An OCaml use case for strong call-by-need reduction

Gabriel Scherer (Partout, INRIA, France)
Nathanaëlle Courant (Cambium, INRIA, France)

2022

Shapes

The compiler artifact produces build artifacts that include, in particular, the "typed tree" of each source file. This is a good representation to use for programming tools (IDEs, code analyzers, etc.), but it is sometimes too complex. Consider the following OCaml program:

```
module Origin = struct let x = 1 end
module Second = struct let x = 2 let y = 2 end

module F(X) = struct
  include X
  include (Second : sig val y : int end)
end

module M = F(Origin)
```

Now suppose a user, or an OCaml-processing tool, wants to know where to find "the definition of" the identifier `M.x`. The answer is: `let x = 1` in the `Origin` module. But how do we know that? Can tools tell without having to reimplement their own understanding of the whole OCaml module system?

To solve this problem for the Merlin language server, Ulysse Gérard, Thomas Refis and Leo White recently introduced a notion of "shapes" to represent the structure of OCaml modules, and track where each term-level construct inside a module is defined. The above program can be erased into a shape, are just lambda-terms with functions (representing functors), records (representing structures), and "locations of definitions" as primitive values. "Normalizing" this shape provides direct access to the definition of `M.x`.

They proposed that the OCaml type-checker should be in charge of computing the shape of each module / compilation unit / source file, pushing all intricate knowledge of the OCaml module system back into the compiler codebase. Those shapes would be stored in the build artifacts, easy to use by OCaml-processing tools. Shapes, they suggested, should be stored in normal form in the build artifacts, so that tools do not have to recompute the reduction each time they are asked about an installed package or library.

The OCaml compiler uses separate compilations, it is fed source files independently and compiles them with only a partial knowledge of its dependencies. In particular, when we "reduce" the shape term for an OCaml source file, the shape of its dependencies are unknown, they are treated as free variables. Inside functors (such as `F(X)` in our example), the input parameter is also a free variable, and we still want to reduce the shape of the body of the functor. This means that we want the OCaml compiler to perform deep reduction of shape terms. Code-processing tools that have access to a whole program can then "link" those shapes together, perform more reduction, and get a closed normal form.

Strong call-by-need

Gérard, Refis and White implemented a very naive strong evaluator for shapes – repeated beta-reductions. This works fine for all the programs they tested, and was merged in development version of the OCaml compiler. The OCaml compiler release process involves building all (opam) OCaml packages, and the next testing round found out that the compiler would blow up on a specific module-heavy source file in the Irmin project: shape reduction would grind to a halt and consume massive amounts of memory[1]. A not-very-native evaluator was called from.

Our contribution is an arguably-simple implementation of strong call-by-need evaluation, meant for shape reduction and integrated into the OCaml compiler. To our knowledge, two aspects are new:

1. This may be the first use-case for strong call-by-need reduction outside proof assistants.
2. Call-by-need evaluation is usually presented as small-step reduction strategies with explicit sharing – on abstract machines (Barras, 1999; Biernacka and Charatonik, 2019) or term calculi (Balabonski, Barenbaum, Bonelli, and Kesner, 2017). We present a "direct evaluator" implementation – a recursive function that returns normal forms – that is arguably simpler, and may make strong call-by-need more approachable to newcomers.

What we *don't* have is a formal argument that our implementation matches the reduction strategy of existing strong call-by-need abstract machines.

Performance numbers[2]: without computing shapes, the problematic Irmin source file compiles in 0.39s and produces a built artifact of 2538Ko – around 2.5Mo. With the naive evaluator plus some specific optimizations proposed by Gérard and Refis, compilation time went to 2.15s and artifact size to 91Mo. With our strong call-by-need evaluator, compile-time is 0.40s and artifact size is 2552Ko.

Note: why the explosion?

It is not entirely clear to us what precisely in the module-heavy Irmin source file made the naive evaluator – and many variants of it – blow up in time and memory. We tried to analyze it, but it is hard to make sense of the megabytes of output it would provide.

