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 (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.

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: α `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:

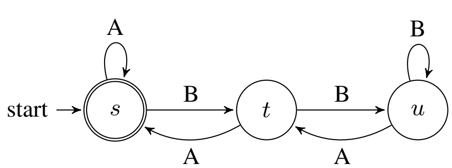
```
let t1 x = .<.~x * succ .~x>.
⇝ val t1 : int code → int code = <fun>
```

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

```
let p12 = .< 1 + 2 >.
let c1 = t1 p12
⇝ val c1 : int code = .<(1 + 2) * succ (1 + 2)>.
```

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:



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

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

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

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

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:

```
let t2 x = .<let y = .~x in y * succ y>.
let c2 = t2 p12
⇝ val c2 : int code = .<let y1 = 1 + 2 in y1 * (succ y1)>.
```

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:

```
let ft1 t x = .<fun u → .~(t x) + .~(t .<.~x + u>.)>.
⇝ val ft1 : (int code → int code) → int code → (int → int) code
```

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

```
let c3 = ft1 t2 p12;;
⇝ val c3 : (int → int) code = .<fun u2 →
    (let y4 = 1 + 2 in y4 * (succ y4)) +
      (let y3 = (1 + 2) + u2 in y3 * (succ y3))>.
```

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
⇝ val p12l : int code = .<let l5 = 1 + 2 in l5>.
```

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

```
let c1l = t1 p12l
⇝ val c1l : int code = .<let l5 = 1 + 2 in l5 * succ l5>.
```

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

```
let ft1 x = .<fun u → .~(t1 x) + .~(t1 (genlet .<.~x + u>.))>.
let c3l = ft1 p12l;;
⇝ val c3l : (int → int) code =
.<let l5 = 1 + 2 in
   fun u10 → let l11 = l5 + u10 in l5*succ l5 + l11*succ l11>.
```

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 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 splus x y = .<.~x + .~y>. in
.<fun x y → .~(fix (fibnr splus .<x>. .<y>.) 5)>.
⇝ − : (int → int → int) code =
.<fun x1 y2 → (((y2+ x1)+ y2)+ (y2+ x1))+ ((y2+ x1)+ y2)>.
```

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
```

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

```
.<fun x y → .~(mfix (fibnr splus .<x>. .<y>.) 5)>.
⇝ − : (int → int → int) code =
.< fun x5 y6 →
    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. 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 tack self m n =
    if m = 0 then .<.~n+ 1>. else
  .<if .~n = 0 then .~(self (m−1)) 1 else
  .~(self (m−1)) (.~(self m) (.~n−1))>.
⇝ val tack : (int→(int→int) code) → int→int code→int code
```

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

```
mfix (fun self m → .<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 mrfix : ((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: the fact reflected in mrfix's (and genletrec's) code and type. The mrfix 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):

```
mrfix tack 2
⇝ − : (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 §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.

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

At first, mutual recursion seems to pose no problem: after all, a group of mutually recursive functions may always be converted to the ordinary recursive function by adding an extra argument: the index of a particular recursive clause in the group[1]:

```
type evod = Even | Odd

let rec evodf self idx m n = match idx with
  | Even →
  if n>0 then self Odd m (n−1) else
  if m>0 then self Odd (m−1) n else
  true
  | Odd →
  if n>0 then self Even m (n−1) else
  if m>0 then self Even (m−1) n else
  false
⇝ val evodf : (evod → int → int → bool) →
                evod → int → int → bool
```

To find out if the sum of 10 and 42 has even parity one writes fix evodf Even 10+ 42. The straightforward staging gives

```
let rec sevodf self idx m n = match idx with
  | Even →
  .<if .~n>0 then .~(self Odd m) (.~n−1) else
  .~(if m>0 then .<.~(self Odd (m−1)) .~n>. else
    .<true>.)>.
  | Odd → ...
⇝ val sevodf : (evod → int → (int → bool) code) →
                 evod → int → int code → bool code
```

[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).

which looks very much like tack from §3. We could thus apply mrfix from that section with trivial 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):

```
mrfix (fun self (idx,m) x → sevodf (fun idx m → self (idx,m)) idx m x)
      (Even,0)
⇝ − : (int → bool) code = .<
let lv6 =
  let rec g1 =
    let lv5 =
      let rec g3 x4 = if x4>0 then g1 (x4−1) else false in g3 in
    fun x2 → if x2>0 then lv5 (x2−1) else true in
  g1 in
lv6>.
```

The odd function (appearing under the generated name g_3) is nested inside even (or, 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 sevodf 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):

