Semantics in Skel and Necro

Louis Noizet[1], Alan Schmitt[2]

[1]Université de Rennes 1, Bretagne, France
[2]INRIA, France

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 languages.

1. Introduction

To formally prove properties of a programming language, or programs in that 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. First, Skel is designed to be light and its semantics simple. A light language facilitates maintainability and the development of tools. Second, Skel is powerful enough to express intricate semantical features, as proven by its use in an ongoing formalization of JavaScript's semantics [1]. Third, a semantics described in Skel can be close to a previously written formulation, be it as inference rules or as an algorithmic specification. Finally, Necro provides a comprehensive and extensible set of tools to manipulate these semantics. For instance, to translate it into an interpreter (Necro ML, see Section 3.2), or to give a formalization in the Coq proof assistant (Necro Coq, see Section 3.3).

Skel is a statically strongly typed language. First introduced in [2], we present its redesign with a syntax close to ML. We also introduce support for polymorphism and higher order, enabling the use of monads in specifications [1].

Skel can be seen as a specification language or as a way to define inductive rules. Both approaches are useful, respectively when writing a semantics whose formalization is a set of algorithms, (e.g. ECMA-262 [3]), or when writing a semantics defined with inference rules (e.g. λ-calculus). Skel can be used to describe arbitrary semantics, including ones with partiality and non-determinism.

Necro contains several tools to manipulate skeletal semantics (semantics written in Skel). First, Necro Lib offers basic operations (parsing, typing, printing, and simple transformations), in the form of a library to write programs that manipulate the AST describing a semantics. Second, Necro includes the tools Necro ML, Necro Coq, and Necro Debug, which use said

Proceedings of the 23rd Italian Conference on Theoretical Computer Science, Rome, Italy, September 7-9, 2022

library to generate respectively an OCaml interpreter, a Coq formalization, and a step-by-step debugger.

The contributions of this work are the Skel language, the accompanying tools, and many examples of semantics available online.[1]

The paper is organized as follows. Section 2 introduces Skel. Section 3 presents the Necro ecosystem. Section 4 compares our work to existing approaches. Section 5 concludes the paper.

2. Skeletal Semantics

2.1. Skel by Example

Let us describe how one specifies a semantics in Skel, using the small-step semantics of λ-calculus as example. A skeletal semantics is a list of declarations, either *type declarations* or *term declarations*, where each declaration can be *unspecified* or *specified*. When a type is unspecified, we only give its name. For instance, when writing the semantics of λ-calculus, we might not want to specify how identifiers for variables are internally represented, so we declare `type ident`.

A specified type may be a variant type (i.e., an algebraic data type, defined by providing the list of its constructors together with their input type), a type alias using the `:=` notation, or a record type (defined by listing its fields and their expected types). For instance, the type of λ-terms would be defined as follows.

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

In this example, `(ident, term)` is the product type with two components, the first one being of type `ident`, and the second of type `term`.

A record type is declared as `type pair = ( left: int , right: int )` and an alias as `type ident := string`. Note that types are all implicitly mutually recursive, so the order in which they are declared does not matter.

We now turn to term declarations. An unspecified term declaration is simply its name and type. To declare a specified term, we also give its definition. An example of a term we might want to let unspecified is substitution, so we would declare `val subst: ident → term → term → term`.

An example of a specified term is the following.

```
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 *)
```

