Necro ML: Generating OCaml Interpreters

Louis Noizet
Alan Schmitt

Abstract

We present Necro ML, a tool which allows to generate interpreters from skeletal semantics in a modular way, using monads to handle different ways to interpret computations.

ACM Reference Format:
Louis Noizet and Alan Schmitt. 2022. Necro ML: Generating OCaml Interpreters. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Skel

Skel [2] is a statically typed domain specific language to describe operational semantics of programming languages. We call Skel codes "skeletal semantics". It is both light and powerful, and the skeletal semantics can be used to generate OCaml interpreters, Coq formalization, and debuggers, among other things.

The Skel language is closely based on ML with an added construct called branching to perform non-deterministic computations. A skeletal semantics is a list of *type declarations* and *term declarations*. Declarations can be unspecified, which means we just declare that a type or a term exists. For terms, we give its type but not its implementation. This lets us hide internal representations that do not matter or whose specification will be done later. Declarations can also be specified, in which case we have to give a definition.

Type definitions can be of three sorts: variant types, i.e., algebraic data types, defined by their constructors and the type of their arguments; record types, defined by their fields and the type they each carry; and type aliases, which cannot be circularly defined. Types also allow arrow types and product types, similarly to other functional languages.

Skeletal expressions are either *terms* or *skeletons*. The latter is used to represent computations, while the former serves to represent evaluated values. This is similar to computational λ-calculus, defined by Moggi [5], extended for ML constructs.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Here is an example of a skeletal semantics with specified and unspecified types and terms:

```
type ident
type term =
    | Var ident
    | Lam (ident, term)
    | App (term, term)

val subst: (ident, term, term) → term
val ss (t:term): term =
    match t with
    | App (t1, t2) ->
        branch
            let t1' = ss t1 in
            App (t1', t2)
        or
            let t2' = ss t2 in
            App (t1, t2')
        or (* beta-reduction of a redex *)
            let Lam (x, body) = t1 in
            let Lam _ = t2 in (* t2 is a value *)
            subst (x, t2, body) (* body[x←t2] *)
        end
    | _ -> (branch end: term)
    end
```

The branching in this example describes a non-deterministic reduction of a λ-expression in small step. Any branch may be taken as long as it succeeds. For instance, the first branch can only be taken if `t1` reduces. The third branch uses a destructuring **`let`** to ensure `t1` is a lambda.

Skel is close to ML languages such as OCaml or SML, but we chose to create a DSL so that the AST can be minimal. Skel's AST is defined in [6], and contains only 114 lines of specification. This allows us to handle and extract skeletal semantics easily.

2 OCaml generation

2.1 Necro

As mentioned above, Necro is an ecosystem to handle skeletal semantics. Its core is Necro Lib [6], an OCaml library file which defines Skel's AST and a set of functions to handle skeletal semantics, including a `parse_and_type` function, which reads a file, and returns a typed skeletal semantics. Anyone thus has access to the AST and to basic functions, so they can create tools to work with skeletal semantics and extend the Necro ecosystem. In addition to Necro Lib, we already provide some tools. One of them, presented here,

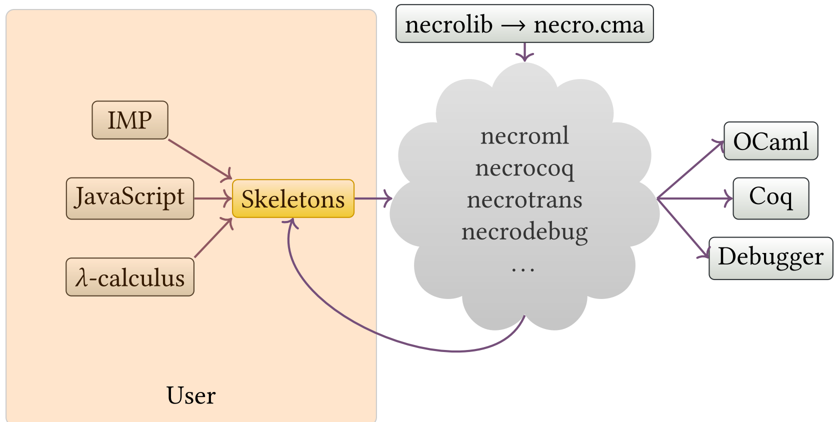

Figure 1. The Necro ecosystem

is Necro ML [4], which generates OCaml interpreters from skeletal semantics.

2.2 Necro ML

Necro ML generates an interpreter for a skeletal semantics. As we explained in Section 1, there are some unspecified data in the skeletal semantics. This is handled by Necro ML in a modular way: we generate a functor `MakeInterpreter`, which produces an interpreter once it is given an OCaml implementation for unspecified types and terms.

The embedding is mainly a shallow one, but we have to handle branches with care, since OCaml does not have a non-deterministic construct. We will show how we handle this in Section 3.

Let us now present the structure of a generated interpreter: First, a module type `TYPES` is generated which contains the unspecified types. Then a module type `UNSPEC` is provided, for unspecified terms and types. There is a functor `Unspec` that, given a module of type `TYPES`, creates a default instantiation where every unspecified function raises an error. One may then apply this functor and override the default implementation of unspecified terms with the actual one. Then, we define an `INTERPRETER` module type, which contains the signature for all specified and unspecified terms and types. Finally, we define the `MakeInterpreter` functor, which takes into argument a module of type `UNSPEC`, and produces a module of type `INTERPRETER`.

The `Unspec` functor takes an other argument, an interpretation monad, which we will describe right away.

