Generating Mutually Recursive Definitions
(Short Paper)

Jeremy Yallop
University of Cambridge, UK
jeremy.yallop@cl.cam.ac.uk

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

Abstract
Many functional programs — state machines [10], top-down and bottom-up parsers [3, 4], evaluators [1], GUI initialization graphs [15], &c. — are conveniently expressed as groups of mutually recursive bindings. One therefore expects program generators, such as those written in MetaOCaml, to be able to build programs with mutual recursion.

Unfortunately, currently MetaOCaml can only build recursive groups whose size is hard-coded in the generating program. The general case requires something other than quotation, and seemingly weakens static guarantees on the resulting code. We describe the challenges and propose a new language construct for assuredly generating binding groups of arbitrary size — illustrating with a collection of examples for mutual, n-ary, heterogeneous, value and polymorphic recursion.

1 Introduction
MetaOCaml (whose current implementation is known as BER MetaOCaml [9]) extends OCaml with support for typed program generation. It makes three additions: a code is the type of unevaluated code fragments, brackets .<>. construct a code fragment by quoting an expression, and splices .-e insert a code fragment into a larger one.

For example, here is a function t1 that builds an int code fragment by inserting its int code argument within a bracketed expression:

```ocaml
let t1 x = .< -x * succ -x >.
```

and here is a call to t1 with a code fragment of the appropriate type:

```ocaml
let p12 = .< 1 + 2 >.
let c1 = t1 p12
```

Combined with higher-order functions, effects, modules and other features of the host OCaml language, these constructs support safe and flexible program generation, permitting typed manipulation of open code while ensuring that the generated code is well-scoped and well-typed.

```not_function
let rec s = function A : r → s r \ B : r → t r | [] → true
   and t = function A : r → s r \ B : r → u r | [] → false
   and u = function A : r → t r \ B : r → u r | [] → false
```

where each function s, t, and u realizes a recognizer, taking a list of A and B symbols and returning a boolean. However, the program that builds such a group from a description of an arbitrary state machine cannot be expressed in MetaOCaml.

The limited support for generating mutual recursion is a consequence of expression-based quotation: brackets enclose expressions, and splices insert expressions into expressions — but a group of bindings is not an expression. There is a second difficulty: generating recursive definitions with ‘backward’ and ‘forward’ references seemingly requires unrestricted, Lisp-like gensym, which defeats MetaOCaml’s static guarantees. It is unclear how to ensure all gensym-ed variables are eventually bound to the intended expressions, and how to ensure that generated code is well-typed.

Related metaprogramming systems such as LMS [12] and Template Haskell [13] which are capable of generating recursive definitions indeed use gensym and ‘compiler magic’ (such as intensional analysis of closures in LMS and dependency analysis in GHC to determine mutually recursive groups).

In practice, MetaOCaml programmers fall back on a variety of workarounds, simulating mutual recursion using ordinary recursion [8] or nested recursion [5], encoding recursion using higher-order state (“Landin’s knot”) [17] or hard-coding templates for a fixed few numbers of binding-group sizes [18]. None of the workarounds are satisfactory:
they do not cover all use cases, are awkward to use, or generate inefficient programs that rely on references or auxiliary data structures.

This paper solves these challenges. Specifically, it describes:

- a low-level primitive for recursive binding insertion (Section 3), building on earlier designs for insertion of ordinary let bindings (Section 2)
- a high-level combinator built on top of the low-level primitive (Section 4) that supports the generation of a wide variety of recursive patterns — mutual, \(n\)-ary, heterogeneous, value and polymorphic recursion.

