Recursion in the Call-by-Value λ-Calculus (*)

Gérard Boudol and *Pascal Zimmer*

INRIA Sophia Antipolis
BP 93 – 06902 Sophia Antipolis Cedex, France

Abstract

We propose an abstract machine to run the call-by-value λ-calculus extended with a call-by-value fixed-point, and we show that this provides us with a correct implementation of our calculus.

1. Introduction

It is a well-known fact that the call-by-value λ-calculus is the functional core of programming languages like Scheme or ML, and that the ability to use fixed-points in this calculus is essential for the expressivity of such programming languages. However, in a language like ML, recursion is usually syntactically restricted to

$$(\mathsf{let\ rec}\ f = \lambda x M\ \mathsf{in}\ \cdots)$$

so that in particular one can only define functions recursively. A reason for this is that it would be dangerous to use an unrestricted recursion construct ($\mathsf{let\ rec}\ f = M\ \mathsf{in}\ \cdots$), where one evaluates the expression M before binding (recursively) its value to f. Indeed, the evaluation of this construct gets stuck if in evaluating M one needs the value of f. An example is

$$(\mathsf{let\ rec}\ f = \mathsf{F}(f\,\mathsf{I})\ \mathsf{in}\ \cdots) \qquad \text{where} \quad \mathsf{F} = \lambda x \lambda y\, y \quad \text{and} \quad \mathsf{I} = \lambda z\, z$$

Nevertheless, one may wish to have the ability to define recursive *non-functional* values, like mutually recursive modules (*cf.* [3, 4]), or objects as recursive records (*cf.* [1, 2]), that is

$$(\mathsf{let\ rec}\ o = (\lambda \mathit{self}\, R) o\ \mathsf{in}\ \cdots)$$

where R is a record of fields and methods. In [1] we have shown that one can determine, by a static analysis – namely, a type system –, that some unrestricted recursions are actually safe. However, the question remained of how to implement this unrestricted recursion, in a more effective (and provably correct) way than what is done in the interpreter built by the second author. For instance, since in an implementation a λ-calculus variable is not a value, one may wonder how we can compute $(\lambda \mathit{self}\, R)o$ without having a value for o. In this paper we introduce an abstract machine for such unrestricted – but typable in our system – recursion, and we show that this implementation is correct. For simplicity reasons, we do this for a minimal calculus, the call-by-value λ-calculus with (call-by-value) recursion, and we do not consider non-functional values like pairs, records or modules. Indeed, we have shown in [1] that the difficulties with recursion lie in this functional core, and that adding more types and related constructs does not cause any particular trouble.

(*) Work partially supported by the CTI "Objets Migrants: Modélisation et Vérification", France Télécom R&D

2. The Calculus

Assuming that a denumerable set $\mathcal{V}ar$ of variables, ranged over by x, y, $z\ldots$, is given, the syntax of our calculus is the following:

$$\begin{array}{rcll} M,\,N\ldots & ::= & V \mid (MN) \mid \mu x M & \textit{expressions} \\ V & ::= & x \mid \lambda x M & \textit{values} \end{array}$$

where $\mu x M$ is the standard fixed-point construct, here computed in a call-by-value regime. Another possible syntax would be ($\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M$), which stands for ($\mathsf{let}\ x = \mu x N\ \mathsf{in}\ M$) – that is, as far as polymorphism is not concerned, $(\lambda x M)(\mu x N)$ –, since $\mu x M$ may be read ($\mathsf{let\ rec}\ x = M\ \mathsf{in}\ x$). We could also consider as a primitive a combinator fix (which is $\lambda f \mu x(fx)$), since $\mu x M$ could then be defined as $\mathsf{fix}(\lambda x M)$, but, as we said in the Introduction, it would not suit our purposes to use the call-by-value fixed-point combinator $\mathsf{Y_v}$ (*cf.* [7]), defined by

$$\mathsf{Y_v} \;=\; (G_\mathsf{v} G_\mathsf{v}) \qquad \text{where} \quad G_\mathsf{v} = \lambda g f. f(\lambda x.(gg)fx)$$

