}{\lambda x.t \Rightarrow \lambda x.t} \qquad \frac{u \Rightarrow \lambda x.t \qquad v \Rightarrow v' \qquad t\{v'/x\} \Rightarrow z}{u\ v \Rightarrow z}$$

The first rule tells us that abstractions are canonical forms. It is instructive to make the contraction of the β-redex $(u'\,v')$ explicit in the second rule:

$$\frac{u \Rightarrow u' \qquad v \Rightarrow v' \qquad u'\ v' \xrightarrow{\beta} t \qquad t \Rightarrow z}{u\ v \Rightarrow z}$$

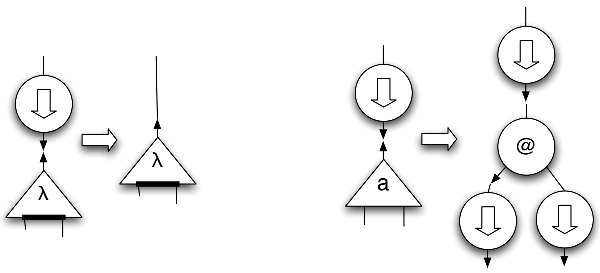
It becomes clear that the rule actually performs two different actions: recursive calls to the evaluation relation, and a β-reduction step. We've seen that β-reduction is treated at the net level by a separate computation rule, so now only the recursion pattern of the rule must be dealt with.

The evaluation semantics can alternatively be expressed as term rewriting rules by introducing in our syntax a new unary constructor $\Downarrow(\cdot)$ for evaluation, and binary constructor $\mathsf{a}(\cdot,\cdot)$ for application (in its syntactic form). In CRS syntax we have

$$\begin{aligned} \Downarrow(\lambda([x]Z(x))) &\to \lambda([x]Z(x)) \\ \Downarrow(\mathsf{a}(u,v)) &\to \Downarrow(@(\Downarrow(u),\Downarrow(v))) \end{aligned}$$

At the net level, evaluation can be controlled by occurrences of a special symbol called a *token*, denoted as $\Downarrow$, with arity 1. Evaluation of a closed term t will then be triggered by connecting the principal port of a token agent $\Downarrow$ to the root of the translation of t into a syntactical net, denoted by $\Downarrow(\mathcal{T}(t))$.

The above rules translate directly into the following interaction rules which we call *evaluation* rules (note that the token has its principal port oriented downwards).



These rules involving the token $\Downarrow$ and the set of syntactic symbols determine how evaluation is performed through the syntax. They may activate a syntactic agent in order to make computational steps possible, and send a new $\Downarrow$ token along the principal port of the activated agent. This guarantees that a computational active pair (i.e. an active pair which constitutes the left-hand side of some computational rule) will eventually be created.

As an example of token-passing evaluation, Figure 4 shows two possible reduction sequences for the λ-term $(\lambda x.x)\ ((\lambda x.x)\ (\lambda x.x))$ in this system. The one shown on top simulates precisely a sequential call-by-value evaluation of the term. The bottom one shows a sequence where the computational rule associated to the outermost redex is applied prior to the evaluation of the argument.

We will now formally define the notions of token-passing rule, system and net.

Definition 4.1 A *computation rule* is an interaction rule between a syntactic symbol and a computation symbol. The right-hand side net of a computation rule cannot contain occurrences of computation symbols.

An *evaluation rule* is an interaction rule between the token symbol $\Downarrow$ and some

