Encoding Iterators in Interaction Nets

José B. Almeida[1], Ian Mackie[2], Jorge Sousa Pinto[1], and Miguel Vilaça[1]

[1] Departamento de Informática / CCTC
Universidade do Minho, Braga, Portugal
[2] LIX, CNRS UMR 7161, École Polytechnique, 91128 Palaiseau Cedex, France

Abstract. We propose a method for encoding iterators (recursion operators) using interaction nets. The method can be used to obtain a visual notation for functional programs, and also to extend with recursion the many translations of the λ-calculus into interaction nets, which have been proposed as efficient implementation mechanisms. We exemplify the method with a number of list-processing examples that illustrate the application to practical functional programming. Our examples also show that the method seems to generate, from appropriate functional programs, many typical examples of interaction net programs.
Keywords: Recursion operators, interaction nets, visual programming.

1 Introduction

The use of visual notations for functional programs has long been an active research topic; the goal is to have a notation that can be used

1. as an intuitive input notation for interpreters and compilers;
2. to animate visually the execution of functional programs.

Both of these have applications in debugging, education, and everyday programming. Even though the functional paradigm seems in principle to be amenable to visual representation, no such notation exists that is widely used by the functional programming community.

In this paper we propose a graphical system for functional programming, based on token-passing interaction nets. The system offers an adequate solution for classic problems of visual notations, including the treatment of higher-order functions, pattern-matching, and recursion (based on the use of iterators and other recursion operators). The system implements a call-by-name semantics, with a straightforward correspondence between functional programs and graphical objects. Programs can be translated into graphical form (or constructed directly at the graphical level), evaluated using the operational semantics of the graphical formalism, and then converted back into textual form.

Technically the main contribution of the paper is an extension of Sinot's token-passing implementation of the λ-calculus [17] to typed languages including recursive types and recursive function definitions based on recursion operators. We illustrate our ideas using the simply-typed λ-calculus with booleans, natural numbers, and their respective iterators. This system is very close to Gödel's System $\mathcal{T}$ [8], but to allow for the examples to have a more realistic programming flavour, list types (and a list iterator) are also included – in fact, the implementation can be extended smoothly for arbitrary polynomial types. We choose to implement normal-order evaluation for this language, but call-by-value and call-by-need could easily be obtained by building on previous results by Sinot. Fixpoint operators have been studied elsewhere for interaction net implementations [4, 13], and also carried over to the token-passing setting [2].

An interesting feature of the work presented in this paper is that the interaction systems output by our encodings result in definitions that are very similar to the typical examples of "direct" interaction net programs. In this sense our work justifies semantically a functional subset of interaction nets. Moreover this provides further evidence that our approach is indeed an appropriate and natural way to represent functional programs visually.

Related Work. Visual Functional Programming. Work in this area has addressed different aspects of visual programming. The Pivotal project [10] offers a visual notation (and Haskell programming environment) for data-structures, but not programs. Visual Haskell [16] more or less stands at the opposite side of the spectrum of possibilities: this is a dataflow-style visual notation for Haskell programs, which allows programmers to *define* their programs visually (with the assistance of a tool) and then have them translated automatically to Haskell code. Kelso's VFP system [11] is a complete environment that allows functional programs to be defined visually and then reduce them step by step. Finally, VisualLambda [6] is a formalism based on graph-rewriting: programs are defined as graphs whose reduction mimics the execution of a functional program. As far as we know none of these systems is widely used.

Visual Haskell and VisualLambda have in common the fact that functions are represented as boxes with input ports for the arguments and an output port for the result; the contents of the box corresponds to the body of the function. They differ in that Visual Haskell uses named variables to refer to function arguments, while VisualLambda uses a graphical notation based on arrows. VFP uses a notation without boxes, inspired by the representations used in implementation-oriented graph-rewriting machines. In particular, it allows for named functions but also for λ-abstractions, and an explicit application node exists. Variables are used for arguments, as in Visual Haskell.

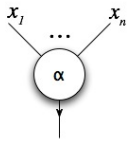
Higher-order programming is a fundamental feature of functional programming. A function f can take a function g as an argument and g can then be applied within the body of f. Expressing this feature is easy if variables are used as in Visual Haskell and VFP; in VisualLambda a special box would be used as a placeholder for g (in the body of f) to be instantiated later, and an arrow would link an input port in the box of f to the box of g. A second difficulty is that a (curried) function of two arguments may be applied to its first argument and return as result a function. In a box-based representation this means that it must be possible for a box to lose its input ports one by one – a quite complicated process. Graph rewriting in general, and our approach in particular, treat this problem naturally as will become clear.

The work presented in this paper uses a pure visual representation of programs, without named variables. In this aspect it resembles VisualLambda, however our work differs significantly from this in that no boxes are used, and all the graph-rewriting operations are *local* in the sense that only two nodes of the graph are involved in each step.

Structure of the Paper. Sections 2 and 3 contain background material on visual programming with interaction nets and on the token-passing encoding of the λ-calculus. Section 4 defines the functional language used in the paper. Section 5 introduces the translation of functional programs into token-passing interaction nets, and Section 6 considers extensions of the language with other recursion operators. We conclude the paper in Section 7.

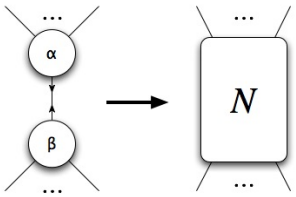
2 Interaction Nets

Interaction nets [12] are constrained graph rewriting systems that can still encode all the computable functions. Interaction nets provide a model of computation in a graphical setting. Programs are represented as particular kinds of graphs, and computation is expressed as graph transformations. Interaction net systems are user-defined, in the same way as term rewriting systems, by giving a signature Σ (a set of symbols with a given arity) and a set of interaction rules R. An occurrence of a symbol is called an *agent*. An agent with arity n has $n + 1$ *ports*: a distinguished one, depicted by an arrow, called the principal port, and n auxiliary ports. Agents are represented graphically in the following way:



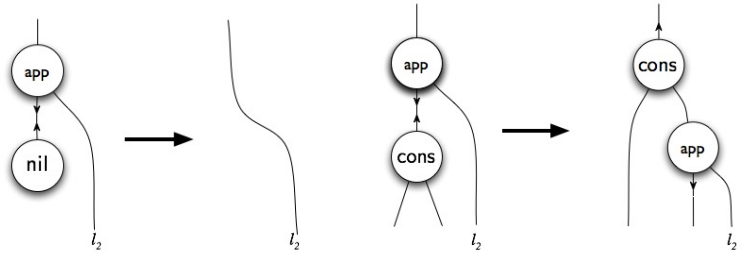
A net N built on a signature Σ is a graph (not necessarily connected) with agents at the vertices. The edges of the net connect agents together at the ports such that there is only one edge at every port. Edges may connect two different ports of the same agent. The ports of an agent that are not connected to another agent are called the *free ports* of the net.

A pair of agents, say (α, β), connected on their principal ports is called an *active pair*, which is the interaction net analogue of a redex. An interaction rule replaces an occurrence of the active pair (α, β) by a net N. The rule has to satisfy a very strong condition: all the free ports are preserved during reduction, and moreover there is at most one rule for each pair of agents. The diagram below illustrates the idea, where N is any net built from the signature:



An interaction net system is therefore fully defined by the pair (Σ, R). We say that a net is in normal form if it does not contain any active pairs. We use the notation $\longrightarrow$ for one-step reduction and $\longrightarrow^*$ for its transitive reflexive closure. Additionally, we write $N \mapsto N'$ if there is a sequence of interaction steps $N \longrightarrow^* N'$, such that N' is a net in normal form. The strong constraints on the definition of interaction rules imply that reduction commutes (the one-step diamond property holds), and thus confluence is easily obtained. Consequently, any normalizing interaction net is strongly normalizing.

Visual Programming with Interaction Nets. The advantages of using interaction nets for visual programming can be understood by looking at a simple example. The following interaction rules define visually the list concatenation operation.



where the symbol app is used for concatenation agents, and nil and cons are the obvious list constructors. The principal port of app is connected to the first list argument, and the result of the operation is obtained in the auxiliary port shown on top.

The interest of visual programming with interaction nets in this way can be summarized as follows.

- Both programs and data are represented in the same simple graphical formalism.
- Programs can be animated without leaving the interaction formalism: instead of resorting to an external interpreter and then displaying the result of each evaluation step, a program can be animated by simply reducing the net. The reader can try this by connecting two lists of some type to an app agent and then applying the rules given above.
- Pattern-matching for external constructors is in-built.
- Recursive definitions are expressed very naturally as interaction rules involving agents (such as app) that are reintroduced on the right-hand side. Rule application then corresponds to the expansion of a recursive definition.