Denoting by $\{x \mapsto N\}M$ the capture avoiding substitution of x by N in M, the reduction – or better, evaluation – relation is given by

$$\begin{array}{rcl} (\lambda x M V) & \to & \{x \mapsto V\}M \\ \mu x V & \to & \{x \mapsto \mu x V\}V \\ M \to M' & \Rightarrow & \mathbf{E}[M] \to \mathbf{E}[M'] \end{array}$$

where the evaluation contexts $\mathbf{E}$ are

$$\mathbf{E} \;::=\; [] \mid (\mathbf{E}N) \mid (V\mathbf{E}) \mid \mu x \mathbf{E}$$

As we said in the Introduction, we use a type discipline to rule out unsafe recursion. This is briefly presented in the Appendix. Our type discipline here is more generous than the one of [1], in the sense that it actually does not impose any restriction on λ-terms – without recursion:

Remark 2.1. *If M is a λ-term, that is an expression written without using μ, then M is typable.*

In [1], we proved that the following holds:

Theorem (Type Safety) 2.2. *Let M be a closed expression. If M is typable, then its evaluation either diverges, or ends with a value.*

Moreover, we have characterized the behaviour of the recursive variables, which can only be "passed around", as argument of functions, or be put inside a value:

Lemma 2.3. *If $\mu x M$ is typable and $M \xrightarrow{*} \mathbf{E}[x]$, where x is not bound by $\mathbf{E}$, then $\mathbf{E} = \mathbf{E}'[(\lambda y N\, [])]$.*

This is the key property that allows us to design a correct abstract machine for our calculus: we know that if a recursive variable comes to be evaluated, that is if we have to evaluate $\mu x \mathbf{E}[x]$, we actually do not need its value, since we can continue with $\mu x \mathbf{E}'[\{y \mapsto x\}N]$.

3. The Abstract Machine

Our machine builds upon standard stack machines for evaluating λ-expressions. As such, it has a *control stack* S, to record the current evaluation context, and an *environment* part σ, to hold the values of the free variables occurring in the *code*, that is, the sub-expression currently evaluated. The stack is made up of *frames*, that are basic evaluation contexts (*cf.* [6]). As usual, substitution of terms for variables is not performed; instead, the machine deals with *closures* [5], that are pairs of an environment and an expression, that we denote σM. As regards recursion, we use the standard trick of Landin's imperative fixpoint: to compute μxM, allocate a new address ℓ, evaluate M in an environment where x is bound to ℓ, and assign to ℓ the result of this evaluation. As noted in [5], this works correctly if M is a function, that is $M = \lambda yN$, though in that case we may have a more direct implementation, binding (circularly) x to λyN in the environment. Then, assuming that a set $\mathcal{L}$ of *locations* is given, our machine has an extra component ξ, which is a memory to store the evaluated recursion's bodies. A configuration of the machine is therefore a tuple

$$\mathcal{M} \;=\; (S, \sigma, M, \xi)$$

where the control stack and environment are built according to the following syntax:

$$\begin{array}{rcl}
S & ::= & \varepsilon \mid S :: \mathbf{F} \\
\mathbf{F} & ::= & ([]\,\sigma N) \mid (\sigma A\,[]) \mid \mu\ell\,[] \\
\sigma,\ \rho\ldots & ::= & \varepsilon \mid \sigma :: \{x \mapsto \rho A\} \mid \sigma :: \{x \mapsto\!\!\!\to A\} \mid \sigma :: \{x \mapsto \ell\} \\
A & ::= & \lambda x M
\end{array}$$

In $\sigma :: \{x \mapsto\!\!\!\to A\}$, the abstraction A is the value of the recursive variable x, and is in the scope of σ (except for what regards x). In a concrete implementation the store ξ simply is a partial mapping assigning values to locations; however, for the correctness proof it is more convenient to have a syntactic description of it, as follows:

$$\xi \;::=\; \emptyset \mid \xi \uplus \{\ell \mapsto \bullet\} \mid \xi[\ell := \sigma A]$$

The domain of a store ξ is defined in the obvious way:

$$\begin{array}{rcl}
\mathsf{dom}(\emptyset) & = & \emptyset \\
\mathsf{dom}(\xi \uplus \{\ell \mapsto \bullet\}) & = & \mathsf{dom}(\xi) \cup \{\ell\} \\
\mathsf{dom}(\xi[\ell := \sigma A]) & = & \mathsf{dom}(\xi)
\end{array}$$

