Stating and Handling Semantics with Skel and Necro

Louis Noizet, Alan Schmitt

Stating and Handling Semantics with Skel and Necro

Louis Noizet, Alan Schmitt

RESEARCH REPORT
N° 9449
January 24, 2022
Project-Team Celtique

ISSN 0249-6399 ISRN INRIA/RR--9449--FR+ENG

Stating and Handling Semantics with Skel and Necro

Louis Noizet, Alan Schmitt

Project-Team Celtique

Research Report n° 9449 — January 24, 2022 — 20 pages

Abstract: We present Skel, a meta language designed to describe the semantics of programming languages, and Necro, a set of tools to manipulate said descriptions. We show how Skel, although minimal, can faithfully and formally capture informal specifications. We also show how we can use these descriptions to generate OCaml interpreters and Coq formalizations of the specified language.

Key-words: Skel, Necro, semantics, skeletal semantics, OCaml, Coq, formalization

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Énoncer et manipuler des sémantiques avec Skel et Necro

Résumé : Ce document présente Skel, un méta-langage conçu pour décrire les sémantiques de langages de programmation, et Necro, un ensemble d'outils pour manipuler lesdites descriptions. Nous montrons comment Skel, bien que minimal, peut fidèlement et formellement capturer des spécifications informelles. Nous montrons également comment ces descriptions peuvent être utiliséès pour générer des interpréteurs OCaml et des formalisations Coq du langage spécifié.

Mots-clés : Skel, Necro, sémantique, sémantique squelettique, OCaml, Coq, formalisation

1 Introduction

In order to formally prove properties of a programming language, or programs in said language, it is necessary to have a formal specification of its semantics. We expect the tool and the language used to describe this formalization to be executable, usable, and easily verifiable, that is, close to a paper-written specification.

Necro provides a language (Skel) and a set of tools to formalize and interpret the semantics of programming languages. Necro fulfills these requirements and more.

- Skel is designed to be light: it is easy to understand its semantics. Also, a light language guarantees an easy maintainability and facilitates the development of tools to handle it.
- Skel is powerful enough to express intricate semantical features, as proven by its use in an ongoing formalization of JavaScript's semantics [**?**].
- A semantics described in Skel can be close to a previously written formulation, be it as inference rules or as an algorithmic specification.
- Necro provides a comprehensive and extensible set of tools to manipulate said semantics. For instance, to translate it into an interpreter (Necro ML, see Section 4.2), or to give a formalization in the proof assistant Coq (Necro Coq, see Section 4.3).

Skel is a statically strongly typed language. First introduced in [**?**], we redesigned it, with a syntax closely based on ML, whose simplicity and expressiveness are well-known. We also added support for polymorphism and higher order.

One can think of Skel as a kind of programming language, or one can think of it as a language to define inductive rules. Both approach make sense, and they are both interesting respectively when writing a semantics whose formalization is a computational one, (e.g. ECMA-262 [**?**]), or when writing a semantics defined with inferrence rules (e.g. λ-calculus). Since it has been created to talk about semantics, Skel allows to talk about partiality and non-determinism

Necro contains several tools to manipulate skeletal semantics (semantics written in the language Skel). First, necrolib offers basic functionalities (parsing, typing, printing, and simple transformations). It is provided as a library to enable anyone to manipulate the AST to produce whatever tool they desire. Furthermore, we also provide the tools necroml, necrocoq, and necrodebug, that use said library, and generate respectively an OCaml interpreter, a Coq formalization, and a debugger.

Let us describe how one specifies a semantics in Skel using the simple imperative language IMP as example.

A skeletal semantics is a list of declarations. There are two kinds of declarations, namely *type declarations* and *term declarations*. These declarations can be either declaration of an unspecified type/term, or a specified one, in which case we give its definition.

When a type is unspecified, we only give its name. For instance, when writing the semantics of IMP, we might not want to specify how literals or integers are internally represented, so we declare:

```
type literal
type int
```

When a type is specified, there are several ways to formalize its definition. One can declare a variant type (i.e., an algebraic data type), by giving the list of its constructors together with their input type. One can declare a type alias with the `:=` notation. One can also declare a record type by listing its fields and their expected types.

For instance, keeping with the semantics of IMP, the type of expressions and statements would be defined that way:

```
type expr =
| Const literal
| Var ident
| Plus (expr, expr)
| Equal (expr, expr)
| Not expr
```

```
type stmt =
| Skip
| Assign (ident, expr)
| Seq (stmt, stmt)
| If (expr, stmt, stmt)
| While (expr, stmt)
```

In this example, `(expr, expr)` is the product type with two components of type `expr`, see Figure 1 for more details.

An example for a record type is the following:

```
type euler_int = ( re: int , im: int )
```

And finally, an example of alias is the following:

```
type nat := int
```

Note that types are all implicitely mutually recursive, so the order in which they are given does not matter.

Now let us present term declarations. To declare an unspecified term, we just give its name and type. To declare a specified term, we must also give its definition.

An instance of unspecified term is the addition. Since the type `int` was not specified, it is impossible to specify the addition, so we would declare:

```
val add: int → int → int
```

Assuming we have defined booleans to be `type boolean = | True | False`, a simple example of a specified term is the following.

```
val neg (b:boolean): boolean =
    branch
        let True = b in False
    or
        let False = b in True
    end
```

The `branch ... or ... end` construct is a Skel primitive to deal with non-deterministic choice. The destructuring pattern matching `let True = b in` asserts that `b` is equal to `True`. If it is not, then the first branch gives no result. Here, the two branches are exhaustive and don't overlap, so `neg` will only yield one result. In general, overlapping branches provide non-determinism, and non-exhaustive branches provide partiality.