The above example is functional in nature: app can be written in a straightforward way as a function of two arguments that performs recursion on its first argument. But the interaction net

formalism does not offer a satisfactory semantic interpretation for the behaviour of that symbol. Moreover, many interaction net systems can be defined that do not have this functional reading.

What is missing is a clear correspondence between functional definitions and interaction systems like the one shown. In this paper we establish a correspondence between functions defined with recursion operators and agents with interaction rules like those given for app. We remark that the inherent inability of interaction nets to match constructors at a level deeper than one raises no problems: the simple form of pattern-matching available in interaction nets is sufficient for iterators and other recursion operators such as primitive recursors or accumulations.

3 The Token-passing Encoding of the λ-calculus

A number of different translations of the λ-calculus into interaction nets exist. These have in common some basic principles:

- Terms are translated into nets of a fixed interaction net system $(\Sigma_{\mathcal{T}}, R_{\mathcal{T}})$.
- If t is a closed λ-term then the net $\mathcal{T}(t)$ has one free port, corresponding to the *root* of the term, which will be drawn at the top of the net.
- Variables are translated simply as edges in $\mathcal{T}(t)$.
- If $x_1 \dots x_n$ are free variables in t, then the net $\mathcal{T}(t)$ has n additional free ports (represented at the bottom) corresponding to each of the variables .
- $\mathcal{T}(\lambda x.t)$ is a net constructed structurally from $\mathcal{T}(t)$. This introduces an abstraction symbol λ at the root of the term, with a port linked to the edge representing the bound variable x and a port linked to the root of the abstraction body net, $\mathcal{T}(t)$. A special case exists when $x \notin \mathcal{FV}(t)$, which is handled by introducing an *erasing agent* ε.
- $\mathcal{T}(t\,u)$ is a net constructed structurally from $\mathcal{T}(t)$ and $\mathcal{T}(u)$. This introduces an application symbol @ with ports connected to the root ports of $\mathcal{T}(t)$ and $\mathcal{T}(u)$. A special case exists when a free variable occurs in both terms, since a single edge must represent this variable at the bottom of the term. This is handled by introducing a *copying agent* c, with its two auxiliary ports connected to the edges representing the free variable in $\mathcal{T}(t)$ and $\mathcal{T}(u)$, and the edge connected to its principal port represents the variable in $\mathcal{T}(t\,u)$.

The *token-passing* encodings [17] use an interaction system where two different symbols exist for application: one is the syntactic symbol @ introduced by the translation; the corresponding agents have their principal ports facing the root of the term and will be depicted by triangles. A second symbol $\widehat{@}$ exists that will be used for computation; to simplify the figures, the corresponding agents will be depicted by circles equally labelled with @. Their principal ports face the net that represents the applied function, to make possible interaction with λ agents.

The translation $\mathcal{T}_{\text{tp}}(\cdot)$ encodes terms in the system $(\Sigma_{\text{tp}}, R_{\text{tp}})$ where $\Sigma_{\text{tp}} = \{\Downarrow, @, \widehat{@}, \lambda, c, \varepsilon, \delta\}$. The translation is shown in Figure 1, where $\mathcal{T}(.)$ stands for $\mathcal{T}_{\text{tp}}(\cdot)$. It generates nets containing no active pairs, so no reduction can happen.

The special symbol $\Downarrow$ is used as an evaluation *token*: an agent $\Downarrow$ traverses the net, transforming occurrences of @ into $\widehat{@}$, thus triggering reductions. The evaluation rules involving $\Downarrow$ can be tailored for a specific evaluation strategy. For call-by-name, R_{tp} consists of the rules in Figure 2 (the arity of each symbol can be inferred from the rules). This comprises evaluation rules involving $\Downarrow$, a computation rule involving @ and λ, and management (copying and erasing) rules. The symbol δ is a mutation of c used for copying abstractions.

To start the reduction (corresponding to normal order evaluation), a $\Downarrow$ symbol must be connected to the root port of the term. Let $\Downarrow N$ denote the net obtained by connecting a $\Downarrow$ agent to the root port of N, then the following correctness result holds: $t \Downarrow z$ iff $\Downarrow \mathcal{T}_{\text{tp}}(t) \longrightarrow^* \mathcal{T}_{\text{tp}}(z)$, where the evaluation relation $\cdot \Downarrow \cdot$ is defined by the standard normal-order evaluation rules:

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.t' \qquad t'[u/x] \Downarrow z}{t\,u \Downarrow z}$$

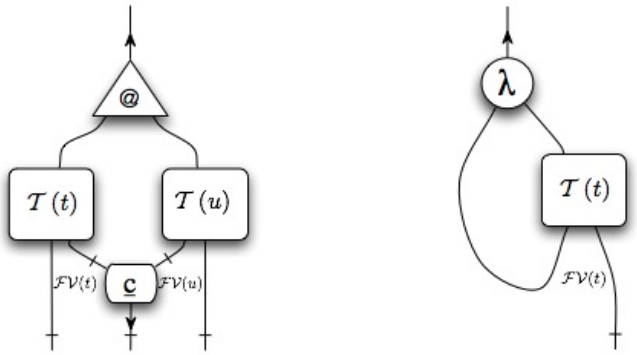


Fig. 1. The token-passing translation of λ-terms: the nets $\mathcal{T}(t\,u)$ and $\mathcal{T}(\lambda x.t)$. $\underline{\mathrm{c}}$ denotes an array of c agents, one for each free variable occurring in both t and u. In $\mathcal{T}(\lambda x.t)$, a special case exists (not depicted) when the bound variable does not occur in the term: an ε agent must be connected to the λ agent instead.

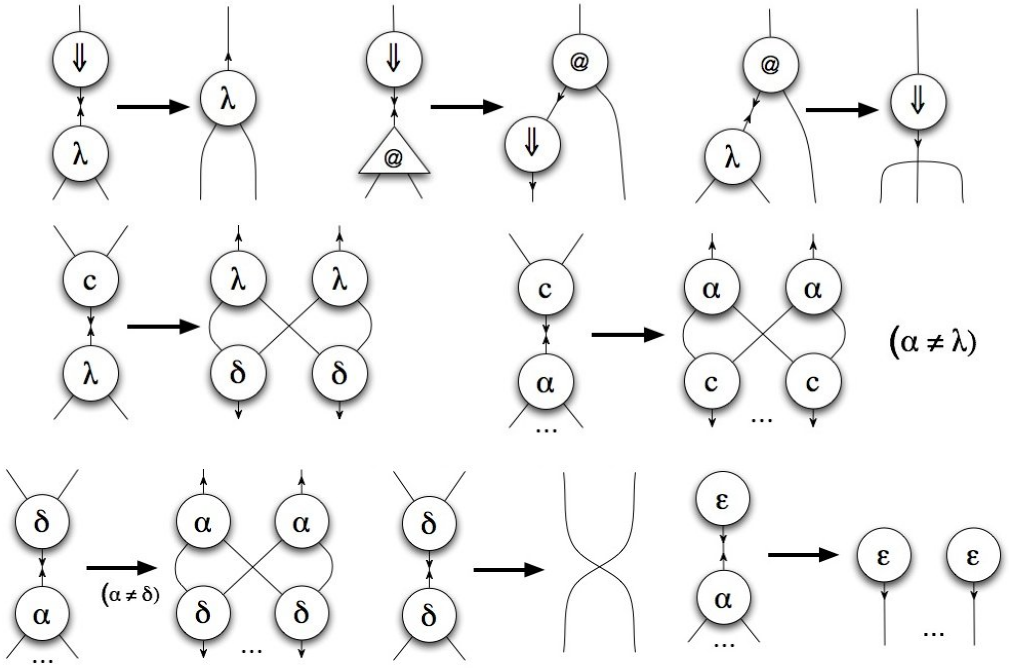


Fig. 2. The token-passing rules R_{tp}. Note the rule *templates* for (c, α), (δ, α), and (ε, α), which generate different rules for each instance of the agent α.