Our current understanding is that while both weak and strong reduction for the lambda-calculus can result in blowup in normal forms, such a blowup is *uncommon* in practice in weakly-evaluated programs, but *more common* in strongly-evaluated program – see the discussion of size explosion in Accatoli and Leberle (2019). Consider for example the following program:

```
module F(A) = struct
  let x = A.x
  let y = A.y
  let z = A.z
end
```

With weak reduction, the functor `F` will only reduce when provided a concrete module `A` providing values for `x, y, z`. The size of the normal form will be the size of `A` plus a constant. With open or strong reduction, the application `F(Lib.Make(Foo))`, where `Lib` is a free variable, musth andle `Lib.Make(Foo)` as an inert normal form and reduce the body of `F`, returning a normal form duplicating the argument `Lib.Make(Foo)` three times.

In the Irmin codebase we have observed quadratic blowups in program size due to subterms of this form, which can be common: we desugar a signature sealing `(X : S)` into a term of this shape – in this example `S` would contain three values `x`, `y`, `z`.

[1] https://github.com/ocaml/ocaml/pull/10796
[2] https://github.com/ocaml/ocaml/pull/10796

Our implementation, step by step

In the interest of space, we restrict here our language of shapes to variables, applications and lambda-abstractions – full shapes also contain records and projections, and contain location information.

```
type var = Ident.t
and t =
  | Var of var
  | Abs of var * t
  | App of t * t
```

We describe here the iterations we went through to obtain our final strong call-by-need evaluator. The full code for all steps (in the simplified setting of our exposition) can be found at https://gitlab.com/gasche-papers/shapes/-/blob/trunk/code.ml; it contains additional explanations for technical details omitted here. The in-compiler implementation is at https://github.com/ocaml/ocaml/blob/3aaa9a7/typing/shape.ml#L217-L438.

Naive strong reduction

The environments we pass around during reduction contain either a value (when replacing a variable by a value) or a free variable (when evaluating under a binder). We will use different evaluation strategies with diffent notions of "value", so our environments are parametrized over the type `'v` of values. They use the `'a Ident.tbl` type of identifier-indexed maps provided by the OCaml compiler.

```
type 'v open_value =
  | Val of 'v
  | Free of var
type 'v open_env = 'v open_value Ident.tbl
```

An (environment-based) naive strong evaluator normalizes terms by first evaluating into a structured type of "normal forms", and then "read back" those normal forms into (normalized) terms. A type of normal forms is as follows:

```
type nf = (* normal forms *)
  | Ne of ne
  | Clos of env * var * t * var * nf
and ne = (* neutral terms *)
  | Var of var
  | App of ne * nf

and env = nf open_env
```

Note that the `Clos` case, representing function closures, is unusual: the normal form of `Abs(x, t)` will be of the form `Clos(env, x, t, x', nf)` where `env` is the current environment (as expected of closure), and `nf` is the strong normal form of `t` where occurrences of the bound variable `x` have been replaced by a fresh variable `x'`. (This fresh `x'` is important to avoid variable capture for input terms that may reuse the same identifier in several bindings. The OCaml compiler maintains unicity of binders, so we could always reuse `x` at the cost of a less general exposition.)

For example, a normal form of `Abs(x, App(id, Var x))` would be, where `id` is the identity function and `y` is a fresh identifier:

```
Clos(Ident.empty,
     x, App(id, Var x),
     y, y)
```

Why store both `t` and `nf`? This is one of the mysteries of strong evaluation; one would expect to store only `nf`, but this is in fact terrible in the application case. Indeed, if you consider a redex `(fun x -> t) u`, you want

to normalize `t[u/x]` (where `N(...)` is the normalization function) and *not* `N(t)[u/x]`: the intermediate term `N(t)` is likely to be much larger than `t`, due to the size explosion phenomenons we mentionend previously, and `N(t)` could contain many more occurrences of `x` that will form redexes once replaced by `u`.

Long story short: a closure `Clos(env, x, t, x', nf)` uses the `t` part when the closure is applied (passed parameters), and the `nf` part when the closure is returned as a value (in strong normal form).

