Certified Abstract Machines for Skeletal Semantics

Guillaume Ambal, Sergueï Lenglet, Alan Schmitt

Certified Abstract Machines for Skeletal Semantics

Guillaume Ambal
Univ Rennes
Rennes, France
guillaume.ambal@irisa.fr

Sergueï Lenglet
Université de Lorraine
Nancy, France
serguei.lenglet@univ-lorraine.fr

Alan Schmitt
Inria
Rennes, France
alan.schmitt@inria.fr

Abstract

Skeletal semantics is a framework to describe semantics of programming languages. We propose an automatic generation of a certified OCaml interpreter for any language written in skeletal semantics. To this end, we introduce two new interpretations, i.e., formal meanings, of skeletal semantics, in the form of non-deterministic and deterministic abstract machines. These machines are derived from the usual big-step interpretation of skeletal semantics using functional correspondence, a standard transformation from big-step evaluators to abstract machines. All these interpretations are formalized in the Coq proof assistant and we certify their soundness. We finally use the extraction from Coq to OCaml to obtain the certified interpreter.

Keywords: Skeletal Semantics, Operational Semantics, Abstract Machines, Certified Interpretation

1 Introduction

Skeletal semantics [8] is a framework, based on a meta-language, to formalize the operational semantics of programming languages. The fundamental idea is to only specify the structure of evaluation functions (e.g., sequences of operations, non-deterministic choices, recursive calls) while keeping abstract basic operations (e.g., updating an environment or comparing two values). The motivation for this semantics is that the structure can be analyzed, transformed, or certified independently from the implementation choices of the basic operations.

Skeletal semantics can be deeply embedded in Coq to be interpreted in a non-deterministic big-step style, called *concrete interpretation*. While useful to prove some properties of a language or of programs, such interpretations cannot reason about non-terminating programs and are quite far from an actual implementation. We thus propose an alternate interpretation in the form of an abstract machine, derived from the big-step one using functional correspondence [1]. The derivation is done in two steps, first creating a non-deterministic abstract machine (NDAM), then removing the non-determinism to obtain a final abstract machine. As the machines take as input the same deep embedding of a skeletal semantics, we can prove their soundness in relation to the big-step semantics.

Using the Coq extraction mechanism [15] on the final abstract machine, we obtain a certified OCaml interpreter that can be instantiated with any language. From a user-defined language written as a skeletal semantics, the existing framework [8] can automatically produce the Coq deep embedding, which itself can be used to instantiate our extracted interpreter. We therefore obtain a certified interpreter for the language at no extra cost for the user.

Our contributions are:

- the derivation of a non-deterministic abstract machine and a deterministic one to interpret skeletal semantics;
- the certification in Coq of their soundness in relation to the usual concrete interpretation;
- an extracted certified OCaml interpreter that can be instantiated with any language.

The paper is structured as follows. We introduce skeletal semantics in Section 2, we derive and prove correct the non-deterministic abstract machine in Section 3 and the deterministic one is Section 4. We finally describe the framework to obtain a certified OCaml interpreter in Section 5.

The implementation of this work, the Coq proofs, examples of interpreters, and appendices are available online [4].

2 Skeletal Semantics

This section presents the concept of skeletal semantics. It uses a meta-language, called Skel, in which we can embed the structure and behavior of any programming language. The first subsection (2.1) introduces Skel by example. The next ones formalize the syntax (2.2) and semantics (2.3) of the meta-language. We finally describe the embedding of Skel in Coq in Section 2.4.

2.1 Informal Example

As a toy example to introduce skeletal semantics, we present the semantics of the basic call-by-value λ-calculus. For readability, this is illustrated with a concrete syntax, the formal definition is presented in Section 2.2.

To define λ-terms, we first need a type for variables.

```
type ident
```

This type is unspecified as we do not assume anything about it or its elements. We use it to create an algebraic datatype for λ-terms.

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

This corresponds to the usual grammar $t ::= \lambda x.t \mid x \mid t\ t$.

To be able to define an evaluation, we first need to introduce types for closures and environments. We represent closures with an algebraic datatype and a single constructor.

```
type clos =
| Clos (ident, lterm, env)
```

I.e., `Clos (x, t, s)` corresponds to the abstraction $\lambda x.t$ with free variables mapped in s.

We could define environments similarly, using for instance a custom implementation of lists. But we do not care about the structure of environments, only that they map variables to closures. Instead, we keep the type unspecified, and we declare unspecified functions to manipulate them.

```
type env

val extEnv: (env, ident, clos) → env
val getEnv: (ident, env) → clos
```

We assume `extEnv` extends an environment with a pair (`ident`,`clos`), and `getEnv` looks up the mapping of an identifier. These *unspecified terms* only specify their type. This way, we can reason about this λ-calculus language independently of the actual implementation of environments.

Finally, we define the main evaluation function, here named `eval`. The body of the function contains a branching, where each branch represents a possible behavior of the `eval` function. Branches may contain pattern-matchings, for instance `let Lam (x, t) = l in ...`, which succeeds only if `l` has the correct shape, and this binds `x` and `t` in the continuation. If the pattern-matching fails, another branch is then taken. The code is straightforward: abstractions are transformed into closures; variables are looked up in the environment; applications first evaluate both terms, with the functional part returning a closure $\texttt{Clos}(x, t, s_1)$ and the argument part a closure c, before evaluating t in the environment s_1 extended with the pair (x, c).

```
val eval (s: env) (l: lterm) : clos =
  branch
    let Lam (x, t) = l in
    Clos (x, t, s)
  or
    let Var x = l in
    getEnv (x, s)
  or
    let App (t1, t2) = l in
    let Clos (x, t, s1) = eval s t1 in
    let c = eval s t2 in
    let s2 = extEnv (s1, x, c) in
    eval s2 t
  end
```

Note that a branching is a non-deterministic construction: each of the three branches might be chosen. If at any point in a branch a pattern-matching fails, then the whole branch is discarded. If no branch remains, the whole branching fails.

The skeletal semantics is strongly typed. Constructors such as `Lam` are specified with the expected types of the arguments. Functions such as `getEnv` and `eval` are also explicitly typed. Even variables within `eval` are implicitly typed. However, this typing is out of the scope of this paper, as it does not conflict with the transformations presented nor the steps of the resulting abstract machines. The following sections formalize the definition of skeletal semantics, but we ignore types to simplify the presentation.

2.2 Syntax

We now introduce the syntax of skeletal semantics. Compared to [8], our language has higher-order functions and pattern-matching.

We write $[]$ for an empty list and $(a :: l)$ for adding an element a to a list l. We note $[a_1; \ldots; a_n]$ for the list $(a_1 :: \ldots :: a_n :: [])$. We write $l_1 +\!\!+ l_2$ the concatenation of two lists, $[a_1; \ldots; a_n] +\!\!+ [b_1; \ldots; b_m] \triangleq [a_1; \ldots; a_n; b_1; \ldots; b_m]$.

The skeletal semantics of a language is composed of:

- a set of unspecified types (without constructors),
- a set of specified types with (typed) constructors,
- a set of (typed) unspecified skelterms,
- a set of (typed) specified skelterms.

We use the word "skelterm" for terms of the Skel metalanguage to avoid confusion with terms of the embedded language (e.g., λ-terms). In the example of Section 2.1, `ident` is an unspecified type; `lterm` a specified type with three constructors, `getEnv` is an unspecified skelterm, and `eval` a specified skelterm.

An *unspecified skelterm* is an identifier representing an abstract object/function not given when defining the language. To execute the language in practice, we would need to determine how to interpret this identifier (see Section 2.3).

We call *specified skelterms* identifiers associated to an explicit skelterm. We note `SpecDecl` the mapping from identifiers (i.e., strings) to their corresponding skelterm.