4 The language BNL

In this paper we use the simply-typed λ-calculus extended with natural numbers, booleans, lists, and iterators for these recursive types. The language BNL is defined by the following syntax for types and terms (x, y range over a set of variables):

$$\begin{aligned}\tau, \sigma &::= \mathbf{Bool} \mid \mathbf{Nat} \mid \mathbf{List}(\tau) \mid \tau \to \sigma \\ t, u, v &::= x \mid \lambda x.t \mid t\,u \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{iterbool}(t,u,v) \\ &\quad \mid 0 \mid \mathsf{suc}(t) \mid \mathsf{iternat}(\lambda x.t, u, v) \mid \mathsf{nil} \mid \mathsf{cons}(t,u) \mid \mathsf{iterlist}(\lambda xy.t, u, v)\end{aligned}$$

and by the typing rules given by:

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

$$\frac{}{\Gamma \vdash \mathsf{tt} : \mathbf{Bool}} \quad \frac{}{\Gamma \vdash \mathsf{ff} : \mathbf{Bool}} \quad \frac{}{\Gamma \vdash 0 : \mathbf{Nat}} \quad \frac{\Gamma \vdash t : \mathbf{Nat}}{\Gamma \vdash \mathsf{suc}(t) : \mathbf{Nat}} \quad \frac{}{\Gamma \vdash \mathsf{nil} : \mathbf{List}(\tau)}$$

$$\frac{\Gamma \vdash h : \tau \qquad \Gamma \vdash t : \mathbf{List}(\tau)}{\Gamma \vdash \mathsf{cons}(h,t) : \mathbf{List}(\tau)} \qquad \frac{\Gamma \vdash t : \mathbf{Bool} \qquad \Gamma \vdash V : \tau \qquad \Gamma \vdash F : \tau}{\Gamma \vdash \mathsf{iterbool}(V,F,t) : \tau}$$

$$\frac{\Gamma \vdash t : \mathbf{Nat} \qquad \Gamma \vdash \lambda x.S : \tau \to \tau \qquad \Gamma \vdash Z : \tau}{\Gamma \vdash \mathsf{iternat}(\lambda x.S, Z, t) : \tau}$$

$$\frac{\Gamma \vdash t : \mathbf{List}(\sigma) \quad \Gamma \vdash \lambda xy.C : \sigma \to \tau \to \tau \quad \Gamma \vdash N : \tau}{\Gamma \vdash \mathsf{iterlist}(\lambda xy.C, N, t) : \tau}$$

The call-by-name evaluation semantics is as follows. Note that constructor terms of a given type are taken to be canonical forms.

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.t' \qquad t'[u/x] \Downarrow z}{t\,u \Downarrow z} \qquad \frac{}{0 \Downarrow 0} \qquad \frac{}{\mathsf{suc}(n) \Downarrow \mathsf{suc}(n)}$$

$$\frac{}{\mathsf{tt} \Downarrow \mathsf{tt}} \qquad \frac{}{\mathsf{ff} \Downarrow \mathsf{ff}} \qquad \frac{t \Downarrow \mathsf{tt} \qquad V \Downarrow z}{\mathsf{iterbool}(V,F,t) \Downarrow z} \qquad \frac{t \Downarrow \mathsf{ff} \qquad F \Downarrow z}{\mathsf{iterbool}(V,F,t) \Downarrow z}$$

$$\frac{t \Downarrow 0 \qquad Z \Downarrow z}{\mathsf{iternat}(\lambda x.S, Z, t) \Downarrow z} \qquad \frac{t \Downarrow \mathsf{suc}(n) \qquad S[\mathsf{iternat}(\lambda x.S, Z, n)/x] \Downarrow z}{\mathsf{iternat}(\lambda x.S, Z, t) \Downarrow z}$$

$$\frac{}{\mathsf{nil} \Downarrow \mathsf{nil}} \qquad \frac{}{\mathsf{cons}(u,v) \Downarrow \mathsf{cons}(u,v)} \qquad \frac{t \Downarrow \mathsf{nil} \qquad N \Downarrow z}{\mathsf{iterlist}(\lambda xy.C, N, t) \Downarrow z}$$

$$\frac{t \Downarrow \mathsf{cons}(u,v) \qquad C[u/x, \mathsf{iterlist}(\lambda xy.C, N, v)/y] \Downarrow z}{\mathsf{iterlist}(\lambda xy.C, N, t) \Downarrow z}$$

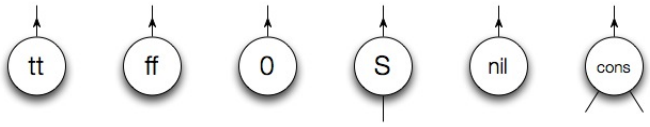
Some variables have been capitalized due to reasons that will become clear later on.

5 A Token-passing Encoding of BNL

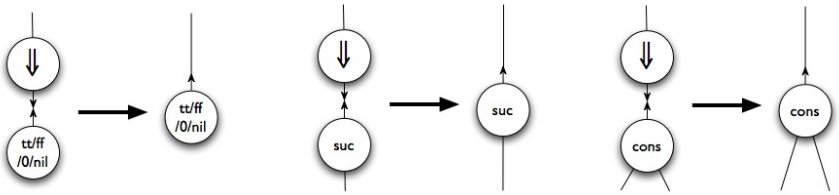
We extend to BNL the token-passing call-by-name translation of the λ-calculus into the interaction system $(\Sigma_{\mathrm{tp}}, R_{\mathrm{tp}})$. We first extend the interaction system and then the translation function. The novelty of this encoding is not the token-passing aspect (which is a natural extension of the encoding of the λ-calculus), but rather the approach to recursion.

Data Structures. Terms of inductively defined types can be represented in interaction nets in the natural way, as *trees* where each node corresponds to a constructor, with its principal port facing the parent node. In a token-passing implementation, there will be an interaction rule between the token agent and each such constructor symbol that will stop evaluation – this corresponds to the fact that constructor terms are canonical forms.

For BNL we define the system $(\Sigma_{\mathrm{BNL}}, R_{\mathrm{BNL}})$ where Σ_{BNL} consists of the symbols tt, ff, 0 and nil with arity 0; suc with arity 1; and cons with arity 2, depicted as



and R_{BNL} consists of the rules given below.



Recursive Programs. A recursive program will be dynamically encoded in an interaction system specifically generated for it. The interaction system will not be extended by introducing a fixed set of symbols; instead a new symbol will be introduced for *each occurrence of a recursion operator*. This will be accompanied by a set of interaction rules, one for each different constructor of its argument type, so a dedicated interaction system (Σ_t^0, R_t^0) is generated for each term t.

This system is constructed by a recursive function $(\Sigma_t^0, R_t^0) = \mathcal{S}(t)$, defined as follows ($\cup$ is occasionally used to denote pairwise union).

$$\mathcal{S}(x) \doteq \mathcal{S}(\mathsf{tt}) \doteq \mathcal{S}(\mathsf{ff}) \doteq \mathcal{S}(0) \doteq \mathcal{S}(\mathsf{nil}) \doteq (\emptyset, \emptyset)$$

$$\mathcal{S}(\lambda x.t) \doteq \mathcal{S}(\mathsf{suc}(t)) \doteq \mathcal{S}(t)$$

$$\mathcal{S}(t\,u) \doteq \mathcal{S}(\mathsf{cons}(t,u)) \doteq \mathcal{S}(t) \cup \mathcal{S}(u)$$

$\mathcal{S}(\mathsf{iterbool}(V,F,b)) \doteq (\{\mathsf{It}^{\mathbf{Bool}}_{V,F}, \widehat{\mathsf{It}^{\mathbf{Bool}}_{V,F}}\} \cup \Sigma, R_{\mathsf{It}^{\mathbf{Bool}}_{V,F}} \cup R)$,
where $(\Sigma, R) = \mathcal{S}(b) \cup \mathcal{S}(V) \cup \mathcal{S}(F)$, and $R_{\mathsf{It}^{\mathbf{Bool}}_{V,F}}$ consists of the interaction rules included in Figures 3(a) and 3(b).

$\mathcal{S}(\mathsf{iternat}(\lambda x.S, Z, n)) \doteq (\{\mathsf{It}^{\mathbf{Nat}}_{S,Z}, \widehat{\mathsf{It}^{\mathbf{Nat}}_{S,Z}}\} \cup \Sigma, R_{\mathsf{It}^{\mathbf{Nat}}_{S,Z}} \cup R)$
where $(\Sigma, R) = \mathcal{S}(n) \cup \mathcal{S}(S) \cup \mathcal{S}(Z)$ and $R_{\mathsf{It}^{\mathbf{Nat}}_{S,Z}}$ consists of the interaction rules included in Figures 3(a) and 3(c).

$\mathcal{S}(\mathsf{iterlist}(\lambda xy.C, N, l)) \doteq (\{\mathsf{It}^{\mathbf{List}}_{C,N}, \widehat{\mathsf{It}^{\mathbf{List}}_{C,N}}\} \cup \Sigma, R_{\mathsf{It}^{\mathbf{List}}_{C,N}} \cup R)$
where $(\Sigma, R) = \mathcal{S}(l) \cup \mathcal{S}(C) \cup \mathcal{S}(N)$ and $R_{\mathsf{It}^{\mathbf{List}}_{C,N}}$ consists of the interaction rules included in Figures 3(a) and 3(d).