The implementation of evaluation and read-back are rather direct from this type.

```
let rec eval env : t -> nf = function
  | Var x ->
    begin match Ident.find_same x env with
    | Val v -> v
    | Free x -> Ne (Var x)
    end
  | Abs (x, t) ->
    let y = fresh x in
    Clos (env, x, t, y,
     let env' = Ident.add x (Free y) env in
     eval env' t)
  | App (t, u) ->
    let f, arg = eval env t, eval env u in
    match f with
    | Ne n -> Ne (App (n, arg))
    | Clos (env', x, body, _y, _v) ->
      eval (Ident.add x (Val arg) env') body

let rec read_back : nf -> t = function
  | Clos (env', _x, _t, y, v) -> Abs (y, read_back v)
  | Ne (App (n, v)) -> App (read_back (Ne n), read_back v)
  | Ne (Var x) -> Var x

let normalize env t =
  read_back (eval env t)
```

There are two apparent issues with this implementation of strong reduction:

- Useless reduction: when evaluating `Abs(x,t)`, we always evaluate `t` to a strong normal form, but normalizing `t` is useless in our strategy if this abstraction is only applied, never returned as a value.
- Loss of sharing: while our syntax does not contain an explicit sharing construct (`let`, explicit substitutions...), you can notice that the normal form of `App(Abs(x,t), v)` does *not* contain several copies of the normal form of `v`, even if `x` occurs several times in `t`. Instead, we return a term where all occurrences of the normal form of `v` come from looking up `x` in the environment, and are thus physically shared in memory. In other words, the natural semantics of the host language provide implicit sharing here.

  Unfortunately this sharing is lost immediately by the `read_back` function, which will map over each occurrence of the normal form of `v` and create physically-distinct read-back forms for them.

Memoized strong reduction

To solve the "loss of sharing" problem without introducing explicit sharing construct (we are too lazy to do that), one functional solution is to introduce memoization of the `eval` and `read_back` function. This is a trivial change to implement with a well-chosen memoizing-fixpoint combinator (using hashtables).

Strong call-by-need

The "useless reduction" issue can be solved by using lazyness in the host language: just make the `nf` component in closures a lazy thunk, computed on demand and shared between consumers.

```
type nf = (* normal forms *)
  | Ne of ne
  | Clos of env * var * t * var * dnf
and ne = (* neutral terms *)
  | Var of var
  | App of ne * nf
and dnf = nf Thunk.t (* delayed normal forms *)

and env = dnf open_env
```

Adapting the evaluator is again fairly simple.

Strong call-by-need with memoization

In fact, if we use memoization, we don't need to use lazy thunks in the host language. A value of type `nf Thunk.t` can be represented exactly by a pair `env * t`, where "forcing" the thunk is just calling the `eval` function. Normally this would duplicate computations, but remember that our `eval` function is memoized! In essence, instead of (internally mutable) lazy thunks we use the memoization table for sharing.

```
type nf = (* normal forms *)
  | Ne of ne
  | Clos of env * var * t * var * dnf
and ne = (* neutral terms *)
  | Var of var
  | App of ne * dnf
and dnf = Delayed of env * t

and env = dnf open_env
```

This was our last stop in the OCaml compiler codebase, which uses this variant.

In fact, it is possible to simplify it further by noticing that the `dnf` element of closures is redundant with the `env` and `t` already stored in them. It is possible to present the exact same implementation in a slightly different style:

```
type nf = (* normal forms *)
  | Ne of ne
  | Clos of env * var * t
and ne = (* neutral terms *)
  | Var of var
  | App of ne * env * t
and dnf = Delayed of env * t

and env = dnf open_env
```

Notice how closures now carry exactly the same information as closures for a weak reduction strategy. The `read_back` case for closures looks like this:

```
| Clos (env, x, t) ->
  let y = fresh x in
```

```
  let env' = Ident.add x (Free y) env' in
  Abs (x, normalize env' t)
```

This is reminiscent of Grégoire and Leroy's "iterated symbolic weak reduction and read-back" strategy (Grégoire and Leroy, 2002), but for a call-by-need rather than call-by-value strategy.

References

Beniamino Accatoli and Maico Leberle. Useful open call-by-need. 2019.

Thibaut Balabonski, Pablo Barenbaum, Eduardo Bonelli, and Delia Kesner. Foundations of strong call-by-need. 2017.

Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives.* PhD thesis, 1999.

Malgorzata Biernacka and Witold Charatonik. Deriving an abstract machine for strong call by need. 2019.

Benjamin Grégoire and Xavier Leroy. A Compiled Implementation of Strong Reduction. 2002.