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 [8], top-down and bottom-up parsers [2, 3], evaluators [1], GUI initialization graphs [10], &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 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 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 [6] or nested recursion [4], encoding recursion using higher-order state (“Landin’s knot”) [12] or hard-coding templates for a few fixed numbers of binding-group sizes [13]. These various workarounds suffer from various drawbacks: they are insufficient for 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.


1. Introduction

MetaOCaml (whose current implementation is known as BER MetaOCaml [7]) extends OCaml with support for typed program generation. There are three additions: α code is the type of unevaluated code fragments, brackets \(\langle e \rangle\). construct code fragments by delaying the evaluation of an expression, and splices \(\cdot e\) insert a code fragment into a larger fragment.

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

\[
\text{let } t1 \text{ x } = \langle \langle x \ast \text{ succ} \rangle \rangle \text{.}
\]

\[
\text{val } t1 : \text{ int } \rightarrow \text{ int } = \langle \text{\langle fun} \rangle \rangle
\]

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

\[
\text{let } c1 = t1 \langle 1 + 2 \rangle ;
\]

\[
\text{val } c1 : \text{ int } \text{ code } = \langle \langle 1 + 2 \ast \text{ succ} \rangle \rangle.
\]

Combined with the features of the host OCaml language, these constructs support safe and flexible program generation, with typed manipulation of code with free variables, use of arbitrary OCaml features (effects, modules, etc.) in the generating program, and a guarantee that generated code is well-scoped and well-typed.

However, MetaOCaml’s 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 [11]. For example, the following state machine:

![State machine diagram]

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

\[
\text{let rec } c3 = \text{ function A : } r \rightarrow s \mid B : r \rightarrow t \mid \text{ true}
\]

\[
\text{and } t = \text{ function A : } r \rightarrow s \mid B : r \rightarrow u \mid \text{ false}
\]

\[
\text{and } u = \text{ function A : } r \rightarrow t \mid B : r \rightarrow u \mid \text{ false}
\]

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 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 [6] or nested recursion [4], encoding recursion using higher-order state (“Landin’s knot”) [12] or hard-coding templates for a few fixed numbers of binding-group sizes [13]. These various workarounds suffer from various drawbacks: they are insufficient for 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:

\[
\text{let } t2 x = .<\langle x \ast \text{ succ}\rangle >.
\]

\[
\text{val } t2 : \text{ t p12} ;
\]

\[
\text{val } c2 : \text{ int code } = .<\langle \text{ let } y_1 \rightarrow 1 \ast \text{ succ} \rangle >.
\]

However, in general let expressions cannot be inserted locally. For example, in the following program, the function \(ft1\) generates a function expression, using the argument \(t\) to build subexpressions:

\[
\text{let } ft1 x = .\langle \text{ fun } u \rightarrow \langle \langle t \rangle \rangle >\rangle.
\]

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

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

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

\[
\text{val } c3 : \text{ (int } \rightarrow \text{ int code } = .<\langle \text{ fun } u_2 \rightarrow (\langle \text{ let } y_4 \rightarrow 1 \ast \text{ succ} \rangle + (\langle \text{ let } y_3 \rightarrow (1 \ast \text{ succ} y_3) \rangle)) >.
\]

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 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, the let is inserted immediately:

```ml
let p12l = genlet ..< 1 + 2 >.
```

Note that the code generation is exponential, as the genlet .. `< ` expression is exponentially slow.

We see the duplicated expressions in the generated code reveal why fibnr becomes a code generator, here building the code value with just that variable. (If e is already an atomic expression, genlet 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, the let is inserted immediately:

```ml
let p12l = genlet ..< 1 + 2 >.
```

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

```ml
let c1l = t1 p12l
```

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

```ml
let ft1 x = let rec
```

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) 5.

```ml
let rec f x y = let rec genletrec : ((alpha -> beta) code) code)
```

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:

```ml
let genletrec = let rec f x y = let rec genletrec : (alpha -> beta) code)
```

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 maintains a mapping from previous arguments to the let-bound variables [9]:

```ml
let mfix f = let rec loop n = try List.assoc n !memo with Not_found ->
```

Recurrent 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):

```ml
let mfix f = let rec loop n = try List.assoc n !memo with Not_found ->
```

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

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

```ml
let rec ack m n = if m = 0 then n + 1 else
```

to the given value of m. Turning ack into the generator of the specialized code is easy in the open-recursion style, by merely annotating the code keeping in mind that n is future-stage:

```ml
let recack m n = if m = 0 then n + 1 else
```