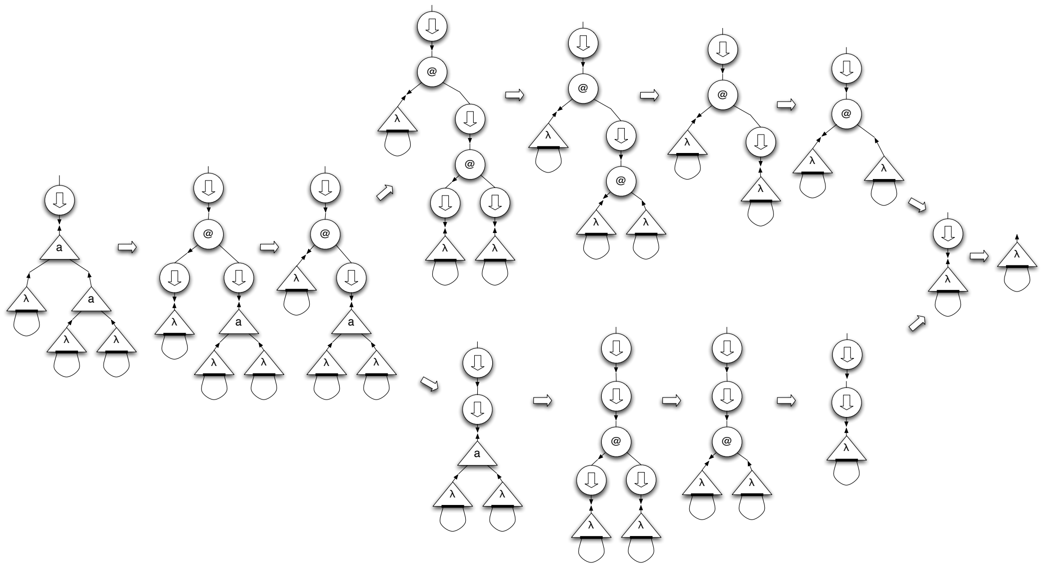


Fig. 4. Reduction sequence for $(\lambda x.x)\ ((\lambda x.x)\ (\lambda x.x))$

syntactic symbol s. Let $\hat{s}$ be the computation symbol corresponding to the syntactic symbol s. An evaluation rule is either a *canonical* or an *activation* rule:

- In a *canonical rule* the net in the right-hand side contains an occurrence of s, and possibly occurrences of the token symbol.
- In an *activation rule* the net in the right-hand side still contains an occurrence of the token connected to the root, with its principal port connected to an occurrence of $\hat{s}$. Furthermore, a second occurrence of the token is connected to the principal port of $\hat{s}$. The net may additionally contain further occurrences of the token connected to (other auxiliary ports of) $\hat{s}$.

Canonical rules correspond to the evaluation of terms that are already canonical forms, or whose evaluation to canonical form requires mere recursive evaluation of subterms, in which case new token agents will be connected to auxiliary ports of the s agent. For terms whose evaluation requires more sophisticated rewriting, an activation rule is needed to enable the application of a subsequent computation rule.

We remark that if a weak notion of reduction is assumed (as was the case with call-by-value for the λ-calculus), activation rules do not introduce $\Downarrow$ agents under binding constructors, corresponding to canonical forms. In this case, whenever an activation rule is applied during the evaluation of a closed term, only syntactic nets corresponding to closed terms are connected to the activated agent – Proposition 3.3 can thus be applied. In the rest of the paper we will consider only weak strategies.

Definition 4.2 A *token-passing system* is an interaction system $\langle \Sigma, \mathcal{R} \rangle$ with $\Sigma = \{\Downarrow, c, \delta, \epsilon\} \cup S \cup C$ where S is a set of syntactic symbols as introduced in Section 3, and C is a set of computation symbols such that each $\hat{s} \in C$ is associated to a unique and distinct syntactic symbol $s \in S$, with the same arity.

The set of interaction rules $\mathcal{R} = \mathcal{C} \cup \mathcal{E}_C \cup \mathcal{E}_A \cup \mathcal{M}$ consists of computation rules ($\mathcal{C}$), canonical rules ($\mathcal{E}_C$), activation rules ($\mathcal{E}_A$), and management rules ($\mathcal{M}$), such that $\mathcal{E}_A \cup \mathcal{E}_C$ contains an evaluation rule for each term constructor. For each syntatical symbol and for each management symbol, $\mathcal{M}$ contains a rule according to the templates of Figure 3. This guarantees that normal forms do not contain tokens and are thus syntactical nets.

Definition 4.3 A *token-passing net* is a net N' constructed in a token-passing system, such that $\Downarrow(N) \to^* N'$ for some syntactical net N.

We end the section with a note on the encoding of recursion. There is scope in the definition of token-passing nets for recursive constructs. These may be handled either by computation rules that re-introduce an occurrence of the syntactic symbol in the right-hand side, or by canonical rules. The former approach can be used for recursion patterns such as iterators or recursors, which perform pattern-matching on their argument [1]. The latter approach is explored in the next section to encode a general binding recursion operator.

5 Encoding of a Functional Language

We now present the encoding of a typed functional language. For the sake of simplicity, we choose to keep the language rather small – it includes types for booleans