We let c, n, and x range over respectively constructors, natural numbers, and identifiers (i.e., strings). For any entity e, we note l_e for lists of elements of this entity: for instance, l_c represents a list of constructors. The grammar of variables (v), patterns (p), skelterms (t), and skeletons (S) is defined as follows.

$$
\begin{aligned}
v &::= \texttt{VLet}(x) \mid \texttt{VUnspec}(x) \mid \texttt{VSpec}(x) \\
p &::= \texttt{PWild} \mid \texttt{PVar}(x) \mid \texttt{PConstr}(c, p) \mid \texttt{PTuple}(l_p) \\
t &::= \texttt{TVar}(v) \mid \texttt{TConstr}(c, t) \mid \texttt{TTuple}(l_t) \mid \texttt{TFunc}(p, S) \\
S &::= \texttt{Branching}(l_S) \mid \texttt{LetIn}(p, S_1, S_2) \\
&\quad \mid \texttt{Apply}(t, l_t) \mid \texttt{Return}(t)
\end{aligned}
$$

A *skelterm* represents a completed computation. It is recursively composed of constructors and tuples, whose leaves are either variables or functions.

A *skeleton* represents a computation to perform. It can either be a LetIn structure with pattern-matching (`let` p = S_1 `in` S_2) chaining two computations, a non-deterministic choice among several computations (`Branching`(l_S)), an application of a function to a list of arguments (`Apply`(t, l_t)), or simply returning a skelterm.

Skeletal variables v correspond to the different strings we can see appear in a skelterm or skeleton. A variable of the language is defined using `PVar`(x) in a pattern and used with `VLet`(x). A specified skelterm is referenced using `VSpec`(x), an unspecified one using `VUnspec`(x).

As an example, the skelterm of the `eval` function of Section 2.1 is of the following form—we only detail the second branch `let Var x = l in getEnv (x,s)`.

$$\begin{aligned}
&\texttt{SpecDecl("eval")} = S\\
S &\triangleq \texttt{TFunc(PVar("s"),Return(TFunc(PVar("l"),}S_0\texttt{)))}\\
S_0 &\triangleq \texttt{Branching([...;LetIn(}p, S_1, S_2\texttt{);...])}\\
p &\triangleq \texttt{PConstr("Var",PVar("x"))}\\
S_1 &\triangleq \texttt{Return(TVar(VLet("l")))}\\
S_2 &\triangleq \texttt{Apply(TVar(VUnspec("getEnv")),[}t\texttt{])}\\
t &\triangleq \texttt{TTuple([TVar(VLet("x"));TVar(VLet("s"))])}
\end{aligned}$$

2.3 Concrete Interpretation

A skeletal semantics is just syntax. An *interpretation* gives meaning to each construct. The interpretation we consider in this paper, called *concrete interpretation*, corresponds to a big-step evaluation of the objects introduced in Section 2.2 (skelterms, skeletons, etc.).

The results of this evaluation are called *concrete values*, or cvalues for short. The grammar of concrete values (r), and environments (Σ) is defined as follows.

$$\begin{aligned}
r ::= {}& \texttt{CVTuple}(l_r) \mid \texttt{CVConstr}(c, r)\\
&\mid \texttt{CVClos}(\Sigma, p, S) \mid \texttt{CVUnspec}(x, n, l_r)\\
&\mid \texttt{CVBase}(a)\\
\Sigma ::= {}& [(x_1, r_1); \ldots; (x_n, r_n)]
\end{aligned}$$

Environments are simply lists mapping identifiers to results. `CVTuple`(l_r) represents a tuple of results, `CVConstr`(c, r) a constructor packing another cvalue. `CVClos`(Σ, p, S) is a closure, corresponding to the skelterm `TFunc`(p, S) bundled with an evaluation environment Σ. `CVUnspec`(x, n, l_r) represents a partially applied unspecified function x, with partial arguments l_r (see below for more details). Finally, `CVBase`(a) is an injection from base values of unspecified types to cvalues. Indeed, the concrete interpretation assumes given an instantiation of unspecified types and unspecified skelterms. For each unspecified type, we expect a set representing its values (e.g., $\mathbb{N}$ for `"nat"`, or $\{\top; \bot\}$ for `"bool"`). In the grammar above, a represents an object in the interpretation of an unspecified type.

For each unspecified skelterm, we require an arity and a function producing a list of possible results. For notation purposes, we group these elements in two main auxiliary functions: `Arity` of type (`string` → `nat`); and `UnspecDecl` of type (`string` → `cvalue list` → `cvalue list`). For an unspecified skelterm x, `UnspecDecl`(x) takes as argument a list of size `Arity`(x) and outputs a list of possible results.

Example 2.1. To simulate an If/Then/Else construction, we can provide unspecified skelterms `"isTrue"` and `"isFalse"` of type (`"bool"` → ()) with the following definitions:

$$\begin{aligned}
&\texttt{Arity("isTrue")} = 1\\
&\texttt{UnspecDecl("isTrue")[CVBase(}\top\texttt{)]} = \texttt{[CVTuple([])]}\\
&\texttt{UnspecDecl("isTrue")[CVBase(}\bot\texttt{)]} = \texttt{[]}
\end{aligned}$$

where unit is represented as an empty tuple, and similarly for `"isFalse"`. This shows that an empty list result can be interpreted as a failure. A conditional branching "If v then … else …" can be simulated with:

```
branch let () = isTrue v in
       ... (* 'then' part *)
or     let () = isFalse v in
       ... (* 'else' part *)
end
```

Example 2.2. A list of multiple results can also represent non-determinism. For instance, for choosing a number in an interval, we can use:

$$\begin{aligned}
&\texttt{Arity("randInt")} = 2\\
&\texttt{UnspecDecl("randInt")[CVBase(5);CVBase(10)]} =\\
&\texttt{[CVBase(5);CVBase(6);...;CVBase(9);CVBase(10)]}
\end{aligned}$$

The non-deterministic inductive inference rules of Figure 1 express how the different structures of skeletal semantics are evaluated. The important rules are those for the evaluation of skelterms ($\Sigma \vdash t \Downarrow_t r$) and skeletons ($\Sigma \vdash S \Downarrow_s r$) to concrete values.

Most of the rules are self-explanatory. For instance, evaluating a skeleton `LetIn`(p, S_1, S_2) under Σ corresponds to evaluating S_1 under Σ to get r', performing a pattern-matching between p and r' to expand the environment to Σ', and finally evaluating S_2 under Σ'. Some premises might not be doable, for instance the pattern-matching between p and r' could fail. In this case, the rule does not apply and the skeleton cannot be evaluated.

Environment lookup ($\Sigma(x) = r$) corresponds to finding the first pair of the form (x, r) in Σ. Pattern-matching ($\Sigma + \{p \mapsto r\} \Downarrow_p \Sigma'$) corresponds to extending Σ into Σ' with pairs (x_i, r_i), and is only valid if p and r have (recursively) tuples of the same arity and use the same constructors.

The evaluation of unspecified skelterms is probably the least intuitive, as the concrete interpretation allows for partially applied unspecified functions. If we encounter an unspecified skelterm of arity 0, we immediately apply its interpretation. Otherwise, (`Arity`$(x) = n+1$), we create a concrete

value `CVUnspec`$(x, n, [])$, keeping track of the missing number of arguments $(n + 1)$ and the already given arguments (starting with an empty list).

The behavior of application for unspecified skelterms depends on the number of arguments. If there is none, we simplify as expected. Else we have $(m + 1)$ arguments. If we have enough new arguments ($m + 1 \geq n + 1$, simplified to $m \geq n$ in Figure 1) we apply the interpretation of the skelterm. Otherwise ($m < n$), we expand the list of partial arguments and diminish the counter of missing arguments.

A few rules of Figure 1 are non-deterministic, as we sometimes pick an element in a list. This happens when choosing a branch in a branching, which can be seen as an internal source of non-determinism; and when selecting a result from the interpretation of unspecified skelterms, which can be seen as external non-determinism.

Note that there is no rule for generating or consuming base values `CVBase`(a) of unspecified types, as they can only be handled by unspecified skelterms.

2.4 Formalization in Coq

The Skel meta-language comes with a transformer from concrete syntax to a deep embedding in the Coq proof assistant [19] (see the artifact). For instance, the example of Section 2.1 placed in a file `lambda.sk` can be automatically transformed into a Coq definition file `Lambda.v`. This embedding uses the types defined in the file `Skeleton.v`, formalizing the grammar of Section 2.2. The concrete interpretation of Section 2.3 is also available in a file `Concrete.v`, encoded as inductive predicates. Formalizing properties of a language can be done by importing the independent files `Concrete.v` and `Lambda.v`.

