Generating Mutually Recursive Definitions
(Short Paper)

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

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 $\langle\rangle$. construct a code fragment by quoting an expression, and splices $\langle\rangle$ 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:

```ocaml
let t1 x = \langle x \rangle \times \langle x \rangle.
val t1 : int code -> int code = <fun>
```

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

```ocaml
let p12 = \langle 1 + 2 \rangle, 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.

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 [16]. For example, the following state machine:

```
\[
\begin{array}{cccc}
A & B & t & B \\
\downarrow & \downarrow & \uparrow & \downarrow \\
\select{n} & s & B & u \\
\end{array}
\]
```

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

```ocaml
let rec s = function
\mathrm{A} \to \mathrm{r} \mid \mathrm{B} \to \mathrm{t} \mid \text{[]} \to true
and t = function
\mathrm{A} \to \mathrm{r} \mid \mathrm{B} \to \mathrm{u} \mid \text{[]} \to false
and u = function
\mathrm{A} \to \mathrm{r} \mid \mathrm{B} \to \mathrm{u} \mid \text{[]} \to 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 few fixed 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:

```ocaml
let t2 x = .<let y = -.x in y * succ y>.
let c2 = t2 p12
```

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:

```ocaml
let ft1 x = .<fun u -> -.t (x) + -.t (.-.x + u>)>.
```

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

```ocaml
let c3 = ft1 t2 p12;
```

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 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:

```ocaml
let p12l = genlet p12
```

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

```ocaml
let ft1 x = .<fun u -> -.t (x) + -.t (genlet (.-.x + u>)>)>.
```

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).

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:

```ocaml
let fibnr plus x y self n =
  if n = 0 then x else
  if n = 1 then y else
    plus (self (n-1)) (self (n-2))
```

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 [14]

```ocaml
let rec loop n = try List.assoc n !memo
  with Not_found ->
    let v = genlet (f loop n) in
    memo := (n,v):: !memo; v
```

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

```ocaml
let c1l = t1 p12l
```

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:

```ocaml
let fibnr plus x y self n =
  if n = 0 then x else
  if n = 1 then y else
    plus (self (n-1)) (self (n-2))
```

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).

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:

```ocaml
let fibnr plus x y self n =
  if n = 0 then x else
  if n = 1 then y else
    plus (self (n-1)) (self (n-2))
```

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 [14]

```ocaml
let rec loop n = try List.assoc n !memo
  with Not_found ->
    let v = genlet (f loop n) in
    memo := (n,v):: !memo; v
```

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

```ocaml
let c1l = t1 p12l
```

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

```ocaml
let ft1 x = .<fun u -> -.t (x) + -.t (genlet (.-.x + u>)>)>.
```
Generates Mutually Recursive Definitions

\[
\text{let } l_0 = l_6 + l_7 \text{ in let } l_{10} = l_9 + l_8 \text{ in } l_{10} >.
\]

Without genlet however, we get the same poor code as with
the ordinary fix: memoization alone speeds up the code gen-
eration 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 sim-
plest example is specializing the Ackermann function

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

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:

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

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

\[
\text{mfix (fun} m \rightarrow .<\text{fun} n \rightarrow .(\text{tack} m .<n>))(2)
\]

Looking at the original ack shows the reason: \(m\) depends
not only on \(a\) (also \(m\) but also on \(a\) 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 \(\text{let rec } g = e\text{ in}\)
body may be re-written as

\[
\text{let } g = \text{let rec } g = e\text{ in } g\text{ in } body
\]

which immediately gives us genletrec:

\[
\text{let genletrec : } ((\alpha \rightarrow \beta) \text{ code} \rightarrow \alpha \text{ code} \rightarrow \beta \text{ code}) \rightarrow (\alpha \rightarrow \beta) \text{ code} =
\text{fun} f \rightarrow \text{genlet. } \text{let rec g x = } .(\text{.f}.<\text{g}>.<\text{x}>) \text{ in } g >.
\]