[1] https://gitlab.inria.fr/skeletons/necro-test

$$
\begin{array}{rrcl}
\textsc{Term} & t & ::= & x \mid C\ t \mid (t, \ldots, t) \mid \lambda p : \tau \to S \mid t.f \mid t.i \mid (f = t, \ldots, f = t) \mid \\
 & & & t \leftarrow (f = t, \ldots, f = t) \\
\textsc{Skeleton} & S & ::= & t_0\ t_1 \ldots t_n \mid \textbf{let}\ p = S\ \textbf{in}\ S \mid \textbf{let}\ p : \tau\ \textbf{in}\ S \mid \\
 & & & \textbf{match}\ t\ \textbf{with}\ "\mid" p \to S \ldots "\mid" p \to S\ \textbf{end} \mid \\
 & & & \textbf{branch}\ S\ \textbf{or}\ \ldots\ \textbf{or}\ S\ \textbf{end} \mid \mathrm{ret}\, t \\
\textsc{Pattern} & p & ::= & x \mid \_ \mid C\ p \mid (p, \ldots, p) \mid (f = p, \ldots, f = p) \\
\textsc{Type} & \tau & ::= & b \mid \tau \to \tau \mid (\tau, \ldots, \tau) \\
\textsc{Term decl} & d_t & ::= & \textbf{val}\ x : \tau \mid \textbf{val}\ x : \tau = t \\
\textsc{Type decl} & d_\tau & ::= & \textbf{type}\ b \mid \textbf{type}\ b = "\mid" C\ \tau \ldots "\mid" C\ \tau \mid \textbf{type}\ b := \tau \mid \\
 & & & \textbf{type}\ b = (f : \tau, \ldots, f : \tau)
\end{array}
$$