Note that the first line, as is usual with OCaml for instance, is syntactic sugar for the following:

```
val neg: boolean → boolean = λ b : boolean →
```

The file `while_better.sk`, on the Necro Test repository[1] gives IMP's semantics. The `eval_stmt` term is a more complete example. We extract the part that evaluates a while loop:

[1] `https://gitlab.inria.fr/skeletons/necro-test/-/blob/FSCD2022/while_better.sk`

$$
\begin{array}{rrcl}
\textsc{Term} & t & ::= & x_i \mid C\ t \mid (t, \ldots, t) \mid \lambda p : \tau \to S \mid t.f \mid (f = t, \ldots, f = t) \mid t \leftarrow (f = t, \ldots, f = t) \\
\textsc{Skeleton} & S & ::= & t_0\ t_1 \ldots t_n \mid \text{let}\, p = S \,\text{in}\, S \mid \text{let}\, p : \tau \,\text{in}\, S \mid \text{branch}\, S \,\text{or}\, .. \,\text{or}\, S \,\text{end} \mid t \\
\textsc{Pattern} & p & ::= & x_i \mid \_ \mid C\ p \mid (p, \ldots, p) \mid (f = p, \ldots, f = p) \\
\textsc{Type} & \tau & ::= & b \mid \tau \to \tau \mid (\tau, \ldots, \tau) \\
\textsc{Term decl} & r_t & ::= & \texttt{val}\ x : \tau \mid \texttt{val}\ x : \tau = t \\
\textsc{Type decl} & r_\tau & ::= & \texttt{type}\ b \mid \texttt{type}\ b = \text{"|"}\ C\ \tau \ldots \text{"|"}\ C\ \tau \mid \texttt{type}\ b := \tau \mid \texttt{type}\ b = (f : \tau, \ldots, f : \tau)
\end{array}
$$

Figure 1: The syntax of Skel (without polymorphism)

```
val eval_stmt (s:state) (t:stmt): state =
branch
    let While (cond, t') = t in
    let Bool b = eval_expr s cond in
    branch
        let True = b in
        let s' = eval_stmt s t' in
        eval_stmt s' t
    or
        let False = b in
        s
    end
or ... end
```

The first line ensures that the evaluated statement is a `While` loop, and it extracts its components, using a destructuring pattern matching. The second line evaluates the expression and makes sure it is a boolean. Depending on the value of the boolean (first line of each inner branch), either the loop body is evaluated before evaluating the whole loop again, or the current state is returned.

2 Skeletal semantics

In this Section, we introduce the formalism, typing, and semantics of Skel.

2.1 Formalism

Figure 1 contains the syntax of Skel terms and skeletons. This BNF does not include polymorphism for simplicity, it will be described in Section 2.4.

As we can see, there are two types of expressions, namely terms and skeletons. This is close to an Abstract Normal Forms [**?**]. The point is to separate what is intuitively an evaluated value to what is a computation. This way, the order of evaluation is always trivial as there is only one way to do an evaluation step in a skeleton.

A term is either a variable, a constructor applied to a term, a (possibly empty) tuple of terms, a λ-abstraction, the access to a given field of a term, a record of terms, or a term with reassignment of some fields.

A skeleton is either the application of a term to other terms, a let-binding, an existential (see Section 2.3), a branching, or simply the return of a term. We sometimes write ret t instead of t, to clearly state that we consider the skeleton and not the term.

A pattern is either a variable, a wildcard, a constructor applied to a pattern, a (possibly empty) tuple of patterns, or a record pattern.

Finally, a type is either a base type (defined by the user), an arrow type, or a (possibly empty) tuple of types. Term and type declarations have already been described in Section 1.

2.2 Typing

Skel is a strongly typed language. As we can see in Section 2.1, there are a lot of type annotations. Every term declaration is given with a type, as well as every λ-abstraction. The reason is that, by design, Skel is meant to be explicitly typed. Polymorphism is ignored in this Section, but it is presented in Section 2.4, and we will see that the type arguments are also explicitly specified. Specifying every type might seem tedious at first, but it helps to improve confidence on the correction of the skeletal semantics. However, a future version of the typer may include a type-inference mechanism, for those adventurous or confident enough, as explained in Section 2.4

To be able to give the typing rules for Skel, we must first define ctype(C) which returns the pair of the declared input type and output type for the constructor C in the type declaration list. For instance, if we define the type `expr` as shown in Section 1, we would have ctype(`Const`) = `(literal, expr)`.

In the same way, we define ftype(f) which returns (τ, ν) where τ is the type of the field f, and ν is the record type to which the field f belongs. For instance, if we define the type `euler_int` as shown in Section 1, we would have ftype(`re`) = `(int, euler_int)`.

A field name cannot belong to two different record types, and a constructor name cannot be used twice, so the ftype and ctype function are well-defined.

The typing rules for terms and skeletons are given in Appendix A. They are respectively of the form $\Gamma \vdash_t t : \tau$ and $\Gamma \vdash_S S : \tau$, where Γ is a typing environment (a partial functions from variable names to types).