2 Let-insertion

The code generated for \(c1\) above contains duplicate expressions, which ideally should be computed only once. We can avoid the duplicated computation by changing \(t1\) to generate a let expression:

\[
\text{let } t2 x = \langle\text{let } y = -x \text{ in } y \times \text{succ } y\rangle.
\]

\[
\text{let } c2 = t2 p12
\]

\[
\sim \text{val } c2 : \text{int code} = \langle\text{let } y_1 = 1 + 2 \text{ in } y_1 \times (\text{succ } y_1)\rangle.
\]

However, in general let expressions cannot be inserted locally. For example, in the following program, \(ft1\) takes a code template \(t\) as argument, using it when building the body of the generated function:

\[
\text{let } ft1 t x = \langle\text{fun } u \rightarrow -(t x) + -(t (-x + u))\rangle.
\]

\[
\sim \text{val } ft1 : (\text{int code} \rightarrow \text{int code}) \rightarrow \text{int code} \rightarrow (\text{int} \rightarrow \text{int}) \text{ code}
\]

Now the let expression generated by \(t2\) is not positioned optimally:

\[
\text{let } c3 = \text{ft1 } t2 \text{ p12};
\]

\[
\sim \text{val } c3 : (\text{int} \rightarrow \text{int}) \text{ code} = \langle\text{fun } u_2 \rightarrow (\text{let } y_4 = 1 + 2 \text{ in } y_4 \times (\text{succ } y_4)) + (\text{let } y_3 = (1 + 2) + u_2 \text{ in } y_3 \times (\text{succ } y_3))\rangle.
\]

since we do not wish to compute 1+2 every time the function generated by \(c3\) is applied. The challenge is inserting let bindings into a wider context rather than into the immediate code fragment under construction.

Recent versions of BER MetaOCaml have a built-in genlet primitive: if \(e\) is a code value, then genlet \(e\) arranges to generate, at an appropriate place, a let expression binding \(e\) to a variable — returning the code value with just that variable. (If \(e\) is already an atomic expression, genlet \(e\) returns \(e\) as it is).

For example, in the following program \(p12\) is bound to a code expression 1+2 that is to be let-bound according to the context. When \(p12\) is printed — that is, used in the top-level context — the let is inserted immediately:

\[
\text{let } p12 = \text{genlet } p12
\]

\[
\sim \text{val } p12 : \text{int code} = \langle\text{let } l_5 = 1 + 2 \text{ in } l_5\rangle.
\]

If we pass \(p12\) to \(t1\), the let is inserted outside the template’s code:

\[
\text{let } c1l = t1 \text{ p12l}
\]

\[
\sim \text{val } c1l : \text{int code} = \langle\text{let } l_5 = 1 + 2 \text{ in } l_5 \times \text{succ } l_5\rangle.
\]

Finally, in the complex \(ft1\) example, the let-binding happens outside the function, as desired:

\[
\text{let } c3l = \text{ft1 } t1 \text{ p12l};
\]

\[
\sim \text{val } c3l : (\text{int} \rightarrow \text{int}) \text{ code} = \langle\text{fun } u_{10} \rightarrow \text{let } l_{11} = l_5 + u_{10} \text{ in } l_5 \times \text{succ } l_5 + l_{11} \times \text{succ } l_{11}\rangle.
\]

Let-insertion and memoization Let-insertion is often used with memoization, as we illustrate with a simplified dynamic-programming algorithm [6]. The \(fibnr\) function computes the \(n\)th element of the Fibonacci sequence whose first two elements are given as arguments \(x\) and \(y\):

\[
\text{let } \text{fibnr } x y \text{ self } n =
\]

\[
\begin{align*}
&\text{if } n = 0 \text{ then } x \text{ else } \\
&\text{if } n = 1 \text{ then } y \text{ else } \\
&\text{plus (self (n-1)) (self (n-2))}.
\end{align*}
\]

The code is written in open-reursive style, and abstracted over the addition operation. Tying the knot with the standard call-by-value fixpoint combinator \(\text{let rec}\) \(\text{fix}\) \(f x = f (\text{fix } f) x\) we compute, for example, the 5th element of the standard sequence as \(\text{fix (fibnr (+ 1) 1) 5}\).