One of the contributions of this paper is the addition of two new interpretations, formalized as `Concrete_ndam.v` (Section 3) and `Concrete_am.v` (Section 4), also dependent on the same syntax in `Skeleton.v`. Furthermore, the Coq extraction mechanism [15] can combine the contents of `Concrete_am.v` and `Lambda.v` into a certified interpreter (Section 5).

3 Deriving the Non-Deterministic Abstract Machine

The first step towards a certified interpreter is to transform the big-step interpretation of skeletons into an abstract machine while keeping the non-determinism of the concrete interpretation. We thus derive a Non-Deterministic Abstract Machine (NDAM) from the concrete interpretation. Section 4 describes the next step of the translation to obtain a deterministic abstract machine.

The derivation follows the known strategy of functional correspondence [1]. The main phases of the derivation are a CPS-transformation [17] (Section 3.2), a phase of defunctionalization [18] (Section 3.3), and then the proper creation of the abstract machine and its evaluation modes (Section 3.4).

As the standard strategy operates on big-step interpreters, we first rewrite the concrete interpretation from inference rules to pseudo-code (Section 3.1). Finally, Section 3.5 presents the Coq equivalence result between the resulting abstract machine and the rules of Figure 1.

3.1 Pseudo-Interpreter

We start by translating the concrete interpretation into pseudo-code that we can manipulate more freely. We use an OCaml-like syntax [14] for the pseudo-code, where natural numbers have constructors `Z` and `S`.

We translate each predicate of Figure 1 into evaluation functions, resulting into the following pseudo-code for skeletons:

```
let eval_sk sk e : cvalue = match sk with
| Branching (skl) ->
  (* oracle picking the correct skeleton *)
  let sk' = pick skl in
  eval_sk sk' e
| ...
| LetIn (p, sk1, sk2) ->
  let r = eval_sk sk1 e in
  let e2 = eval_pat p r e in
  eval_sk sk2 e2
```

The non-determinism of the evaluation of a branching is reflected by the `pick` function which behaves like an oracle, as it is able to choose an appropriate element in a list.

Similarly, we use the same function for unspecified skelterms. As presented before, the evaluation of such a skelterm depends on its arity. If it does not require arguments, we immediately call its interpretation, and then arbitrarily select a result from the obtained list, using `pick`. Otherwise, we create a partially applied skelterm initialized with zero arguments.

```
let eval_trm t e : cvalue = match t with
| TVar (v) -> match v with
  | VLet (x) -> lookup e x
  | VSpec (h) -> eval_trm (specdecl h) []
  | VUnspec (f) -> match arity f with
    | Z -> let rl = unspecdecl f [] in
           (* oracle picking the correct cvalue *)
           let r = pick rl in
           r
    | S m -> CVUnspec (f, m, [])
| TConstr (c, t') ->
  let r = eval_trm t' e in
  CVConstr (c, r)
| ...
```

We also define evaluation functions for lists of skelterms and lists of pattern-matchings. The former corresponds to

Interpretation of Skelterms:

$$\frac{\Sigma(x) = r}{\Sigma \vdash \texttt{TVar}(\texttt{VLet}(x)) \Downarrow_\text{t} r} \qquad \frac{\texttt{Arity}(x) = 0 \qquad r \in \texttt{UnspecDecl}(x)[]}{\Sigma \vdash \texttt{TVar}(\texttt{VUnspec}(x)) \Downarrow_\text{t} r} \qquad \frac{\texttt{Arity}(x) = n + 1}{\Sigma \vdash \texttt{TVar}(\texttt{VUnspec}(x)) \Downarrow_\text{t} \texttt{CVUnspec}(x, n, [])}$$

$$\frac{\texttt{SpecDecl}(x) = t \qquad [] \vdash t \Downarrow_\text{t} r}{\Sigma \vdash \texttt{TVar}(\texttt{VSpec}(x)) \Downarrow_\text{t} r} \qquad \frac{\Sigma \vdash t \Downarrow_\text{t} r}{\Sigma \vdash \texttt{TConstr}(c, t) \Downarrow_\text{t} \texttt{CVConstr}(c, r)}$$

$$\frac{\forall i,\ \Sigma \vdash t_i \Downarrow_\text{t} r_i}{\Sigma \vdash \texttt{TTuple}([t_1, \ldots, t_n]) \Downarrow_\text{t} \texttt{CVTuple}([r_1, \ldots, r_n])} \qquad \frac{}{\Sigma \vdash \texttt{TFunc}(p, S) \Downarrow_\text{t} \texttt{CVClos}(\Sigma, p, S)}$$

Interpretation of Skeletons:

$$\frac{S \in l \qquad \Sigma \vdash S \Downarrow_\text{s} r}{\Sigma \vdash \texttt{Branching}(l) \Downarrow_\text{s} r} \qquad \frac{\Sigma \vdash t \Downarrow_\text{t} r}{\Sigma \vdash \texttt{Return}(t) \Downarrow_\text{s} r} \qquad \frac{\Sigma \vdash t \Downarrow_\text{t} r_0 \qquad \forall i,\ \Sigma \vdash t_i \Downarrow_\text{t} r_i \qquad r_0\ \$\ [r_1, \ldots, r_n] \Downarrow_\text{a} r}{\Sigma \vdash \texttt{Apply}(t, [t_1, \ldots, t_n]) \Downarrow_\text{s} r}$$

$$\frac{\Sigma \vdash S_1 \Downarrow_\text{s} r' \qquad \Sigma + \{p \mapsto r'\} \Downarrow_\text{p} \Sigma' \qquad \Sigma' \vdash S_2 \Downarrow_\text{s} r}{\Sigma \vdash \texttt{LetIn}(p, S_1, S_2) \Downarrow_\text{s} r}$$

Interpretation of Application:

$$\frac{}{r\ \$\ [] \Downarrow_\text{a} r} \qquad \frac{\Sigma + \{p \mapsto r_0\} \Downarrow_\text{p} \Sigma' \qquad \Sigma' \vdash S \Downarrow_\text{s} r_1 \qquad r_1\ \$\ l_r \Downarrow_\text{a} r}{\texttt{CVClos}(\Sigma, p, S)\ \$\ (r_0 :: l_r) \Downarrow_\text{a} r}$$

$$\frac{m \geq n \qquad r' \in \texttt{UnspecDecl}(x)(l +\!\!+ [r_0, \ldots, r_n]) \qquad r'\ \$\ [r_{n+1}, \ldots, r_m] \Downarrow_\text{a} r}{\texttt{CVUnspec}(x, n, l)\ \$\ [r_0, \ldots, r_m] \Downarrow_\text{a} r}$$

$$\frac{m < n \qquad n' = n - (m + 1) \qquad l' = l +\!\!+ [r_0, \ldots, r_m]}{\texttt{CVUnspec}(x, n, l)\ \$\ [r_0, \ldots, r_m] \Downarrow_\text{a} \texttt{CVUnspec}(x, n', l')}$$

Pattern-Matching:

$$\frac{}{\Sigma + \{\texttt{PWild} \mapsto r\} \Downarrow_\text{p} \Sigma} \qquad \frac{}{\Sigma + \{\texttt{PVar}(x) \mapsto r\} \Downarrow_\text{p} (x, r) :: \Sigma} \qquad \frac{\Sigma + \{p \mapsto r\} \Downarrow_\text{p} \Sigma'}{\Sigma + \{\texttt{PConstr}(c, p) \mapsto \texttt{CVConstr}(c, r)\} \Downarrow_\text{p} \Sigma'}$$

$$\frac{}{\Sigma + \{\texttt{PTuple}([]) \mapsto \texttt{CVTuple}([])\} \Downarrow_\text{p} \Sigma} \qquad \frac{\Sigma + \{p \mapsto r\} \Downarrow_\text{p} \Sigma' \qquad \Sigma' + \{\texttt{PTuple}(l_p) \mapsto \texttt{CVTuple}(l_r)\} \Downarrow_\text{p} \Sigma''}{\Sigma + \{\texttt{PTuple}(p :: l_p) \mapsto \texttt{CVTuple}(r :: l_r)\} \Downarrow_\text{p} \Sigma''}$$

Environment Lookup:

$$\frac{}{((x, r) :: \Sigma)(x) = r} \qquad \frac{x \neq y \qquad \Sigma(x) = r}{((y, r) :: \Sigma)(x) = r}$$

Figure 1. Concrete Interpretation

mapping the evaluation function for skelterms over the list, while the latter behaves like a fold over the list, in accordance with the rules of Figure 1.