Iterator symbols are introduced in pairs $(\mathsf{It}^{\cdots}_{\cdots}, \widehat{\mathsf{It}^{\cdots}_{\cdots}})$ where the first symbol is used for syntactic agents and the second for computation agents. To simplify the graphical presentation, syntactic agents are depicted by triangles. The arity of each symbol can be inferred from the interaction rules. In Figures 3(b) to 3(d), $\underline{c}$ denotes an array of c agents and $\underline{\varepsilon}$ denotes an array of ε agents. The size of this array depends on the number of free variables in the corresponding terms.

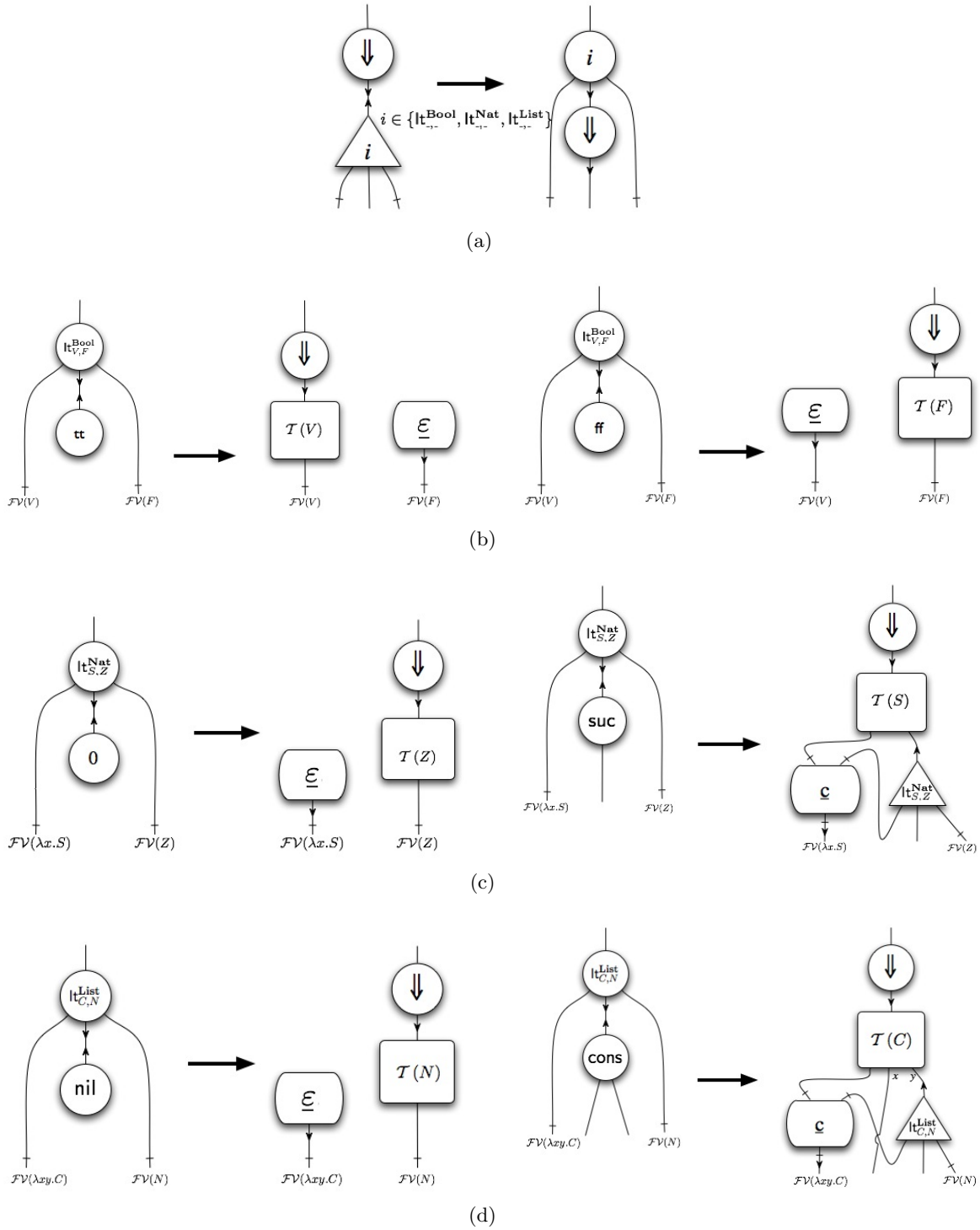
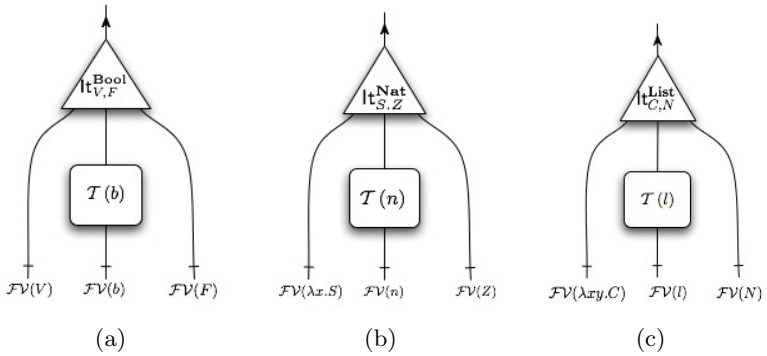


Fig. 3. Interaction rules for iterators



(a) (b) (c)

Fig. 4. Translations of iterators. We remark that, if the same variable occurs in more than one of the named sets (say, $\mathcal{FV}(V)$ and $\mathcal{FV}(F)$ for $\mathsf{iterbool}(V, F, b)$), c agents must be used to group the edges, analogously to what happens in the encoding of an application $t\,u$ (not shown in this figure).

The Translation. A BNL program t will be translated into an interaction net defined in the system $(\Sigma_t, R_t) = (\Sigma_{\mathrm{tp}} \cup \Sigma_{\mathrm{BNL}} \cup \Sigma_t^0, R_{\mathrm{tp}} \cup R_{\mathrm{BNL}} \cup R_t^0)$ where $(\Sigma_{\mathrm{tp}}, R_{\mathrm{tp}})$ was defined in Section 3.

Definition 1. *Given a* BNL *program t, the net $\mathcal{T}(t)$ is given as follows.*

- *If t is an abstraction, variable or application, then $\mathcal{T}(t)$ is defined as in Section 3.*
- *If t is one of* tt, ff, 0, *or* nil, *then $\mathcal{T}(t)$ is an instance of the corresponding symbol.*
- *If $t = \mathsf{suc}(t')$, then $\mathcal{T}(t)$ is constructed by connecting the auxiliary port of a* suc *agent to the root port of $\mathcal{T}(t')$.*
- *If $t = \mathsf{cons}(h, t')$, then $\mathcal{T}(t)$ is constructed by connecting the auxiliary ports of a* cons *agent to the root ports of $\mathcal{T}(h)$ and $\mathcal{T}(t')$.*
- *If $t = \mathsf{iterbool}(V, F, b)$ then $\mathcal{T}(t)$ is given by the net in Figure 4(a).*
- *If $t = \mathsf{iternat}(\lambda x.S, Z, n)$ then $\mathcal{T}(t)$ is given by the net in Figure 4(b).*
- *If $t = \mathsf{iterlist}(\lambda xy.C, N, l)$ then $\mathcal{T}(t)$ is given by the net in Figure 4(c).*

Remarks. As is characteristic of token-passing implementations, all terms (including iterators) are translated as syntax trees. Syntactic iterator agents i are turned into their computation counterparts $\widehat{i}$ by token agents, in the same way as the @ agents in the encoding of the λ-calculus.

A first key aspect of our approach is that the interaction rules of the (computation) iterator agents internalise the iterator's parameters. For instance the net $\mathcal{T}(\mathsf{iterlist}(\lambda xy.C, N, \mathsf{cons}(h, t)))$ reduces in one step to $\mathcal{T}(C[h/x, \mathsf{iterlist}(\lambda xy.C, N, t)/y])$, with an evaluation token on top to control normal-order evaluation.

A second key aspect is that each such new symbol will have auxiliary ports in a one-to-one correspondence with the free variables in the iterator term, since iterator terms are not restricted to be closed. The significance of this will be clear from the examples.