If, instead of passing the standard addition function + for \(\text{fibnr}\)’s plus argument, we pass a code-generating implementation of plus then \(\text{fibnr}\) also becomes a code generator, here building code that computes the 5th element, given the first two:

\[
\text{let } \text{splus } x y = \langle-x - y\rangle. \text{ in}
\]

\[
\begin{align*}
&\langle\text{fun } x y \rightarrow -(\text{fix (splus })<\langle x \rangle \ldots <\langle y \rangle >) 5)\rangle. \\
&\sim \text{val } c1l : \text{int code} = \langle\text{fun } x_1 y_2 \rightarrow ((y_2 + x_1) + y_2) + (y_2 + x_1) + ((y_2 + x_1) + y_2)\rangle.
\end{align*}
\]

The duplicated expressions in the generated code reveal why \(\text{fibnr}\) is exponentially slow.

A memoizing fixpoint combinator inserts a let-binding for the result of each call, and maintains a mapping from previous arguments to the let-bound variables [14]

\[
\text{let } \text{mfix } (f : (\alpha \rightarrow \beta \text{ code})) \rightarrow (\alpha \rightarrow \beta \text{ code}) (x : \alpha) : \beta \text{ code} = \\
\text{let memo } = \text{ref }[] \text{ in}
\]

\[
\text{let rec loop } n = \text{try List.assoc } n !\text{memo}
\]

\[
\begin{align*}
&\text{with } \text{Not found } \rightarrow \text{let } v = \text{genlet } (f \text{ loop } n) \text{ in} \\\n&\text{memo } := (n,v) :: !\text{memo}; v
\end{align*}
\]

\[
\text{in } \text{loop } x
\]

letting us to compute the \(n\)-th element fast and generate fast code:

\[
\begin{align*}
&\langle\text{fun } x y \rightarrow -(\text{mfix (splus })<\langle x \rangle \ldots <\langle y \rangle >) 5)\rangle. \\
&\sim \text{val } c1l : \text{int code} = \langle\text{fun } x_5 y_6 \rightarrow \text{let } l_7 = y_6 + x_5 \text{ in } l_8 = l_7 + y_6 \text{ in}
\end{align*}
\]
Generating Mutually Recursive Definitions

Without genlet however, we get the same poor code as with the ordinary fix: memoization alone speeds up the code generation without affecting the efficiency of the generated code. The crucial role of let-insertion in these applications has been extensively discussed by Swadi et al. [14].

3 Inserting recursive let

As we have seen, the specialization of recursive functions calls for generating definitions. More complicated recursive patterns require generating recursive definitions. The simplest example is specializing the Ackermann function

```plaintext
let rec ack m n =
  if m = 0 then n + 1 else
  if n = 0 then ack (m - 1) 1 else
  ack (m - 1) (ack m (n - 1))
```

for a given value of \( m \), a challenge originally posed by Neil Jones. Turning ack into a generator of specialized code is easy in the open-recursion style, by merely annotating the code keeping in mind that \( n \) is future-stage:

```plaintext
let rec tack self m n =
  if m = 0 then <.n+1> else
    .<if .n = 0 then -(self (m-1)) 1 else
      -(self (m-1)) -(self m) -(n-1)>
    .
  
  ∼ val tack : (int→(int→int code)) → int→int code→int code

All that is left is to set the desired value of \( m \) and apply the mfix — which promptly diverges:

```plaintext
mfix (fun self m → .<fun n → -(tack self m).n>).) 2
```

Looking at the original ack shows the reason: \( m \) depends not only on \( m \) (also) but also on \( m \) itself.

Generating recursive definitions was deemed for a long time a difficult problem. One day, a two-liner solution emerged, from the insight that a recursive definition let rec \( g = e \) in body may be re-written as

```plaintext
let g = let rec g = e in g in body
```

which immediately gives us genletrec:

```plaintext
let genletrec : ((α→β code → α code → β code) → (α→β) code =
  fun f → genlet. let rec g x = .(f .<g> .<x>).) in g.
```

The new memoizing fixpoint combinator becomes