The full pseudo-code is available in Appendix B.1 of the supplementary material.

3.2 CPS-Transform

From now on, we follow the textbook strategy [1] to create an abstract machine. In each code snippet illustrating the different phases of the transformation, we use a gray background to indicate changes relative to previous sections.

We first rephrase the pseudo-code in Continuation Passing Style. We modify the evaluation functions to include an extra argument, a continuation k, indicating what is left to compute after the current function is finished. When a function is done computing, it passes the result to the continuation k instead of directly returning it.

This CPS transformation is only applied to the main evaluation functions: `eval_trm`, `eval_sk`, `eval_pat`, `lookup`, etc. We do not modify the parametric functions (`pick` / `specdecl` / `arity` / `unspecdecl`) nor the basic functions such as the concatenation of two lists.

```
let eval_trm t e (k:cvalue -> cvalue) : cvalue =
match t with
| TVar (v) -> match v with
  | VLet (x) -> lookup e x k
  | VSpec (h) -> eval_trm (specdecl h) [] k
  | VUnspec (f) -> match arity f with
    | Z -> let r = pick (unspecdecl f []) in
           k r
    | S m -> k (CVUnspec (f, m, []))
| TConstr (c, t') ->
  eval_trm t' e (fun r -> k (CVConstr (c, r)))
| ...

let eval_sk sk e (k:cvalue -> cvalue) : cvalue =
match sk with
| Branching (skl) ->
  let sk' = pick skl in
  eval_sk sk' e k
| ...
| LetIn (p, sk1, sk2) ->
  eval_sk sk1 e (fun r ->
  eval_pat p r e (fun e2 ->
  eval_sk sk2 e2 k))
```

The arity zero case is an example where the result is passed to the continuation. Calls to unmodified functions, such as `pick`, are left unchanged. A call to an evaluation function f is changed so that we create a continuation for the rest of the code and make a single tail-call to f with it. When several of such calls are chained, we build nested continuations, like for the constructor `LetIn`: the continuation after evaluating `sk1` consists of performing the pattern-matching with a continuation evaluating `sk2`.

The full pseudo-code at this point is available in Appendix B.2 of the supplementary material.

3.3 Defunctionalization

The CPS transformation introduced several anonymous functions, on which we cannot do pattern-matching in the target abstract machine. As such, we perform a phase of defunctionalization, creating new types, constructors, and dispatch functions to replace continuations.

For each anonymous function generated in Section 3.2, we create a fresh constructor, whose arguments are the free variables of the function. In the code, we replace each anonymous continuation by its new corresponding constructor.

```
type krt =
| KRID
| KRLet of pattern * skeleton * env * krt
| ...

let eval_sk sk e (k: krt) : cvalue =
match sk with
| Branching (skl) ->
  let sk' = pick skl in
  eval_sk sk' e k
| ...
| LetIn (p, sk1, sk2) ->
  eval_sk sk1 e (KRLet (p,sk2,e,k))
```

We create the type `krt` to represent continuations expecting a result, i.e., a concrete value. For example, the continuation of the evaluation of `sk1` in the LetIn case of Section 3.2 is replaced by the constructor `KRLet`, whose arguments are the free variables of the continuation. Additionally, we create a constructor $\mathrm{kr}_{\mathrm{id}}$ representing the identity continuation. It is used to start and stop a computation, and corresponds to the function $\lambda r.r$.

Reifying continuations means that we can no longer simply apply them to compute. Instead, we introduce an external *dispatch* function `disp_kr` which pattern-matches on object continuations and triggers the corresponding computation.

```
let disp_kr (k: krt) (r: cvalue) : cvalue =
match k with
| KRLet (p,sk,e,k') ->
    eval_pat p r e (KELet (sk,k'))
| ...

let eval_trm t e (k: krt) : cvalue =
match t with
| TVar (v) -> match v with
  | VLet (x) -> lookup e x k
  | VSpec (h) -> eval_trm (specdecl h) [] k
  | VUnspec (f) -> match arity f with
```

```
    | Z -> let r = pick (unspecdecl f []) in
           disp_kr k r
    | S m -> disp_kr k (CVUnspec (f, m, []))
| ...
```

For instance, the application of the continuation `k r` in the arity zero case is replaced by a call `disp_kr k r`. Note that there is no dispatch rule for the identity continuation kr_{id}, as there is no computation left to perform in that case: we expect the abstract machine to stop when encountering it.

Continuations expecting concrete values are not the only possibility in our CPS-transformed evaluator: some expect either a list of concrete values or an environment. Therefore, we also create new types `klt` and `ket`, corresponding to continuations expecting respectively lists of cvalues and environments. These two types are also given their own identity continuation: kl_{id} and ke_{id}.

```
type ket =
| KEID
| KELet of skeleton * krt
| ...
let disp_ke (k: ket) (e: env) : cvalue =
match k with
| KELet (sk, k') -> eval_sk sk e k'
| ...
```

We can see `KELet` being used when we dispatch the continuation `KRLet` in `disp_kr`: it corresponds to the continuation of the evaluation of the pattern `p` in the LetIn case of Section 3.2.

The full pseudo-code at this point is available in Appendix B.3 of the supplementary material.

3.4 Abstract Machine

The defunctionalization phase results in a pseudo-code in big-step style, because the evaluation functions still return a fully computed result. However, it is in a shape appropriate to be translated into an abstract machine: there is no anonymous function, and every evaluation function performs a pattern-matching immediately followed by a tail-call. The parametric functions (such as `specdecl` and `pick`), can be translated as guarding conditions of the abstract machine.

Each evaluation and dispatch function of this pseudo-interpreter becomes a mode of the abstract machine, with the same arguments as the ones of the function. For instance, the function `eval_sk S Σ k` is turned into the state $\langle S, \Sigma, k\rangle_{\text{sk}}$, while `disp_kr k r` corresponds to $\langle k, r\rangle_{\text{kr}}$.

Each path in the code of Section 3.3 produces a step of the abstract machine, where the resulting state corresponds to the tail-call. For instance, the LetIn case of the pseudo-code generates the following step:

$$\langle \texttt{LetIn}(p, S_1, S_2), \Sigma, k\rangle_{\text{sk}} \longrightarrow \langle S_1, \Sigma, \texttt{KRLet}(p, S_2, \Sigma, k)\rangle_{\text{sk}}$$

$$\begin{aligned}
\langle \text{kr}_{\text{id}}, r\rangle_{\text{kr}} &\twoheadrightarrow \texttt{(* end of computation *)}\\
\langle \texttt{KRLet}(p, S, \Sigma, k), r\rangle_{\text{kr}} &\longrightarrow \langle p, r, \Sigma, \texttt{KELet}(S, k)\rangle_{\text{pat}}\\
\cdots &\longrightarrow \cdots\\
\langle \texttt{KELet}(S, k), \Sigma\rangle_{\text{ke}} &\longrightarrow \langle S, \Sigma, k\rangle_{\text{sk}}\\
\cdots &\longrightarrow \cdots\\
\langle \texttt{Branching}(l), \Sigma, k\rangle_{\text{sk}} &\longrightarrow \langle S, \Sigma, k\rangle_{\text{sk}} \quad \texttt{for } (S \in l)\\
\langle \texttt{LetIn}(p, S_1, S_2), \Sigma, k\rangle_{\text{sk}} &\longrightarrow \langle S_1, \Sigma, \texttt{KRLet}(p, S_2, \Sigma, k)\rangle_{\text{sk}}
\end{aligned}$$

Figure 2. Non-Deterministic Abstract Machine

This correctly produces a stepping relation. Some of the steps of the NDAM are given in Figure 2.

The full abstract machine is available in Appendix B.4 of the supplementary material.

The initial states of the NDAM correspond to injections $\langle t, [], \text{kr}_{\text{id}}\rangle_{\text{trm}}$ and $\langle S, [], \text{kr}_{\text{id}}\rangle_{\text{sk}}$ for evaluating respectively a skelterm t and a skeleton S. The result r of the evaluation is given by a state of the form $\langle \text{kr}_{\text{id}}, r\rangle_{\text{kr}}$.