Lemma 1. *Let t be a closed* BNL *term; then: $t \Downarrow z \implies \Downarrow \mathcal{T}(t) \longrightarrow^* \mathcal{T}(z)$.*

Lemma 2. *Let t be a closed* BNL *term and z a canonical form, then: $\Downarrow \mathcal{T}(t) \longrightarrow^* \mathcal{T}(z) \implies t \Downarrow z$.*

The proofs of these results can be found in a long version of this paper [1]. The following is a consequence of the lemmas:

Proposition 1 (Correctness). *If t is a closed* BNL *term and z a canonical form, then: $t \Downarrow z \iff \Downarrow \mathcal{T}(t) \longrightarrow^* \mathcal{T}(z)$.*

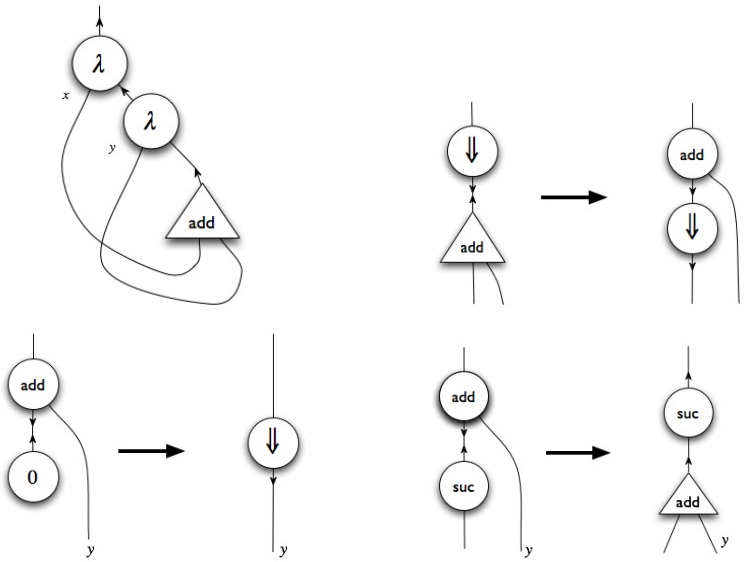


Fig. 5. Encoding of add and corresponding interaction rules

Example 1. Let *add* of type $\mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Nat}$ be defined as $add = \lambda xy.\mathsf{iternat}(\lambda r.\mathsf{suc}(r), y, x)$. The free variable y in the second argument of the iterator gives rise to an auxiliary port in the symbol $\mathsf{It}^{\mathbf{Nat}}_{\mathsf{suc}(r),y}$. The net corresponding to the encoding of the function and the interaction rules generated are given in Figure 5, where add stands for $\mathsf{It}^{\mathbf{Nat}}_{\mathsf{suc}(r),y}$. We remark that the last rule, whose right-hand side contained an active pair, was normalized by reducing that pair. The same will happen in the following examples.
The interaction rules for the computation agent add constitute a highly intuitive visual definition of addition, as should happen in any framework for visual programming. Figure 6 shows an example evaluation of a program.

Example 2. The reader is invited to work out the encoding of the append function: $app : \mathbf{List}(\tau) \to \mathbf{List}(\tau) \to \mathbf{List}(\tau)$, defined as $app = \lambda l_1 l_2.\mathsf{iterlist}(\lambda hr.\mathsf{cons}(h, r), l_2, l_1)$ and to compare it to the rules given in Section 2 for the agent app as an example of a direct interaction net program.

Example 3. Our final example corresponds to a higher-order function. The function $\mathsf{map} : (\tau \to \sigma) \to \mathbf{List}(\tau) \to \mathbf{List}(\sigma)$, defined as $map = \lambda fl.\mathsf{iterlist}(\lambda hr.\mathsf{cons}(f\,h, r), \mathsf{nil}, l)$. This example differs from the previous in that a free variable (f) now occurs in the *first* argument of the iterator. Again this generates an auxiliary port in $\mathsf{It}^{\mathbf{List}}_{\mathsf{cons}(f\,h,r),\mathsf{nil}}$. The function is encoded as as the net in Figure 7, where the name map is used for the symbol $\mathsf{It}^{\mathbf{List}}_{\mathsf{cons}(f\,h,r),\mathsf{nil}}$. Its interaction rules are also shown in the figure.
Again the visual representation is intuitive. The role of the copying agent in the second rule is to produce two copies of the encoding of the function f: one to be applied to the head of the argument list, and another to be used in the recursive mapping of the tail.

6 Extending the Language with New Operators

A recursor for natural numbers can be added to the language with the following syntax, typing and evaluation rules: $t, u, v ::= \ldots \mid \mathsf{recnat}(\lambda xy.u, v, t)$,

$$\frac{\Gamma \vdash t : \mathbf{Nat} \qquad \Gamma \vdash \lambda xy.S : \tau \to \mathbf{Nat} \to \tau \qquad \Gamma \vdash Z : \tau}{\Gamma \vdash \mathsf{recnat}(\lambda xy.S, Z, t) : \tau}$$

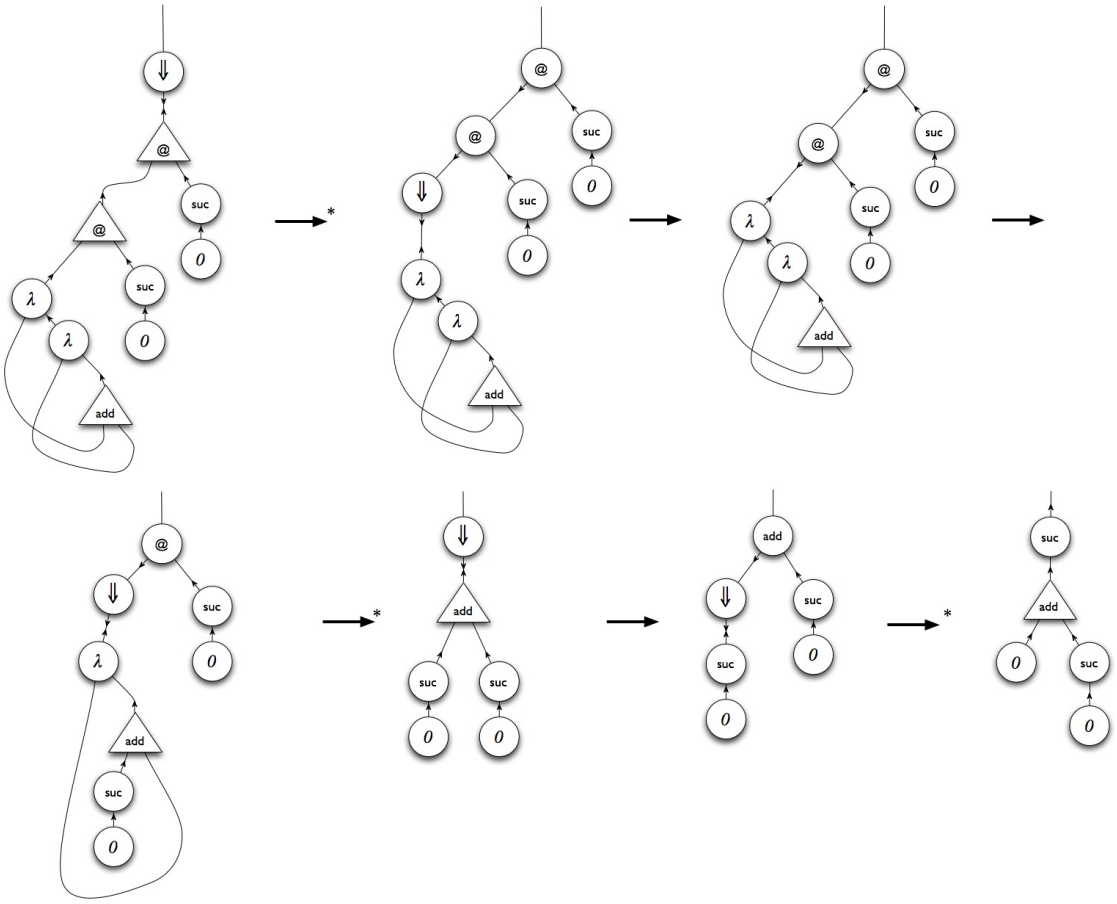


Fig. 6. Snapshots of the evaluation of the program $(\lambda xy.\mathsf{iternat}(\lambda r.\mathsf{suc}(r), y, x))\,(\mathsf{suc}(0))\,(\mathsf{suc}(0))$