Figure 1: Syntax of Skel (without Polymorphism)

```
        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 **branch** . . . **or** . . . **end** construct is a Skel primitive to deal with non-deterministic choice, similar to McCarthy's ambiguous operator [4]. There is no order in a branching, so any branch which yields a result can be chosen. Overlapping branches provide non-determinism, and non-exhaustive branches provide partiality. For instance the **branch end** at the end will never yield a result, so `ss` is partially defined.

The destructuring pattern matching **let Lam** `(x, body) = t1` **in** `...` asserts that `t1` is a lambda-abstraction. If it is not, then the considered branch yields no result. If it is, then `x` and `body` will be assigned to the proper values.

The **match** . . . **with** . . . **end** construct is a pattern matching, similar to any other language's pattern matching. In particular, it is deterministic. Only the first $(pattern, skeleton)$ pair for which the pattern corresponds with the matched value is taken.

2.2. Formalism

Figure 1 contains the syntax of Skel terms and skeletons. It does not include polymorphism, which will be described in Section 2.5. There are two types of expressions: terms and skeletons. Our syntax is close to Moggi's computational λ-calculus [5] and to Abstract Normal Forms [6]: we separate what is intuitively an evaluated value to what is a computation. This makes the evaluation order unambiguous and the manipulation of skeletal semantics simpler.

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, the access to a member of a tuple, a record of terms, or a term with reassignment of some fields. A skeleton is either the application of

$$\Gamma + x \leftarrow \tau \triangleq \Gamma, x : \tau \qquad \Gamma + \_ \leftarrow \tau \triangleq \Gamma$$
$$\Gamma + C\ p \leftarrow \nu \triangleq \Gamma + p \leftarrow \tau, \text{ where } \mathrm{ctype}(C) = (\tau, \nu)$$
$$\Gamma + (p_1, \ldots, p_n) \leftarrow (\tau_1, \ldots, \tau_n) \triangleq ((\Gamma + p_1 \leftarrow \tau_1) \ldots) + p_n \leftarrow \tau_n$$
$$\Gamma + (f_1 = p_1, \ldots, f_n = p_n) \leftarrow \nu \triangleq ((\Gamma + p_1 \leftarrow \tau_1) \ldots) + p_n \leftarrow \tau_n, \quad \text{where } \mathrm{fields}(\nu) = \{f_1 : \tau_1, \ldots, f_n : \tau_n\}$$

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

Figure 2: Pattern Matching of Types and Values

a term to other terms, a let-binding, an existential (see Section 2.4), a branching, a match, or simply the return of a term. We sometimes omit ret in $\mathrm{ret}\, t$. 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 2.1

2.3. Typing

Skel is a strongly typed language with explicit type annotations. Every term declaration is given a type, as well as every pattern in a λ-abstraction. Polymorphism, presented in Section 2.5, also uses explicitly specified type arguments. Specifying every type might seem tedious at first, but it helps to improve confidence on the correctness of the skeletal semantics. A future version of the typer will include an optional type-inference mechanism.

To give the typing rules for Skel, we first define $\mathrm{ctype}(C)$, which returns the pair of the declared input type and output type for the constructor C in its type declaration. For instance, we have $\mathrm{ctype}(\texttt{Var}) = \texttt{(ident, term)}$. Similarly, we define $\mathrm{ftype}(f)$ to return (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, we have $\mathrm{ftype}(\texttt{left}) = \texttt{(int, pair)}$. Note that a field name may not belong to two different record types, and a constructor name may not be used twice, so these functions are well-defined. Finally, we write $\mathrm{fields}(\nu)$ for the fields and types of record type ν.

The typing rules for terms and skeletons are straightforward, we give them 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 function from variable names to types). To deal with pattern matching in λ-abstractions and let-bindings, we use the partial function $\Gamma + p \leftarrow \tau$ defined at the top of Figure 2.

2.4. 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 interpreta-

tion, which stands for the usual natural semantics of a language [7]. 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. Such an interpretation is beyond the scope of this paper.

Every skeleton and term is interpreted as a value. Given sets of values V_τ for each unspecified type τ, we build values for variant and product types in the expected way. For arrow types, we use closures for the specified functions, and we delay the evaluation of unspecified terms until we have all arguments. To this end, for each declaration `val x : τ1 → ... → τn → τ`, we assume given an arity $n = \text{arity}(x) \in \mathbb{N}$ and a relation $[\![x]\!] \in \mathcal{P}(V_{\tau_1} \times \cdots \times V_{\tau_n} \times V_\tau)$.

The rules for the concrete interpretation are straighforward. They 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 mapping variables to values), the term t or the skeleton S can evaluate to a value v.

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. This pattern matching, also used for the let-in constructs and closures, is defined in Figure 2 as the partial function $E + p \leftarrow v$ that returns an extended environment.

The concrete interpretation is relational: a skeleton can be interpreted to 0, 1, or several values. A branching with no branch has no result, causing partiality. A branching with several branches can have several results, causing non-determinism. Pattern-matching can fail, causing partiality. Finally, non-terminating computations also cause partiality.

2.5. Polymorphism and Type Inference

Our type system allows for polymorphism, with explicit type annotations specified using angle brackets. Any declared and defined type can be polymorphic.

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

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

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

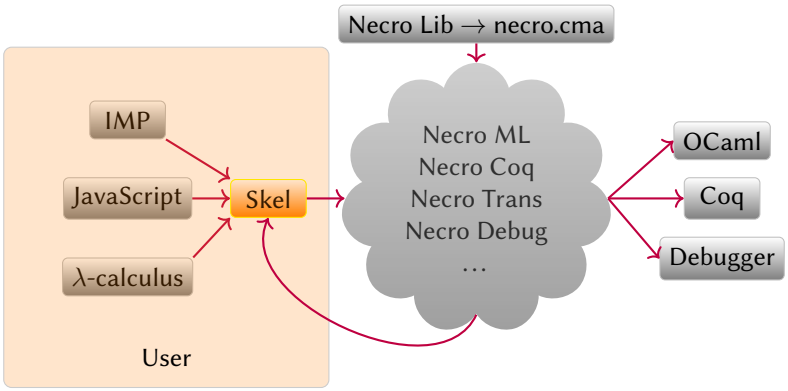

Figure 3: The Necro Ecosystem

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 (e.g., `let Nil = l in ...`).

The explicit typing is by design, as explicit type annotations reduce the risk of error. In the future, we will add an option to perform type inference. This option could also infer the type of the arguments in constructs of the form λ`p:`τ `→ sk`.

3. The Necro Ecosystem

Necro is an ecosystem with several tools to perform different operations on skeletal semantics, as illustrated in Figure 3.

3.1. Necro Lib

Necro Lib [8] is an OCaml library file, `necro.cma`. It 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 that 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 Necro ML (Section 3.2) and Necro Coq (Section 3.3).

3.2. Necro ML

Necro ML [9] is a generator of OCaml interpreters. Given a skeletal semantics, it produces an OCaml functor. This functor expects as arguments an OCaml specification 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 fitting the branching construct (pattern matching is deterministic in OCaml). So types and terms are shallowly embedded, but skeletons are deeply embedded. We use an interpretation monad, which

specifies, amongst other things, how skeletons are represented, and how `let in`s, applications and `branch`es are computed.

3.2.1. OCaml Interpreter

When Necro ML is executed on a skeletal semantics, it generates an OCaml file, which contains several modules, functors, and module types. To create an interpreter, one needs to instantiate them, in order to indicate how unspecified types and terms are interpreted, and which interpretation monad is chosen.

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

3.2.2. Interpretation Monad

The interpretation monad is defined as follows, 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
```