For all states a and b, we write $a \longrightarrow^* b$ for the reflexive and transitive closure of the stepping relation of the abstract machine, and $a \longrightarrow^n b$ for a sequence of n steps with $n \in \mathbb{N}$.

3.5 Certification

The NDAM is formalized in Coq in the file `Concrete_ndam.v` (see artifact), and we prove this NDAM to be sound and complete with respect to the concrete interpretation. The Coq proof is done at the meta level (parametric in the skeletal semantics), as such it is valid independently from the language we are interested in (e.g., λ-calculus).

Intuitively, each big-step relation of the concrete interpretation (see Figure 1) corresponds to an evaluation mode of the NDAM. For instance, we have:

$$\Sigma \vdash S \Downarrow_{\text{s}} r \;\text{ iff }\; \langle S, \Sigma, \text{kr}_{\text{id}}\rangle_{\text{sk}} \longrightarrow^* \langle \text{kr}_{\text{id}}, r\rangle_{\text{kr}}$$

We call kr_{id}, ke_{id}, and kl_{id} the *basic continuations* of the abstract machine. States stuck evaluating a basic continuation, e.g., $\langle \text{kr}_{\text{id}}, r\rangle_{\text{kr}}$, are called *final states*.

Since the NDAM also manipulates reified continuations, we need a few results to reshape them when appropriate. We first define *continuation composition* $k[k']$. Given two continuations k and k', we write $k[k']$ for the substitution of the basic continuation inside k by k', defined as follows:

$$\begin{aligned}
\text{kr}_{\text{id}}[k'] &\triangleq k'\\
\texttt{KRLet}(p, S, \Sigma, k)[k'] &\triangleq \texttt{KRLet}(p, S, \Sigma, k[k'])\\
\cdots &\triangleq \cdots
\end{aligned}$$

This continuation composition can be naturally extended to machine states, e.g., $\langle S, \Sigma, k\rangle_{\text{sk}}[k'] \triangleq \langle S, \Sigma, k[k']\rangle_{\text{sk}}$.

Because the NDAM never pattern-matches on basic continuations, machine steps are preserved by continuation composition.

Lemma 3.1. For all a, b, and k, $a \to b$ implies $a[k] \to b[k]$.

This lemma is sufficient to prove the completeness part. Henceforth, we state lemmas and theorems for the skeleton mode of the machine, but they can be stated similarly for the other modes.

Theorem 3.2. For all S, Σ, and r, if $\Sigma \vdash S \Downarrow_s r$, then $\langle S, \Sigma, \mathrm{kr_{id}}\rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r\rangle_{\mathrm{kr}}$.

From the concrete interpretation to the NDAM, we proceed by structural induction on the inductive properties of the concrete interpretation (Figure 1), and simply apply the induction hypothesis and merge sequences, using Lemma 3.1.

Sketch. For instance, for the constructor LetIn:

$$\frac{\Sigma \vdash S_1 \Downarrow_s r' \qquad \Sigma + \{p \mapsto r'\} \Downarrow_p \Sigma' \qquad \Sigma' \vdash S_2 \Downarrow_s r}{\Sigma \vdash \mathtt{LetIn}(p, S_1, S_2) \Downarrow_s r}$$

Applying the induction hypothesis on each premise gives us:

(1) $\langle S_1, \Sigma, \mathrm{kr_{id}}\rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r'\rangle_{\mathrm{kr}}$
(2) $\langle p, r', \Sigma, \mathrm{ke_{id}}\rangle_{\mathrm{pat}} \to^* \langle \mathrm{ke_{id}}, \Sigma'\rangle_{\mathrm{ke}}$
(3) $\langle S_2, \Sigma', \mathrm{kr_{id}}\rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r\rangle_{\mathrm{kr}}$

We then merge them to reconstruct a sequence, held by a few additional steps. The identity continuations are lifted using Lemma 3.1 when necessary.

$$\begin{array}{lll}
& \langle \mathtt{LetIn}(p, S_1, S_2), \Sigma, \mathrm{kr_{id}}\rangle_{\mathrm{sk}} & \\
\to & \langle S_1, \Sigma, \mathtt{KRLet}(p, S_2, \Sigma, \mathrm{kr_{id}})\rangle_{\mathrm{sk}} & \mathtt{Definition} \\
\to^* & \langle \mathtt{KRLet}(p, S_2, \Sigma, \mathrm{kr_{id}}), r'\rangle_{\mathrm{kr}} & (1) + \mathtt{Lemma} \\
\to & \langle p, r', \Sigma, \mathtt{KELet}(S_2, \mathrm{kr_{id}})\rangle_{\mathrm{pat}} & \mathtt{Definition} \\
\to^* & \langle \mathtt{KELet}(S_2, \mathrm{kr_{id}}), \Sigma'\rangle_{\mathrm{ke}} & (2) + \mathtt{Lemma} \\
\to & \langle S_2, \Sigma', \mathrm{kr_{id}}\rangle_{\mathrm{sk}} & \mathtt{Definition} \\
\to^* & \langle \mathrm{kr_{id}}, r\rangle_{\mathrm{kr}} & (3)
\end{array}$$

□

For the reverse implication, we make explicit where continuations are needed by splitting sequences of steps at the point where the continuation is actually focused on. Up to that point, the continuation could be replaced by a basic continuation, as stated in the following result. If the sequence does not use the continuation (e.g., $\langle \mathtt{LetIn}(p, S_1, S_2), \Sigma, k\rangle_{\mathrm{sk}} \to \langle S_1, \Sigma, \mathtt{KRLet}(p, S_2, \Sigma, k)\rangle_{\mathrm{sk}}$), there is no splitting point for us to exploit. To avoid this case, the lemma assumes the sequence leads to a final state, ensuring the continuation is used.

Lemma 3.3. For all S, Σ, k, n, and final state b, if we have $\langle S, \Sigma, k\rangle_{\mathrm{sk}} \to^n b$ then there exist n_1, n_2, and r such that $\langle S, \Sigma, \mathrm{kr_{id}}\rangle_{\mathrm{sk}} \to^{n_1} \langle \mathrm{kr_{id}}, r\rangle_{\mathrm{kr}}$, $\langle k, r\rangle_{\mathrm{kr}} \to^{n_2} b$, and $n = n_1 + n_2$.

The proof is by strong induction on the length of the sequence of steps. Intuitively, it holds because we stop at the first step pattern-matching the initial continuation; the steps before the cut are still valid after changing the unused continuation.

With this lemma, we can tackle the soundness part of the certification.

Theorem 3.4. For all S, Σ, and r, if $\langle S, \Sigma, \mathrm{kr_{id}}\rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r\rangle_{\mathrm{kr}}$, then $\Sigma \vdash S \Downarrow_s r$.

We proceed by strong induction on the length of the sequence of steps, and make use of Lemma 3.3.

Sketch. For instance, for the constructor LetIn, we start with:

$$\langle \mathtt{LetIn}(p, S_1, S_2), \Sigma, \mathrm{kr_{id}}\rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r\rangle_{\mathrm{kr}}.$$

Using the definition of the NDAM (Figure 2) and Lemma 3.3 several times, we can cut the sequence into the following pieces:

- $\langle S_1, \Sigma, \mathrm{kr_{id}}\rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r'\rangle_{\mathrm{kr}}$
- $\langle p, r', \Sigma, \mathrm{ke_{id}}\rangle_{\mathrm{pat}} \to^* \langle \mathrm{ke_{id}}, \Sigma'\rangle_{\mathrm{ke}}$
- $\langle S_2, \Sigma', \mathrm{kr_{id}}\rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r\rangle_{\mathrm{kr}}$

Each piece is strictly shorter than the initial sequence, so we can apply the induction hypothesis on each of them. We obtain the premises of the LetIn concrete interpretation (cf. Figure 1) so we can conclude. The other cases are similar. □

4 Deriving the Deterministic Abstract Machine

The abstract machine of Section 3 is non-deterministic, and thus does not offer a computable semantics for the empirical evaluation of skelterms and skeletons. In this section, we show how to obtain a deterministic machine using explicit backtracking. The deterministic machine returns at most one of the possible results; computing all of them is impossible in presence of non-terminating executions. We choose to try branches in order and to backtrack if necessary.

