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 [Hinze and Paterson 2003; Hutton and Meijer 1996], evaluators [Abelson et al. 1984], GUI initialization graphs [Syme 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.

CCS Concepts  • Software and its engineering → Recursion; Functional languages; Source code generation;

Keywords  Recursion, fixed points, multi-stage programming, metaprogramming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

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: a code is the type of unevaluated code fragments, brackets,.<e>, 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:

\[
\text{let } t1 \text{ x } = < \text{.} - \text{x} \text{.} \text{succ} < - \text{x} > > .
\]

\[
\text{let } t1 : \text{int code } \rightarrow \text{int code } = < \text{fun} >
\]

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

\[
\text{let } p12 = < 1 + 2 > .
\]

\[
\text{let } c1 = t1 \text{ p12}
\]

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 [Taha 1999]. For example, the following state machine:

\[
\begin{array}{c}
\text{A} \\
<e>
\end{array}
\]

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

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

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

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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:
are eventually bound to the intended expressions, and how to ensure that generated code is well-typed.

Related metaprogramming systems such as LMS [Rompf 2016] and Template Haskell [Sheard and Jones 2002] 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). None of them therefore catch the left-unbound variables until the code is fully generated and compiled – at which point the error in the generator becomes very difficult to find (as illustrated in [Ofenbeck et al. 2016]).

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 (Section 3), building on earlier designs for insertion of heterogeneous, value and polymorphic recursion.

• 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, 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:

```
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, t1 takes a code template t as argument, using it when building the body of the generated function:

```
let ft1 t x = <fun u -> -(t x) + -(t (.<-x + u)).>. let c3l = ft1 p12;
```

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

```
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:

```
let p12l = genlet p12
```

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

```
let ft1 x = <fun u -> -(t1 u) + -(t1 (genlet (.<-x + u))).>. let c3l = ft1 p12l;
```

```
fun u10 -> let l11 = l5 + u10 in l5>succ l5 + l11>succ l11>
```

2.1 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:

```
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:

```
let plus x y = <.-x + -.y>. in
  <fun x y -> -(fix (fibnr plus.<x>.<y>)) 5>.
```

```
Generating Mutually Recursive Definitions

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 [Swadi et al. 2006]

```
let mfix (f : (α → β code) → (α → β code)) (x : α) : β code =
  let memo = ref [] in
  let rec loop n = try List.assoc n !memo
                    with Not_found →
                     let v = genlet (f loop n) in
                     memo := (n,v) :: !memo; v
  in loop x
```

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

```
< fun x y -> -(mfix (fibnr splus .<x>. .<x>. .<x>)) 5 > .
~< = : (int → int → int) code =
  .< fun x y ->
  let l7 = y6 + x5 in let l8 = l7 + y6 in
  let l9 = l8 + l7 in let l10 = l9 + l8 in l10 >.
```

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. [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 specializing the Ackermann function

```
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:

```
let tac k 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 tac : (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:

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

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

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

which immediately gives us genletrec:

```
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