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. But it chooses the first branch that succeeds, with no ability to backtrack, which can be problematic in some cases. A more interesting one is the continuation monad, which keeps as a failure continuation the branches not taken, and can then backtrack if need be [10].

```
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
```

Nevertheless, the continuation monad is not complete w.r.t the concrete interpretation, as it always executes the first branch first, 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 these monads and some others is available online.[2] Note that to change the interpretation monad, the user simply has to swap it at module instantiation, no code needs to be rewritten.

3.3. Necro Coq

Necro Coq [11] is a tool to embed a given skeletal semantics into a Coq formalization. It can then be used to prove language properties or correctness of a given program.

3.3.1. 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 3.3.2. 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.

3.3.2. 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 the basic constructs (the definition of a type, a skeleton, ...).

3.3.3. Values

The values that are 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.
```

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

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`. The next two constructors are straightforward. The fourth one is a closure constructor, to store λ-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 below.

3.3.4. Interpretation

The file `Concrete.v`[3] provides the concrete interpretation for skeletons. It uses Coq's induction to define the relations `interp_skel` and `interp_term`, which relate respectively skeletons and 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 $e' = e + p \leftarrow v$

The file `Concrete.v` defines the interpretation using big-step evaluation, but we also provide a file `Concrete_ss.v` which does a small-step evaluation. An 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 one may iterate this step. It is closest to the initial definition in [2]. The purpose of this file is to be able to perform a strong induction on `interp_skel` in a simple way.

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 time. 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 need to use this one is to prove an induction property on the semantics of a lambda calculus.[4] As Skel is deeply embedded, one evaluation step in the language corresponds to 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 to be equivalent, it suffices to only prove it for `Concrete_ss.v`:

[3] https://gitlab.inria.fr/skeletons/necro-coq/-/blob/master/files/Concrete.v
[4] https://gitlab.inria.fr/skeletons/necro-coq/-/blob/master/test/lambda/EvalInd.v#L55

```
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.

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

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

Finally, interpretations in the form of abstract machines have been defined in [12]. They reuse the deep embedding provided by Necro Coq and are proven correct in relation to the concrete interpretation.

3.3.5. Unspecified Functional Values

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 that, 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. We choose the operation approach as it does not need to evaluate ahead of time: it just waits to be applied to enough arguments before computing. We thus use the following constructor, which denotes an unspecified functional term that is 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.

3.3.6. 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 correctness 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 λ-calculus extended with natural numbers (`test/lambda` folder). In addition, it has been used in [13] to provide the *a posteriori* proof of an automatic generic transformation of a big step semantics into a small step semantics.

4. 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 [2]. We significantly improve on this 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 [2] were written by hand, they can now be automatically generated. Generation of OCaml code was also proposed in [14], but it was only available for the language of [2], which was less powerful than the current language. It also did not consider interpretation monads.