To this end, we consider a more complex CPS transformation with a failure continuation (Section 4.1). We then derive the resulting deterministic abstract machine through defunctionalization in Section 4.2, and present the Coq certification of its soundness in Section 4.3.

4.1 CPS-Transform

Starting from the big-step evaluator of Section 3.1, we modify every evaluation function to take two continuations [11]:

- a success continuation `k` as in Section 3.2, indicating what to do if the computation succeeds, and
- a failure continuation `fk`, remembering the last checkpoint to backtrack to in case of failure.

When the evaluator of Section 3.1 chooses between several results using `pick`, we instead decide to always evaluate the first of the possible choices and create a checkpoint for the other possibilities. For branchings, it means evaluating the first branch, and remembering the others in the failure continuation. When we reach an empty branching, it means that

all the evaluations of all the branches failed, and we need to backtrack by calling `fk`. Once again, we use a gray background to indicate changes or new functions, here relative to the initial pseudo-code of Section 3.1.

```
let eval_sk sk e
  (k: cvalue -> (() -> cvalue) -> cvalue)
  (fk: () -> cvalue) : cvalue =
match sk with
| Branching (skl) -> match skl with
  | [] -> fk ()
  | sk'::l ->
    eval_sk sk' e k
    (fun _ -> eval_sk (Branching(l)) e k fk)
| ...
| LetIn (p, sk1, sk2) ->
  eval_sk sk1 e (fun r fk2 ->
  eval_pat p r e (fun e2 fk3 ->
  eval_sk sk2 e2 k fk3) fk2) fk
```

The other source of non-determinism is the evaluation of unspecified skelterms, which returns a list of possible results (see Figure 1). In this case, we create a new evaluation function `select_list` which follows the same principle as for branchings: it sequentially tries the elements of the list, and otherwise calls the failure continuation.

```
let select_list rl
  (k: cvalue -> (() -> cvalue) -> cvalue)
  (fk: () -> cvalue) : cvalue =
match rl with
| [] -> fk ()
| r::l -> k r (fun _ -> select_list l k fk)
```

Also, we complete the pattern-matchings of the pseudo-code to make them total, and backtrack in the problematic cases. Initially, the pseudo-code follows the rules of Figure 1 and does not cover cases that cannot evaluate. For instance, the rules for looking up a variable in an environment assumes the environment to have at least one entry:

```
def lookup e x : cvalue = match e with
| (y,r)::e2 -> if x=y then r
               else lookup e2 x
```

Here, to cover all possible behaviors, we add a case for when the environment is empty. We add similar backtracking cases at several points throughout the pseudo-code.

```
def lookup e x k fk : cvalue = match e with
| [] -> fk ()
| (y,r)::e2 -> if x=y then k r fk
               else lookup e2 x k fk
```

Lastly, we need to pass the current failure continuation as an argument of the success continuation `k`. The reason is that a computation can seemingly succeed at first and fail later on; we would then need to backtrack to checkpoints unknown to `k`. This is obvious for functions such as `select_list`: it succeeds in selecting an element of the list, but the continuation might fail later.

The extra failure continuation does not fundamentally change how deterministic constructors are CPS-transformed, as we can see with the resulting code for `LetIn` in the above `eval_sk` function: we just need to pass along the failure continuation, and be mindful of the new type of success continuations.

The full pseudo-code after CPS transformation is available in Appendix C.1 of the supplementary material.

4.2 Defunctionalization and Abstract Machine

On top of the types `krt`, `klt`, and `ket` of Section 3.3, this defunctionalization phase generates a new type `fkt` and its dispatch function `disp_fk` for failure continuations.

```
type fkt =
| FEmpty
| FSK of skeleton * env * krt * fkt
| FList of (cvalue list) * krt * fkt
let disp_fk fk = match fk with
| FSK(sk, e, k, fk') -> eval_sk sk e k fk'
| FList(rl, k, fk') -> select_list rl k fk'
```

The constructors `FSK` and `FList` correspond to the anonymous functions of Section 4.1 where we construct backtracking checkpoints for trying respectively the arguments of a branching or the list of possible interpretations of an unspecified skelterm. We also create a constructor `FEmpty` representing an empty failure continuation, used to start a computation. It has no rule in the dispatch function, since no backtrack is possible.

```
let select_list rl (k: krt) (fk: fkt): cvalue =
match rl with
| [] -> disp_fk fk
| r::l -> k r (FList (l, k, fk))

let eval_sk sk e (k: krt) (fk: fkt): cvalue =
match sk with
| Branching (skl) -> match skl with
  | [] -> disp_fk fk
  | sk'::l -> eval_sk sk' e k
              (FSK (Branching(l), e, k, fk))
| ...
```

As previously, the new constructors replace the anonymous functions, and we add a call to the dispatch function at every backtrack point.

The full pseudo-code at this point is available in Appendix C.2 of the supplementary material.

$$\langle \mathrm{kr}_{\mathrm{id}}, r, f\rangle_{\mathrm{kr}} \twoheadrightarrow \texttt{(* end *)} \qquad \langle \texttt{FEmpty}\rangle_{\mathrm{fk}} \twoheadrightarrow \texttt{(* fail *)}$$

$$\langle \texttt{KRLet}(p, S, \Sigma, k), r, f\rangle_{\mathrm{kr}} \to \langle p, r, \Sigma, \texttt{KELet}(S, k), f\rangle_{\mathrm{pat}} \qquad \langle \texttt{FSK}(S, \Sigma, k, f)\rangle_{\mathrm{fk}} \to \langle S, \Sigma, k, f\rangle_{\mathrm{sk}}$$

$$\cdots \to \cdots \qquad \langle \texttt{FList}(l, k, f)\rangle_{\mathrm{fk}} \to \langle l, k, f\rangle_{\mathrm{lst}}$$

$$\langle \texttt{KELet}(S, k), \Sigma, f\rangle_{\mathrm{ke}} \to \langle S, \Sigma, k, f\rangle_{\mathrm{sk}} \qquad \cdots \to \cdots$$

$$\cdots \to \cdots \qquad \langle \texttt{Branching}([\,]), \Sigma, k, f\rangle_{\mathrm{sk}} \to \langle f\rangle_{\mathrm{fk}}$$

$$\langle [\,], k, f\rangle_{\mathrm{lst}} \to \langle f\rangle_{\mathrm{fk}} \qquad \langle \texttt{Branching}(S :: l), \Sigma, k, f\rangle_{\mathrm{sk}} \to \langle S, \Sigma, k, \texttt{FSK}(\texttt{Branching}(l), \Sigma, k, f)\rangle_{\mathrm{sk}}$$

$$\langle r :: l, k, f\rangle_{\mathrm{lst}} \to \langle k, r, \texttt{FList}(l, k, f)\rangle_{\mathrm{kr}} \qquad \langle \texttt{LetIn}(p, S_1, S_2), \Sigma, k, f\rangle_{\mathrm{sk}} \to \langle S_1, \Sigma, \texttt{KRLet}(p, S_2, \Sigma, k), f\rangle_{\mathrm{sk}}$$

Figure 3. Deterministic Abstract Machine

Transforming the defunctionalized evaluator produces a deterministic abstract machine, as each machine state is either stuck or reduces to exactly one machine state. Compared to the non-deterministic machine of Section 3.4, the states carry an extra argument corresponding to the failure continuation.

The deterministic machine has two additional modes `lst` and `fk`, which correspond to the functions (`select_list` and `disp_fk`). We present their steps in Figure 3: as expected, the `lst` mode tries the continuation on each element of the list, and triggers the backtracking mode `fk` on the empty list; it is also invoked on an empty branching. The `fk` mode then restores the backtracking checkpoint, unless the failure continuation is empty, in which case the machine stops.

We initialize the machine with either $\langle t, [\,], \mathrm{kr}_{\mathrm{id}}, \texttt{FEmpty}\rangle_{\mathrm{trm}}$ to evaluate a skelterm t or $\langle S, [\,], \mathrm{kr}_{\mathrm{id}}, \texttt{FEmpty}\rangle_{\mathrm{sk}}$ for a skeleton S. There are three possible outcomes:

- the machine gets stuck at $\langle \mathrm{kr}_{\mathrm{id}}, r, f\rangle_{\mathrm{kr}}$, which means r is a result (there might be other correct results, but we stop at the first);
- it gets stuck at $\langle \texttt{FEmpty}\rangle_{\mathrm{fk}}$, which means all possible branches have been tried and there is no result;
- the evaluation does not terminate, as the abstract machine can loop on an infinite branch (there might be correct results on other branches).

