Extended recursive definitions in call-by-value languages with applications to mixin modules and recursive modules

Tom Hirschowitz — ENS Lyon
Xavier Leroy — INRIA Rocquencourt
J.B. Wells — Heriot-Watt University

Recursive definitions and call-by-value Call-by-name or lazy languages naturally support the evaluation of arbitrary recursive definitions $x = e[x]$. On-demand unrolling of e leads either to the fixpoint when it exists, or to divergence otherwise (as in $x = 1 + x$). This is not possible with a strict call-by-value strategy, where the right-hand side $e[x]$ must be evaluated exactly once and at definition time. Therefore, CBV languages must restrict the expressions $e[x]$ allowed as r.h.s. of recursive definitions – typically, in ML, $e[x]$ must be a function abstraction $\lambda x.e'$.

However, less drastic restrictions can remain compatible with CBV. A famous example is Scheme's `letrec` construct [8, 9], which builds on a "backpatching" imperative semantics. Operationally, to evaluate (`letrec` $(x\ e)\ f$), first, x is bound to a reference initialized to a special undefined value; then, e is evaluated (causing an error if the undefined value of x is used); finally, x is updated (via reference assignment) with the value thus obtained.

A variant of the backpatching scheme is the in-place update scheme [3] used in Objective Caml. Here, the additional reference is avoided by restricting the right-hand side to evaluate to a heap-allocated block of statically-predictable size; then, x is initialized to a block of that size containing undefined fields, which is eventually updated in place by copying the fields of the value of e.

Reduction semantics without a store A weakness of these extended CBV recursive definitions is that their semantics is given in terms of an updatable store. In [7, 5], we give a store-less reduction semantics that captures the expressiveness and limitations of the in-place update scheme. The semantics can be viewed as a CBV evaluation strategy imposed (via evaluation contexts) on Ariola and Blom's equational theory for recursive definitions [1]. The five fundamental rules, suitably restricted using evaluation contexts and side conditions to avoid variable capture, are the following (b ranges over mutually recursive bindings $x_1 = e_1, \ldots, x_n = e_n$).

$$\begin{array}{rcll}
C[\texttt{letrec}\ b\ \texttt{in}\ e] & \rightarrow & \texttt{letrec}\ b\ \texttt{in}\ C[e] & \text{lifting} \\
\texttt{letrec}\ b_1, x = (\texttt{letrec}\ b_2\ \texttt{in}\ e), b_3\ \texttt{in}\ f & \rightarrow & \texttt{letrec}\ b_1, b_2, x = e, b_3\ \texttt{in}\ f & \text{internal merging} \\
\texttt{letrec}\ b_1\ \texttt{in}\ \texttt{letrec}\ b_2\ \texttt{in}\ f & \rightarrow & \texttt{letrec}\ b_1, b_2\ \texttt{in}\ f & \text{external merging} \\
\texttt{letrec}\ b_1, x = e, b_2\ \texttt{in}\ C[x] & \rightarrow & \texttt{letrec}\ b_1, x = e, b_2\ \texttt{in}\ C[e] & \text{external substitution} \\
\texttt{letrec}\ b_1, x = e, b_2, y = C[x], b_3\ \texttt{in}\ f & \rightarrow & \texttt{letrec}\ b_1, x = e, b_2, y = C[e], b_3\ \texttt{in}\ f & \text{internal substitution}
\end{array}$$

We reflect the constraints imposed by the in-place update scheme in the semantics, using annotations on every recursive definition giving the expected size for the right-hand side if known: $x =_{[n]} e$ or $x =_{[?]} e$. The semantics gets stuck if the value of a not-yet-evaluated definition is needed.

Compilation scheme and its correctness We formalize the in-place update scheme as a translation from a pure CBV λ-calculus with `letrec` down to an imperative λ-calculus with explicit store, but no `letrec`. In the translation T, a source `letrec` definition becomes a series of `let` binding variables of known size n to initial blocks of size n, followed by left-to-right evaluation of the definitions: $x =_{[n]} e$ becomes `update`$(x, T(e)); \ldots$, while $x =_{[?]} n$ becomes `let` $x = T(e)$ `in` $\ldots$ We prove that this compilation scheme preserves the semantics of the source program: if e reduces to a value, or reduces infinitely, or gets stuck, then so does $T(e)$. The proof is not a simple simulation property "if $e \to e'$, then $T(e) \xrightarrow{*} T(e')$" (which does not hold), but makes use of a more complicated translation scheme S that performs some administrative reductions on the fly (i.e. $T(e) \xrightarrow{*} S(e)$) and is such that if $e \to e'$, then either e' and $S(e)$ are stuck, or $S(e) \xrightarrow{*} S(e')$, or there exists e'' such that $e' \to e''$ and $S(e) \xrightarrow{+} S(e'')$.