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau} \qquad \frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash t\ u : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathbb{B}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathbb{B}} \qquad \frac{\Gamma \vdash b : \mathbb{B} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}$$

$$\frac{}{\Gamma \vdash [] : [\tau]} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : [\tau]}{\Gamma \vdash (e_1 :: e_2) : [\tau]} \qquad \frac{\Gamma \vdash t : [\tau] \quad \Gamma \vdash e_1 : \sigma \quad \Gamma, x : \tau, y : [\tau] \vdash e2 : \sigma}{\Gamma \vdash (\mathsf{Case}\ t\ \mathsf{of}\ [] \to e_1\ ;\ (x :: y) \to e_2) : \sigma}$$

$$\frac{\Gamma, y : \tau \vdash t : \tau}{\Gamma \vdash (\mathsf{Rec}\ y.t) : \tau}$$

$$\frac{}{V \Rightarrow V}\ V \in \{\mathsf{true}, \mathsf{false}, [], \lambda x.t\} \qquad \frac{e_1 \Rightarrow w \quad e_2 \Rightarrow z}{e_1 :: e_2 \Rightarrow w :: z}$$

$$\frac{t \Rightarrow \mathsf{true} \quad e_1 \Rightarrow z}{\mathsf{if}\ t\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Rightarrow z} \qquad \frac{t \Rightarrow \mathsf{false} \quad e_2 \Rightarrow z}{\mathsf{if}\ t\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Rightarrow z}$$

$$\frac{l \Rightarrow [] \quad e_1 \Rightarrow z}{\mathsf{Case}\ l\ \mathsf{of}\ [] \to e_1\ ;\ (h :: t) \to e_2 \Rightarrow z} \qquad \frac{l \Rightarrow w :: y \quad e_2\{w/h,\ y/t\} \Rightarrow z}{\mathsf{Case}\ l\ \mathsf{of}\ [] \to e_1\ ;\ (h :: t) \to e_2 \Rightarrow z}$$

$$\frac{u \Rightarrow \lambda x.t \quad v \Rightarrow w \quad t\{w/x\} \Rightarrow z}{u\ v \Rightarrow z} \qquad \frac{f\{\mathsf{Rec}\ y.f/y\} \Rightarrow z}{\mathsf{Rec}\ y.f \Rightarrow z}$$

Fig. 5. Typing and Evaluation rules for BLR

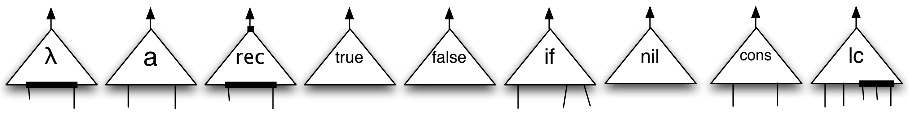
and lists, pattern-matching and general recursion (hence its name BLR). However, it should be fairly evident that the proposed encoding extends smoothly to additional types or type constructors.

Terms and types for the language are given by the following grammar:

$$\begin{aligned}
\mathcal{T} = {} & \mathcal{V} \mid \lambda\mathcal{V}.\mathcal{T} \mid \mathcal{T}\ \mathcal{T} \mid \\
& \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ \mathcal{T}\ \mathsf{then}\ \mathcal{T}\ \mathsf{else}\ \mathcal{T} \mid \\
& [] \mid \mathcal{T} :: \mathcal{T} \mid \mathsf{Case}\ \mathcal{T}\ \mathsf{of}\ [] \to \mathcal{T}\ ;\ (\mathcal{V} :: \mathcal{V}) \to \mathcal{T} \mid \\
& \mathsf{Rec}\ \mathcal{V}.\mathcal{T} \\
\mathrm{Tp} = {} & \mathbb{B} \mid [\mathrm{Tp}] \mid \mathrm{Tp} \to \mathrm{Tp}
\end{aligned}$$

where $\mathcal{V}$ denotes a set of variables. Type assignment and evaluation rules for the language are given in Figure 5. The evaluation strategy shown is *call-by-value*, as visible in the application or the list-cons rule. Note however that pattern matching (*if-then-else* and *list-case* expressions) has as usual a *call-by-name* flavour to allow for an effective use of recursion (see, e.g. [14]).

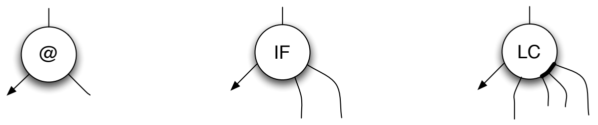
We will now design a token-passing system for this language by addressing each of the points identified in the previous sections. To encode the syntax, we shall consider an symbol for each term constructor of the language. Observe that the language includes additional binding constructors besides λ. In fact, the typing rules clearly state that the third argument of the *list-case* constructor binds two variables, and the single argument of the *rec* constructor binds one variable. Taking this into account, we are led to the following choice of syntactical symbols (we use nil for [], cons for :: and lc for pattern-matching over lists).