The operation $\xi \uplus \{\ell \mapsto \bullet\}$ is the extension of ξ with a new location ℓ to which is assigned a "dummy" value, while $\xi[\ell := \sigma A]$ is the updating operation. Then a store ξ represents a partial function, defined as:

$$(\xi \uplus \{\ell \mapsto \bullet\})(\ell') \;=\; \begin{cases} \bullet & \text{if } \ell' = \ell \\ \xi(\ell') & \text{otherwise} \end{cases}$$

$$(\xi[\ell := \sigma A])(\ell') \;=\; \begin{cases} \sigma A & \text{if } \ell' = \ell \\ \xi(\ell') & \text{otherwise} \end{cases}$$

To evaluate a closed expression M, we start from the configuration $(\varepsilon, \varepsilon, M, \emptyset)$. Now we describe the behaviour of the machine, that is the transition relation $\mathcal{M} \to \mathcal{M}'$ between configurations. The first three rules for evaluating an application (MN) are standard: first, we record the closure of the argument N on the control stack and evaluate the function M. Then we record (the closure of) its value $A = \lambda xM'$ on the stack, and turn to the evaluation of the argument. When a value for it is computed, we update the environment of the function with a binding for x to this value, and we compute M'. There is a special case when the argument is a recursive variable, whose value is not yet computed: this is dealt with by a fourth rule for application, which is new. Here are the rules:

$$\boxed{\begin{array}{rcll}
(S, \sigma, (MN), \xi) & \to & (S :: ([]\,\sigma N), \sigma, M, \xi) & \\
(S :: ([]\,\rho N), \sigma, A, \xi) & \to & (S :: (\sigma A\,[]), \rho, N, \xi) & \\
(S :: (\sigma\lambda xM\,[]), \rho, A, \xi) & \to & (S, \sigma :: \{x \mapsto \rho A\}, M, \xi) & \\
(S :: (\sigma\lambda yM\,[]), \rho :: \{x \mapsto \ell\}, x, \xi) & \to & (S, \sigma :: \{y \mapsto \ell\}, M, \xi) & \quad \xi(\ell) = \bullet
\end{array}}$$

As we said, to evaluate a recursion we use Landin's imperative fixpoint technique, except that we make a special case when the body of the recursion is known to be a function:

$$\boxed{\begin{array}{rcll} (S,\sigma,\mu x A,\xi) & \to & (S,\sigma :: \{x\mapsto A\},A,\xi) & \\ (S,\sigma,\mu x M,\xi) & \to & (S :: \mu\ell\,[\,],\sigma :: \{x\mapsto \ell\},M,\xi\uplus\{\ell\mapsto\bullet\}) & M\neq\lambda yN,\ \ell\notin\mathsf{dom}(\xi) \\ (S :: \mu\ell\,[\,],\sigma,A,\xi) & \to & (S,\sigma,A,\xi[\ell := \sigma A]) & \end{array}}$$

Finally we have rules to fetch the value of a variable from the environment σ. These rules are standard: one discards the bindings that do not concern the given variable, and then, depending on the value of the variable in the environment, one fetches the appropriate closure, which is unravelled, or the abstraction that is the recursive value of the variable.

$$\boxed{\begin{array}{rcll} (S,\sigma :: \{y\mapsto Z\},x,\xi) & \to & (S,\sigma,x,\xi) & x\neq y \\ (S,\sigma :: \{x\mapsto \rho A\},x,\xi) & \to & (S,\rho,A,\xi) & \\ (S,\sigma :: \{x\mapsto A\},x,\xi) & \to & (S,\sigma :: \{x\mapsto A\},A,\xi) & \\ (S,\sigma :: \{x\mapsto \ell\},x,\xi) & \to & (S,\rho,A,\xi) & \xi(\ell)=\rho A \end{array}}$$

