Compiling recursion in OCaml

Vincent Laviron

September 27th, 2024

OCamlPro

Context

- The OCaml language supports multiple kinds of recursive definitions
- The OCaml compiler needs to translate those to simpler languages: bytecode or assembly
- This talk explains some of the challenges and solutions

Functions

Recursive functions in the target language

- Most (all?) assembly languages support recursive functions
- However, only closed functions can be defined
- Bytecode doesn't support recursive definitions at all

The current solution: translate all recursive functions to non-recursive equivalent versions

Recursion in λ-calculus

- Recall the simple λ-calculus:

$$M ::= x \mid \lambda x.M \mid MN$$

- No recursive definitions... at first glance
- Recursion can be emulated with fix-point combinators:

$$\text{fact0} = \lambda f.\lambda x.\ \text{if } x = 0 \text{ then } 1 \text{ else } x * f(x-1)$$

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

$$\text{fact} = Y\ \text{fact0}$$

OCaml without `let rec`

- We can use similar techniques in OCaml:

```
type 'a rec_def = Rec of ('a rec_def -> 'a)
let fact0 (Rec fact) x =
  if x = 0 then 1 else x * fact (Rec fact) (x - 1)
let fact = fact0 (Rec fact0)
```

- The compiler could automatically translate regular definitions to this form
- The actual compilation scheme for recursive functions is in fact very close

Digression on closure conversion

- Both the bytecode and native backends of OCaml compile functions as closures
- Here is an example of a possible compilation scheme:

```
let a = ... and b = ...
let f x y = a + b + x + y
(* compiled into *)
let f_closed x y env = env.a + env.b + x + y
let f_env = { a; b }
let f = (f_closed, f_env)
```

- Closed functions can be compiled to independent code
- Environments are regular blocks

closure conversion: call sites

- Calling a function passes the environment to the closed function:

```
let r = f 0 1
(* compiled into *)
let r =
  let (closed_f, env) = f in
  closed_f 0 1 env
```

- Curryfication changes a number of things, but the idea stays the same
- Takeaway: closure conversion already forces us to add an extra parameter to functions
- We will reuse this extra parameter to find the closure itself

Compilation scheme for single functions

- In practice, OCaml closures are flat records:

```
let f = { fun_ptr = f_closed; a; b }
```

- That means the extra parameter `env` is the closure of the function being called
- As a result, we get to compile recursive functions for free:

```
let rec fact x =
  if x = 0 then 1 else x * fact (x - 1)
(* compiled into *)
let fact_closed x self =
  if x = 0 then 1 else x * self.fun_ptr (x - 1) self
let fact = { fun_ptr = fact_closed }
```

Multiple functions

- The scheme doesn't naturally extend to mutually recursive functions.
- The solution used in the compiler is to have multiple-entry closures:

```
let rec even x = if x = 0 then true else odd (x - 1)
and odd x = if x = 0 then false else even (x - 1)
(* compiled into *)
let even_closed x self =
  if x = 0 then true else self.odd_ptr (x - 1) (self : even :> odd)
let odd_closed x self =
  if x = 0 then false else self.even_ptr (x - 1) (self : odd :> even)
let closure = { even_ptr = even_closed; odd_ptr = odd_closed }
let even = (closure :> even)
let odd = (closure :> odd)
```

Multiple functions (continued)

- The `self :> foo` operation is needed to support indirect calls.
- It is implemented as a pointer offset: `self :> foo` points directly to the field for the function `foo` instead of the beginning of the block.
- It requires special support in the runtime (mostly the garbage collector).
- It means that projections from `self` need to take into account the offset for the current function (so `self.var` is `(self : curr_fun :> first_fun).var`).
- In Flambda 2, the record fields corresponding to recursive functions are called *function slots*, while the fields for the free variables are called *value slots*. The whole record is called a *set of closures*.

Generalised `let rec`

Examples

```
(* cyclic structure with mutation *)
type 'a dll = { data : 'a; mutable prev : dll; mutable next : dll }
let singleton data =
  let rec dll = { data; prev = dll; next = dll } in
  dll

(* infinite list, with lazyness *)
Module LazyList : sig
  type 'a t = private Nil | Cons of 'a * 'a t Lazy.t
  val cons : 'a -> 'a t Lazy.t -> 'a t
end
let rec lazy_cycle = lazy (LazyList.cons 0 lazy_cycle)
```

More examples

```ocaml
(* infinite list, no lazyness *)
let rec cycle = 0 :: cycle

(* sequence *)
type 'a node = Nil | Cons of 'a * 'a t
and 'a t = unit -> 'a node
let rec inf_node = Cons (0, inf_seq)
and inf_seq () = inf_node
```

The rules

Recursive value definitions in OCaml must follow the following rules:

- Only variables on the pattern side
- The value of the recursive variables must not be inspected until all variables are fully initialised
- The values produced by the definitions must be compatible with the compiler's pre-allocation scheme

The second and third rule are checked (mostly) conservatively by an algorithm described in the following article:

Reynaud, Alban, Gabriel Scherer, and Jeremy Yallop. 2021. "A Practical Mode System for Recursive Definitions". *Proceedings of the ACM on Programming Languages* 5 (POPL): 1–29

Compilation scheme: Outline

- Bind the recursive variables to uninitialised values
- Use the definitions to produce a value for each variable
- Copy the contents of the new values back into the uninitialised blocks

```
let cycle = caml_alloc_dummy(2)
let cycle' = 0 :: cycle
let () = caml_update_dummy(cycle, cycle')
```

What can be pre-allocated

Obvious candidates:

- Records, tuples
- Non-constant variant constructors (`a :: b`, `Some x`, ...)

Less obvious cases:

- Lazy blocks
- Boxed numbers (float, int32, ...)
- Constant-size arrays
- First-class modules

Complex cases:

- Closures
- Constant variant constructors and integers

What *cannot* be pre-allocated

- Function applications
- Branching expressions
- Objects

```
let b : bool = ...
let rec l = b :: l (* Ok *)
let rec l = List.cons b l (* Rejected *)
let rec l = if b then true :: l else false :: l (* Rejected *)
```

Complex case 1: integers

```
let rec not_rec = 42
let rec not_really_rec = let _ = not_really_rec in 53
```

- Integers cannot be pre-allocated
- The final value is an integer, so cannot contain occurrences of anything else.
- The definition cannot inspect the values of any recursive variable (second rule)

Solution: replace recursive variables with dummy values

```
let dummy = Obj.magic 0
let not_really_rec = let _ = dummy in 53
```

Complex case 2: closures

- Size of closures is hard to compute:

```
let x = Some 0
(* Which are the free variables of [f] ? *)
let f () = Option.map succ x
```

- A function may even be defined as one of a set of recursive functions:

```
let rec f =
  let rec g () = f ()
  and h () = g ()
  in h
```

Closures: the old way

- Wait until functions have been compiled to explicit closures
- Use the same scheme as for blocks

Pros:

- Can easily mix blocks and functions
- Handles non-syntactic functions too (`let rec f = let () = () in fun x -> f x`)

Cons:

- Less efficient than the normal function scheme
- Code duplication (one for each backend)
- Distance between the check and the compilation

Closures: the new way

- Split functions from other recursive definitions
- Compile in order:
  - Non-recursive definitions
  - Pre-allocation of non-function definitions
  - Functions definitions (mutually recursive)
  - Computation of non-function definitions
  - Backpatching of non-function definitions

Handling non-syntactic functions

- The recursive check only allows two other cases:
  - Sequential code ending with a syntactic function
  - Sequential code ending in a variable (known to be bound to a function)
- By η-expanding the second case we reduce to the first case only
- We handle the first case by a partial closure conversion

Partial closure conversion

- An extra recursive variable is added for the function's local environment
- The original variable is bound to just the original syntactic function, with local variables replaced by accesses from the environment variable
- The environment variable is bound to the original definition, with the function replaced by a block allocation containing all local variables used by the function
- The environment variable can then be pre-allocated and back-patched

Partial closure conversion: example

```
let rec f =
  let rec g () = f () and h () = g () in
  h
(* Eta-expand *)
let rec f =
  let rec g () = f () and h () = g () in
  fun x -> h x
(* Partial closure *)
let rec f = fun x -> f_env.h x
and f_env =
  let rec g () = f () and h () = g () in
  { h }
```

Recursive modules

Using the previous scheme

- Regular modules are blocks of known size
- Functors are functions

-> The algorithm for recursive values should “just work”

A typical example

```
module rec Tree : sig ... end = struct
  type t = TreeSet.t
  let compare = TreeSet.compare
end
and TreeSet : sig ... end =
  Set.Make(Tree)
```

Inspects recursively bound variables: rejected by the recursive value check

Design choices

- Allow referencing module fields in definitions
- Allow using the recursive modules as functor arguments
- Allow definitions that are not obviously well-founded
- Fail at runtime for actually problematic definitions

More examples

```
module type S = sig val f : unit -> unit end
module rec M1 : S = M1
module rec M2 : S = struct
  let f = M2.f
end
module rec M3 : S = struct
  let f () = M3.f ()
end
module Id(X : S) : S = X
module rec M4 : S = Id(M4)
```

Semantics

- Conservative model:
  - Similar to `let rec`, but dynamic
  - The module itself exists, and can be stored in blocks and closures
  - Any other use of the module throws an exception at runtime
- Problem: hard to implement
- Current version: more permissive (reading module fields is allowed, using the fields may throw a runtime exception or not)
- Some definitions are rejected at compile time