```plaintext
let mfix : ((α→β) code → (α→β code →γ code)) → ((α→β) code =
  fun f x →
  let memo = ref ([],[]) in
  let rec loop n =
    try List.assoc n (fst !memo @ snd !memo)
    with Not_found →
      let v = genletrec (fun y →
        let old = snd !memo in
        memo := (fst !memo, (n,g) :: old);
        let v = (f loop n y) in

  mfix v
```

Recursive definitions have to be the definitions of functions: the fact reflected in mfix’s (and genletrec’s) code and type. The mfix code has another peculiarity: splitting of the memo table into the ‘global’ and ‘local’ parts. We let the reader contemplate its significance (until we return to this point in Section 4).

Finally we are able to specialize the Ackermann function to a particular value of \( m \) (which is two, in the code below):

mfix tack 2

```plaintext
∼ ∼ : (int → int code =
  .< let l13 = let rec g11 x12 = x12 + 1 in g11 in
    let l14 = let rec g9 x10 =
      if x10 = 0 then l13 1 else l13 (g9 (x10 - 1))
    in g9 in
    let l15 = let rec g7 x8 =
      if x8 = 0 then l14 1 else l14 (g7 (x8 - 1))
    in g7
    in l15>
```

One clearly sees recursive definitions that were not present in the original ack.

4 Generating mutually-recursive functions

In many practical cases of generating recursive definitions one wants to produce mutually recursive definitions, such as the state machine shown in Section 1. To illustrate the challenges brought by mutual recursion, we take a simpler running example, contrived to be in the shape of the earlier Ackermann function. The example is the ‘classical’ even-odd pair, but taking two integers \( m \) and \( n \) and returning a boolean, telling if the sum \( m+n \) has even or odd parity, resp.

```plaintext
let rec even m n = if n>0 then odd m (n-1) else
  if m>0 then odd (m-1) n else true
and odd m n = if n>0 then even m (n-1) else
  if m>0 then even (m-1) n else false
```

At first, mutual recursion seems to pose no problem: after all, a group of mutually recursive functions may always be converted to the ordinary recursive function by adding an extra argument: the index of a particular recursive clause in the group:

```plaintext
type evod = Even | Odd
```

```plaintext
let rec evodf self idx m n =
  match idx with
  | Even f loop n y =
```
which looks very much like tack. To find out if the sum of m and n has even parity we write:

\[
\begin{align*}
\text{let rec } \text{sevodf } \text{self idxx m n } &= \text{match idxx with} \\
| \text{Even } &\to \text{if n>0 then self Odd m (n-1) else} \\
&\text{if m>0 then self Even m (n-1) else true} \\
| \text{Odd } &\to \text{if n>0 then self Even m (n-1) else} \\
&\text{if m>0 then self Odd m (n-1) else false} \\
\end{align*}
\]

To find out if the sum of 10 and 42 has even parity one writes:

\[
\text{let rec } \text{sevodf } \text{self idxx m n } = \\
\text{match idxx with} \\
| \text{Even } &\to \text{if n>0 then self Odd m (n-1) else} \\
&\text{if m>0 then self Even m (n-1) else true} \\
| \text{Odd } &\to \text{if n>0 then self Even m (n-1) else} \\
&\text{if m>0 then self Odd m (n-1) else false} \\
\text{~ val } \text{sevodf : (evod } \to \text{int } \to \text{int } \to \text{bool } \to \text{int } \to \text{int } \to \text{bool code} \\
\text{evod } \to \text{int } \to \text{int code } \to \text{bool code}
\]

which looks very much like tack from Section 3. We could thus apply m fix to that section with some adaptations and obtain the code for even m n specialized to a particular value of m, say 0 (which is just the ordinary even function):

\[
\begin{align*}
\text{mrfix } \text{(fun self (idxx,m) x } \to \text{ evodf } \text{(fun idxx m } \to \text{ self (idxx,m)) idxx m x}) \\
\text{(Even,0)} \\
\text{~ val } \text{sevodf : (evod } \to \text{int } \to \text{int } \to \text{bool code} \\
\text{evod } \to \text{int } \to \text{int code } \to \text{bool code}
\end{align*}
\]

The earlier genlet (and, hence genletrec) inserted the requested definition in the widest possible context (while ensuring the absence of unbound variables in the generated code). With the new interface the insertion point (and hence the scope of the inserted bindings) is explicitly marked using genletrec_locus and each call to genletrec indicates which group of recursive bindings should contain the generated definition. Correspondingly, in a call genletrec locus ((fun g x } \to \text{...})\text{the identifier for the binding (bound to g) scopes beyond genletrec’s body (but within the scope denoted by locus). We would like to generate the mutually recursive definition:

\[
\begin{align*}
\text{let rec } od0 n &\equiv \text{if } n>0 \text{ then ev1 (n-1) else lev0 n in od1 in} \\
\text{if } n>0 \text{ then lod1 (n-1) else lod0 n in ev1 in lev1}
\end{align*}
\]

Here, the function ev1, the specialization of even m n to m=1 calls od0 and od1. The latter calls ev1 and fun n } \to \text{...}, whose code was already generated and memoized, under the name lev0. Unfortunately, the scope of lev0 does not extend beyond the scope of od0 definition, and hence mentioning lev0 within od1 is scope extrusion.

The generator of mutually recursive bindings has to be a MetaOCaml primitive. What should be its interface? After quite a bit of thought, it turns out that genletrec, if made primitive, would suffice. For the sake of better error detection, one would generalize it slightly. We add a second function, genletrec_locus, which marks the location where a group of recursive definitions should be inserted; the generated locus_t value representing the location can be passed as first argument of genletrec:

\[
\begin{align*}
\text{type locus_t } \\
\text{val genletrec_locus : (locus_t } \to \text{ α code } \to \text{ α code} \\
\text{val genletrec : locus_t } \to \text{ ((α} \to \text{β) code } \to \text{ α code } \to \text{ β code } \to \text{ (α} \to \text{β) code})
\end{align*}
\]

The earlier genlet (and, hence genletrec) inserted the requested definition in the widest possible context (while ensuring the absence of unbound variables in the generated code). With the new interface the insertion point (and hence the scope of the inserted bindings) is explicitly marked using genletrec_locus and each call to genletrec indicates which group of recursive bindings should contain the generated definition. Correspondingly, in a call genletrec locus ((fun g x } \to \text{...})\text{the identifier for the binding (bound to g) scopes beyond genletrec’s body (but within the scope denoted by locus). The new genletrec let us write m fix essentially just like the simpler m fix, without the splitting of the memo table

\[
\begin{align*}
\text{let lod0 } &\equiv \text{ (\star odd 0 n \star) } \\
\text{let rec } od0 n &\equiv \text{ (\star even 0 n \star) } \\