The new memoizing fixpoint combinator becomes

\[
\text{let mfix : } ((\alpha \rightarrow (\beta \rightarrow \gamma) \text{ code}) \rightarrow (\alpha \rightarrow \beta \text{ code} \rightarrow \gamma \text{ code}) \rightarrow
(\alpha \rightarrow (\beta \rightarrow \gamma) \text{ code}) =
\text{fun} f \rightarrow \text{let memo = ref } ([],[]) \text{ in }
\text{let rec loop } n =
\text{try} \text{ List.assoc } n \text{ (fst !memo) snd !memo) }
\text{with} \text{ Not_found} \rightarrow
\text{let } v = \text{genletrec (fun } g y \rightarrow
\text{let old } = \text{snd !memo in }
\text{memo } := (\text{fst !memo, (n,g) :: old});
\text{let } v = (\text{f loop } n y) \text{ in }
\]

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:

\[
\text{mfix tack 2}
\]

\[
\n
\text{let rec } l_{13} = \text{let rec } g_{11} x_{12} = x_{12} + 1 \text{ in } g_{11} \text{ in }
\text{let l_{14} = let rec } g_{9} x_{10} =
\text{if } x_{10} = 0 \text{ then } l_{13} 1 \text{ else } l_{13} (g_{9} (x_{10} - 1))
\text{in } g_{9} \text{ in }
\text{let l_{15} = let rec } g_{7} x_{8} =
\text{if } x_{8} = 0 \text{ then } l_{14} 1 \text{ else } l_{14} (g_{7} (x_{8} - 1))
\text{in } g_{7} \text{ in } l_{15} >.
\]

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.

\[
\text{let rec } \text{even } m n = \text{if } n\geq 0 \text{ then } \text{odd } m (n - 1) \text{ else}
\text{if } m\geq 0 \text{ then } \text{odd } (m - 1) n \text{ else}
\text{true}
\text{and odd } m n = \text{if } n\geq 0 \text{ then } \text{even } m (n - 1) \text{ else}
\text{if } m\geq 0 \text{ then } \text{even } (m - 1) n \text{ else}
\text{false}
\]

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\(^1\):

\[
\text{type evod } = \text{Even | Odd}
\]

\[
\text{let rec } evodsf self idx m n =
\text{match idx with}
\]

\(^1\)Since the functions even and odd have the same types, the index here is
the ordinary data type evod. The general case calls for generalized algebraic
data types (GADTs), as Section 5.3 shows.
which looks very much like tack from Section 3. We could thus apply mfix from that section with suitable 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):

\[
\text{mfix (fun self (idx,m) x \mapsto \sevod (fun idx m \mapsto self (idx,m)) idx m x)}
\]

\[
\text{Even,0)
\]

\[
\text{let \_ : (int \mapsto bool) code = .<}
\]

\[
\text{let lv0 =}
\]

\[
\text{let rec g1 =}
\]

\[
\text{let lv5 =}
\]

\[
\text{let rec g3 x4 = if x4>0 then g1 (x4-1) else false in g3 in}
\]

\[
\text{fun x2 \mapsto if x2>0 then lv5 (x2-1) else true in}
\]

\[
g1 \text{ in}
\]

\[
lv5>.
\]

The odd function (appearing under the generated name g3) is nested inside even (or, g1) 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. There is a deeper problem than mere code duplication: specializing even m n to m=1 (that is, applying the tied-knot sevod to (Even,1)) generates no code. An exception is raised instead, telling us that MetaOcaml detected scope extrusion: an attempt to use a variable outside the scope of its binding. Indeed, we have attempted to produce something like the following (identifiers are renamed for clarity):

\[
\text{let lod0 = (+ odd 0 n)}
\]

\[
\text{let rec od0 n} =
\]

\[
\text{let levo = (+ even 0 n)}
\]

\[
\text{let rec ev0 n = if n>0 then od0 (n-1) else true in ev0 in}
\]

\[
\text{let lv1 = (+ even 1 n)}
\]

\[
\text{let rec ev1 n} =
\]

Here, the function ev1, the specialization of even m n to m=1 calls od0 and od1. The latter calls ev1 and fun n \mapsto even 0 n, 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.