They are pretty straightforward, and very similar to what exists in other functional languages. To type a λ-abstraction and a let-binding, we use a pattern-typing defined thereafter, using the partial function $\Gamma + p \mapsto \tau$:

$$\Gamma + x \mapsto \tau \triangleq \Gamma, x : \tau \qquad \Gamma + \_ \mapsto \tau \triangleq \Gamma$$

$$\Gamma + C\ p \mapsto \nu \triangleq \Gamma + p \mapsto \tau, \text{ where } \text{ctype}(C) = (\tau, \nu)$$

$$\Gamma + (p_1, \ldots, p_n) \mapsto (\tau_1, \ldots, \tau_n) \triangleq (\Gamma + p_1 \mapsto \tau_1) \ldots + p_n \mapsto \tau_n$$

$$\Gamma + (f_1 = p_1, \ldots, f_n = p_n) \mapsto \nu \triangleq (\Gamma + p_1 \mapsto \tau_1) \ldots + p_n \mapsto \tau_n, \quad \text{where fields}(\nu) = \{f_1 : \tau_1, \ldots, f_n : \tau_n\}$$

An empty branch can therefore be typed to any given type, but since Skel is strongly typed, it will reject any empty branching whose type is not explicitly specified.

2.3 Concrete interpretation

As such, Skel is a concrete syntax to describe a programming language. The *meaning* associated to a Skel description is called an *interpretation*. We present in this section a concrete interpretation, that corresponds to the usual natural semantics of a language [**?**]. One may also define an abstract interpretation, where unspecified types are given values in some abstract domain and

where the results from branches are all collected. Defining such an interpretation is beyond the scope of this paper

Every skeleton and term is interpreted as a value, so we need to define the set in which we choose those values.

First, for every unspecified type b, we assume a set V_b. For every variant type b', the set $V_{b'}$ is the set freely generated by the constructors with their types. For tuple types, we use the cartesian product: the set $V_{(\tau_1,\dots,\tau_n)}$ is $V_{\tau_1} \times \cdots \times V_{\tau_n}$. For a record type, the set is a family of values. For instance, if b'' has fields $\{f_1 : \tau_1, \dots, f_n : \tau_n\}$, we have

$$V_{b''} = \{(f_1 = v_1, \dots, f_n = v_n) \mid v_i \in V_{\tau_i}\}.$$

For arrow types, we use relations, as the same skeleton can be evaluated to zero, one, or several values, because of branchings. So $V_{\tau\to\nu} = \mathcal{R}(V_\tau, V_\nu) = \mathcal{P}(V_\tau \times V_\nu)$.

Next, we need to give a meaning to every unspecified term. So for each declaration `val x : τ`, we assume given a set of value $[\![x]\!] \subseteq V_\tau$.

The rules for the concrete interpretation are given in Appendix B. They are of the form $E, t \Downarrow_t v$ for terms and $E, S \Downarrow_S v$ for skeletons, meaning that in the environment E (partial function matching variables to values), the term t (resp. the skeleton S) can evaluate to a value v.

In the App rule, the R^* notation is used to represent the uncurried version of the relation. We define:

$$R^*(v_1, \dots, v_n, w) \triangleq \exists R_2, \dots, R_n, R(v_1, R_2) \wedge R_2(v_2, R_3) \cdots \wedge R_n(v_n, w)$$

A construct that is specific to Skel is the existential. To interpret the existential `let p:τ in sk`, we take any value of type τ, and match `p` to this value, before evaluating the continuation in the extended environment.

In the rules for the existential, as well as in the rules for let-in and closures, we use the partial function $E + p \mapsto v$. It is defined as follows.

$$E + x \mapsto v \triangleq E, x : v \qquad E + \_ \mapsto v \triangleq E \qquad E + C\ p \mapsto C\ v \triangleq E + p \mapsto v$$
$$E + (p_1, \dots, p_n) \mapsto (v_1, \dots, v_n) \triangleq (E + p_1 \mapsto v_1) \dots + p_n \mapsto v_n$$
$$E + (f_1 = p_1, \dots, f_n = p_n) \mapsto (f_1 = v_1, \dots, f_n = v_n) \triangleq (E + p_1 \mapsto v_1) \dots + p_n \mapsto v_n$$

As we can see, the concrete interpretation is relational. A skeleton or a term can be interpreted to 0, 1, or several values. The sources of partiality and non-determinism are threefold.

- A branching with no branch has no result, and can cause partiality; a branching with several branches can have several results and cause non-determinism.
- Unspecified terms can have any arbitrary number of interpretation, so they can cause both partiality and non-determinism.
- Pattern-matching can fail, and is therefore a source of partiality.

2.4 Polymorphism and type inference

For practical reasons, including the definition of monadic binders in Section 3, the type system also provides polymorphism. All type annotations are explicit. They are specified using angle brackets. Unspecified types, variant types, record types, and aliases can all be polymorphic.

```
(* Unspecified *)
type list<_>

(* Variant *)
type union<a,b> =
| InjL a
| InjR b
```

```
(* Record *)
type pair<a,b> = (left: a, right: b)

(* Alias *)
type option<a> := union<a,()>
```

Terms defined at toplevel can also be polymorphic, but let-bound terms are necessarily monomorphic.

```
val map<a,b> (f: a → b) (l: list<a>): list<b> =
    branch
        let Nil = l in
        Nil<b>
    or
        let Cons (a, qa) = l in
        let b = f a in
        let qb = map<a,b> f qa in
        Cons<b> (b, qb)
    end
```

Type annotations are explicitly given when constructing a term (e.g `Nil<b>`) or when using a polymorphic term (e.g., `map<a,b> f qa`), but they can be locally inferred in patterns, so they are not specified in them.

The explicit typing is by design, considering that explicit type annotations reduce the risk of error. In the future, we will possibly add an option that would allow the typer to perform type inference. This option would possibly also infer the type of the arguments in `λp:τ → sk` constructs.

3 Monads in Skel

3.1 Custom binders

Some executable semantics specifications, such as ECMA-262 [**?**], make extensive but implicit use of monads, for instance to carry state, propagate exceptions, or suspend computations. Skel has a built-in system to handle monadic binds, making it easier to use them, and making the written Skel code easily readable and close to the specification [**?**].