Compilation scheme: outline

- Pre-allocate each module with a block full of type-safe dummy values
- Compute the actual definitions
- Patch each module field using the fields of the new computed module

Dummy values are constructed so that examining one leads to a runtime error

Safe modules

- Not all types allow constructing safe dummy values
- Dummy values can be constructed for function types
- Plus a few other cases (lazy, sub-modules, classes)
- A module where all the runtime fields allow dummy values is called safe
- Safety only depends on the module type

Compiling unsafe modules

- Definitions containing unsafe modules can be allowed
- Each cycle in the runtime module dependency graph must contain at least one safe module
- Initialisation can then proceed in topological order
- Dependencies include all references in the compiled definition (no types)

Tree example again

```
module rec Tree : sig ... end = struct
  type t = TreeSet.t
  let compare = TreeSet.compare
end
and TreeSet : sig ... end =
  Set.Make(Tree)
```

- `Tree` is safe (`compare` is a function)
- `TreeSet` is unsafe (`empty` is not a function)
- `TreeSet` doesn't depend directly on `TreeSet`, so the definition is accepted

Tree example, compiled

```
(* initialisation of safe modules *)
module Tree = struct
  let compare = (fun _ -> raise Undefined_recursive_module)
end
(* computation of modules in topological order *)
module TreeSet = Set.Make(Tree)
module Tree_new = struct
  let compare = TreeSet.compare
end
(* back-patching *)
CamlinternalMod.update_mod Tree Tree_new
```

Additional things

- A functor is not a safe module
  - Makes it harder to define recursive functors
  - Makes it more likely that a recursive definition including functors and regular modules initialises properly
- Support for non-function constants in safe modules was considered but never finalised
- The dummy safe functions actually check at runtime if their module has been initialised, and call the new function in that case

Recursive modules vs. recursive values

- Modules:
  - Support focused on modules full of functions
  - No well-foundedness requirement, can fail at runtime
  - Topological sort can re-order definitions
- Values:
  - Large support for many language constructs
  - Strict well-foundedness check, no runtime failures
  - Minimal re-ordering (non-recursive definitions can be lifted out)

Classes

Objects are not recursive

- Object structures can specify a `self` variable
- Methods can use this variable to refer to the current object
- Very similar to closure conversion
- This removes the need for recursive object bindings

Recursion in classes

- All classes are recursive
- Recursion can take two forms:
  - Recursion in class expressions
  - Recursion through `new`

```
class type ct = object method x : int end
class c : ct = c
class c : ct = object method x =
    let module M = struct class d : ct = c end in (new M.d).x
  end
class c : ct = object method x = (new c)#x end
class c : ct = let o = new c in object method x = o#x end
class c : ct = let o () = new c in object method x = (o ())#x end
```

Class expressions

- In class expressions, only classes defined strictly earlier are allowed

```
class c : ct = c (* Bad *)
class c : ct = object method x = 0 end
and d : ct = c (* Good *)
class d : ct = c
and c : ct = object method x = 0 end (* Bad *)
```

Use through `new`

- `new c` expressions are allowed in any context that will end up under a function after class compilation.
- In practice, only toplevel `let`-bindings are not allowed to use the class except under a function

```
class c : ct = let o = new c in object method x = o#x end (* Bad *)
class c : ct = let o () = new c in object method x = (o())#x end (* Good *)
class c : ct = (fun () -> let o = new c in object method x = o#x end) () (* Good *)
```

Compilation scheme

- The runtime representation of a class is a record of functions
- The compiler first translates class expressions to normal expressions producing such records
- This definition is then compiled using the scheme for recursive values:
  - The restriction on the use of `new` ensures that uninitialised classes are not used to create objects
  - The restriction on class expressions ensures that the definition order is compatible with the topological order

Bibliography

[1] A. Reynaud, G. Scherer, and J. Yallop, “A practical mode system for recursive definitions,” *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–29, 2021.

[2] X. Leroy *et al.*, “The OCaml system release 5.2: Documentation and user's manual.” 2024.

[3] X. Leroy *et al.*, “The OCaml system release 5.2: Documentation and user's manual.” 2024.

[4] X. Leroy *et al.*, “The OCaml system release 5.2: Documentation and user's manual.” 2024.

[5] V. Laviron and L. Ayanides, “Compile recursive bindings in Lambda.” 2023.

Bibliography

[6] X. Leroy, "A proposal for recursive modules in Objective Caml," *Available from the author's website*, 2003.

[7] D. Rémy and J. Vouillon, "Objective ML: An effective object-oriented extension to ML," *Theory and practice of object systems*, vol. 4, no. 1, pp. 27–50, 1998.

[8] J. Vouillon, "Conception et réalisation d'une extension du langage ML avec des objets," 2000.