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

```ml
let rec g = in body may be re-written as
```

which immediately gives us genletrec:

```ml
let genletrec = (alpha -> beta) code) code)
```

The new memoizing fixpoint combinator becomes

```ml
let mfix f x y = let rec loop n = try List.assoc n !memo with Not_found ->
```

Let-insertion is often used with memoization, as we illustrate with a simplified dynamic-programming algorithm [5]. The fibnr function computes the nth element of the Fibonacci sequence whose first two elements are given as arguments x and y:

```ml
let fibnr x y = if n <= 0 then x else if n = 1 then y else
```

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) 5.

```ml
let rec f x y = let rec genletrec : ((alpha -> beta) code) code)
```

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:

```ml
let genletrec = let rec f x y = let rec genletrec : (alpha -> beta) code)
```

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 maintains a mapping from previous arguments to the let-bound variables [9]:

```ml
let mfix f = let rec loop n = try List.assoc n !memo with Not_found ->
```

Recurrent 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):

```ml
let mfix f x y = let rec loop n = try List.assoc n !memo with Not_found ->
```

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 §1. We take a simpler running example: generating the odd-even function. At first, the method of the previous section applies: 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 the particular recursive clause in the group:

\[
\text{let rec odd evod self idx n = match idx with}
\]

```ocaml
| Even -> if n = 0 then true else self Odd (n-1)
| Odd -> if n = 0 then false else self Even (n-1)
```

To find the parity of 42, one writes `evod Even 42`. The straightforward staging of `evod` and applying `mrfix` and the index `Even` gives the following code for even parity:

```ocaml
let rec even = let rec odd =
                let l12 = genletrec locus (l12 -> if x17 = 0 then true else l12 (x17 - 1)) in
                fun x17 -> if x17 = 0 then true else l12 (x17 - 1) in
                fun x17 -> if x17 = 0 then true else l12 (x17 - 1) in
                let l20 = let rec g18 x19 =
                          if x19 = 0 then false else g16 (x19 - 1) in
                          g18 in
```

The odd function is nested inside of even rather than being ‘parallel’ with it. It means odd is not accessible from the outside; if we also want to compute odd parity, we have to duplicate the code.

We would like to generate the mutually recursive definition `let rec even = ... and odd = ...` that defines both even and odd in the same scope. Alas, this is impossible using only brackets and escapes: code values represent OCaml expressions, but the set of bindings is not an expression. There is also a bigger, semantic challenge. While generating the code for the i-th recursive clause in a group we may refer to clauses with both smaller and larger indices. It seems we have to resort to Lisp-like gensym, explicitly creating a name and only later binding it. However, what static assurances to we have that all generated names will be bound, and to their intended clauses. How do we maintain the MetOCamel guarantee that the fully generated code is always well-typed?

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:

```ocaml
type locus_t
val genletrec_locus : (locus_t -> α code) -> α code
val genletrec : locus_t ->
    ((α->β) code -> α code -> β code) -> (α->β) code
```

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). In the new `genletrec`, the insertion point is explicitly marked by `genletrec_locus`, which hence sets the scope for all identifiers in the recursive group. Correspondingly, in the new `genletrec` `(fun g x -> ... )`, 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 `mrfix` 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.

```ocaml
let mrfix :
  ((α -> (β->γ) code) -> (α -> β code -> γ code)) ->
  (α -> (β->γ) 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 `mrfix` but the same old `evodf` we obtain the familiar mutually recursive even-odd code: `mrfix evodf Even`. Recognizers of finite state automata are produced by the following generic, textbook generator:

```ocaml
type token = A | B

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
  .<match .stream with
  | A :: r -> ."(self (next A))" r
  | B :: r -> ."(self (next B))" r
  | [] -> accept>
```

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

```ocaml
let au1 =
  {finals = [S];
   trans =
     [[(S, [(A, S); (B, T)]); (T, [(A, S); (B, U)]);
      (U, [(A, T); (B, U)])];
     .<match .stream with
     | A :: r -> ."x1" r
     | B :: r -> ."x2" r
     | [] -> false
     and x2 y = .match y with
     | A :: r -> ."x1" r
     | B :: r -> ."x2" r
     | [] -> false
     in x1}
```

**Status** Currently the proof-of-concept of the described `genletrec` is prototyped using plain MetaOCaml as well as MetaOCaml with delimited control effects (such as those provided by Multicore OCaml or the `delimcc` library): `https://github.com/yall0p/metaocaml-genletrec`. 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.

**References**