Turning now to the dynamics of the system, we start by writing a set of term-rewriting rules corresponding to the evaluation semantics given. The evaluation rules for lists are written in CRS syntax as

$$\begin{aligned}
\Downarrow(\mathsf{nil}) &\to \mathsf{nil}\\
\Downarrow(\mathsf{cons}(x,y)) &\to \mathsf{cons}(\Downarrow(x),\Downarrow(y))\\
\Downarrow(\mathsf{lc}(l,e_1,e_2)) &\to \Downarrow(\mathsf{LC}(\Downarrow(l),e_1,e_2))\\
\mathsf{LC}(\mathsf{nil},x,[a,b]Z(a,b)) &\to x\\
\mathsf{LC}(\mathsf{cons}(x,y),z,[a,b]Z(a,b)) &\to Z(x,y)
\end{aligned}$$

Two more computation symbols are now introduced in the system: $\hat{\mathsf{lc}}$ (denoted LC) and $\hat{\mathsf{if}}$ (denoted IF). Note that LC is an example of a computation symbol with binding. The full set of (activated) computation symbols is thus:



The recursion rule translates directly to a rewriting rule as

$$\Downarrow(\mathsf{rec}([y]Z(y))) \to \Downarrow(Z(\mathsf{rec}([y]Z(y))))$$

The rules of the token-passing system for BLR are given in Figure 6, where the first two rows contain computation rules, and the next two rows contain respectively canonical and activation rules. Observe that non-linear uses of variables or meta-variables in the rules are explicitly handled by the use of the agents ϵ, c and δ. More precisely, erasing and copying of variables are handled by ϵ and c agents respectively (Proposition 3.3), and of meta-variables by ϵ and δ agents (Lemma 3.2).

The correctness of the encoding is stated in the following proposition.

Proposition 5.1 *For every closed term t such that $t \Rightarrow z$ (with derivation tree τ), there exists a reduction sequence $\Downarrow(\mathcal{T}(t)) \twoheadrightarrow \mathcal{T}(z)$. Moreover, the computational steps in this sequence correspond precisely to the reductions found in τ.*

Proof. We construct the reduction sequence by induction on the derivation of $t \Rightarrow z$ (τ):

- If $t \in \{\lambda x.t', \mathsf{true}, \mathsf{false}, []\}$, then $t \Rightarrow t$. On the other hand, $\Downarrow(\mathcal{T}(t)) = \mathcal{T}(t)$ (by the corresponding interaction rules).
- If $t = h :: u$ then $t \Rightarrow w :: v$ where $h \Rightarrow w$ and $u \Rightarrow v$. On the other hand, $\Downarrow(\mathcal{T}(h :: u)) \to (\Downarrow(\mathcal{T}(h)) :: \Downarrow(\mathcal{T}(u)))$ by $(\Downarrow, \mathsf{cons})$-rule which reduces to $(\mathcal{T}(w) :: \mathcal{T}(v)) = \mathcal{T}(w :: v)$ by induction hypothesis.