Assume that a term `bind<a,b>` is defined, with the type `m<a> → (a → m<b>) → m<b>` where `m<·>` is a monad, for instance the state monad. Then, one can use the following notation:

```
let p =%bind s1 in s2
```

In this case, the type arguments for bind are inferred (they can also be explicitly given if desired), and the result is semantically the same as `bind s1 (λp → s2)` (with all the proper type annotations added).

For instance, we can rewrite the semantics of IMP within a state monad, which makes the code more easily readable, and avoids mistakes in state manipulations. Here is the monad:

```
type st<a> := state → (a, state)
val bind<a, b> (a:st<a>) (f:a → st<b>): st<b> =
```

```
    λ s : state → let (a, s) = a s in f a s
val ret<a> (a:a): st<a> = λ s : state → (a, s)
```

Using this monad, we can rewrite the evaluation of a while loop presented in Section 1 as follows.

```
val eval_stmt (t:stmt): st<()> =
branch
    let While (cond, t') = t in
    let Bool b =%bind eval_expr cond in
    branch
        let True = b in
        eval_stmt t' ;%bind
        eval_stmt t
    or
        let False = b in
        ret<()> ()
    end
or ... end
```

Note that a semicolon construct `s1; s2` is syntactic sugar for `let _ = s1 in s2`. Monadic notation also applies to this sequencing operator.

The type of `bind` does not have to be the type of a monadic bind to be used as notation, it can be any type as long as the second argument is an arrow type, and it is interpreted in the exact same way. This may be necessary when handling several monads at the same time, or when a bind-like operator is not monadic.

3.2 Necro Trans

Necro Lib (cf. Section 4.1) provides a tool, called Necro Trans, which performs transformations on skeletal semantics. One of the provided transformers inlines custom binders. This allows to remove all monads out of a semantics. Along with the `explode` transformer, which duplicates code so as to push all branches to the same level, an option monad can be inlined and still provide readable code. In the same way, using the `eta` transformer, which performs η-expansion and then some β-reduction, we can inline a state monad and still get something readable.

For instance, here is a part of the semantics of IMP, using state monads:

```
val eval_expr (e:expr): st<value> =
    branch
        let Plus (e1, e2) = e in
        let f1 =%bind eval_expr e1 in
        let Int v1 = f1 in
        let f2 =%bind eval_expr e2 in
        let Int v2 = f2 in
        let v = add v1 v2 in
        let r = Int v in
        ret<value> r
    or ... end
```

After `necrotrans inlinemonads`, `necrotrans eta` and `necrotrans inline ret`, we get the following code:

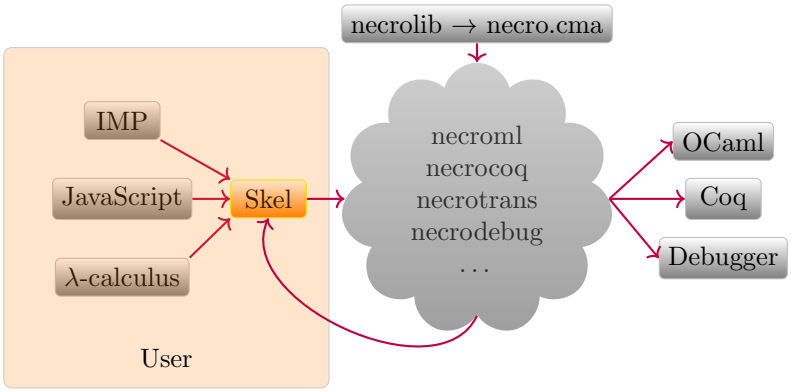

Figure 2: The Necro ecosystem

```
val eval_expr (e:expr): st<value> =
    λ state : state →
    branch
        let Plus (e1, e2) = e in
        let a' = eval_expr e1 in
        let (a, s) = a' state in
        let Int v1 = a in
        let a' = eval_expr e2 in
        let (a', s) = a' s in
        let Int v2 = a' in
        let v = add v1 v2 in
        let r = Int v in
        (r, s)
    or ... end
```

which is close to what would have been written by hand and is easily understandable. This is useful for instance when debugging a semantics, using necrodebug (see Section 4.4).

4 The Necro Ecosystem

Necro is an ecosystem with several tools to perform different operations on skeletal semantics, as illustrated in Figure 2.

4.1 Necro Lib

Necro Lib [**?**] is Necro's centerpiece. It generates an OCaml library file, `necro.cma`. This library provides a parser and a typer, in order to transform a Skel file into a Skel AST. The AST is described in the file `main/skeltypes.mli` of the repository. It also contains a pretty-printer, which displays the AST in the format of a Skel file; a set of transformers, along with the tool Necro Trans (Section 3.2), which calls the transformers on an AST and prints the result; and a set of utility functions to manipulate the AST.

The `necro.cma` file is the basis for the tools Necro ML (Section 4.2), Necro Coq (Section 4.3), and Necro Debug (Section 4.4).

4.2 Necro ML

Necro ML [**?**] is a generator of OCaml interpreters. Given a skeletal semantics, it produces an OCaml functor. This functor expects as arguments a specification (written in OCaml) of all unspecified types and terms, it then provides an interpreter that can compute any given skeleton.

Skel cannot be shallowly embedded in OCaml, since OCaml does not have an operator corresponding to the branching construct (pattern matching is deterministic in OCaml). So types and terms are shallowly embedded, but the skeletons are deeply embedded. We use an interpretation monad that specifies, among other things, how skeletons are represented, and how `let-in`s, applications and `branch`es are computed.