Regarding meta-languages to describe semantics, existing tools are much more complex than Skel. This is the case of Lem [15] and Ott [16]. The simplicity of Skel (the file `skeltypes.mli` describing Skel's AST is only 114 lines of specification) allows anyone to easily write a tool handling skeletal semantics. This is less immediate with Lem and Ott, as one has to deal with 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 [17] 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[5], which is one of our future goals. 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` [18] to go the other way). One may then execute the semantics using the OCaml version and prove 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, as OCaml or Coq AST are not easy to manipulate.

5. Conclusion

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 features. We have focused on dynamic semantics, but one may also describe static semantics in Skel. The tools Necro provide, such as Necro ML, help in the process of writing a semantics. Necro Coq allows to manipulate and certify these semantics once written. They give the necessary framework to prove program correctness 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 [1]. We plan to write a formalization of Python based on an existing precise description [19].

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

The Necro ecosystem includes other tools, such as Necro Debug,[6] which is a step-by-step execution of a semantics. An example execution can be found online.[7]. In addition, people can easily produce a new tool using Necro Lib.

Although not mentioned in this paper, Skel allows to split the definition of a semantics in several files, and to access them using an `include` construct. The OCaml generation tool can handle these multi-file semantics using modules, but at the moment this is not the case for Necro Coq. A future task is to implement this functionality using Coq modules. 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. Initial work on this standard library can be found online.[8]

Finally, Skel and Necro are currently being used to describe semantics style transformations, both at the object level [13] and at the meta-language level [12]. Work has also started to automatically derive control flow analyses from a language description.

[6] https://gitlab.inria.fr/skeletons/necro-debug
[7] https://skeletons.inria.fr/debugger/index_while.html
[8] https://gitlab.inria.fr/skeletons/necro/-/tree/master/examples/necro_in_necro

References

[1] A. Khayam, L. Noizet, A. Schmitt, Jskel: Towards a formalization of javascript's semantics, in: JLFA 2021 - Journées Francophones des Langages Applicatifs, 2021.

[2] M. Bodin, P. Gardner, T. Jensen, A. Schmitt, Skeletal Semantics and their Interpretations, Proceedings of the ACM on Programming Languages 44 (2019) 1–31. URL: https://hal.inria.fr/hal-01881863. doi:10.1145/3290357.

[3] TC39, ECMA-262, https://262.ecma-international.org/, ???? URL: https://262.ecma-international.org/.

[4] J. McCarthy, A basis for a mathematical theory of computation, in: P. Braffort, D. Hirschberg (Eds.), Computer Programming and Formal Systems, volume 26 of *Studies in Logic and the Foundations of Mathematics*, Elsevier, 1959, pp. 33–70. URL: https://www.sciencedirect.com/science/article/pii/S0049237X09700990. doi:https://doi.org/10.1016/S0049-237X(09)70099-0.

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

[6] A. Sabry, M. Felleisen, Reasoning about programs in continuation-passing style, in: LISP AND SYMBOLIC COMPUTATION, 1993, pp. 288–298.

[7] G. Kahn, Natural semantics, in: F. Brandenburg, G. Vidal-Naquet, M. Wirsing (Eds.), STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings, volume 247 of *Lecture Notes in Computer Science*, Springer, 1987, pp. 22–39. URL: https://doi.org/10.1007/BFb0039592. doi:10.1007/BFb0039592.

[8] L. Noizet, Necro Library, https://gitlab.inria.fr/skeletons/necro, ???? URL: https://gitlab.inria.fr/skeletons/necro.

[9] E. C. Martin Bodin, Nathanaëlle Courant, L. Noizet, Necro Ocaml Generator, https://gitlab.inria.fr/skeletons/necro-ml, ???? URL: https://gitlab.inria.fr/skeletons/necro-ml.

[10] O. Danvy, A. Filinski, Abstracting control, in: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP '90, Association for Computing Machinery, New York, NY, USA, 1990, p. 151–160. URL: https://doi.org/10.1145/91556.91622. doi:10.1145/91556.91622.