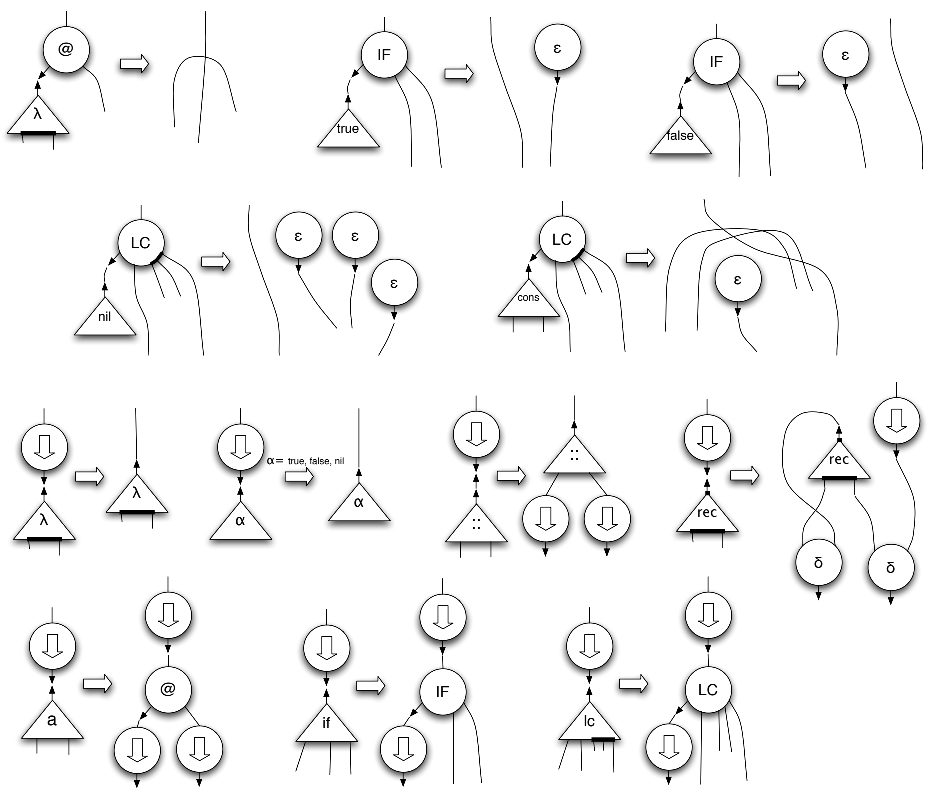


Fig. 6. Reduction rules for BLR

- If $t = (u\ v)$ then $t \Rightarrow z$ where $u \Rightarrow \lambda x.t'$, $v \Rightarrow v'$ and $t'\{v'/x\} \Rightarrow z$. By $(\Downarrow, \mathsf{a})$-rule, we have that $\Downarrow(\mathcal{T}(t)) \rightarrow \Downarrow(@(\Downarrow(u), \Downarrow(v)))$ which, by induction hypothesis, reduces to $\Downarrow(@(\lambda x.t', v')$. Applying $(@, \lambda)$-rule and Corollary 3.4, it reduces to $\Downarrow(t'\{v'/x\})$ that reduces to $\mathcal{T}(z)$ (again by induction hypothesis).
- If t is a pattern-matching construct: we illustrate for the case where t = $\mathsf{Case}\ l\ \mathsf{of}\ [] \rightarrow e_1\ ;\ (x{::}y) \rightarrow e_2$ and $l \Rightarrow u{::}v$ (other cases are similar). We have $t \Rightarrow z'$ where $e_2\{u/x,\ v/y\} \Rightarrow z$. Applying $(\Downarrow, \mathsf{lc})$-rule and induction hypothesis we get $\Downarrow(\mathcal{T}(t)) \rightarrow \Downarrow(\mathsf{LC}(\Downarrow(l), \mathcal{T}(e_1), [x, y]\mathcal{T}(e_2))) \rightarrow^*$ $\Downarrow(\mathsf{LC}(\mathsf{cons}(\mathcal{T}(u), \mathcal{T}(v)), \mathcal{T}(e_1), [x, y]\mathcal{T}(e_2))$. By $(\mathsf{LC}, \mathsf{cons})$-rule and Corollary 3.4 it reduces to $\Downarrow(e_2\{u/x,\ v/y\})$ which, by induction hypothesis reduces to $\mathcal{T}(z)$.
- If $t = \mathsf{Rec}\ y.f$ then $t \Rightarrow z$ where $f\{\mathsf{Rec}\ y.f/y\} \Rightarrow z$. Applying $(\Downarrow, \mathsf{rec})$-rule, Lemma 3.2 and Corollary 3.4 it reduces to $\Downarrow(\mathcal{T}(f\{\mathsf{Rec}\, y.f/y\})$ which, by induction hypothesis, reduces to $\mathcal{T}(z)$.

The argument that the computational steps correspond to reductions in τ follows from the construction of the sequence (in each inductive step, a reduction is simulated through the appropriate application of a computational rule). □

Note that the reduction sequence constructed in the proof simulates the sequential call-by-value strategy. However, the system allows for a considerable degree of freedom in the order of interactions, not only in the administrative tasks but also in the evaluation itself (Figure 4 illustrates the potential for parallelism). By strong confluence these sequences are all permutations of the standard one.

Proposition 5.2 *For every closed term t, if $\Downarrow(\mathcal{T}(t)) \twoheadrightarrow Z$, then $Z = \mathcal{T}(z)$ and $t \Rightarrow z$.*

Proof. The confluence of the system may be explored to rearrange the given reduction sequence. In particular, we postpone the application of rules involving agents @, IF and LC (computational rules) to when these agents have their arguments in normal form. Also, when dealing with a reduction sequence that acts on two disjoint subnets, we split it in two halves containing the interactions of each subnet.

As a notational convenience, we denote nets by their corresponding CRS terms (e.g. $@(N_1, N_2)$ denotes the net obtained by connecting the @ agent to N_1 to its principal port and N_2 to its first auxiliary port). The proof proceeds by induction on the length of the reduction sequence and case analysis:

- If $t \in \{\lambda x.t', \mathsf{true}, \mathsf{false}, []\}$ then $\Downarrow(\mathcal{T}(t)) \twoheadrightarrow \mathcal{T}(t)$. On the other hand $t \Rightarrow t$.
- If $t = h :: u$ then $\Downarrow(\mathcal{T}(t)) \to \mathsf{cons}(\Downarrow(\mathcal{T}(h)), \Downarrow(\mathcal{T}(u)))$. Since the nets $\mathcal{T}(h)$ and $\mathcal{T}(u)$ are disjoint, we split the remainder sequence in $\Downarrow(\mathcal{T}(h)) \twoheadrightarrow W$, $\Downarrow(\mathcal{T}(u)) \twoheadrightarrow V$ where $Z = \mathsf{cons}(W, V)$. By induction hypothesis and applying the appropriate evaluation rule we get $t \Rightarrow z = w::v$ where $W = \mathcal{T}(w)$ and $V = \mathcal{T}(v)$.
- If $t = (u\ v)$ then $\Downarrow(\mathcal{T}(t)) \to \Downarrow(@(\Downarrow(\mathcal{T}(u)), \Downarrow(\mathcal{T}(v))))$. By assumption, we postpone the application of the rule involving the introduced @ agent as much as possible. Let us consider the first fragment of the remainder sequence that goes till the application of that particular rule. Since $\mathcal{T}(u)$ and $\mathcal{T}(v)$ are disjoint, we split it in $\Downarrow(\mathcal{T}(u)) \twoheadrightarrow U'$ and $\Downarrow(\mathcal{T}(v)) \twoheadrightarrow V'$. Therefore, by induction hypothesis, $u \Rightarrow u'$ and $v \Rightarrow v'$ where $U' = \mathcal{T}(u')$ and $V' = \mathcal{T}(v')$. We turn now the attention to what rests in the original sequence. After the application of $(@, \lambda)$-rule (well-typedness of t force $u' = \lambda x.t'$), we postpone the interaction with the topmost $\Downarrow$ agent — that would guaranty that Corollary 3.4 applies and we get $@(\mathcal{T}(\lambda x.t'), \mathcal{T}(v')) \twoheadrightarrow \mathcal{T}(t'\{v'/x\})$. Finally, again by induction hypothesis, we use what rests in the sequence to get $t'\{v'/x\} \Rightarrow z$. Summing up, and applying corresponding evaluation rule, we get $t \Rightarrow z$ with $Z = \mathcal{T}(z)$.
- If t is a pattern-matching construct: we illustrate for the case where $t = \mathsf{Case}\ l\ \mathsf{of}\ [] \to e_1\ ;\ (x::y) \to e_2$ (the other cases are similar). We have that $\Downarrow(\mathsf{lc}(\mathcal{T}(l), \mathcal{T}(e_1), [x, y]\mathcal{T}(e_2))) \to \Downarrow(\mathsf{LC}(\Downarrow(\mathcal{T}(l)), \mathcal{T}(e_1), \mathcal{T}([x, y]e_2)))$. By assumption we postpone the application of the LC rule and, therefore the first part of the sequence is $\Downarrow(\mathcal{T}(l)) \twoheadrightarrow L'$. By induction hypothesis $l \Rightarrow l'$ and $L' = \mathcal{T}(l')$. By well-typdness, l' should be either $[]$ or $u::v$ – we consider the second alternative (the first one is similar). After the application of $(\mathsf{LC}\text{-}\mathsf{cons})$-rule, we postpone the reduction involving the topmost $\Downarrow$ agent. Then by Lemma 3.3 and Corollary 3.4 we get $\mathsf{LC}(\mathsf{cons}(\mathcal{T}(u), \mathcal{T}(v)), \mathcal{T}(e_1), \mathcal{T}([x, y]e_2)) \twoheadrightarrow \mathcal{T}(e_2\{u/x,\ v/y\})$. By induction hypothesis on the remainder of the sequence and applying the corresponding evaluation rule finally give $t \Rightarrow z$ with $Z = \mathcal{T}(z)$.
- If $t = \mathsf{Rec}\ y.\mathcal{T}(f)$, we assume that after the application of $(\Downarrow, \mathsf{rec})$ the reduction involving the topmost $\Downarrow$ is postponed. By Lemma 3.2 and Corollary 3.4 we get $\mathcal{T}(t) \to\ \mathcal{T}(f)\{\mathcal{T}(\mathsf{Rec}\ y.f)/y\}$. Then, by induction hypothesis on the remainder of the sequence and the appropriate evaluation rule we get $t \Rightarrow z$ with $Z = \mathcal{T}(z)$.

□

6 Discussion

We have studied a new system for token-passing by extending the encoding of the λ–calculus in this paradigm to a richer language, much more suited for real world programming. Although only a small fragment is presented in this paper, other extensions have already been coded and proved to be simple to implement.