The complete deterministic abstract machine is available in Appendix C.3 of the supplementary material.

4.3 Certification

The deterministic Abstract Machine (AM) is formalized in Coq in the file `Concrete_am.v` (see artifact), and once again the certification is independent from the skeletal semantics (language) we are interested in.

We prove in Coq that the AM is sound with respect to the NDAM: if the AM finds a result, then the NDAM can also find the same result. Because the NDAM is sound with respect to the concrete interpretation, so is the AM. However it is not complete: the AM can find at most one of the results, and may loop in an infinite branch even if there is a valid result elsewhere.

To avoid confusion, the stepping relations of the abstract machines are written ($\to_{\mathrm{AM}}$) and ($\to_{\mathrm{ND}}$) in this section. We note $\widetilde{x}$ for a sequence $(x_1, \ldots, x_n)$, $\langle \widetilde{x}\rangle_{\mathrm{m}}$ for a NDAM state of mode m, and $\langle \widetilde{x}, f\rangle_{\mathrm{m}}$ for an AM state of mode m—failure continuations are always the last argument in the deterministic case.

Unlike in Section 3.5, both semantics use success continuations so there is no need for lemmas to manipulate them. However, only the AM uses failure continuations, so we do need a few results to handle them.

Firstly, we define *failure continuation composition* $f[f']$ which replaces the empty failure continuation in f by f'.

$$\begin{aligned} \texttt{FEmpty}[f'] &\triangleq f' \\ \texttt{FSK}(S, \Sigma, k, f)[f'] &\triangleq \texttt{FSK}(S, \Sigma, k, f[f']) \\ \texttt{FList}(l, k, f)[f'] &\triangleq \texttt{FList}(l, k, f[f']) \end{aligned}$$

We extend it to AM states so that $\langle \widetilde{x}, f\rangle_{\mathrm{m}}[f'] \triangleq \langle \widetilde{x}, f[f']\rangle_{\mathrm{m}}$.

As previously, steps hold after composition:

Lemma 4.1. For all a, b, and f, $a \to_{\mathrm{AM}} b$ implies $a[f] \to_{\mathrm{AM}} b[f]$.

Secondly, we prove another lemma to discard failure continuations. If a sequence never uses it, we can remove it from both the head and tail states. Otherwise, we can split this sequence at the point where it is called, and the first part can be written without using the failure continuation.

Lemma 4.2. For all n, AM mode m, and states $\langle \widetilde{x}, f\rangle_{\mathrm{m}}$ and b, if $\langle \widetilde{x}, f\rangle_{\mathrm{m}} \to^n_{\mathrm{AM}} b$ then either:

- there exists b' such that $\langle \widetilde{x}, \texttt{FEmpty}\rangle_{\mathrm{m}} \to^n_{\mathrm{AM}} b'$ and $b = b'[f]$, or
- there exist n_1 and n_2 such that $\langle \widetilde{x}, \texttt{FEmpty}\rangle_{\mathrm{m}} \to^{n_1}_{\mathrm{AM}} \langle \texttt{FEmpty}\rangle_{\mathrm{fk}}$, $\langle f\rangle_{\mathrm{fk}} \to^{n_2}_{\mathrm{AM}} b$, and $n = n_1 + n_2$.

The proof is done by strong induction on the length of the sequence of steps, and uses a few basic results about composition, such as associativity.

Lastly, we can pose our main theorem. It states that the AM is sound with respect to the NDAM.

Theorem 4.3. For all l, k, r, and f, if $\langle l, k, \texttt{FEmpty}\rangle_{\mathrm{lst}} \to^*_{\mathrm{AM}} \langle \mathrm{kr}_{\mathrm{id}}, r, f\rangle_{\mathrm{kr}}$ then there exists $r' \in l$ such that $\langle k, r'\rangle_{\mathrm{kr}} \to^*_{\mathrm{ND}} \langle \mathrm{kr}_{\mathrm{id}}, r\rangle_{\mathrm{kr}}$.

For all NDAM mode m (i.e., other than `lst` and `fk`), for all $\langle \widetilde{x}, \texttt{FEmpty} \rangle_{\mathrm{m}}$, r, and f, if $\langle \widetilde{x}, \texttt{FEmpty} \rangle_{\mathrm{m}} \rightarrow^*_{\mathrm{AM}} \langle \mathrm{kr}_{\mathrm{id}}, r, f \rangle_{\mathrm{kr}}$ then $\langle \widetilde{x} \rangle_{\mathrm{m}} \rightarrow^*_{\mathrm{ND}} \langle \mathrm{kr}_{\mathrm{id}}, r \rangle_{\mathrm{kr}}$.

In particular, for the mode evaluating skeletons `sk`, the following holds.

Corollary 4.4. For all S, Σ, r, and f, if

$$\langle S, \Sigma, \mathrm{kr}_{\mathrm{id}}, \texttt{FEmpty} \rangle_{\mathrm{sk}} \rightarrow^*_{\mathrm{AM}} \langle \mathrm{kr}_{\mathrm{id}}, r, f \rangle_{\mathrm{kr}},$$

then

$$\langle S, \Sigma, \mathrm{kr}_{\mathrm{id}} \rangle_{\mathrm{sk}} \rightarrow^*_{\mathrm{ND}} \langle \mathrm{kr}_{\mathrm{id}}, r \rangle_{\mathrm{kr}}.$$

The mode `lst` is treated differently in the theorem because there is no corresponding mode in the NDAM. It amounts to choosing the first cvalue $r' \in l$ that would produce a result. In the NDAM, we would have an oracle picking the right element of l to evaluate.

The proof is done by strong induction on the sequence of the AM. We follow the AM sequence, and most steps correspond to similar steps in the NDAM. We restrict ourselves to sequences with an empty failure continuation to make use of the induction hypothesis. When the AM would create a checkpoint, we use Lemma 4.2; a case disjunction on the result allows us to pick the correct branch for the NDAM and keep an empty failure continuation.

Remark 4.5. In the proof of Theorem 4.3, an AM state with an empty failure continuation and the corresponding NDAM state have exactly the same success continuation. This is why we no longer need the continuation composition of Section 3.5 (and the associated lemmas) to handle them.

Finally, since the NDAM is sound w.r.t. the concrete interpretation, so is the AM.

Theorem 4.6. For all S, Σ, r, and f, if

$$\langle S, \Sigma, \mathrm{kr}_{\mathrm{id}}, \texttt{FEmpty} \rangle_{\mathrm{sk}} \rightarrow^*_{\mathrm{AM}} \langle \mathrm{kr}_{\mathrm{id}}, r, f \rangle_{\mathrm{kr}},$$

then

$$\Sigma \vdash S \Downarrow_{\mathrm{s}} r.$$

Proof. From Theorems 4.3 and 3.4. □

5 OCaml interpreter

We show how to use the deterministic abstract machine of Section 4 to automatically generate a certified OCaml interpreter from a skeletal description of a language.

As stated before, the two abstract machines have been implemented and certified in Coq. Since the NDAM is not computable, it is coded as a relation:

```
Inductive step: state -> state -> Prop
```

However, the AM is computable and coded as a partial function. The only states not being mapped are the final states (i.e., blocked on $\mathrm{kr}_{\mathrm{id}}$/$\mathrm{kl}_{\mathrm{id}}$/$\mathrm{ke}_{\mathrm{id}}$ or `FEmpty`):

```
Definition step (a: state) : option state
```

This stepping function can be repeated to form a partial evaluation function. Since Coq only accepts terminating functions, we need a fuel parameter (i.e., maximum number of steps).

```
Fixpoint evalfuel (n: nat) (a: state) : option cvalue
```

This function extracts the result r of a final state $\langle \mathrm{kr}_{\mathrm{id}}, r, f \rangle_{\mathrm{kr}}$, it fails on $n = 0$ or $a = \langle \texttt{FEmpty} \rangle_{\mathrm{fk}}$, and it steps then recurses otherwise.