```
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 (fst !memo @ snd !memo)
  with Not_found →
  let v = genletrec (fun g y →
    let old = snd !memo in
    memo := (fst !memo, (n,g) :: old);
    let v = (f loop n y) in
    memo := (fst !memo, old);
    v) in
  memo := ((n,v) :: fst !memo, snd !memo); v
  in loop x
```

Recursive definitions have to be the definitions of functions: this fact is 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
```

```
< fun x y -> -(mfix (fibnr splus .<x>. .<x>. .<x>)) 5 >
```

```
~< = : (int → 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 non-negative integers \( m \) and \( n \) and returning a boolean, telling if the sum \( m+n \) has even or odd parity, resp.

\[
\begin{align*}
\text{let rec } & \text{even } m n = \text{if } n > 0 \text{ then odd } (m-1) \text{ else } \\
& \text{if } m > 0 \text{ then odd } (m-1) n \text{ else } \text{true} \\
\text{and } & \text{odd } m n = \text{if } n > 0 \text{ then even } (m-1) \text{ else } \\
& \text{if } m > 0 \text{ then even } (m-1) n \text{ else } \text{false}
\end{align*}
\]

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

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

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

To find out if the sum of 10 and 42 has even parity one writes \( \text{evodf Even 10+42} \). The straightforward staging gives

\[
\begin{align*}
\text{let rec } & \text{sevodf self idx m n = } \\
& \text{match idx with } \\
& \text{| Even } \rightarrow . \text{if } n > 0 \text{ then . . . else . . .} \\
& \text{| Odd } \rightarrow . \text{if } m > 0 \text{ then . . . else . . .}
\end{align*}
\]

which looks very much like \text{tack} from Section 3. We could thus apply \text{mrfix} from that section with trivial adaptations

\[
\begin{align*}
\text{and obtain the code for even } m n \text{ specialized to a particular value of } m, \text{ say, 0 (which is just the ordinary even function):}
\end{align*}
\]

\[
\begin{align*}
\text{mrfix (fun self (idx,m) x } \rightarrow \\
& \text{sevodf (fun idx m } \rightarrow \text{self (idx,m)) idx m x) \\
& \text{(Even,0)} \\
& \text{(Even,0)} \\
& \text{~(self Odd (m-1)) n ~(self Even (m-1)) n ~else true} \\
& \text{~(self Even (m-1)) n ~else true} \\
& \text{~(self Even (m-1)) n ~else true} \\
& \text{~(self Even (m-1)) n ~else true} \\
& \text{~(self Even (m-1)) n ~else true}
\end{align*}
\]

\[
\begin{align*}
\text{let lv0 = } \\
& \text{let rec g1 = } \\
& \text{let lv5 = } \\
& \text{let rec g3 x4 = if x4 > 0 then g1 (x4-1) else false in } \\
& \text{g3 in } \\
& \text{fun lv2 } \rightarrow \text{if lv2 > 0 then lv5 (lv2-1) else true in } \\
& \text{g1 in } \\
& \text{lv5.}
\end{align*}
\]

The odd function (appearing under the generated name \( g_3 \)) is nested inside even \( g_1 \) 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 \text{sevodf} to \( (\text{Even,1}) \)) 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):

\[
\begin{align*}
\text{let lod0 = } (\ast \text{ odd } 0 n \ast) \\
& \text{let rec od0 n = } \\
& \text{let lev0 = } (\ast \text{ even } 0 n \ast) \\
& \text{let rec ev0 n = if n > 0 then od0 (n-1) else true in ev0 in } \\
& \text{if n > 0 then lev0 (n-1) else false in od0 in } \\
& \text{let lev1 = } (\ast \text{ even } 1 n \ast) \\
& \text{let rec ev1 n = } \\
& \text{let lod1 = } (\ast \text{ odd } 1 n \ast) \\
& \text{let rec od1 n = if n > 0 then ev1 (n-1) else lev0 n in od1 in } \\
& \text{if n > 0 then lod1 (n-1) else lod0 n in ev1 in } \\
& \text{in lev1}
\end{align*}
\]

Here, the function \( ev1 \), the specialization of even \( m n \) to \( m=1 \) calls \( od0 \) and \( od1 \). The latter calls \( ev1 \) and \( fun n \rightarrow \text{even } 0 n \), whose code was already generated and memoized, under the name \( lev0 \). Unfortunately, the scope of \( lev0 \) does not extend beyond the scope of \( od0 \)’s definition, and hence mentioning \( lev0 \) within \( od1 \) is a scope extrusion.

We would like to generate the mutually recursive definition \text{let rec even = … and odd = …} that defines both even and odd \text{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

---

\( ^{\text{1}}\)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.
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?

Finally, what should the interface for the generator of mu-
tually recursive bindings be in the first place? After quite a
bit of thought, it turns out that genletrec’s interface would
suffice. For the sake of better error detection, one would gen-
eralize it slightly. We add a second function, genletrec_locus,
which marks the location where a group of recursive defi-
nitions should be inserted; the generated locus_t value rep-
resenting the location can be passed as first argument of
genletrec:

```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 re-
quested definition in the widest possible context (while en-
suring 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

```
let rec f x
  genletrec locus (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 mfix essentially just like the
simpler mfix, without the splitting of the memo table
to global and local parts:\(^3\): now, the definitions have the
same scope.

```
let mfix :
  ((α → (β→γ) code) → (α → β code → 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
```

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

\(^3\)Previously, genletrec relied on the trick let g = let rec g = e in g in body,
which binds two different gs, one within and one outside the scope of
the local let rec. Therefore, the memo table had two parts. The local part tracks
the identifiers that are valid only while we are generating the let rec body;
the global part, to which we only add, collects the externally visible gs.

With this new mfix but the same sevdof 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\(^4\):

```
type token = A | B
val state = S | T | U
val (α,σ) automaton =
  { finals: σ list;
    trans: (σ × (α × σ) list) list}
```

```
let makeau :
  (token, α) automaton →
  (α → (token list → bool) code) →
  α → token list code → bool code =
  fun {finals; trans} self state stream →
  let accept = List.mem state finals in
  let next token = List.assoc token (List.assoc state trans) in
  < match state ever with
    | A : r → -(self (next A)) r
    | B : r → -(self (next B)) r
    | [] → 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 mfix (makeau au1) S generates:

```
let rec x1 y = match y with
  | A:r → x1 r
  | B:r → x5 r
  | [] → true
and x5 y = match y with
  | A:r → x1 r
  | B:r → x9 r
  | [] → false
and x9 y = match y with
  | A:r → x5 r
  | B:r → x9 r
  | [] → false
in x1
```

of the type token list → bool.