3 Interpretation monad

We explained above how we cannot use solely a shallow embedding, because of the non-deterministic branching construct. To this effect, we use an embedding monad, or interpretation monad, to describe computations. So terms of type `'a` are shallowly embedded as OCaml expressions of type `'a`, while skeletons of type `'a` are deeply embedded as OCaml expressions of type `'a M.t` Several interpretation monads exist. These monad must define the type `'a t`, and how to handle `let`-bindings and branches. The monad's module type is defined this way:

```
module type MONAD = sig
    type 'a t
    val ret: 'a -> 'a t
    val bind: 'a t -> ('a -> 'b t) -> 'b t
    val branch: (unit -> 'a t) list -> 'a t
    val fail: string -> 'a t
    val apply: ('a -> 'b t) -> 'a -> 'b t
    val extract: 'a t -> 'a
end
```

The `fail` operator takes as input a string which constitutes the error message. It is used for instance when trying to match a pattern with a non-matching value. The `bind` is used to embed `let`-bindings. The `branch` is the one that performs the non-deterministic choice, and `extract` is a usability construct without any theoretical meaning, which allows to extract a value from the monad.

The first and the simplest way to instantiate this module type is the identity monad, where `'a M.t = 'a`. For branches, we just take the first branch that succeeds:

```
module ID = struct
  exception Branch_fail of string
  type 'a t = 'a
  let ret x = x
  let rec branch l =
    match l with
    | [] -> raise (Branch_fail "No branch matches")
    | b1 :: bq ->
      try b1 () with Branch_fail _ -> branch bq
  let fail s = raise (Branch_fail s)
  let bind x f = f x
  let apply f x = f x
```

```
  let extract x = x
end
```

This seems naive but is useful in most cases with deterministic programming languages where branches are used only in non-overlapping cases. Moreover, we can randomize the monad using the randomizing functor so that it is not the first branches that is taken, but any random succeeding branch. The randomizing functor is defined this way:

```
let shuffle l =
    let () = Random.self_init () in
    let lrand = List.map (fun c ->
        (Random.bits (), c)) l in
    List.sort compare lrand |> List.map snd

module Rand (M: MONAD) = struct
    include M
    let branch l =
        M.branch (shuffle l)
end
```

Of course this does not work in all cases, and can cause an issue if we find out later that a branch was not the right one, in the sense that it contains a computation which fails after the branching has completed. An example thereof is the following:

```
let f =
    branch
        (λ _ → (branch end:()))
    or
        (λ _: () → ())
    end
in f ()
```

The **ID** monad will take the first branch (and the randomized one *may* take it), and it will succeed. Afterwards, when applying it to (), it will produce an error. The problem is that there is no way to backtrack. This idea gave birth to a continuation monad with a backtracking point. This monad is defined as follows :

```
module ContPoly = struct
    type 'b fcont = string -> 'b (* backtracking
    ↪  point *)
    type ('a,'b) cont = 'a -> 'b fcont -> 'b
    type 'a t = { cont: 'b. (('a,'b) cont -> 'b
    ↪  fcont -> 'b) }
    let ret (x: 'a) = { cont = fun k fcont -> k
    ↪  x fcont }
    let bind (x: 'a t) (f: 'a -> 'b t) : 'b t =
        { cont = fun k fcont -> x.cont (fun v
        ↪  fcont' -> (f v).cont k fcont') fcont
        ↪  }
    let fail s = { cont = fun k fcont -> fcont s
    ↪  }
    let rec branch l = { cont = fun k fcont ->
                begin match l with
                | [] -> fcont "No branch matches"
                | b :: bs -> (b ()).cont k (fun _ ->
                ↪  (branch bs).cont k fcont)
                end}
    let apply f x = f x
    let extract x = x.cont (fun a _ -> a) (fun s
    ↪  -> failwith s)
end
```

There are several other interpretation monads, and they all have their purposes. The main point is that it is very easy to change the interpretation monad, as no code has to be rewritten. The only piece of code that has to change is the choice of monad used to instantiate the **Unspec** functor.

4 Conclusion

Necro ML is a flexible tool to generate interpreters for programming languages. It performs a shallow embedding of types, and a semi-deep embedding of expressions, to handle non-determinism. It has been shown to work on significant files, since it has been used on an ongoing formalization of JavaScript [3] which can already run simple programs.

Other projects have a significant overlap or development which might be related. First, Necro Debug is a step-by-step execution of a skeletal semantics, to describe the computation of a term.[1] It uses the same approach and module types than Necro ML, hence one can use the specification of unspecified types and terms both in the generated interpreter with Necro ML, and in the generated debugger with Necro Debug. Second, there are works to generate interpreters in ML with Necro Coq, by using Coq's extraction mechanism [1].

References

[1] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt. Certified Abstract Machines for Skeletal Semantics. In *Certified Programs and Proofs*, Philadelphia, United States, January 2022.

[2] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. Skeletal Semantics and their Interpretations. *Proceedings of the ACM on Programming Languages*, 44:1–31, 2019. URL: https://hal.inria.fr/hal-01881863, doi:10.1145/3290357.

[3] Adam Khayam, Louis Noizet, and Alan Schmitt. JSkel: Towards a Formalization of JavaScript's Semantics, 2022. Submitted for publication.

[4] Enzo Crance Martin Bodin, Nathanaëlle Courant and Louis Noizet. Necro Ocaml Generator, https://gitlab.inria.fr/skeletons/necro-ml. URL: https://gitlab.inria.fr/skeletons/necro-ml.

[5] Eugenio Moggi. Computational lambda-calculus and monads. *Proceedings. Fourth Annual Symposium on Logic in Computer Science*, 1988.

[6] Louis Noizet. Necro Library, https://gitlab.inria.fr/skeletons/necro. URL: https://gitlab.inria.fr/skeletons/necro.

[1] https://skeletons.inria.fr/debugger/index_while.html