\text{let rec } ev0 n &\equiv \text{ if n>0 then od0 (n-1) else false in ev0 in} \\
\text{if } n>0 \text{ then lev0 (n-1) else false in od0 in} \\
\text{let lev1 } &\equiv \text{ (\star even 1 n \star) } \\
\text{let rec } ev1 n &\equiv \text{ (\star odd 1 n \star) }
\end{align*}
\]
Generating Mutually Recursive Definitions

into global and local parts: now, the definitions have the same scope.

   let mrfix : (\(\alpha \to (\beta \to \gamma)\) code) \to (\(\alpha \to \beta\) code \to \gamma\) code) \to
   (\(\alpha \to (\beta \to \gamma)\) code) =
   fun f x \to
   genletrec_locus @@ fun locus \to
   let memo = ref [] in
   let rec loop n =
   try List.assoc n !memo
   with Not_found \to
   genletrec locus (fun g y \to
   memo := (n,g) :: !memo;
   f loop n y)
   in loop x

With this new mrfix but the same sevof from Section 4 we are able to generate the specialized even \(1\) \(n\) code, with four mutually recursive definitions.

Finite State Automata, reprise Recognizers of finite state automata are produced by the following generic, textbook generator:

   type token = A | B
   type state = S | T | U
   type (\(\alpha,\sigma\)) automaton = {finals: \(\sigma\) list;
   trans: (\(\alpha \times (\alpha \times \sigma)\) list) list}

   let makeau [finals;trans] self state stream =
   let accept = List.mem state finals in
   let next token = List.assoc token (List.assoc state trans) in
   \(\text{.<}\) match \(\text{-stream with}\)
   | A :: r \to .-(self (next A)) r
   | B :: r \to .-(self (next B)) r
   | [] \to accept >.

In particular, the automaton in Section 1 is represented by the following description

   let au1 =
   [finals = [S];
   trans = [(S, [(A, S); (B, T)]);
   (T, [(A, S); (B, U)]);
   (U, [(A, T); (B, U)])]

Then mrfix (makeau au1) S generates:

\[\text{let rec } x_1 y = \text{match } y \text{ with } A:r \to x_1 r \]
\[| B:r \to x_5 r \]
\[| [] \to \text{true}\]
\[\text{and } x_5 y = \text{match } y \text{ with } A:r \to x_1 r \]
\[| B:r \to x_9 r \]
\[| [] \to \text{false}\]

5 Further extensions

We sketch some extensions to the mrfix combinator of Section 4.

5.1 Arbitrary bodies in let rec expressions

The mrfix combinator has the following type:

   val mrfix : ((\(\alpha \to (\beta \to \gamma)\) code) \to (\(\alpha \to \beta\) code \to \gamma\) code)) \to
   \(\alpha \to (\beta \to \gamma)\) code

There are two arguments: the first is a function that builds recursive definitions; the second (of type \(\alpha\)) is an index that selects the identifier associated with one of the definitions to appear in the body of the generated let rec expression. For example, in the code generated for the Ackermann function by the call mrfix tick 2 in Section 3, the body of the generated expression is \(1_{5}\), the identifier associated with the definition generated by tick 2. And in the code generated for the finite state automaton in Section 4 the body of the generated expression is \(x_1\), the name of the function that corresponds to the start symbol.

However, it is sometimes convenient to generate let rec expressions with bodies that are more complex than single identifiers. The following function, mrfixk, generalizes mrfix to additionally support generation of arbitrary bodies:

   val mrfixk : ((\(\alpha \to (\beta \to \gamma)\) code) \to (\(\alpha \to \beta\) code \to \gamma\) code)) \to
   ((\(\alpha \to (\beta \to \gamma)\) code) \to \gamma\) code

Rather than an index, the second argument is now a function that calls its argument to insert recursive definitions and builds a body of type \(\gamma\) code. For example, here is the code that builds a recursive group representing the state machine from previous examples, whose body is a tuple returning all the recognizer functions:

   mrfixk (makeau au1) \((\text{fun } f \to \text{.<} \text{-}(\text{-}(f S), \text{-}(f U), \text{-}(f T)) >)\)

The generated code is the same as the code generated by mrfix, except for the more complex body:

   let rec x_1 y = \text{match } y \text{ with } A:r \to x_1 r \]
   \[| B:r \to x_5 r \]
   \[| [] \to \text{true}\]
\[\text{and } x_5 y = \text{match } y \text{ with } A:r \to x_1 r \]
\[| B:r \to x_9 r \]
\[| [] \to \text{false}\]
5.2 A syntax extension

Third-order functions such as \( \text{mrfixk} \) are not always easy to understand and use. The following small syntax extension improves readability in many cases:

\[
\begin{align*}
\text{let}\%\text{staged rec } & f \ p \ p' = e \text{ in } e' \\
\sim & \text{mrfixk}(\text{fun } f \ p \ p' \to e)(\text{fun } f \to e')