\(^4\)The generator makeau is indeed polymorphic over the type of the state;
the dependence on the alphabet shows in the match statement. Incidentally,
MetaOCaml also has a facility to generate pattern-match clauses of statically
unknown length and content. With its help, we can make makeau fully
general.
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:

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

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 \( l_{15} \), 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:

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

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:

\[ \text{mrfixk} \text{ (makeau au1)} (\text{fun} f \to \langle (\sim(f S)), (\sim(f U)), (\sim(f T)) \rangle) \]

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

\[ \text{let rec} \ x_1 \ y = \text{match} \ y \text{ with} \\
\begin{array}{l}
A: \tau \to x_1 \ r \\
\mid B: \tau \to x_2 \ r \\
\mid [ \to \text{true} \\
\end{array} \]

\[ \text{and} \ x_5 \ y = \text{match} \ y \text{ with} \\
\begin{array}{l}
A: \tau \to x_1 \ r \\
\mid B: \tau \to x_9 \ r \\
\mid [ \to \text{false} \\
\end{array} \]

\[ \text{and} \ x_9 \ y = \text{match} \ y \text{ with} \\
\begin{array}{l}
A: \tau \to x_5 \ r \\
\mid B: \tau \to x_9 \ r \\
\mid [ \to \text{false} \\
\end{array} \]

\[ \text{in} \ (x_1, x_9, x_5) \]

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:

\[ \text{let%staged rec} \ f \ p \ p' = e \in e' \]

\[ \leadsto \text{mrfixk} \ (\text{fun} \ f \ p \ p' \to e) \ (\text{fun} \ f \to e') \]

Here %staged is an attribute that indicates the need for a rewrite by a plug-in program that expands the syntax as shown above. Then ack can be written as follows

\[ \text{let%staged rec} \ \text{ack} \ m \ n = \\
\begin{array}{l}
\text{if} \ m = 0 \ \text{then} \ .<,-n+1>. \ \text{else} \\
.\text{if} \ .-n = 0 \ \text{then} \ .<(\text{ack} \ (m-1)). \ \text{else} \\
.\text{ack} \ (m-1)). \ (\sim(\text{ack} \ m)) (-n-1)>), \text{in ack 2} \\
\end{array} \]

As this example shows, the syntax extension avoids the need for explicitly higher-order code and for open recursion; the identifier ack serves as the self argument in the expanded syntax, and so the calls to 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 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 GADTs.\(^5\) For example, the following GADT supports generating mutually-recursive bindings for functions of types \( \text{int} \to \text{bool} \) and \( \text{float} \to \text{bool} \)

\[ \text{type} \ \alpha \text{ eo} = \text{Even} : (\text{int} \to \text{bool}) \text{ eo} \]

\[ \mid \text{Odd} : (\text{float} \to \text{bool}) \text{ eo} \]

The type of the mrfixk function is generalized accordingly: the type \( \alpha \to (\beta \to \gamma) \) code of functions that map indexes to variables becomes \( \forall \alpha. \tau \to \alpha \text{ code} \), where the higher-kinded type variable \( \tau \) stands for an arbitrary parameterized index type, such as eo. Here is the fully generalized type:

\[ \text{val} \ \text{mrfixk} : \forall \gamma \forall \tau. (\forall \beta. (\forall \alpha. \tau \to \alpha \text{ code}) \to (\beta \to \gamma) \text{ code}) \to \\
((\forall \alpha. \tau \to \alpha \text{ code}) \to \gamma \text{ code}) \to \gamma \text{ code} \]

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 ntree of perfectly balanced trees and a polymorphic function swivel that interchanges left and right elements of ntree values:

\(^5\text{This (progressively fancier and fancier) indexing is closely related to the generalized arity in Plotkin and Power’s formulation of algebraic effects [Plotkin and Power 2003]. Like them, we represent a tuple as a set of indices plus the function that maps each index to a value. The index set does not have to have a fixed finite cardinality. Adding more structure to the set of indices lets us, like it let Plotkin and Power, represent more interesting collections.}
Generating Mutually Recursive Definitions

```haskell
type α ntree = EmptyN
  | TreeN of α * (α * α) ntree

let rec swivel : α. (α → α) → α ntree → α ntree =
  fun f t → match t with
  | EmptyN → EmptyN
  | TreeN (v,t) → TreeN (f v, swivel (fun (x, y) → (f y, f x)) t)
```

This is polymorphic-recursive because the recursive call uses swivel at a different type than the type of the definition: the passed function f acts on pairs α * α, not values of type α.

Generating polymorphic-recursive definitions like swivel involves indexing by a polymorphic type. Here is a suitable index for generating swivel:

```haskell
type swivel = { swivel: α. (α → α) → α ntree → α ntree }
type _ index = Swivel : swivel index
```

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 (α → α) → α ntree → α 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. This work was partially supported by JSPS KAKENHI Grant Number 18H03218.

Status

The described genletrec is prototyped in full 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 a forthcoming release of MetaOCaml.

References


https://github.com/yallop/metaocaml-genletrec