Generating Mutually Recursive Definitions

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 (Krishnamurthi 2006), top-down and bottom-up parsers (Hutton and Meijer 1996; Hinze and Paterson 2003), evaluators (Abelson et al. 1984), GUI initialization graphs (Syne 2006), &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 (Kiselyov 2014)) extends OCaml with support for typed program generation. It makes three additions:

• code is the type of unevaluated code fragments, brackets \langle e \rangle .
• construct a code fragment by quoting an expression, and splices \langle e \rangle 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:

\begin{verbatim}
let t1 x = \langle x + 1 \rangle
\end{verbatim}

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

\begin{verbatim}
let y = \langle x + 1 \rangle
\end{verbatim}

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-scooped and well-typed.

However, support for generating recursive programs is currently limited: there is no support for generating mutually-recursive definitions whose size is not hard-coded in the generating program (Taha 1999). For example, the following state machine:

\begin{center}
\begin{tikzpicture}[->,>=stealth',shorten >=1pt,auto, semithick]

\node[state] (s) at (0,0){s};
\node[state] (t) at (1,0){t};
\node[state] (u) at (2,0){u};
\node (A) at (2,1){A};
\node (B) at (2,-1){B};
\draw (s) edge [] node [left] {A} (t);
\draw (u) edge [] node [right] {B} (t);
\draw (t) edge [] node [above] {B} (u);
\draw (s) edge [loop above] node {A} (s);
\end{tikzpicture}
\end{center}

is naturally expressed as a mutually-recursive group of bindings:

\begin{verbatim}
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
\end{verbatim}

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.

In practice, MetaOCaml programmers fall back on a variety of workarounds, simulating mutual recursion using ordinary recursion (Kiselyov 2013) or nested recursion (Inoue 2014), encoding recursion using higher-order state (“Landin’s knot”) (Yallop 2016) or hard-coding templates for a few fixed numbers of binding-group sizes (Yallop 2017). 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 (§3), building on earlier designs for insertion of ordinary let bindings (§2)

• a high-level combinator built on top of the low-level primitive (§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:

\begin{verbatim}
let t2 x = \langle \text{let } y = x \text{ in } y * (x + 1) \rangle
\end{verbatim}

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:

\begin{verbatim}
let ft1 t x = \langle \text{fun } u \rightarrow (t x) + \langle t \cdot x + u \rangle \rangle
\end{verbatim}

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

\begin{verbatim}
let c3 = ft1 t2 p12
\end{verbatim}

\begin{verbatim}
let c3 : (int \rightarrow int) code = \langle \text{fun } u2 \rightarrow (let y4 = 1 + 2 \text{ in } y4 * (succ y4)) + (let y3 = 1 + 2 + u2 \text{ in } y3 * (succ y3)) \rangle.
\end{verbatim}
since we do not wish to compute $1 + 2$ every time the function
generated by $c3$ is applied. The challenge is inserting let bindings
to 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 p12l is bound to a code
expression $1 + 2$ that is to be let-bound according to the context.
When p12l is printed — that is, used in the top-level context — the
let is inserted immediately:

\begin{verbatim}
let p12l = genlet p12
  ~ val p12l : int code = .<let l3 = 1 + 2 in l3>.
\end{verbatim}

If we pass p12l to t1, the let is inserted outside the template’s code:

\begin{verbatim}
let c1l = t1 p12l
  ~ val c1l : int code = .<let l5 = 1 + 2 in l5 * succ l5>.
\end{verbatim}

Finally, in the complex if1 example, the let-binding happens out-
side the function, as desired:

\begin{verbatim}
let ft1 x = .<fun u = .("(t1 x) + ."(t1 (genlet .<.x + u>).)).>.
let c3l = ft1 p12l;
  ~ val c3l : (int -> int) code = .<let l9 = 1 + 2 in
  fun u10 -> let l11 = l5 + u10 in l5 * succ l5 + l11 * succ l11>.
\end{verbatim}

**Let-insertion and memoization** Let-insertion is often used with
memoization, as we illustrate with a simplified dynamic-programming
algorithm (Kameyama et al. 2011). The fibnr function computes the
nth element of the Fibonacci sequence whose first two elements are
given as arguments x and y:

\begin{verbatim}
let fibnr plus x y self n =
  if n=0 then x else
  plus (self (n-1)) (self (n-2))
\end{verbatim}

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

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

\begin{verbatim}
let plus x y = <.x + .> y . in
  .<fun x y = ."(fix (fibnr plus .<x>. .<y>).) 5>.
  ~ = : (int -> int -> int) code =
    .<fun x1 y2 -> .((y2 + x1) + (x2 + y1)) + y2 + y1 + x1 + y1>.
\end{verbatim}

The duplicated expressions in the generated code reveal why fibnr
is exponentially slow.