For instance, the reader can check that if $M=\mu z(\mathsf{K}\,z)$ where $\mathsf{K}=\lambda x\lambda y\,x$, then

$$\begin{array}{rcll} (\varepsilon,\varepsilon,M,\emptyset) & \xrightarrow{*} & (\varepsilon :: \mu\ell\,[\,] :: (\rho\mathsf{K}\,[\,]),\rho,z,\xi) & \text{where } \rho=\varepsilon :: \{z\mapsto\ell\},\ \xi=\emptyset\uplus\{\ell\mapsto\bullet\} \\ & \to & (\varepsilon :: \mu\ell\,[\,],\sigma,\lambda y\,x,\xi) & \text{where } \sigma=\rho :: \{x\mapsto\ell\} \\ & \to & (\varepsilon,\sigma,\lambda y\,x,\xi[\ell := \sigma\lambda y\,x]) & \end{array}$$

A final state of the machine is a configuration of the form $\mathcal{M}=(\varepsilon,\sigma,A,\xi)$, and we may define the result of the computation as the expression $\mathcal{V}al(\mathcal{M})=\langle\xi\rangle\circ\langle\sigma\rangle A$ associated with this configuration, where $\langle\sigma\rangle$ is a simultaneous substitution for the variables, defined in such a way that it satisfies in particular:

$$\begin{array}{rcl} \langle\sigma :: \{x\mapsto\rho A\}\rangle x & = & \langle\rho\rangle A \\ \langle\sigma :: \{x\mapsto A\}\rangle x & = & \langle\sigma\rangle\mu x A \\ \langle\sigma :: \{x\mapsto\ell\}\rangle x & = & \ell \end{array}$$

where we assume for convenience that the language is extended so that locations are a new kind of variables, and $\langle\xi\rangle$ is a substitution for locations, such that

$$\langle\xi[\ell := \rho A]\rangle\ell \ = \ \mu x\{\ell\mapsto x\}(\langle\xi\rangle\circ\langle\rho\rangle)A$$

where x is fresh. Then we define a partial function $\mathcal{E}val$ from closed expressions to closed values as follows: for any closed expression M, $\mathcal{E}val(M)$ is defined if there exists a (unique, since the machine is deterministic) final configuration $\mathcal{M}$ such that $(\varepsilon,\varepsilon,M,\emptyset)\xrightarrow{*}\mathcal{M}$, and in this case $\mathcal{E}val(M)=\mathcal{V}al(\mathcal{M})$. Then our main result is:

Theorem (Correctness) 3.1. *Let M be a closed expression. Then*

(i) *if $\mathcal{E}val(M)$ is defined then $M\xrightarrow{*}\mathcal{E}val(M)$;*

(ii) *if M is typable and $M\xrightarrow{*}A$ then $\mathcal{E}val(M)$ is defined and $\mathcal{E}val(M)=A$.*

To see that the typing is needed, one can check for instance that $\mu x(\mathsf{K}(\mu z\,x))$, which is not typable, reduces to $\lambda y\mu x\lambda y\,x$, while the evaluation of this expression in the machine gets stuck in the configuration

$$(\varepsilon :: \mu\ell\,[\,] :: (\rho\mathsf{K}\,[\,]) :: \mu\ell'\,[\,],\rho,x,\emptyset\uplus\{\ell\mapsto\bullet\}\uplus\{\ell'\mapsto\bullet\}) \qquad \text{where} \quad \rho=\varepsilon :: \{x\mapsto\ell\}$$

The abstract machine we presented, being based on named variables, is well-suited for establishing a correspondance with the calculus. However, it should not be difficult to built a more concrete, nameless version of the machine, based on de Bruijn's indices.

References

[1] G. Boudol, *The recursive record semantics of objects revisited,* to appear (2002). Available from the web page of the author.

[2] L. Cardelli, *A semantics of multiple inheritance,* Semantics of Data Types, Lecture Notes in Comput. Sci. 173 (1984) 51-67. Also published in Information and Computation, Vol. 76 (1988).

[3] K. Crary, R. Harper, S. Puri, *What is a recursive module?,* PLDI'99 (1999) 50-63.

[4] T. Hirschowitz, X. Leroy, *Mixin modules in a call-by-value setting,* ESOP'02, Lecture Notes in Comput. Sci. 2305 (2002) 6-20.

[5] P. J. Landin, *The mechanical evaluation of expressions,* Computer Journal Vol. 6 (1964) 308-320.

[6] A. Pitts, *Operational semantics and program equivalence,* Lectures given at the International Summer School on Applied Semantics, APPSEM'00 (2000). Available from the web page of the author.