Application to mixin modules In [6, 5], we define a language of CBV mixin modules and a compilation scheme by translation to CBV λ-calculus with extended `letrec`. The gist of the translation is that output components of mixins become record fields parameterized by the input components referenced. For instance, a mixin structure with two inputs x, y and two outputs $y = e[x, y]$ and $z = f[x]$ becomes the record $\{y = \lambda x.\lambda y.e[x, y];\ z = \lambda x.f[x]\}$. Most mixin operators translate to simple record operations, directed by the types of the mixins. For instance, mixin composition $A + B$ becomes record concatenation. However, mixin operations that connect identically-named inputs and outputs, such as freezing and closing, give rise to recursive definitions of the form $x = A.f\ x$. For instance, closing a mixin A with inputs x, y and outputs x, y, where x depends on x and y on x and y, is compiled to `letrec` $x = A.x\ x, y = A.y\ x\ y$ `in` $\{x = x; y = y\}$. These recursive definitions can be handled by our extended CBV `letrec` construct, and compiled to efficient imperative code using the in-place update scheme. The expected size annotations can be derived from the types of the source mixins.

Application to recursive modules We have recently extended the Objective Caml language with the ability to define structures (basic modules) via mutually-recursive definitions. Most practical uses of recursively-defined modules involve functor applications in the right-hand sides, as in

```
    module rec A : SIGA = struct ... ASet.t ... end and ASet : SIGASET = Set.Make(A)
```

It is therefore important that this construct does not restrict the r.h.s. to be syntactic structures `struct`... `end`. Again, the compilation of these recursive module definitions relies on the in-place update scheme, using the signatures of the structures to determine the expected sizes. An issue that remains largely open is the static detection of ill-founded recursive module definitions, which is currently handled by an ad-hoc combination of typing restrictions and initialization of structure fields to "bottom" values such as $\lambda x.$ `raise Undefined` for function types.

Enforcing correct recursive definitions via static typing As mentioned earlier, our extended CBV recursive binding cannot evaluate arbitrary definitions. In particular, it gets stuck on ill-founded definitions ($x = x + 1$) and on definitions where a value is needed too early ($x = f\ 0, f = \lambda y.1$). It is therefore desirable to statically rule out these non-evaluable definitions using a type system. The first such type system was given by Boudol [2]; it relies on annotated function types $\tau \xrightarrow{1} \sigma$ and $\tau \xrightarrow{0} \sigma$ to denote functions that are non-strict (1) or possibly strict (0) in their argument. Then, a recursive definition $x = f\ x$ is allowed only if f has a $\xrightarrow{1}$ type. This type system was refined by Hirschowitz [5] to handle curried applications. There, function types $\tau \xrightarrow{n} \sigma$ are annotated by an integer n giving the number of function applications that can be performed without using the value of the parameter of type τ. This type system is expressive enough to show that the terms arising from the translation of mixin modules are well-typed, and therefore evaluate safely. Recent work by Dreyer [4] develops a more powerful type and effect systems that tracks references to recursively-defined variables as formal computational effects. All these type systems are intended for intermediate languages and are arguably too complex to be applied to source languages and exposed to the programmers. For the application to Caml's recursive modules, we are currently looking for a simpler (and less expressive) extension to the type system of the module language that rules out bad recursive definitions.

References

[1] Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of pure and applied logic*, 117(1–3):95–178, 2002.

[2] G. Boudol. The recursive record semantics of objects revisited. In D. Sands, editor, *European Symposium on Programming*, volume 2028 of *LNCS*, pages 269–283. Springer-Verlag, 2001.

[3] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[4] D. Dreyer. A type system for well-founded recursion. In *31st symp. Principles of Progr. Lang*, pages 293–305. ACM Press, 2004.

[5] T. Hirschowitz. *Mixin modules, modules, and extended recursion in a call-by-value setting.* PhD thesis, University Paris 7, Dec. 2003.

[6] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In D. Le Métayer, editor, *Programming Languages and Systems, ESOP'2002*, volume 2305 of *LNCS*, pages 6–20. Springer-Verlag, 2002.

[7] T. Hirschowitz, X. Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *Int. Conf. Principles and Practice of Declarative Progr.*, pages 160–171. ACM Press, 2003.

[8] R. Kelsey, W. Clinger, and J. Rees. The revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

[9] O. Waddell, D. Sarkar, and R. K. Dybvig. Robust and effective transformation of letrec. In *Electronic proceedings of the 2002 Scheme Workshop*, 2002.