A memoizing fixpoint combinator inserts a let-binding for the result
of each call, and cached a mapping from previous arguments to the
let-bound variables (Swadi et al. 2006)

\begin{verbatim}
let mfix f : (a -> b code) -> (a -> b code) x : a : b code =
  let memo = ref [] in
  let rec loop n =
    try List.assoc n !memo with Not_found ->
    let v = genletrec (fun g y ->
      let old = snd !memo in
      memo := (fst !memo, (n,g) :: old); v in
      let v = (f loop n y) in
      memo := (fst !memo, old); v) in
    in loop x
\end{verbatim}

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

\begin{verbatim}
<fun x y -> ."(mfix (fibnr plus .<x>. .<y>).) 5>.
\end{verbatim}

Without genlet however, we get the same poor code as with the ordin-
tary 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. (2006).

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 special-
izing the Ackermann function

\begin{verbatim}
let rec ack m n =
  if m = 0 then n + 1
  else
    let t1 = n in
    let t2 = m in
    let t3 = t1 t2 t2 t3 in
    t2 t1 t1 t2 t1 t3 t2 t3 t2 t1 t1.
\end{verbatim}

for a given value of m. 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:

\begin{verbatim}
let tack self m n =
  if m = 0 then ."<.x + n+1>. else
    ."(tack self m ."(self (m-1)) 1 else
    ack (m-1) (ack m (m-1))
\end{verbatim}

Looking at the original ack shows the reason: ack m depends not
only on ack (m−1) but also on ack 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

\begin{verbatim}
let g = let rec g = e in g in body
\end{verbatim}

which immediately gives us genletrec:

\begin{verbatim}
let genletrec : ((α -> β) code -> α code -> β code) -> (α -> β) code =
  fun f -> genlet .<let rec g x = ."(f ."<g>. .<x>).) in g>.
\end{verbatim}

The new memoizing fixpoint combinator becomes

\begin{verbatim}
let mfix : ((α -> (β -> γ) code) -> (α -> β code -> γ code)) ->
  (α -> (β -> γ) code) =
  fun f x ->
  let memo = ref [()] in
  let rec loop n =
    try List.assoc n !memo with Not_found ->
    let v = genletrec (fun g y ->
      let old = snd !memo in
      memo := (fst !memo, (n,g) :: old);
      in loop x
\end{verbatim}

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 §4).
Finally we are able to specialize the Ackermann function to a particular value of \( m \) (which is two, in the code below):

\[
\text{mrfix\ t\ 2} \\
\rightarrow \ : (\text{int} \to \text{int}) \ \text{code} = \\
\quad \lt \text{let}\ t_3 = \text{let}\ \text{rec}\ g_{11} x_{12} = x_{12} + 1 \ \text{in}\ g_{11} \ \text{in}\ \\
\quad \text{let}\ t_4 = \text{let}\ \text{rec}\ g_9 x_{10} = \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \qu
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

locaus's body (but within the scope denoted by locus).

The new genletrec let us write mrfx essentially just like the simpler mfix, without the splitting of the memo table into global and local parts: now, the definitions have the same scope.

let mrfx =
  (\(\alpha \rightarrow (\beta \rightarrow \gamma)\) code) \(\rightarrow (\alpha \rightarrow \beta\) code \(\gamma\) code)) \(\rightarrow (\alpha \rightarrow (\beta \rightarrow \gamma)\) code

\(\rightarrow (\alpha \rightarrow \beta\) code \(\gamma\) code)

fun f x ->
genletrec_locus @\@ fun locus ->
let memo = ref [] in
let rec loop n =
  try List.assoc n memo with Not_found ->
  genletrec_locus (fun g y -> memo := (n,g) :: !memo;
    f loop n y)
in loop x

With this new mrfx but the same sevelf from §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:

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

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

In particular, the automaton in §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 mrfx (makeau au1) S generates:

\[
\text{let rec } \text{x}_1 \ y = \text{match } y \text{ with}
  \text{A::r -> } \text{x}_1 \ r
  \text{B::r -> } \text{x}_5 \ r
  [] -> \text{true}
\]

and \(\text{x}_5 \ y = \text{match } y \text{ with}
  \text{A::r -> } \text{x}_1 \ r
  \text{B::r -> } \text{x}_9 \ r
  [] -> \text{false}
\]

in \(\text{x}_1\)

Status Currently the proof-of-concept of the described genletrec is prototyped\(^3\) using plain MetaOCaml as well as MetaOCaml with delimited control effects, such as those provided by Multicore OCaml (Dolan et al. 2015) or the delimcc library (Kiselyov 2012). We are working at supporting it above-the-board in the forthcoming release of MetaOCaml. The presentation will additionally describe extensions to nested mutually-recursive bindings (to show that generation is modular), heterogeneous and polymorphic recursion.

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.

References


Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. Effective concurrency through algebraic effects. OCaml Users and Developers Workshop 2015, September 2015.


\(^3\)https://github.com/yallop/metaocaml-letrec
A. Further extensions

We sketch some extensions to the \texttt{mrfix} combinator of Section 4.

\subsection*{A.1 Arbitrary bodies in let rec expressions}

The \texttt{mrfix} combinator has the following type:

\begin{verbatim}
val mrfix : ((α → (β→γ) code) → (α → β code → γ code)) →
            α → (β→γ) code
\end{verbatim}

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 \texttt{let rec} expression. For example, in the code generated for the Ackermann function by the call \texttt{mrfix \textit{tack} 2} in Section 3, the body of the generated expression is \texttt{l15}, the identifier associated with the definition generated by \texttt{tack 2}. And in the code generated for the finite state automaton in Section 4 the body of the generated expression is \texttt{x1}, the name of the function that corresponds to the start symbol.

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

\begin{verbatim}
val mrfixk : ((α → (β→γ) code) → (α → β code → γ code)) →
            ((α → (β→γ) code) → γ code) →
            γ code → γ code
\end{verbatim}

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:

\texttt{mrfixk (makeau au1) (fun \textit{f} → \texttt{<""}(\textit{f} \texttt{S}), \texttt{""}(\textit{f} \texttt{U}), \texttt{""}(\textit{f} \texttt{T}) \texttt{>)

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

\begin{verbatim}
let rec \texttt{x1} \texttt{y} = \texttt{match} \texttt{y} with
| \texttt{A}::\texttt{r} \to \texttt{x1} \texttt{r} | \texttt{B}::\texttt{r} \to \texttt{x5} \texttt{r} |
| \texttt{false} \to \texttt{true} and \texttt{x5} \texttt{y} = \texttt{match} \texttt{y} with
| \texttt{A}::\texttt{r} \to \texttt{x1} \texttt{r} | \texttt{B}::\texttt{r} \to \texttt{x9} \texttt{r} |
| \texttt{false} \to \texttt{true} and \texttt{x9} \texttt{y} = \texttt{match} \texttt{y} with
| \texttt{A}::\texttt{r} \to \texttt{x5} \texttt{r} | \texttt{B}::\texttt{r} \to \texttt{x9} \texttt{r} |
| \texttt{false} \to \texttt{false} in \texttt{x1}, \texttt{x9}, \texttt{x5})
\end{verbatim}

\subsection*{A.2 A syntax extension}

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

\begin{verbatim}
let%staged rec \texttt{f \textit{p} \textit{p}' = e in e'}
\to \texttt{mrfixk \texttt{(fun \textit{f \textit{p} \textit{p}' → e}) \texttt{(fun \textit{f → e')}}}
\end{verbatim}

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

Then \texttt{ack} can be written as follows:

\begin{verbatim}
let%staged rec \texttt{ack \textit{m} \textit{n} =
| \texttt{if \textit{m} = 0 then \textit{.<""n+1>.} else
| \texttt{.<""n}}
\texttt{if .\textit{n} = 0 then \texttt{("ack (m-1)) 1 else
| \texttt{<(ack (m-1)) ("ack \textit{m}) ("n-1)}>.
| \texttt{in \textit{ack} 2}
\end{verbatim}

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