4.2.1 OCaml Interpreter

When Necro ML is executed on a skeletal semantics, it generates an OCaml file, which contains several modules and module types.

- The `TYPES` module type contains all unspecified types. The user must produce a module for them.
- The `MONAD` module type is the module type for the interpretation monad mentioned above and detailed in Section 4.2.2. It states how let-ins, branches, and applications are computed.
- The `UNSPEC` module type contains all types, specified and unspecified, the signature of unspecified terms, and the interpretation monad. The latter is present since the choice of monad can have an effect on the definition of the unspecified terms.
- The `Unspec` functor takes the instantiation of the unspecified types and the chosen monad, and it produces a default module of type `UNSPEC` containing all type information and a default implementation of unspecified terms. This implementation simply raises a NotImplemented error. It is not meant to be used as is, but to be overridden with the actual implementation of the unspecified terms.
- The `INTERPRETER` module type is the module type for the interpreter of the described language. It extends the `UNSPEC` type with the signature for specified terms.
- The `MakeInterpreter` functor takes the instantiation of unspecified types and unspecified terms, typically derived from the application of the `Unspec` functor, and it produces an interpreter for the language.

In summation, to produce an interpreter, the user only has to do the following:

- write a module of type `TYPES`, to implement the unspecified types;
- choose an interpretation monad of type `MONAD` or write their own; the proposed interpretation monads are in file `necromonads.ml` of Necro ML's repository;
- apply the `Unspec` functor then override all unspecified terms with their actual implementation;
- apply the `MakeInterpreter` functor to generate the code for the specified terms.

Working examples can be found in the `test` folder of Necro ML's repository.

4.2.2 Interpretation monad

The interpretation monad is defined in the following way, where terms are assumed to be pure while skeletons are monadic (of type `'a t`).

```
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
```

There is some similarity between `bind` and `apply`. The difference is that `bind` is used to translate `let-in` constructs, therefore it expects a skeleton of type `'a t`, and apply is used to translate application to a term, and therefore expects a value of type `'a` (remember that skeletons are deeply embedded, while terms are only shallowly embedded).

The `fail` operator takes a string as input which is an error message that should be raised, and the `extract` operator is a usability construct to extract a result from the monad, typically to display it.

There are several proposed ways to instantiate this monad, and the user can also define their own.

The standard identity monad, which is closest to a shallow embedding, tries each branch in turn. It is defined as follows.

```
module ID = struct
    exception Branch_fail of string
    type 'a t = 'a
    let ret x = x
    let rec branch l =
        begin match l with
        | [] -> raise (Branch_fail "No branch matches")
        | b1 :: bq -> try b1 () with Branch_fail _ -> branch bq
        end
    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 monad is the easiest to use, and it works perfectly fine in most cases. However, the issue with this monad is that we always take the first branch that succeeds, but if we find out later that it was not the right branch, there is no way to get back to another branch. For instance, for the following code, calling `fail ()` will raise the `Branch_fail` exception even though `()` is a valid interpretation.

```
val fail (x:()): () =
    let f =
        branch
```

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

As we mentioned, the main problem is the inability to backtrack. To fix this, we propose a continuation monad, which uses failure continuations to keep track of all branches that were not taken, therefore giving the possibility to backtrack if a branch fails [**?**].

```
module ContPoly = struct
    type 'b fcont = string -> 'b
    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
```

This time, the execution of the same example will choose the first branch, but it keeps in its failure continuation the content of the second branch. Then, when executing `f x`, it will fail, and unstack its failure continuation. It will then backtrack to the second branch, and succeed with the result `()`.

Nevertheless, the continuation monad is not complete in relation to the concrete interpretation, as it always begins by executing the first branch, and can therefore be caught in an infinite loop. The following function is an example thereof.

```
val loop (_:()): () =
    branch loop () or () end
```

To fix this issue, we propose another interpretation monad called BFS, which does one step in each branch, until it gets a result. The file containing all the proposed monads is available online[2]

4.3 Necro Coq

Necro Coq [**?**] is a tool that allows to embed a given skeletal semantics into a Coq formalization. It can then be used to prove correction of a given program or to prove language properties.

[2] `https://gitlab.inria.fr/skeletons/necro-ml/-/blob/FSCD2022/necromonads.ml`

4.3.1 Structure

Necro Coq uses a deep embedding of Skel. The embedding of Skel is defined in the file `files/Skeleton.v`, which is presented in Section 4.3.2. The command `necrocoq file.sk` provides a Coq file that contains the AST of the original skeletal semantics. To give meaning to the skeletons, we then provide a file that defines the concrete interpretation for skeletons and terms, presented in Section 4.3.4

4.3.2 Skel's embedding

The embedding is a straightforward deep embedding. It defines a number of variables, which are the data of a given skeletal semantics (its constructors, base types, ...), and provides with the basic constructs (the definition of a type, a skeleton, ...).

For instance, here is the mutual definition of terms and skeletons:

```
Inductive term: Type :=
| term_constructor : string -> list type -> term -> term
| term_var: typed_var -> term
| term_tuple: list term -> term
| term_func: pattern -> skeleton -> term
| term_field: term -> list type -> string -> term
| term_nth: term -> list type -> nat -> term
| term_record: option term -> list (string * list type * term) -> term
with skeleton: Type :=
| skel_branch : type -> list skeleton -> skeleton
| skel_return : term -> skeleton
| skel_apply : term -> list term -> skeleton
| skel_letin : pattern -> skeleton -> skeleton -> skeleton
| skel_exists : pattern -> type -> skeleton -> skeleton.
```

4.3.3 Values

The values used to store the result of the evaluation of a term or skeleton are deeply embedded as well. The Coq type which supports values is defined with five constructors. Here are the first four.