[11] L. Noizet, Necro Gallina Generator, https://gitlab.inria.fr/skeletons/necro-coq, ???? URL: https://gitlab.inria.fr/skeletons/necro-coq.

[12] G. Ambal, S. Lenglet, A. Schmitt, Certified Abstract Machines for Skeletal Semantics, in: Certified Programs and Proofs, Philadelphia, United States, 2022.

[13] G. Ambal, A. Schmitt, S. Lenglet, Automatic Transformation of a Big-Step Skeletal Semantics into Small-Step, Research Report RR-9363, Inria Rennes - Bretagne Atlantique, 2020. URL: https://hal.inria.fr/hal-02946930.

[14] N. Courant, E. Crance, A. Schmitt, Necro: Animating Skeletons, in: ML 2019, Berlin, Germany, 2019.

[15] D. Mulligan, S. Owens, K. Gray, T. Ridge, P. Sewell, Lem: Reusable engineering of real-world semantics, ACM SIGPLAN Notices 49 (2014). doi:10.1145/2628136.2628143.

[16] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, R. Strnisa, Ott:

Effective tool support for the working semanticist, J. Funct. Program. 20 (2010) 71–122. doi:10.1017/S0956796809990293.
[17] L. Li, E. L. Gunter, IsaK: A Complete Semantics of $\mathbb{K}$, Technical Report, Computer Science, Univ. of Illinois Urbana-Champaign, 2018.
[18] N. Labs, coq-of-ocaml, https://clarus.github.io/coq-of-ocaml, ???? URL: https://clarus.github.io/coq-of-ocaml.
[19] R. Monat, Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries, Ph.D. thesis, Sorbonne Université, 2021.

A. Typing Rules for Skeletal Semantics

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_t x : \tau}\ \textsc{Var} \qquad \frac{\texttt{val x:}\tau}{\Gamma \vdash_t x : \tau}\ \textsc{TermUnspec} \qquad \frac{\texttt{val x:}\tau\texttt{ = t}}{\Gamma \vdash_t x : \tau}\ \textsc{TermSpec}$$

$$\frac{\Gamma \vdash_t t : \tau \qquad \text{ctype}(\mathrm{C}) = (\tau, \tau')}{\Gamma \vdash_t \mathrm{C}\, t : \tau'}\ \textsc{Const} \qquad \frac{\Gamma \vdash_t t_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash_t t_n : \tau_n}{\Gamma \vdash_t (t_1, \ldots, t_n) : (\tau_1, \ldots, \tau_n)}\ \textsc{Tuple}$$

$$\frac{\Gamma + p \leftarrow \tau \vdash_S S : \tau'}{\Gamma \vdash_t (\lambda p : \tau \to S) : \tau \to \tau'}\ \textsc{Clos} \qquad \frac{\Gamma \vdash_t t : \nu \qquad \text{ftype}(f) = (\tau, \nu)}{\Gamma \vdash_t t.f : \tau}\ \textsc{FieldGet}$$

$$\frac{\Gamma \vdash_t t : (\tau_1, \ldots, \tau_m) \qquad 1 \leq i \leq m}{\Gamma \vdash_t t.i : \tau_i}\ \textsc{TupleGet}$$

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

$$\frac{\Gamma \vdash_t t : \tau \qquad \forall i \in [\![1; m]\!], \Gamma \vdash_t t_i : \tau_i \qquad \forall i \in [\![1; m]\!], \text{ftype}\, f_i = (\tau_i, \tau)}{\Gamma \vdash_t t \leftarrow (f_1 = t_1, \ldots, f_m = t_m) : \tau}\ \textsc{FieldSet}$$

$$\frac{\Gamma \vdash_t t : \tau}{\Gamma \vdash_S \text{ret}\, t : \tau}\ \textsc{Ret} \qquad \frac{\Gamma \vdash_S S_1 : \tau \qquad \ldots \qquad \Gamma \vdash_S S_n : \tau}{\Gamma \vdash_S (S_1 \ldots S_n) : \tau}\ \textsc{Branch}$$