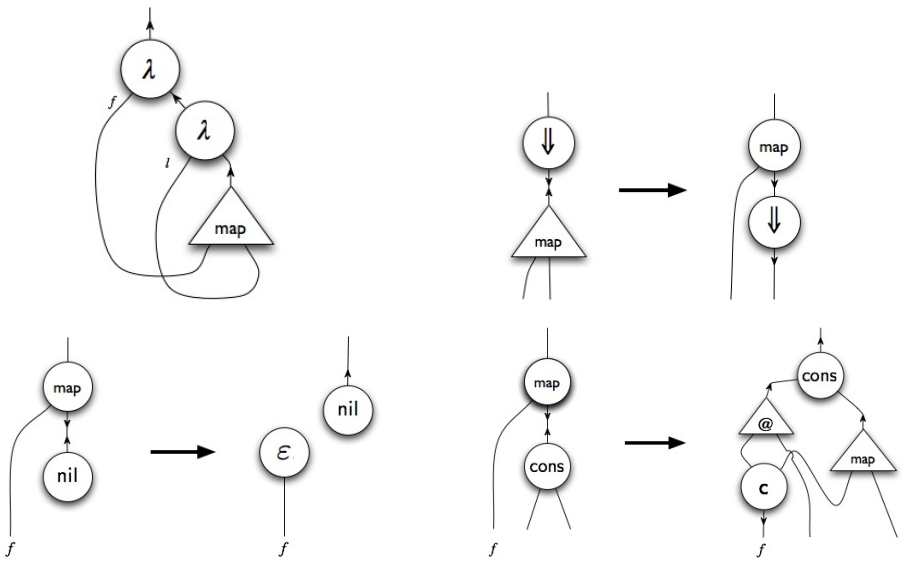
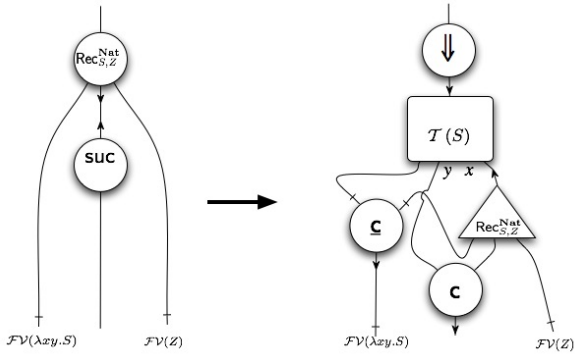


Fig. 7. Encoding of map and corresponding interaction rules

$$\frac{t \Downarrow 0 \qquad Z \Downarrow z}{\mathsf{recnat}(\lambda xy.S, Z, t) \Downarrow z} \qquad \frac{t \Downarrow \mathsf{suc}(n) \qquad S[\mathsf{recnat}(\lambda xy.S, Z, n)/x, n/y] \Downarrow z}{\mathsf{recnat}(\lambda xy.S, Z, t) \Downarrow z}$$

The computational power of this recursor operator comes from the fact that it has access to its *argument*, in addition to the recursive result on that argument. The factorial function, for instance, can be defined in this way, but not with an iterator. Replacing the iterator with this recursor requires only minor changes in the interaction system: an agent $\mathsf{Rec}^{\mathbf{Nat}}_{S,Z}$ must be used in the translation of the expression $\mathsf{recnat}(\lambda xy.S, Z, t)$ instead of $\mathsf{It}^{\mathbf{Nat}}_{S,Z}$. Its interaction with the successor symbol is given by the rule shown below, where we note that for an argument $\mathsf{suc}(n)$, the net representing n must now be duplicated.



Extending the language with other recursion operators is not only a matter of expressiveness, but also of convenience. We take as example the Haskell `foldl` (left folding) list operator: even though it can be encoded with the more common `foldr` (right folding operator), it is still convenient to have it in the language. For instance, a linear time, tail-recursive function for reversing lists can be written in the two following ways:

```
revt l = foldr (\h r a -> r(h:a)) id l []
revt l = foldl (\r h -> h:r) [] l
```

The latter is clearly preferable for its simplicity. The first version can be written in BNL as $\mathsf{revt} = \lambda l.\mathsf{iterlist}(\lambda xya.y\,(\mathsf{cons}(x,a)),(\lambda x.x),l)\,\mathsf{nil}$. Applying the encoding of Section 5 results in the introduction of an agent $\mathsf{It}^{\mathbf{List}}_{(\lambda a.y\,\mathsf{cons}(x,a)),(\lambda x.x)}$. Naturally, the interaction rules for this agent introduce encodings of abstractions in their right-hand sides, which results in a quite complicated definition. To accommodate the second, simpler definition, we now consider the extension of BNL with an accumulation operator, $t, u, v ::= \ldots \mid \mathsf{acclist}(\lambda xy.t, u, v)$, with the following typing and evaluation rules.

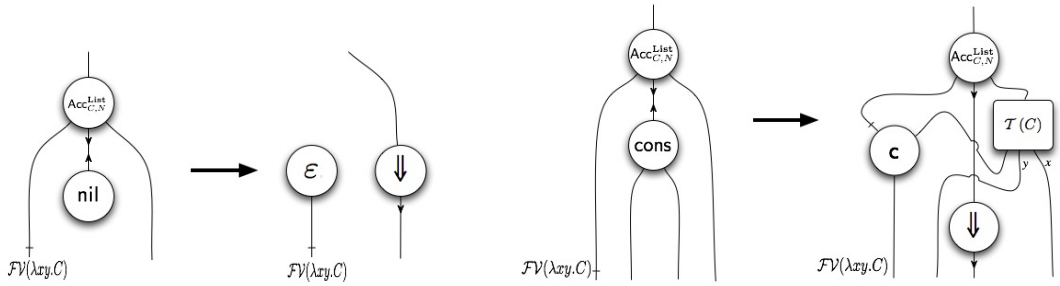
$$\frac{\Gamma \vdash t : \mathbf{List}(\tau) \qquad \Gamma \vdash \lambda xy.C : \sigma \to \tau \to \sigma \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathsf{acclist}(\lambda xy.C, N, t) : \sigma}$$

$$\frac{t \Downarrow \mathsf{nil} \qquad N \Downarrow z}{\mathsf{acclist}(\lambda xy.C, N, t) \Downarrow z} \qquad \frac{t \Downarrow \mathsf{cons}(h, u) \qquad \mathsf{acclist}(\lambda xy.C, C[N/x, h/y], u) \Downarrow z}{\mathsf{acclist}(\lambda xy.C, N, t) \Downarrow z}$$

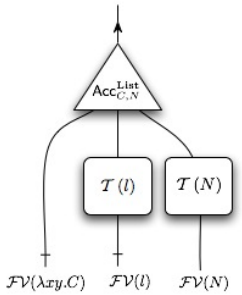
The function $\mathcal{S}(\cdot)$ is extended to create the accumulator Interaction Net System as follows.

$$\begin{aligned}\mathcal{S}(\mathsf{acclist}(\lambda xy.C, N, l)) &= (\{\mathsf{Acc}^{\mathbf{List}}_{C,N}, \widehat{\mathsf{Acc}^{\mathbf{List}}_{C,N}}\} \cup \Sigma, R_{\mathsf{Acc}^{\mathbf{List}}_{C,N}} \cup R) \\ &\text{where } (\Sigma, R) = \mathcal{S}(l) \cup \mathcal{S}(C) \cup \mathcal{S}(N)\end{aligned}$$

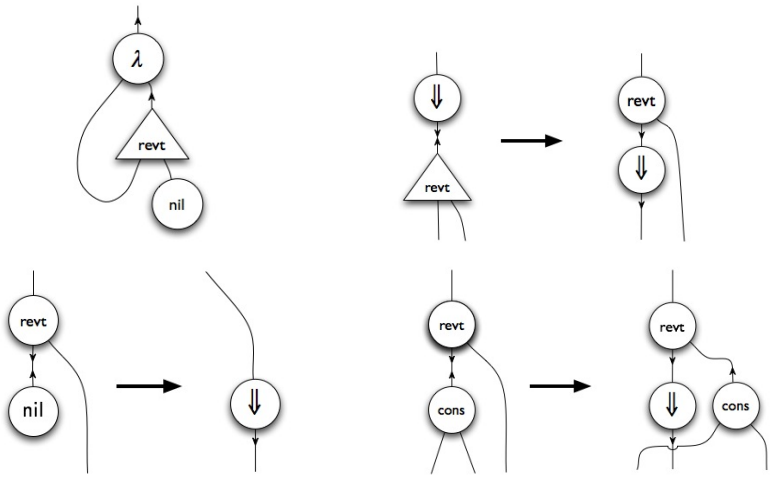
and $R_{\mathsf{Acc}^{\mathbf{List}}_{C,N}}$ consists of the following rules (together with the obvious evaluation token rule).