```
Inductive cvalue : Type :=
| cval_base : forall A, A -> cvalue
| cval_constructor : constr -> cvalue -> cvalue
| cval_tuple: list cvalue -> cvalue
| cval_closure: pattern -> skeleton -> list (string * cvalue) -> cvalue.
```

The `cval_base` constructor allows values of an unspecified type to be represented in Coq with any given type `A`. For instance, the value 1, in an unspecified type `int`, could be stored as `cval_base Z 1` (here, 1 is in the scope of Z, which means it is equal to `Zpos xH`)

The next two constructors are straightforward. The fourth one is a closure constructor, to store lambda-abstractions. The three arguments of the constructor are the bound pattern, the skeleton to evaluate, and the current environment.

The fifth constructor is used for unspecified functional terms, it will be presented in Section 4.3.5.

4.3.4 Interpretation

The file Concrete.v provides the concrete interpretation for skeletons. It uses Coq's induction to define the relations `interp_skel` and `interp_term`, which relates skeletons (respectively terms) in a given environment to their possible interpretations as a value. It mostly follows the semantics of Appendix B. For instance, this is the rule for a `let-in` construct:

```
Inductive interp_skel: env -> skeleton -> cvalue -> Prop :=
| i_letin: forall e e' p s1 s2 v w,
        interp_skel e s1 v ->
        add_asn e p v e' ->
        interp_skel e' s2 w ->
        interp_skel e (skel_letin p s1 s2) w
```

The `add_asn e p v e'` proposition states that the environment e can be extended into e' by mapping p to v, that is $e' = e + p \mapsto v$

The file `Concrete.v` provides the interpretation using big-step evaluation, but we also provide a file `Concrete_ss.v` which does a small-step evaluation.

Another alternative is `ConcreteRec.v`, which defines interpretation from the bottom up. That is, instead of using Coq's induction, it only defines how to do one step, which doesn't use recursive calls, and then we may iterate this step. It is closest to the initial definition in [**?**]. The purpose of this file is to be able to perform a strong induction on `interp_skel` in a very easy fashion.

These interpretations are proven (in Coq) to be equivalent, so one can use indifferently one or the other, and one may even switch between several of them depending on what is more useful at the given moment.

The big step definition is usually the easiest to use. The small step one allows to reason about non-terminating behaviors, and it provides a simple way to prove the subject reduction of Skel (see below). The iterative one allows, as we mentioned, to perform a strong induction. An instance where we may want to use this one is, given the skeletal semantics of lambda-calculus, if we want to derive the induction property for lambda-terms. This property is the following:

$$(\forall x \in X, P_x) \to (\forall MN \in \Lambda, P_M \to P_N \to P_{M\ N}) \to \\ (\forall x \in X, \forall M \in \Lambda, P_M \to P_{\lambda x \cdot M}) \to \forall M \in \Lambda, P_M.$$

Since Skel is deeply embedded, one evaluation step in the language matches several steps in Necro Coq. Because of this, the inductive interpretation is not convenient to prove that property, whereas this is much simpler using the iterative version with a strong induction on the height of the derivation tree.

As Skel is strongly typed, we also have a file `WellFormed.v` to check that a term or skeleton is well-formed, i.e., that it can be typed. We prove that the concrete interpretation respects the subject reduction property with regards to well-formedness. Since interpretations are all shown equivalent, it is sufficient to prove it for `Concrete_ss.v`:

```
Theorem subject_reduction_skel:
    forall sk sk' ty,
        type_ext_skel sk ty ->
        interp_skel_ss sk sk' ->
        type_ext_skel sk' ty.
```

This translates roughly to:

$$\frac{S : \tau \qquad S \to S'}{S' : \tau}$$

where S and S' are extended skeletons.

Extended skeletons can be actual skeletons, in which case the `type_ext_skel` property is equivalent to `well_formed_skel` defined in the file `WellFormed.v`. They can also be values,

signaling that the computation is over, in which case the `type_ext_skel` property asserts that the value is in V_τ

With this theorem proven, and since `Concrete_ss.v` and `Concrete.v` are equivalent, we have the following:

$$\frac{\emptyset \vdash S : \tau \qquad S_S \Downarrow v}{v \in V_\tau}$$

4.3.5 Unspecified functional values

There is one other issue that needs to be addressed. Since Skel allows to declare unspecified terms, we must be able to interpret them. The natural idea would be to ask for a `cvalue` for each given unspecified term. But for unspecified functions (like the addition), that would mean giving a closure, which is equivalent to specifying the function. We would lose Skel's power of partial specification.

Instead, we ask for a relation which, given a list of cvalues as arguments, provides the result of the application of the term to the arguments. For instance, for the addition, it would be $\{([x; y], x + y) \mid x, y \in \mathbb{N}\}$.

There are operational and denotational approaches to represent this in Coq. The file `Concrete.v` (as well as its variants `Concrete_ss.v` and `ConcreteRec.v`) provides an implementation for the operational approach.

In these files, there is a constructor for unspecified functional terms that are not fully applied yet:

```
| cval_unspec: nat -> unspec_value -> list type -> list cvalue -> cvalue.
```

The first nat argument being `n` means that there are `S n` arguments missing. We add one, because there cannot be 0 argument missing, since if there is 0 argument missing, it is not a partial application. The list of types is the type annotation for polymorphic unspecified terms. the list of values is the list of arguments that have already been provided.

The file `Concrete2.v` (as well as `Concrete2_ss.v` and `Concrete2Rec.v`) provides an implementation for the denotational approach.

In these files, unspecified functions are defined using the following extensive constructor:

```
| cval_func: forall A, (A -> cvalue * cvalue) -> cvalue
```

This is basically the same as using a relation, but in a declarative way instead of predicative, to agree with Coq's positivity requirements.

As we have not identified that one approach is better than the other, we provide both and let the user choose the most convenient for the task at hand.

4.3.6 Applications and usability

We have considered several applications of Necro Coq, some of them can be found in the `test` folder of the repository. For instance, we have proved the correction of an IMP code that computes the factorial function (`test/certif` folder). Necro Coq has also been used to prove the equivalence between a small-step and a big-step semantics for a lambda-calculus extended with natural numbers (`test/lambda` folder).

In addition, it has been used in [**?**] to provide the a posteriori proof of an automatic generic transformation of a big step semantics into a small step semantics.

4.4 Necro Debug

In the process of writing a semantics, it is usual and natural to make mistakes. To track them, any good language should provide a way to debug a code written using it. This is what Necro Debug offers. It is a relatively recent tool, and is therefore still in its infancy, but it is already usable.

Necro Debug [**?**] is a debugger generator. Given the skeletal semantics of a language and the AST of a program written in this language, it provides a step by step execution, using an abstract machine interpretation of Skel. We provide two user interfaces. One is in a terminal, and the second one is an HTML page, using `js_of_ocaml`.

To use it, it is required to provide the interpretation for unspecified types and terms, in the same way that they would be provided in Necro ML with the ID monad. The user can also provide a printing function for unspecified types, or use the default value, which prints for instance `<int>` for every value of type `int`. The user can therefore use the same instantiation for the debugger and necroml, and only needs to extend it with printing functions in the debugger.

5 Related Work

We review existing approaches that are generic, in the sense that they can be used to describe and manipulate any semantics.

Our work is an extension of the work undertaken in [**?**]. We significantly improve on their approach by having a more expressive language (with higher-order functions and polymorphism) and a set of tools to manipulate skeletal semantics. The Coq formalizations of [**?**] were written by hand, they can now be automatically generated. Generation of OCaml code was also proposed in [**?**], but it was only available for the language of [**?**], which was less powerful than the current language.

On the process of formalizing semantics, several tools already exist, but they are heavier and more complex than Skel. This is the case of Lem [**?**] and Ott [**?**]. The lightness of Skel (the file `skeltypes.mli` describing Skel's AST is only 108 lines of specification) allows anyone to easily write a tool handling skeletal semantics. This is more complex with Lem and Ott, as they provide many additional features. For instance, Lem natively defines set comprehension, relations, and maps. Also, Coq generation is done as a shallow embedding, hence functions must be proven to terminate. In addition, shallow embedding of large semantics are not easily manipulated in Coq, due to the space complexity of the `induction` and `inversion` tactics.

The K framework [**?**] also allows to formally define the semantics of a language and prove programs, and it is designed to be easy to use. It does not allow, however, to prove meta-theory properties of a language.[3] Furthermore, there are no Coq backend for K at the present time. And since K is a large language, writing new backends is far from trivial.

Finally, another common way to describe a semantics is to implement it both in OCaml and in Coq, or other similar tools, either directly or through tools to transform them (Coq extraction to OCaml or `coq-of-ocaml` [**?**] to go the other way). One may then execute the semantics using the OCaml version and ++++e properties using the Coq one. In this case, the Coq formalization is simpler to manipulate, but changing design choices (such as going from a shallow to a deep embedding) is very costly.

[3] `https://sympa.inria.fr/sympa/arc/coq-club/2020-02/msg00066.html`

6 Conclusion

The language Skel offers a way to specify semantics of programming languages, using a language light enough to be easily readable and maintainable, yet powerful enough to express many semantical feature. The tools Necro provide, especially Necro ML and Necro Debug, help in the process of writing a semantics. Necro Coq and Necro Trans allow to manipulate and certify these semantics once written. They give the necessary framework to prove program correction and language properties.