$$\frac{\Gamma \vdash_S S : \tau \qquad \Gamma + p \leftarrow \tau \vdash_S S' : \tau'}{\Gamma \vdash_S \texttt{let}\, p = S\, \texttt{in} S' : \tau'}\ \textsc{LetIn} \qquad \frac{\Gamma + p \leftarrow \tau \vdash_S S : \tau'}{\Gamma \vdash_S \text{let}\, p : \tau \,\text{in}\, S : \tau'}\ \textsc{Exist}$$

$$\frac{\Gamma \vdash_t t : \tau \qquad \Gamma + p_1 \leftarrow \tau \vdash_S S_1 : \nu \qquad \ldots \qquad \Gamma + p_n \leftarrow \tau \vdash_S S_n : \nu}{\Gamma \vdash_S \texttt{match}\, t\, \texttt{with} | p_1 \to S_1 | \ldots | p_n \to S_n : \nu}\ \textsc{Match}$$

$$\frac{\Gamma \vdash t_0 : \tau_1 \to \cdots \to \tau_n \to \tau \qquad \Gamma \vdash_t t_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash_t t_n : \tau_n}{\Gamma \vdash_S (t_0\, t_1 \ldots t_n) : \tau}\ \textsc{App}$$

B. Concrete Interpretation of Skeletal Semantics

First we defined rules to handle environments. We define the relation $p \not\mapsto v$ which holds if p cannot represent the value v, and the partial function $E + v\,\mathrm{match}(p_1, \ldots, p_n)$ as follows:

$$\frac{p \not\mapsto v}{Cp \not\mapsto Cv}\ \textsc{NoMatchSameConstr} \qquad \frac{C \neq C'}{Cp \not\mapsto C'v}\ \textsc{NoMatchDiffConstr}$$

$$\frac{p_i \not\mapsto v_i \qquad 0 \leq i \leq n}{(p_1, \ldots, p_n) \not\mapsto (v_1, \ldots, v_n)}\ \textsc{NoMatchTuple}$$

$$\frac{p_i \not\mapsto v_i \qquad 0 \leq i \leq m \leq n}{(f_1 = p_1, \ldots, f_m = p_m) \not\mapsto (f_1 = v_1, \ldots, f_n = v_n)}\ \textsc{NoMatchRecord}$$

$$\frac{E + p_1 \leftarrow v = E'}{E + v\,\mathrm{match}(p_1, \ldots, p_n) = (1, E')}\ \textsc{GetMatchFirst}$$

$$\frac{p_1 \not\mapsto v \qquad E + v\,\mathrm{match}(p_2, \ldots, p_n) = (i, E')}{E + v\,\mathrm{match}(p_1, \ldots, p_n) = (i + 1, E')}\ \textsc{GetMatchNext}$$

Now we define the rules for concrete interpretation:

$$\frac{E(x) = v}{E, x \Downarrow_t v}\ \textsc{Var} \qquad \frac{\texttt{val x : } \tau \qquad \text{arity}(x) = 0 \qquad [\![x]\!](v)}{E, x \Downarrow_t v}\ \textsc{TermUnspecZero}$$

$$\frac{\texttt{val x : } \tau \qquad \text{arity}(x) > 0}{E, x \Downarrow_t \lceil x, () \rceil}\ \textsc{TermUnspecSucc} \qquad \frac{\texttt{val x:}\tau\texttt{ = t} \qquad \epsilon, t \Downarrow_t v}{E, x \Downarrow_t v}\ \textsc{TermSpec}$$

$$\frac{E, t \Downarrow_t v}{E, (\mathrm{C}\, t) \Downarrow_t \mathrm{C}\, v}\ \textsc{Const} \qquad \frac{E, t_1 \Downarrow_t v_1 \qquad \dots \qquad E, t_n \Downarrow_t v_n}{E, (t_1, \dots, t_n) \Downarrow_t (v_1, \dots, v_n)}\ \textsc{Tuple}$$

