Iterators, Recursors and Interaction Nets

Ian Mackie[1], Jorge Sousa Pinto[2], and Miguel Vilaça[2]
mackie@lix.polytechnique.fr,{jsp,jmvilaca}@di.uminho.pt

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

Abstract. We propose a method for encoding iterators (and recursion operators in general) using interaction nets (INs). There are two main applications for this: the method can be used to obtain a visual notation for functional programs; and it can be used to extend the existing translations of the λ-calculus into INs to languages with recursive types.

1 Introduction

The use of visual notations for functional programs has long been an active research topic, whose main goal is to have a notation that can be used (i) to define functional programs visually, and (ii) to animate their execution.

In this paper we propose a graphical system for functional programming, based on token-passing INs [9]. 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 recursion operators). The system implements a call-by-name semantics, with a straightforward correspondence between functional programs and graphical objects.

Most approaches to visual programming simply propose a notation for programs. Program evaluation is animated by representing visually the intermediate programs that result from executing reduction steps on the initial program, using the operational semantics of the underlying functional language. Our approach is different in that we use a graph-rewriting formalism with its own semantics.

The advantages of using INs for visual programming are:

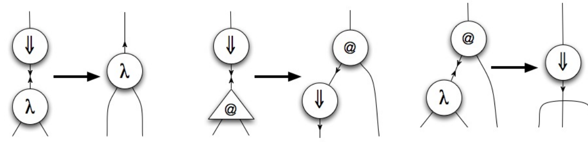
- Both programs and data are represented in the same graphical formalism.
- Programs can be animated without leaving the interaction formalism.
- Pattern-matching for external constructors is in-built.
- Recursive definitions are expressed very naturally as interaction rules involving agents that are reintroduced on the right-hand side.

But the interaction net formalism does not offer a satisfactory semantic interpretation for the behaviour of functional symbols. Moreover, many interaction net systems can be defined that do not have a functional reading. What is missing is a clear correspondence between functional definitions and interaction systems.In this paper we establish a correspondence between agents with "obviously functional" interaction rules and functions defined with recursion operators.

2 The Token-passing Encoding of the λ-calculus

The *token-passing* encodings [9] 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 $\mathcal{T}_{\mathrm{tp}}(\cdot)$ encodes terms in the system $(\Sigma_{\mathrm{tp}}, R_{\mathrm{tp}})$ where $\Sigma_{\mathrm{tp}} = \{\Downarrow, @, \widehat{@}, \lambda, c, \varepsilon, \delta\}$. It generates nets containing no active pairs.The special symbol $\Downarrow$ is used as an evaluation *token*: an agent $\Downarrow$ traverses the net, transforming occurrences of @ into $\widehat{@}$. The evaluation rules involving $\Downarrow$ can be tailored to a specific evaluation strategy. For call-by-name, R_{tp} consists of the following rules which comprises evaluation rules involving $\Downarrow$ and a computation rule involving @ and λ. Management (copying and erasing) rules are omitted here.



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

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

$$\begin{aligned} t, u, v ::= & \; x \mid \lambda x.t \mid t\,u \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{iterbool}(t, u, v) \mid 0 \mid \mathsf{suc}(t) \mid \mathsf{iternat}(\lambda x.t, u, v) \\ \mid & \; \mathsf{nil} \mid \mathsf{cons}(t, u) \mid \mathsf{iterlist}(\lambda xy.t, u, v) \end{aligned}$$

3 A Token-passing Encoding of BNL

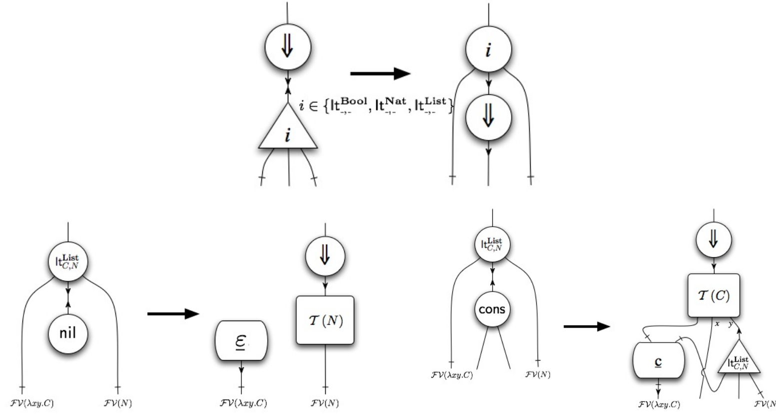
We extend to BNL the token-passing call-by-name translation of the λ-calculus into the interaction system $(\Sigma_{\mathrm{tp}}, R_{\mathrm{tp}})$. The novelty of this encoding is not the token-passing aspect, but rather the approach to recursion.

We first consider data structures. In a token-passing implementation, there will be an interaction rule between the token agent and each constructor symbol that will stop evaluation. 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. Each recursive program will be encoded in an interaction system specifically generated for it. This is a major novelty of our approach. The interaction system for the λ-calculus 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, with an interaction rule for each different constructor, 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:

$\mathcal{S}(x) \doteq \mathcal{S}(\mathsf{tt}) \doteq \mathcal{S}(\mathsf{ff}) \doteq \mathcal{S}(0) \doteq \mathcal{S}(\mathsf{nil}) \doteq (\emptyset, \emptyset)$
$\mathcal{S}(\lambda x.t) \doteq \mathcal{S}(\mathsf{suc}(t)) \doteq \mathcal{S}(t)$
$\mathcal{S}(t\,u) \doteq \mathcal{S}(\mathsf{cons}(t,u)) \doteq \mathcal{S}(t) \cup \mathcal{S}(u)$
$\mathcal{S}(\mathsf{iterbool}(V,F,b)) \doteq (\{\mathsf{It}^{\mathbf{Bool}}_{V,F}, \widehat{\mathsf{It}^{\mathbf{Bool}}_{V,F}}\} \cup \Sigma, R_{\mathsf{It}^{\mathbf{Bool}}_{V,F}} \cup R)$,
where $(\Sigma, R) = \mathcal{S}(b) \cup \mathcal{S}(V) \cup \mathcal{S}(F)$.
$\mathcal{S}(\mathsf{iternat}(\lambda x.S, Z, n)) \doteq (\{\mathsf{It}^{\mathbf{Nat}}_{S,Z}, \widehat{\mathsf{It}^{\mathbf{Nat}}_{S,Z}}\} \cup \Sigma, R_{\mathsf{It}^{\mathbf{Nat}}_{S,Z}} \cup R)$
where $(\Sigma, R) = \mathcal{S}(n) \cup \mathcal{S}(S) \cup \mathcal{S}(Z)$
$\mathcal{S}(\mathsf{iterlist}(\lambda xy.C, N, l)) \doteq (\{\mathsf{It}^{\mathbf{List}}_{C,N}, \widehat{\mathsf{It}^{\mathbf{List}}_{C,N}}\} \cup \Sigma, R_{\mathsf{It}^{\mathbf{List}}_{C,N}} \cup R)$
where $(\Sigma, R) = \mathcal{S}(l) \cup \mathcal{S}(C) \cup \mathcal{S}(N)$

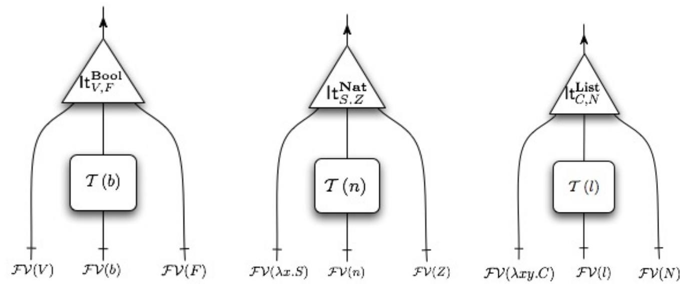
$R_{\mathsf{It}^{\mathbf{List}}_{C,N}}$ consists of the following interaction rules (the others are similar).



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 (similarly to $@, \widehat{@}$).

A BNL program t will be translated into a net defined in the system $(\Sigma_t, R_t) = (\Sigma_{\text{tp}} \cup \Sigma_{\text{BNL}} \cup \Sigma_t^0, R_{\text{tp}} \cup R_{\text{BNL}} \cup R_t^0)$ where $(\Sigma_{\text{tp}}, R_{\text{tp}})$ was defined in Section 2.

Given a BNL program t, where t is $\mathsf{iterbool}(V,F,b)$, $\mathsf{iternat}(\lambda x.S, Z, n)$ or $\mathsf{iterlist}(\lambda xy.C, N, l)$, then the net $\mathcal{T}(t)$ is given as follows.



In token-passing implementations, all terms are translated as syntax trees. Syntactic iterator agents i are turned into their computation counterparts $\widehat{i}$ by token agents. 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 to $\Downarrow \mathcal{T}\,(C[h/x, \mathsf{iterlist}(\lambda xy.C, N, t)/y])$.

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. We end the section with a correctness result. The proofs can be found in [1].

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

4 Conclusions and Future Work

We have presented an approach to encoding in INs functional programs defined with recursion operators, and given the full details of the application of this approach to the token-passing implementation of a call-by-name 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 other recursion operators and also to new strategies. The novel characteristics of the encoding are (i) the interaction system is generated dynamically from the program, and (ii) the internalisation of some of the parameters of the recursion operator in the interaction rules. With respect to previous work on encoding recursion in interaction nets, fixpoint operators have been studied elsewhere for interaction net implementations [4, 6], and we have shown elsewhere how a binding recursion operator (as in PCF) can be implemented in the token-passing setting [2].

A prototype system for visual functional programming has been developed, integrated in the tool INblobs [3, 10] for interaction net programming. The tool consists of an evaluator for interaction nets together with a visual editor and a compiler module that translates programs into nets. The latter module allows users to type in a functional program, visualize it, and then follow its evaluation visually step by step. The current compiler module automatically generates call-by-name or call-by-value systems. Additionally, a visual editing mode is available that allows users to construct nets corresponding to functional programs. In the current implementation there is no way to convert visual programs back to textual ones.

The token-passing translation is not however very representative of most work in this area, which has concentrated on designing *efficient* translations; [5, 7, 8] are some samples. Let $\mathcal{T}\,(\cdot)$ be one such translation. Typically $\mathcal{T}\,(t\,u)$ 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 removing the evaluator tokens and introducing the iterator agents with the principal port immediately facing the argument. When the iterated function is a closed term, a correctness result can be easily established. Let $\lambda x.S$ be a closed term, then $\mathcal{T}\,(\mathsf{iternat}(\lambda x.S, Z, 0)) \longrightarrow \mathcal{T}\,(Z)$ and $\mathcal{T}\,(\mathsf{iternat}(\lambda x.S, Z, \mathsf{suc}(n))) \longrightarrow \mathcal{T}\,(S[\mathsf{iternat}(\lambda x.S, Z, n)/x])$

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 Nets for Functional Languages. In J. Giesl, editor, *Proceedings of the 7th International Workshop on Reduction Strategies in Rewriting and Programming (WRS'07)*, volume 204 of *Electronic Notes in Theoretical Computer Science*, pages 181–198, 2007.
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. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. M. Vilaça. Inblobs webpage. http://haskell.di.uminho.pt/jmvilaca/INblobs/.