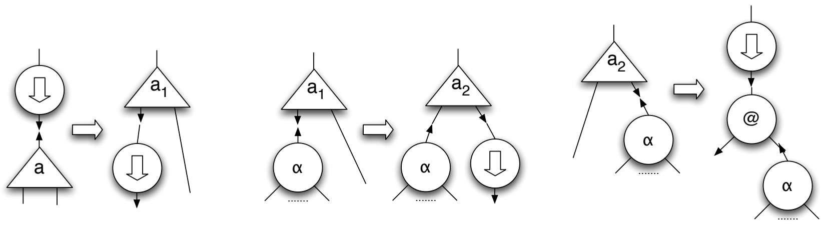
We remark that we have not tried in this paper to solve problems related with the limitations of interaction nets to encode CRS term-rewriting in general; we are certainly unable to cover rules involving more than two constructors, or sets of rules in which the same external constructor is involved in rewriting with different arguments in different rules (necessary to encode parallel or). Several extensions of interaction nets have been studied with these goals in mind, which could perhaps be adapted to our setting.

Our approach to token-passing nets differs in several aspects from their original presentation. In particular, from the methodological point of view, we avoid the "small step semantics" used in [11,13] as an intermediate step in the conversion from big-step rules to token-passing nets, since it did not allow us to reason about management (copying and erasing) operations. A second difference is that in [12], the author deliberately avoids a characterization of token-passing nets, which we have precisely attempted to give here.

Sequentiality

We have also relaxed the sequential impositions of the original presentation, corresponding to a linearity constraint on the evaluation token. Sequentiality forces the existence of symbols that are small variations on other symbols. Removing this constraint we are led to considerably smaller and more intuitive interaction systems, without the extra symbols and bureaucracy.

We will now illustrate how sequentiality might be recovered. Consider again the evaluation rule for application. We have seen that the corresponding interaction rule introduces three evaluation agents $\Downarrow$. In order to satisfy the linearity restriction, one should decompose the rule into 3, each introducing a $\Downarrow$ in turn.



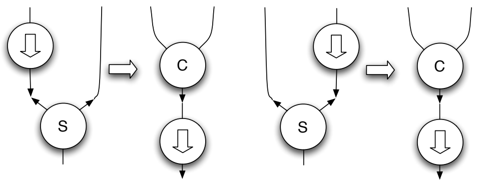
In these rules, α denotes an arbitrary agent. Note the need for additional agents (a_1 and a_2) that are analogous to a but with a different choice of principal port. These agents are responsible for scheduling the application of the several steps.

Call-by-need

In our presentation we have adopted the call-by-value strategy. All the development could however be reproduced for call-by-name, which would in fact be slightly easier.

More refined evaluation strategies such as *call-by-need* require additional efforts. The problem lies in the fact that the corresponding big-step rules are no longer purely syntactic – they incorporate evaluation contexts or side conditions. In order to capture these extended rules, we could either extend the language syntax to internalise these features, or adapt the "internals" of the interaction system to capture the intended semantics.

We will now follow the second approach and describe how Sinot's treatment of call-by-need might be considered in our setting. This treatment [13] can be summarized as follows: starting from the *call-by-name* token passing system, one modifies the translation of terms to replace the symbol c by a new symbol s, used to signal sharing. This symbol s is not a pure interaction-net symbol – it possesses two principal ports and only one auxiliary port. Its reduction rules are



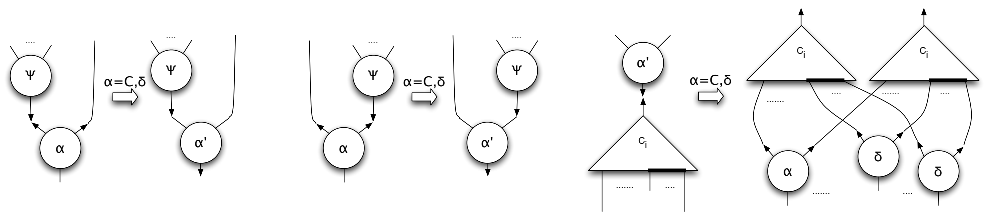
Once again, this system is considerably simpler than the one presented in [13], due to the absence of a linearity restriction on the evaluation token.

We omit a detailed proof of the correctness of this encoding (and some technicalities involving the interaction of s with δ). As noted in [13], the departure from the interaction net formalism does have an associated cost – confluence is no longer immediate from the locality/independence of interaction. In fact, the system actually fails to be confluent, which is harmless since non-confluence is limited to non-normalizing terms interacting with ϵ.

Lazy Copying

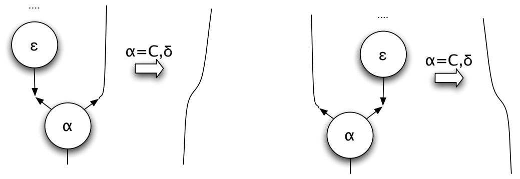
In the proof of Proposition 5.2, in order to reason about the reduction sequence, an interleaving where the administrative tasks (copy and erasure) are performed eagerly is considered. But it is interesting to consider the opposite scenario, where these tasks are postponed until they are really needed. This can be enforced by the system as long as we are willing to accept a generalization of the interaction net formalism allowing symbols with multiple principal ports.