The translation is then extended as follows. $\widehat{\mathcal{T}}\,(\mathsf{acclist}(\lambda xy.C, N, l))$ is defined to be the net



We remark that in the reduction rules for $\mathsf{acclist}(\lambda xy.C, N, l)$ the second argument N is not fixed throughout iteration; as such it cannot be internalized as part of the definition of the agent $\mathsf{Acc}^{\mathbf{List}}_{C,N}$. Instead the corresponding net is connected to an auxiliary port in that agent (we could have removed the N index from the name of the symbol). The second version can now be written $\mathsf{revt} = \lambda l.\mathsf{acclist}(\lambda xy.\mathsf{cons}(y, x), \mathsf{nil}, l)$, and $\widehat{\mathcal{T}}\,(\mathsf{revt})$ is the following net, where revt stands for $\mathsf{Acc}^{\mathbf{List}}_{\mathsf{cons}(y,x),\mathsf{nil}}$, with the rules below.



7 Conclusions and Future Work

We have presented an approach to encoding in interaction nets functional programs defined with recursion operators, and given the full details of the application of this approach to the token-passing implementation of a normal-order language, which results in a very convenient visual notation for this language. The approach can be easily extended to richer sets of recursive types and to other recursion operators and also to new strategies.

We have left types mostly out of our discussion. A net can be typed by assigning a type to every port. In our context, the types are those defined for the functional language BNL, except

that they may occur either positively (in ports corresponding to data structures) or negatively (in ports corresponding to function or constructor arguments). In a correctly-typed net every edge connects two ports typed with $+A$ and $-A$ for some type A. So typing extends smoothly to the visual setting.

The novel characteristics of the translation are the fact that the interaction system is generated dynamically from the program, and the internalisation of some of the parameters of the recursion operator, in the interaction rules of the symbol that encodes the operator's behaviour.

Tool Support. A prototype system for visual functional programming is currently being developed, integrated in the tool INblobs [3] for interaction net programming. The tool consists of an evaluator for interaction nets together with a compiler module that translates programs to nets. It will allow users to type in a functional program, visualize it, and then follow its evaluation visually step by step. Additionally, a visual editing mode will be available that will allow users to construct nets corresponding to functional programs. This raises a topic we have left out of the discussion in the paper, which is to give a direct (i.e. not resulting from a translation) characterization of this class of nets.

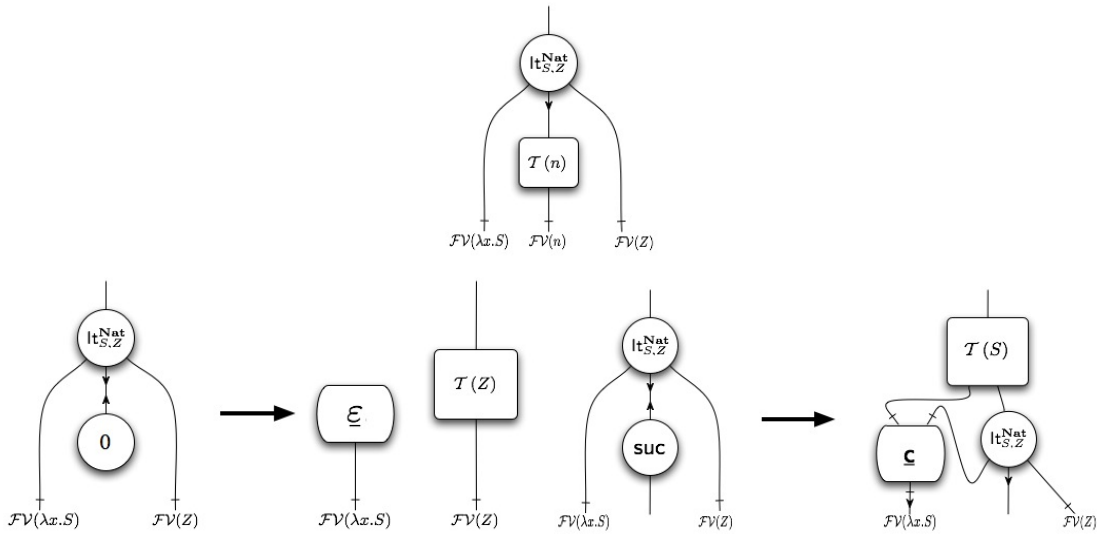
Program Transformation. A different line of work is inspired by work of the datatype-generic programming community and the school of program calculation [5]. This prompts the investigation of visual fusion laws for instance. Fusion laws simplify compositional functional programs before their application to arguments: before calculating $f(g(x))$ one may in certain conditions, by eliminating intermediate data structures, obtain a more efficient function h equivalent to $f \cdot g$, and calculate instead $h(x)$. A classic case is when g is an iterator. We conjecture that these laws can be proved in the interaction net setting by using notions of contextual equivalence [7]. Extending the visual programming tool with fusion capabilities would make possible to perform program transformations at the visual level.

Handling Other Encodings. The token-passing translation of the λ-calculus has the advantage of implementing a simple evaluation order and maintaining a structure in the nets that is always immediately recognizable and understandable in terms of the evaluation semantics. As such it is totally appropriate for our goal of providing a visual representation for functional programs.

Interaction nets have been extensively studied as an implementation mechanism for the λ-calculus. The main motivation for this approach is that it results in highly efficient evaluation strategies, made possible by the close control kept on the erasing and duplication of terms. The token-passing translation is not representative of most work in this area, which has concentrated on designing *efficient*, rather than simple, translations. These translations are not controlled by an evaluation token (in fact they produce nets already containing active pairs) and impose reduction strategies that cannot be defined using term-based abstract machines.

There are a number of interaction net encodings of the λ-calculus, which follow different strategies. To give just a sample, Gonthier, Abadi and Lévy [9] presented an implementation of optimal β-reduction. Mackie [14, 15] has proposed several systems, each corresponding to a different strategy for reduction in the λ-calculus.

Let $\mathcal{T}(\cdot)$ be one such translation. $\mathcal{T}(t\,u)$ is constructed from $\mathcal{T}(t)$ and $\mathcal{T}(u)$ by introducing an application symbol @ with its principal port connected to the root port of $\mathcal{T}(t)$. Our treatment of iterators can be adapted to this setting by simply removing the evaluator tokens and introducing the iterator agents with the principal port immediately facing the argument. For instance we have that $\mathcal{T}(\mathsf{iternat}(\lambda x.S, Z, n))$ may be given by the following net, with the rules below.



When the iterated function is a closed term, a correctness result can be easily established: Let $\lambda x.S$ be a closed term, then

1. $\mathcal{T}\,(\mathsf{iternat}(\lambda x.S, Z, 0)) \longrightarrow \mathcal{T}\,(Z)$
2. $\mathcal{T}\,(\mathsf{iternat}(\lambda x.S, Z, \mathsf{suc}(n))) \longrightarrow \mathcal{T}\,(S[\mathsf{iternat}(\lambda x.S, Z, n)/x])$

We remark that it is always possible to work with iterators with closed functions – thus this result applies to all programs. In general the correctness of the resulting translation of BNL has to be proved for each base translation of the λ-calculus. If such a result can be established, it still has to be studied if, and in what way, the reduction strategy imposed by the translation for the λ-calculus is modified by this treatment of recursion.

References

1. J. B. Almeida, I. Mackie, J. S. Pinto, and M. Vilaça. Encoding iterators in interaction nets. Available from `http://www.di.uminho.pt/~jmvilaca`.
2. J. B. Almeida, J. S. Pinto, and M. Vilaça. Token-passing Implementations of Recursion and Structured Types. In *Proceedings of the 7th International Workshop on Reduction Strategies in Rewriting and Programming (WRS'07)*, 2007. To appear in Elsevier ENTCS.
3. J. B. Almeida, J. S. Pinto, and M. Vilaça. A Tool for Programming with Interaction Nets. In *Proceedings of the The Eighth International Workshop on Rule-Based Programming (RULE'07)*, 2007. To appear in Elsevier ENTCS.
4. A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
5. R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.
6. L. Dami and D. Vallet. Higher-order functional composition in visual form. Technical report, 1996.
7. M. Fernández and I. Mackie. Operational equivalence for interaction nets. *Theoretical Computer Science*, 297(1–3):157–181, February 2003.
8. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
9. 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.
10. K. Hanna. Interactive Visual Functional Programming. In S. P. Jones, editor, *Proc. Intnl Conf. on Functional Programming*, pages 100–112. ACM, October 2002.
11. J. Kelso. *A Visual Programming Environment for Functional Languages*. PhD thesis, Murdoch University, 2002.
12. 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.

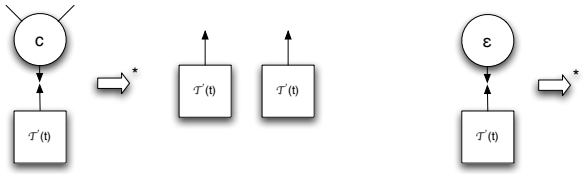
13. I. Mackie. The geometry of interaction machine. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 198–208. ACM Press, January 1995.
14. I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, 1998.
15. 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.
16. H. J. Reekie. *Realtime Signal Processing – Dataflow, Visual, and Functional Programming.* PhD thesis, University of Technology at Sydney, 1995.
17. F.-R. Sinot. Call-by-name and call-by-value as token-passing interaction nets. In P. Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2005.