```
let lod0 = (* odd 0 n *)
  let rec od0 n =
    let lev0 = (* even 0 n *)
      let rec ev0 n = if n>0 then od0 (n−1) else true in ev0 in
    if n>0 then lev0 (n−1) else false in od0 in
let lev1 = (* even 1 n *)
  let rec ev1 n =
   let lod1 = (* odd 1 n *)
    let rec od1 n = if n>0 then ev1 (n−1) else lev0 n in od1 in
  if n>0 then lod1 (n−1) else lod0 n in ev1
in lev1
```

Here, the function ev1, the specialization of even m n to m= 1 calls od0 and od1. The latter calls ev1 and **fun** n → 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 **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 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:

```
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). 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 → ...), 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[3]: now, the definitions have the same scope.

```
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 sevodf 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[4]:

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

let makeau {finals;trans} self state stream =
  let accept = List.mem state finals in
  let next token = List.assoc token (List.assoc state trans) in
  .<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

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

Then mrfix (makeau au1) S generates:

```
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
```

Status Currently the proof-of-concept of the described genletrec is prototyped[5] 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

Hal Abelson, Jerry Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984. ISBN 0-262-01077-1.

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

Ralf Hinze and Ross Paterson. Derivation of a typed functional LR parser, 2003.

Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

Jun Inoue. Supercompilation via staging. In *Fourth International Valentin Turchin Workshop on Metacomputation*, June 2014.

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage: Staging with delimited control. *J. Funct. Program.*, 21(6):617–662, November 2011.

Oleg Kiselyov. Delimited control in ocaml, abstractly and concretely. *Theor. Comput. Sci.*, 435:56–76, June 2012. ISSN 0304-3975. . URL `http://dx.doi.org/10.1016/j.tcs.2012.02.025`.

Oleg Kiselyov. Simplest poly-variadic fix-point combinators for mutual recursion, August 2013. `http://okmij.org/ftp/Computation/fixed-point-combinators.html#Poly-variadic`.

Oleg Kiselyov. The design and implementation of BER MetaOCaml. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, volume 8475 of *Lecture Notes in Computer Science*, pages 86–102. Springer International Publishing, 2014.

Shriram Krishnamurthi. Educational pearl: Automata via macros. *J. Funct. Program.*, 16(3):253–267, 2006. . URL `https://doi.org/10.1017/S0956796805005733`.

Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pašalić. A monadic approach for avoiding code duplication when staging memoized functions. In *PEPM*, pages 160–169, 2006.

Don Syme. Initializing mutually referential abstract objects: The value recursion challenge. In *Proceedings of the ACM-SIGPLAN Workshop on ML (2005)*. Elsevier, March 2006.

[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.

[3] Previously, genletrec relied on the trick let g = let rec g = e in g in body, which binds two different g, one of which is in scope of the local let rec, and another is out. 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.

[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] `https://github.com/yallop/metaocaml-letrec`

Walid Mohamed Taha. *Multistage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. AAI9949870.

Jeremy Yallop. Staging generic programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 85–96, New York, NY, USA, 2016. ACM. .

Jeremy Yallop. Staged generic programming. *Proc. ACM Program. Lang.*, 1(ICFP):29:1–29:29, August 2017. ISSN 2475-1421. . URL http://doi.acm.org/10.1145/3110273.

A. Further extensions

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

A.1 Arbitrary bodies in let rec expressions

The mrfix combinator has the following type:

```
val mrfix : ((α → (β→γ) code) → (α → β code →γ code)) →
        α → (β→γ) code
```

There are two arguments: the first is a function that builds recursive definitions; the second (of type α) 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:

```
val mrfixk : ((α → (β→γ) code) → (α → β code →γ code)) →
        ((α → (β→γ) code) → γ code) → γ 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 γ code. For example, here is the code that builds a recursive group representing the state machine from previous examples, whose body is a tuple returning all the recognizer functions:

```
mrfixk (makeau au1) (fun f → .< (.~(f S), .~(f U), .~(f T)) >.)
```

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

```
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, x9, x5)
```

A.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:

```
let%staged rec f p p' = e in e'
⇝ mrfixk (fun f p p' → e) (fun f → 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

```
let%staged 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))>.
 in ack 2
```

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.