\end{align*}
\]

Here \( %\text{staged} \) is an attribute that indicates the need for a rewrite by an plug-in program that expands the syntax as shown above.

Then ack can be written as follows

\[
\begin{align*}
\text{let}\%\text{staged rec ack m n} = & \\
\text{if } m = 0 \text{ then } <.-n+1> \text{. else} \\
<.if \ .-n = 0 \text{ then } -.\text{ack}(m-1)> 1 \text{ else} \\
-.\text{ack}(m-1)>.&
\end{align*}
\]

\[
\text{in ack 2}
\]

As this example shows, the syntax extension avoids the need for explicitly higher-order code and for open recursion; the identifier \( \text{ack} \) serves as the self argument in the expanded syntax, and so the calls to \( \text{ack} \) appear as standard recursion.

5.3 Heterogeneously-typed recursive groups

In the examples up to this point the bindings in each recursive group have all been of a single type. In practice, however, it is common for \( \text{let rec} \) to bind definitions of different types. Supporting this general case requires several changes to the type of the fixpoint combinator to make it more polymorphic.

The central idea is to generalize the index types used to select recursive bindings from regular algebraic data types to GADTs⁵. For example, the following GADT supports generating mutually-recursive bindings for functions of types \( \text{int} \to \text{bool} \) and \( \text{float} \to \text{bool} \)

\[
\text{type } \alpha \text{ eo} = \begin{cases} 
\text{Even} : (\text{int} \to \text{bool}) \text{ eo} \\
| \text{Odd} : (\text{float} \to \text{bool}) \text{ eo}
\end{cases}
\]

The type of the \( \text{mrfixk} \) function is generalized accordingly:

\[
\begin{align*}
\text{val } \text{mrfixk} : \forall \gamma. & \forall \beta. (\forall \alpha. (\alpha \to \beta) \to \gamma \to \beta) \to \\
& (\forall \alpha. (\alpha \to \beta) \to \gamma \to \beta) \to \\
& (\forall \alpha. (\alpha \to \beta) \to \gamma \to \beta)
\end{align*}
\]

Since the higher-rank and higher-kindled polymorphism found in this type cannot be expressed directly in OCaml, our implementation uses standard encodings based on OCaml’s polymorphic record fields and functors.

5.4 Polymorphic recursion

The extensions needed to support heterogeneous recursion (Section 5.3) are also sufficient to support polymorphic recursion. For example, here is a nested data type \( \text{ntree} \) of perfectly balanced trees and a polymorphic function \( \text{swivel} \) that interchanges left and right elements of \( \text{ntree} \) values:

\[
\begin{align*}
\text{type } & \alpha \text{ ntree} = \text{EmptyN} | \text{TreeN of } \alpha \times (\alpha \times \alpha) \text{ ntree} \\
\text{let rec } & \text{swivel : } \alpha. (\alpha \to \alpha) \to \alpha \text{ ntree } \to \alpha \text{ ntree } = \\
& \text{fun } f \ t \to \text{match } t \text{ with} \\
& | \text{EmptyN } \to \text{EmptyN} \\
& | \text{TreeN}(v,t) \to \text{TreeN}(f v, \text{swivel}(\text{fun } (x, y) \to (f y, f x)) \ t)
\end{align*}
\]

This is polymorphic-recursive because the recursive call uses \( \text{swivel} \) at a different type to the type of the definition: the passed function \( f \) acts on pairs \( \alpha \times \alpha \), not values of type \( \alpha \).

Generating polymorphic-recursive definitions like \( \text{swivel} \) involves indexing by a polymorphic type. Here is a suitable index for generating \( \text{swivel} \):

\[
\begin{align*}
\text{type } & \text{swivel} = \{ \text{swivel : } \alpha. (\alpha \to \alpha) \to \alpha \text{ ntree } \to \alpha \text{ ntree} \} \\
\text{type } & \_ \text{ index} = \text{Swivel} : \text{swivel index}
\end{align*}
\]

At each use of the index the polymorphic record field can be instantiated afresh, making it possible to call the generated function recursively at any instance of the type \( \alpha \to \alpha \) \( \to \alpha \text{ ntree } \to \alpha \text{ ntree} \).

Acknowledgments

We thank Jun Inoue for earlier discussions and posed challenges and Atsushi Igarashi for hospitality. We are grateful to anonymous reviewers for many helpful suggestions.

Status

Currently the proof-of-concept of the described genletrec is prototyped⁶ using plain MetaOCaml as well as MetaOCaml with delimited control effects, such as those provided by Multicore OCaml [2] or the delimcc library [7]. We are working at supporting it above-the-board in a forthcoming release of MetaOCaml.

References

[4] URL elided for anonymous review

⁵This (progressively fancier and fancier) indexing is closely related to the generalized arity in Plotkin and Power’s formulation of algebraic effects [11].