Skel has been used to write the semantics of a set of basic languages such as IMP, but it has also been used to formalize more massive languages, such as WASM (unpublished), and an ongoing formalization of JavaScript in the JSkel project [**?**].

Though we did not mention it in this paper, Skel allows to define semantics in several files, but in this case, Necro Coq is not yet able to generate a Coq formalization. A future task is to implement this functionality in Necro Coq. Once this has been done, the logical next step is to implement a standard library for Skel, defining basic types like lists, with properties on these types proven using Necro Coq.

Another tool that is under implementation is Necro Tex, a generator of formatted TEX rules corresponding to a skeletal semantics.

A Typing rules for Skeletal Semantics

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_t x : \tau}\ \textsc{Var} \qquad \frac{\texttt{val x:}\tau\ \texttt{(= t)}}{\Gamma \vdash_t x : \tau}\ \textsc{Term} \qquad \frac{\Gamma \vdash_t t : \tau \qquad \text{ctype(C)} = (\tau, \tau')}{\Gamma \vdash_t \text{C}\, t : \tau'}\ \textsc{Const}$$

$$\frac{\forall i, \Gamma \vdash_t t_i : \tau_i}{\Gamma \vdash_t (t_1, \ldots, t_n) : (\tau_1, \ldots, \tau_n)}\ \textsc{Tuple} \qquad \frac{\Gamma + p \mapsto \tau \vdash_S S : \tau'}{\Gamma \vdash_t (\lambda p : \tau \to S) : \tau \to \tau'}\ \textsc{Clos}$$

$$\frac{\Gamma \vdash_t t : \nu \qquad \text{ftype}(f_i) = (\tau, \nu)}{\Gamma \vdash_t t.f_i : \tau}\ \textsc{FieldGet} \qquad \frac{\Gamma \vdash_t t : (\tau_1, \ldots, \tau_m) \qquad 1 \leq i \leq m}{\Gamma \vdash_t t.i : \tau_i}\ \textsc{TupleGet}$$

$$\frac{\text{fields}(\tau) = \{f_1 : \tau_1, \ldots, f_n : \tau_n\} \qquad \forall i, \Gamma \vdash_t t_i : \tau_i}{\Gamma \vdash_t (f_1 = t_1, \ldots, f_n = t_n) : \tau}\ \textsc{Rec}$$

$$\frac{\Gamma \vdash_t t : \tau \qquad \forall i, \Gamma \vdash_t t_i : \tau_i \qquad \forall i, \text{ftype}\, f_i = (\tau_i, \tau)}{\Gamma \vdash_t t \leftarrow (f_1 = t_1, \ldots, f_m = t_m) : \tau}\ \textsc{FieldSet} \qquad \frac{\Gamma \vdash_t t : \tau}{\Gamma \vdash_S \text{ret}\, t : \tau}\ \textsc{Ret}$$

$$\frac{\Gamma \vdash_S S_1 : \tau \qquad \ldots \qquad \Gamma \vdash_S S_n : \tau}{\Gamma \vdash_S (S_1 \ldots S_n) : \tau}\ \textsc{Branch} \qquad \frac{\Gamma \vdash_S S : \tau \qquad \Gamma + p \mapsto \tau \vdash_S S' : \tau'}{\Gamma \vdash_S \text{let}\, p = S \,\text{in}\, S' : \tau'}\ \textsc{LetIn}$$

$$\frac{\Gamma + p \mapsto \tau \vdash_S S : \tau'}{\Gamma \vdash_S \text{let}\, p : \tau \,\text{in}\, S : \tau'}\ \textsc{Exist} \qquad \frac{\forall i\ \Gamma \vdash_t t_i : \tau_i \qquad \Gamma \vdash t_0 : \tau_1 \to \cdots \to \tau_n \to \tau}{\Gamma \vdash_S (t_0\ t_1 \ldots t_n) : \tau}\ \textsc{App}$$

B Concrete Interpretation of Skeletal Semantics

$$\frac{E(x) = v}{E, x \Downarrow_t v}\ \textsc{Var} \qquad \frac{\texttt{val x:}\tau\ \texttt{=}\ \texttt{t} \qquad \emptyset, t \Downarrow_t v}{E, x \Downarrow_t v}\ \textsc{TermSpec}$$

$$\frac{\texttt{val x :}\ \tau \qquad v \in [\![x]\!]}{E, x \Downarrow_t v}\ \textsc{TermUnspec} \qquad \frac{E, t \Downarrow_t v}{E, (\mathrm{C}\, t) \Downarrow_t \mathrm{C}\, v}\ \textsc{Const}$$

$$\frac{E, t_1 \Downarrow_t v_1 \qquad \dots \qquad E, t_n \Downarrow_t v_n}{E, (t_1, \dots, t_n) \Downarrow_t (v_1, \dots, v_n)}\ \textsc{Tuple}$$

$$\frac{R \subseteq \{(v, w) \mid v \in V_\tau \wedge (E + p \mapsto v), S \Downarrow_S w\}}{E, (\lambda p : \tau \to S) \Downarrow_t R}\ \textsc{Clos}$$

$$\frac{E, t \Downarrow_t (f_1 = v_1, \dots, f_n = v_n)}{E, t.f_i \Downarrow_t v_i}\ \textsc{FieldGet} \qquad \frac{E, t \Downarrow_t (v_1, \dots, v_m) \qquad 1 \le i \le m}{E, t.i \Downarrow_t v_i}\ \textsc{TupleGet}$$

$$\frac{\forall i, E, t_i \Downarrow_t v_i}{E, (f_1 = t_1, \dots, f_n = t_n) \Downarrow_t (f_1 = v_i, \dots, f_n = v_n)}\ \textsc{Rec}$$

$$\frac{E, t \Downarrow_t (f_1 = v_1, \dots, f_n = v_n) \qquad \forall i, E, t_i \Downarrow_t w_{j_i} \qquad i \notin \{j_1, \dots, j_m\} \to w_i = v_i}{E, t \leftarrow (f_{j_1} = t_1, \dots, f_{j_m} = t_m) \Downarrow_t (f_1 = w_1, \dots, f_n = w_n)}\ \textsc{FieldSet}$$

$$\frac{E, t \Downarrow_t v}{E, \mathrm{ret}\, t \Downarrow_S v}\ \textsc{Ret}$$

$$\frac{E, S_i \Downarrow_S v}{E, (S_1 \dots S_n) \Downarrow_S v}\ \textsc{Branch} \qquad \frac{E, S \Downarrow_S v \qquad E + p \mapsto v, S' \Downarrow_S w}{E, \mathrm{let}\, p = S\, \mathrm{in}\, S' \Downarrow_S w}\ \textsc{LetIn}$$

$$\frac{v \in V_\tau \qquad E + p \mapsto v, S \Downarrow_S w}{E, \mathrm{let}\, p : \tau\, \mathrm{in}\, S \Downarrow_S w}\ \textsc{Exist}$$

$$\frac{\forall i\ \ E, t_i \Downarrow_t v_i \qquad E, t_0 \Downarrow_t R \qquad R^*(v_1, \dots, v_n, w)}{E, (t_0\ t_1 \dots t_n) \Downarrow_S w}\ \textsc{App}$$

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399