$$\frac{}{E, (\lambda p : \tau \to S) \Downarrow_t \langle p, E, S \rangle}\ \textsc{Clos} \qquad \frac{E, t \Downarrow_t (f_1 = v_1, \dots, f_n = v_n)}{E, t.f_i \Downarrow_t v_i}\ \textsc{FieldGet}$$

$$\frac{E, t \Downarrow_t (v_1, \dots, v_m) \qquad 1 \le i \le m}{E, t.i \Downarrow_t v_i}\ \textsc{TupleGet}$$

$$\frac{E, t_1 \Downarrow_t v_1 \qquad \dots \qquad E, t_n \Downarrow_t v_n}{E, (f_1 = t_1, \dots, f_n = t_n) \Downarrow_t (f_1 = v_i, \dots, f_n = v_n)}\ \textsc{Rec}$$

$$\frac{\begin{array}{c} E, t \Downarrow_t (f_1 = v_1, \dots, f_n = v_n) \qquad \forall i \in [\![1; m]\!], E, t_i \Downarrow_t w_{j_i} \\ \forall i \in [\![1; n]\!] \setminus \{j_1, \dots, j_m\}, w_i = v_i \end{array}}{E, t \leftarrow (f_{j_1} = t_1, \dots, f_{j_m} = t_m) \Downarrow_t (f_1 = w_1, \dots, f_n = w_n)}\ \textsc{FieldSet} \qquad \frac{E, t \Downarrow_t v}{E, \text{ret}\, t \Downarrow_S v}\ \textsc{Ret}$$

$$\frac{E, S_i \Downarrow_S v \qquad 1 \le i \le n}{E, (S_1 \dots S_n) \Downarrow_S v}\ \textsc{Branch} \qquad \frac{E, S \Downarrow_S v \qquad E + p \leftarrow v, S' \Downarrow_S w}{E, \texttt{let}\ p = S\ \texttt{in}\ S' \Downarrow_S w}\ \textsc{LetIn}$$

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

$$\frac{E, t \Downarrow_t v \qquad E + v \,\text{match}\, p_1 \dots p_n = (i, E') \qquad E', S_i \Downarrow_S w}{E, \texttt{match}\ t\ \texttt{with} | p_1 \to S_1 | \dots | p_n \to S_n \texttt{end} \Downarrow_S w}\ \textsc{Match}$$

$$\frac{E, t_0 \Downarrow_t f \qquad E, t_1 \Downarrow_t v_1 \qquad \dots \qquad E, t_n \Downarrow_t v_n \qquad f\ v_1\ \dots\ v_n \Downarrow_{\text{app}} w}{E, (t_0\ t_1 \dots t_n) \Downarrow_S w}\ \textsc{App}$$

$$\frac{}{v \Downarrow_{\text{app}} v}\ \textsc{AppZero} \qquad \frac{E + p \leftarrow v_1, S \Downarrow_S g \qquad g\ v_2\ \dots\ v_n \Downarrow_{\text{app}} w}{\langle p, E, S \rangle\ v_1\ \dots\ v_n \Downarrow_{\text{app}} w}\ \textsc{AppClos}$$

$$\frac{\text{arity}(x) > n}{\lceil x, (v_1, \dots, v_m) \rceil\ v_{m+1}\ \dots\ v_n \Downarrow_{\text{app}} \lceil x, (v_1, \dots, v_n) \rceil}\ \textsc{AppUnspecNext}$$

$$\frac{\text{arity}(x) = m \le n \qquad [\![x]\!](v_1, \dots, v_m, g) \qquad g\ v_{m+1}\ \dots\ v_n \Downarrow_{\text{app}} w}{\lceil x, (v_1, \dots, v_m) \rceil\ v_{m+1}\ \dots\ v_n \Downarrow_{\text{app}} w}\ \textsc{AppUnspecCont}$$