We would like to generate the mutually recursive definition \text{let rec even = ... and odd = ...} that defines both even and odd \textit{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 do we have that all generated names will be bound, and to their intended clauses. How do we maintain the MetaOcaml 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. 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:

\[
\text{type locus_t}
\]

\[
\text{val genletrec_locus: (locus_t \mapsto \alpha code) \mapsto \alpha code}
\]

\[
\text{val genletrec : locus_t \mapsto}
\]

\[
\text{\((\alpha\mapsto\beta) code \mapsto \alpha code \mapsto \beta code) \mapsto (\alpha\mapsto\beta) 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). 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\(^2\). Correspondingly, in a call genletrec_locus (fun g x \mapsto \ldots)\(^3\) 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 mfix essentially just like the simpler mfix, without the splitting of the memo table

\(^2\)It hence becomes the programmer’s responsibility to place genletrec_locus correctly. We are yet to explore and resolve the trade-off between automatically floating genlet and genletrec whose scope is to be set manually.
Generating Mutually Recursive Definitions

into global and local parts\(^3\): now, the definitions have the same scope.

```ocaml
let mrfix : 
  \((\alpha \to (\beta \to \gamma) \text{ code}) \to (\alpha \to \beta \text{ code} \to \gamma \text{ code}) \to 
  (\alpha \to (\beta \to \gamma) \text{ code}) = 

\text{fun } f \text{ x }\to 
\text{genletrec \_locus @@ fun \_locus }\to 
\text{let memo }= \text{ ref }[\text{ i }\text{ in }]
\text{let rec }\text{ loop }n = 
  \text{try }\text{List.assoc }n\text{ \_memo }\text{ with }\text{ Not\_found }\to 
  \text{genletrec \_locus \_fun }g\text{ y }\to 
  \text{memo }= (n,g) :: !\text{memo}; 
  f \text{ loop }n y 
\text{in }\text{ loop }x 
```

With this new mrfix but the same seovdf 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\(^1\):

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

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

```ocaml
let au1 = 
  \[(\text{finals }= [\text{S}]; 
  \text{trans }= \{(\text{S}, [(\text{A, S); (B, T))]; 
  \text{(T, [(A, S); (B, U))]; 
  \text{(U, [(A, T); (B, U))]); 
  \}
```

Then mrfix (makeau au1) S generates:

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

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

\[\text{in }x_1\]

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:

\[\text{val mrfix }:\((\alpha \to (\beta \to \gamma) \text{ code}) \to (\alpha \to \beta \text{ code} \to \gamma \text{ code}) \to 
  (\alpha \to (\beta \to \gamma) \text{ 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 (tack 2) in Section 3, the body of the generated expression is \(I\_S\), the identifier associated with the definition generated by tack 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:

\[\text{val mrfixk }:\((\alpha \to (\beta \to \gamma) \text{ code}) \to (\alpha \to \beta \text{ code} \to \gamma \text{ code}) \to 
  ((\alpha \to (\beta \to \gamma) \text{ code}) \to \gamma \text{ code}) \to \gamma \text{ 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:

\[\text{mrfixk (makeau au1) (fun }f \to \ldots \text{-(f S), -(f U), -(f T)) }\ldots )\]

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

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

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

In x_1
```
and \( x_0 y = \text{match } y \text{ with } A : \rightarrow x_5 r \)
\[
\mid B : r \rightarrow x_0 r \\
\mid [] \rightarrow \text{false}
\]
\[\text{in } (x_1, x_9, x_8)\]

### 5.2 A syntax extension

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

\[\text{let staged rec } f p p' = e \text{ in } e'\]
\[\sim mrfixk (\text{fun } f p p' \rightarrow e) (\text{fun } f \rightarrow e')\]

Here \%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

\[\text{let staged rec } ack m n =\]
\[\text{if } m = 0 \text{ then } \langle \cdot \rangle, \text{ else }\]
\[\langle \cdot \rangle. \text{ match } t \text{ with }\]
\[| \text{EmptyN } \rightarrow \text{EmptyN} \]
\[| \text{TreeN } v, t \rightarrow \text{TreeN } (f v, \text{swivel } \text{fun } (x, y) \rightarrow (f y, f x)) t\]

This is polymorphic-recursion because the recursive call uses 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-recursion definitions like swivel involves indexing by a polymorphic type. Here is a suitable index for generating swivel:

\[\text{type } \text{swivel } = \{ \text{swivel } : (\alpha \rightarrow \alpha) \rightarrow \alpha \text{ ntree } \rightarrow \alpha \text{ ntree }\}\]

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 \((\alpha \rightarrow \alpha) \rightarrow \alpha \text{ ntree } \rightarrow \alpha \text{ ntree}\).

### Acknowledgments

We thank Nada Amin and Jun Inoue for helpful 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 prototyped\(^6\) 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


\(^6\)URL elided for anonymous review