A Proof of Lemma 1

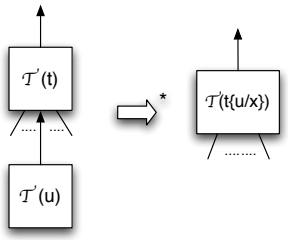
The following auxiliary Lemma has a simple inductive proof:

Lemma 3. *Let $t \in$ BNL be a closed term, then*

1. *the interaction net obtained by connecting an agent ε to $\mathcal{T}(t)$ reduces to the empty net, and*
2. *the interaction net obtained by connecting a c agent to $\mathcal{T}(t)$ reduces to two copies of the net $\mathcal{T}(t)$.*



Lemma 4. *For every term t and closed term u we have the following.*



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

And now for Lemma 1:
Let t be a closed BNL term, then

$$t \Downarrow z \quad \Longrightarrow \quad \Downarrow \mathcal{T}(t) \longrightarrow^* \mathcal{T}(z)$$

Proof. By induction on the length of the derivation of $t \Downarrow z$:

- base cases: if t is one of $\lambda x.t'$, tt, ff, 0, $\mathsf{suc}(t')$, nil, or $\mathsf{cons}(t', u)$, then $t \Downarrow t$ and $\Downarrow \mathcal{T}(t) \longrightarrow \mathcal{T}(t)$
- if $v\,u \Downarrow z$ then two induction hypotheses apply:

$$\Downarrow \mathcal{T}(v) \longrightarrow^* \mathcal{T}(\lambda x.v') \tag{1}$$
$$\Downarrow \mathcal{T}(v'[u/x]) \longrightarrow^* \mathcal{T}(z) \tag{2}$$

  and $\Downarrow \mathcal{T}(v\,u) \longrightarrow^* \mathcal{T}(z)$ by a $\Downarrow \bowtie @$ interaction step, followed by (1), followed by a $\widehat{@} \bowtie \lambda$ interaction step. Now since u must be closed, Lemma 4 applies to finish the substitution process, thus $\Downarrow \mathcal{T}(v\,u) \longrightarrow^* \Downarrow \mathcal{T}(v'[u/x])$. Finally using (2) we get $\Downarrow \mathcal{T}(v\,u) \longrightarrow^* \mathcal{T}(z)$.
- for $\mathsf{iternat}(\lambda x.S, Z, t) \Downarrow z$ two rules exist with this conclusion. For each case two induction hypotheses apply.
  - Case 1

$$\Downarrow \mathcal{T}(t) \longrightarrow^* \mathcal{T}(0) \tag{3}$$
$$\Downarrow \mathcal{T}(Z) \longrightarrow^* \mathcal{T}(z) \tag{4}$$

  - Case 2

$$\Downarrow \mathcal{T}(t) \longrightarrow^* \mathcal{T}(\mathsf{suc}(n)) \tag{5}$$
$$\Downarrow \mathcal{T}(S[\mathsf{iternat}(\lambda x.S, Z, n)/x]) \longrightarrow^* \mathcal{T}(z) \tag{6}$$

In both cases the reduction of $\Downarrow \mathcal{T}\,(\mathsf{iternat}(\lambda x.S, Z, t))$ starts with a $\Downarrow \bowtie \mathsf{It}^{\mathbf{Nat}}_{S,Z}$ interaction step. Then:

- Case 1. I.H. (3) applies, followed by a $\mathsf{It}^{\mathbf{Nat}}_{S,Z} \bowtie 0$ interaction and I.H. (4). Finally, since $\mathsf{iternat}(\lambda x.S, Z, t)$ is closed, Lemma 3 applies to erase subterms.
- Case 2. I.H. (5) applies, then a $\mathsf{It}^{\mathbf{Nat}}_{S,Z} \bowtie \mathsf{suc}$ interaction. Since $\mathsf{iternat}(\lambda x.S, Z, t)$ is closed, Lemma 3 applies to duplicate subterms, resulting in $\Downarrow \mathcal{T}\,(S[\mathsf{iternat}(\lambda x.S, Z, n)/x])$. Finally, I.H. 5 applies.

Thus in both cases $\Downarrow \mathcal{T}\,(\mathsf{iternat}(\lambda x.S, Z, t)) \longrightarrow^* \mathcal{T}\,(z)$.

– The remaining cases (conditional and list iterator) are similar.

□

B Proof of Lemma 2

Let t be a closed BNL term and z a canonical form according to the rules in Section 4; then

$$\Downarrow \mathcal{T}\,(t) \longrightarrow^* \mathcal{T}\,(z) \quad \Longrightarrow \quad t \Downarrow z$$

Proof. By induction on the length of the interaction net reduction sequence $\Downarrow \mathcal{T}\,(t) \longrightarrow^* \mathcal{T}\,(z)$ and case analysis:

– if t is a canonical form $\lambda x.t'$, tt, ff, 0, $\mathsf{suc}(t')$, nil, or $\mathsf{cons}(t', u)$, then we have a base case for induction (1 reduction step) since

$$\Downarrow \mathcal{T}\,(t) \longrightarrow \mathcal{T}\,(t) \quad \Longrightarrow \quad t \Downarrow t$$

– if $t = t'\,u$ then the net $\Downarrow \mathcal{T}\,(t)$ reduces in one step to a net with an agent $\widehat{@}$ at the root. Now if this net reduces to $\mathcal{T}\,(z)$ for some canonical form z, it must be the case that the $\widehat{@}$ agent is involved in a reduction step, and the only rule that applies is the β rule with the λ agent; thus there must be a reduction sequence

$$\Downarrow \mathcal{T}\,(t') \longrightarrow^* \mathcal{T}\,(\lambda x.t'') \tag{7}$$

for some abstraction $\lambda x.t''$, and then a further reduction yields $\Downarrow \mathcal{T}\,(t) \longrightarrow^* \Downarrow \mathcal{T}\,(t''[u/x])$. To complete the reduction sequence the following must hold:

$$\Downarrow \mathcal{T}\,(t''[u/x]) \longrightarrow^* \mathcal{T}\,(z) \tag{8}$$

Now applying the induction hypothesis to the reductions (7) and (8) yields $t' \Downarrow \lambda x.t''$ and $t''[u/x] \Downarrow z$, and the application of the appropriate evaluation rule from Section 4 concludes this case.

– if t is an iterator term, a similar reasoning applies. We illustrate this for $t = \mathsf{iternat}(\lambda x.S, Z, n)$. The net $\Downarrow \mathcal{T}\,(t)$ reduces in one step to a net with an agent $\widehat{\mathsf{It}^{\mathbf{Nat}}_{S,Z}}$ at the root. If this net reduces to $\mathcal{T}\,(z)$ for some canonical form z, it must be the case that the agent at the root is involved in a reduction step, and the only rules that apply are those in Figure 3(c); thus one of the two reduction sequences must exist:

$$\Downarrow \mathcal{T}\,(n) \longrightarrow^* \mathcal{T}\,(0)\,, \text{ or} \tag{9}$$
$$\Downarrow \mathcal{T}\,(n) \longrightarrow^* \mathcal{T}\,(\mathsf{suc}(n')) \tag{10}$$

for some n'. One further step of reduction, together with Lemmas 3 and 4, yields in each case

$$\Downarrow \mathcal{T}\,(t) \longrightarrow^* \Downarrow \mathcal{T}\,(Z)\,, \text{ or}$$
$$\Downarrow \mathcal{T}\,(t) \longrightarrow^* \Downarrow \mathcal{T}\,(S[\mathsf{iternat}(\lambda x.S, Z, n')/x])$$

Reduction is completed as follows in both cases:

$$\Downarrow \mathcal{T}\,(Z) \longrightarrow^* \mathcal{T}\,(z)\,, \text{ or} \tag{11}$$

$$\Downarrow \mathcal{T}\,(S[\mathsf{iternat}(\lambda x.S, Z, n')/x]) \longrightarrow^* \mathcal{T}\,(z) \tag{12}$$

Applying the induction hypothesis to 9 and 11 yields $n \Downarrow 0$ and $Z \Downarrow z$. In the other case, applying the I.H. to 10 and 12 yields $n \Downarrow \mathsf{suc}(n')$ and $S[\mathsf{iternat}(\lambda x.S, Z, n')/x] \Downarrow z$. In both cases the proof is finished by applying evaluation semantics rules, resulting in $t \Downarrow z$.

□