The overall approach is quite simple: the principal and secondary ports of symbols δ and c are swapped. In this way, these symbols would have two principal ports that could monitor if any agent is wiling to interact with what is meant to be copied. Only when such an agent is detected, one copy step is performed (by variants of the mentioned symbols: δ' and c'). The relevant rules would be

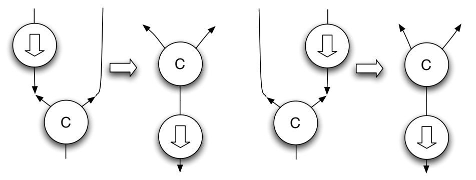


where Ψ denotes an arbitrary agent.

We conjecture that this system is confluent, when restricted to valid nets. Thus this extension does not compromise the meta-properties of the system, as long as we restrict attention to token passing nets. In fact, the only consequence would be to allow normal forms that possess some kind of *sharing* (syntactical nets with unperformed duplications). It also allows us to accommodate an optimization in some reduction sequences by specializing the interaction with ϵ:



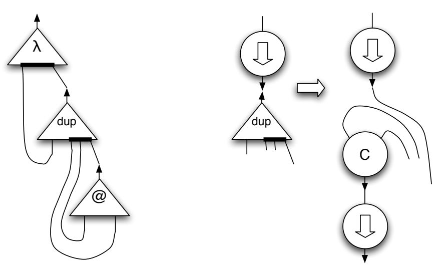
This second extension still doesn't disturb confluence but the system is no longer strongly confluent. The relevance of this extension is further strengthened when it is noted that such an explicit representation of sharing allow us to easily accommodate a *call-by-need* reduction strategy. In fact, starting with call-by-name evaluation rules, a further specialization for the interaction with $\Downarrow$ and c would allow to perform a single evaluation on shared sub-terms. Namely:



But now, and since evaluation might not terminate, we no longer have confluence when considering the ϵ optimization mentioned above. Note that this is essentially Sinot's call-by-need token passing system – in fact, it is arguably closer to the standard evaluation system as it captures sharing of results.

Extensions and Future Work

The present work suggests alternatives to the handling of sharing that do not imply leaving the interaction net formalism. One might replace the sharing symbol s by a proper syntax constructor dup that duplicates a free occurrence of a variable x by two fresh ones y, z. The following figure shows an example of use of this agent and the corresponding evaluation rule.



Of course, evaluation is triggered by duplication, and we do not have any guarantees that the result will actually be required by the computation. But even if the resulting strategy is "more eager" than call-by-need, the more structured nature of syntactical nets might lead to greater flexibility in the encoding of complex strategies. It would be interesting to explore the applicability of this idea to encode closed reduction as defined by Fernández et al. [3].

Additional language features and/or different evaluation strategies may be incorporated in a fairly structured manner, as exemplified for the case of sequential evaluation or the call-by-need strategy.

We are currently working on using the proposed system in parallel implementations, exploring the fact that token-based evaluation promotes a fairly controlled propagation of interaction through the net. Moreover, the token agents themselves always act as single points of contact between distinct sub-nets and are thus natural candidates for *synchronization points* between different reduction threads.

References

[1] José B. Almeida, Ian Mackie, Jorge S. Pinto, and Miguel Vilaça. Encoding Iterators in Interaction Nets (extended abstract). Unpublished.

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

[3] Maribel Fernández, Ian Mackie, and François-Régis Sinot. Closed reduction: explicit substitutions without alpha-conversion. *Mathematical Structures in Computer Science*, 15(2):343–381, 2005.

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

[5] Gilles Kahn. Natural semantics. In *STACS 87, Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 1987*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[6] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theor. Comput. Sci.*, 121(1&2):279–308, 1993.

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

[8] Sylvain Lippi. Encoding left reduction in the lambda-calculus with interaction nets. *Mathematical Structures in Computer Science*, 12(6):797–822, 2002.

[9] Ian 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.

[10] Ian 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.

[11] François-Régis Sinot. Call-by-name and call-by-value as token-passing interaction nets. In Pawel Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2005.

[12] François-Régis Sinot. Call-by-need in token-passing nets. *Mathematical Structures in Computer Science*, 16(4):639–666, 2006.

[13] François-Régis Sinot. Token-passing nets: Call-by-need for free. *Electr. Notes Theor. Comput. Sci.*, 135(3):129–139, 2006.

[14] Glynn Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing. The MIT Press, 1993.