[7] G. Plotkin, *Call-by-name, call-by-value and the λ-calculus,* Theoret. Comput. Sci. 1 (1975) 125-159.

Appendix: The Type System

Our notion of type is a refinement of the usual one, featuring boolean *degrees* 0 or 1, to discriminate which variables are certainly safe (having degree 1) or potentially unsafe (degree 0) for recursion. For lack of space, we shall not comment in detail here on our type system. The interested reader is referred to [1, 4]. The degrees and types are respectively given by

$$\begin{array}{rcl} \alpha,\ \beta\ldots & := & 0 \mid 1 \mid (\alpha \wedge \beta) \\ \tau,\ \theta\ldots & ::= & * \mid (\theta^{\alpha} \rightarrow \tau) \end{array}$$

We consider types up to an equality asserting that $*$ is, as a data type, Scott's D_∞ domain of functions:

$$* = (*^0 \rightarrow *)$$

Then our type system will accept any λ-term. We also consider types up to equality of degrees $\alpha = \beta$, defined as $\alpha \leqslant \beta$ & $\beta \leqslant \alpha$, where the preorder $\leqslant$ satisfies the following laws:

$$\begin{gathered} 0 \leqslant \alpha \leqslant 1 \\ \alpha \leqslant \alpha \wedge \beta \qquad \text{and} \qquad \alpha \leqslant \beta \wedge \alpha \\ \alpha \leqslant \alpha_0 \ \&\ \alpha \leqslant \alpha_1 \ \Rightarrow\ \alpha \leqslant \alpha_0 \wedge \alpha_1 \end{gathered}$$

The typing judgements are $\Gamma^\gamma \vdash M : \tau$ where Γ is a typing assumption, that is a mapping from a finite set of variables to types, and γ is a degree assumption, from the same set of variables to degrees. We use the following notations:

- we abusively denote by 0 and 1 the degree assumptions (of any appropriate domain) that uniformly assign respectively 0 and 1 to any variable.
- for any degree α and expression N, we denote by α_N the degree assumption defined by $\alpha_N(x) = \alpha$ if $x \in \mathsf{fv}(N)$, and $\alpha_N(x) = 1$ otherwise.
- $\delta \leqslant \gamma$ on X, where δ and γ are degree assumptions with the same domain and X is a set of variables, means that $\delta(x) \leqslant \gamma(x)$ for all $x \in X$.
- $\delta \wedge \gamma$ is the degree assumption pointwise defined by $(\delta \wedge \gamma)(x) = \delta(x) \wedge \gamma(x)$.

The typing rules are as follows:

$$\frac{\Gamma^\gamma \vdash M : \tau \qquad \delta \leqslant \gamma \text{ on } \mathsf{fv}(M)}{\Gamma^\delta \vdash M : \tau} \qquad \frac{}{\Gamma^1, x : \tau^0 \vdash x : \tau} \qquad \frac{\Gamma^\gamma, x : \theta^\alpha \vdash M : \tau}{\Gamma^1 \vdash \lambda x M : (\theta^\alpha \to \tau)}$$

$$\frac{\Gamma^\gamma \vdash M : \theta^\alpha \to \tau \qquad \Gamma^\gamma \vdash N : \theta}{\Gamma^{0_M \wedge \delta} \vdash (MN) : \tau} \quad \text{where} \quad \delta(x) = \begin{cases} \alpha & \text{if } N = x \\ (\alpha_N \wedge \gamma)(x) & \text{otherwise} \end{cases}$$

$$\frac{\Gamma^\gamma, x : \tau^1 \vdash M : \tau}{\Gamma^\gamma \vdash \mu x M : \tau}$$

The only constraint that may not be satisfied in typing a term is that any recursive variable must have degree 1 in the typing context for its body. There are three ways in which a variable may get degree 1 in the typing context for an expression M: either it does not occur in M, or x occurs either within abstractions in M or in arguments of "protective" functions, that are expressions of type $\theta^1 \to \tau$. For instance the function $\mathsf{K} = \lambda x \lambda y\, x$ is protective, since it puts its argument within an abstraction, and therefore $\mu z(\mathsf{K} z)$ is typable.