From these definitions, using Coq extraction to OCaml [15], we generate an executable version of the deterministic abstract machine. This extraction is only needed to be done once and it is independent of the target language. The output has been slightly reorganized to make better use of OCaml modules.

At this point, the generated OCaml functions still need to be instantiated with an actual skeletal semantics. To this end, we use the Skel tool to deeply embed the semantics into Coq, which we then extract to OCaml. A small script is provided to automatically transform a skeletal semantics into an OCaml module containing the deterministic abstract machine specialized for the given language.

To exploit this module, a user needs to provide OCaml types and functions for the interpretation of unspecified types and skelterms (e.g., `ident`, `env`, `extEnv`, and `getEnv` for the λ-calculus of Section 2.1). The user then has access to the extracted functions, notably `evalfuel` to launch an evaluation.

The soundness proofs of Sections 3.5 and 4.3 are parameterized by the used skeletal semantics. As such, they are valid for any language. So this OCaml interpreter, extracted from the AM and specialized with a language, is sound with respect to the initial concrete interpretation of Section 2.3. If an execution of `evalfuel` produces a result, then this result is a correct behavior of the skeletal semantics. Once again, if the interpreter does not produce a result, we have no guaranties: the interpreter might need more fuel, it might be stuck in an infinite loop, or there might be no correct result at all.

The advantage of working at the meta-level, i.e., proving correction once and for all languages, has a drawback: the execution happens in the meta-language, namely Skel, while a user may prefer working at the level of the language, e.g., λ-terms. This deep embedding requires the user to understand the Skel meta-language. Using OCaml macros can alleviate notations, but cannot resolve fully this gap. Furthermore, an abstract machine for the language itself would be more efficient than the abstract machine for the meta-language.

As examples, we instantiated this meta-interpreter with different languages. The main ones are an imperative language with mutable state, and an extended lambda-calculus with features (pairs, fix-point recursion, etc.). The specialized interpreters, as well as the rest of this work, are available online [4].

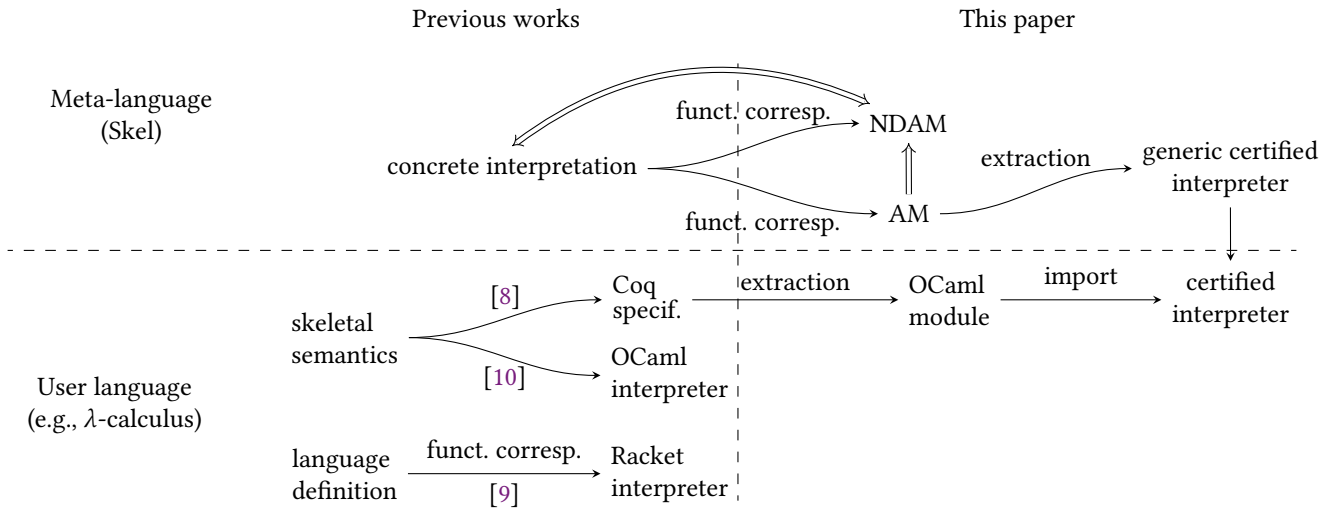

Figure 4. Summary of the paper and comparison with related work

6 Related Work

The technique of functional correspondence, introduced in [1], has been manually applied to many languages with many different features [2, 3, 5–7, 12, 13, 16], showing its robustness and usefulness. Automatic application of the technique is only very recent. Buszka and Biernacki [9] present an algorithm and a tool to automatically generate abstract machines from evaluators, based on the same transformations with a more complex static analysis. We probably could have used their tool to create the NDAM, but the AM would have required the introduction of backtracking by hand. Their approach is general, in the sense that they can automatically generate an abstract machine for any language. On our side, we created a single AM for the meta-language Skel, that can then be automatically specialized for any language. They did not certify their tool, while we prove the soundness of our AM in Coq.

In [10], the authors propose a tool to automatically generate a shallow embedding of a skeletal semantics into an OCaml interpreter. Skel types are directly translated into OCaml types, and skelterms into OCaml functions. Because the meta-language Skel is completely transparent, the interpreter is at the level of the language we are interested in, and more intuitive to use. However, unlike our approach, their ad-hoc translation is not certified and offers no guarantees on the behavior of the interpreter.

7 Conclusion

We present two new semantics for the meta-language Skel of skeletal semantics, in the form of a non-deterministic and a deterministic abstract machine. They are derived from the initial big-step semantics using known transformation techniques. A novelty of our approach is to use these classic tools (CPS transformation and defunctionalization) at the meta-level. This yields a generic abstract machine than can be proved sound once and for all, independently of the input language.

We implement the machines in the Coq proof assistant to certify their soundness. Using previous tools and the Coq extraction mechanism, we can automatically generate a certified OCaml interpreter for the deterministic abstract machine specialized for any skeletal semantics. This can be used as a certified interpreter for any language written using skeletal semantics. We summarize the transformations needed to reach our goal in Figure 4.

As a future work, we would like to create a deterministic abstract machine in the form of a breadth-first search of all possible behaviors. Unlike the one presented in this paper, it would not risk being stuck in a loop, ensuring it to eventually find a result if there is one.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is partially supported by PHC Polonium.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*. ACM, 8–19. https://doi.org/10.1145/888251.888254

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2004. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inf. Process. Lett.* 90, 5 (2004), 223–232. https://doi.org/10.1016/j.ipl.2004.02.012

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2005. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theor. Comput. Sci.* 342, 1 (2005), 149–172. https://doi.org/10.1016/j.tcs.2005.06.008

[4] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt. 2021. Certified Abstract Machines for Skeletal Semantics. https://skeletons.inria.fr/cpp2022/

[5] Malgorzata Biernacka, Dariusz Biernacki, Witold Charatonik, and Tomasz Drab. 2020. An Abstract Machine for Strong Call by Value. In *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12470)*, Bruno C. d. S. Oliveira (Ed.). Springer, 147–166. https://doi.org/10.1007/978-3-030-64437-6_8

[6] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. 2005. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *Log. Methods Comput. Sci.* 1, 2 (2005). https://doi.org/10.2168/LMCS-1(2:5)2005

[7] Dariusz Biernacki and Olivier Danvy. 2003. From Interpreter to Logic Engine by Defunctionalization. In *Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3018)*, Maurice Bruynooghe (Ed.). Springer, 143–159. https://doi.org/10.1007/978-3-540-25938-1_13

[8] Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. 2019. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.* 3, POPL (2019), 44:1–44:31. https://doi.org/10.1145/3290357

[9] Maciej Buszka and Dariusz Biernacki. 2021. Automating the Functional Correspondence between Higher-Order Evaluators and Abstract Machines, In LOPSTR 2021. *CoRR* abs/2108.07132. arXiv:2108.07132 https://arxiv.org/abs/2108.07132

[10] Nathanaël Courant, Enzo Crance, and Alan Schmitt. 2019. Necro: Animating Skeletons. In *ML 2019*. Berlin, Germany.

[11] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*. ACM, 151–160. https://doi.org/10.1145/91556.91622

[12] Olivier Danvy and Jacob Johannsen. 2010. Inter-deriving semantic artifacts for object-oriented programming. *J. Comput. Syst. Sci.* 76, 5 (2010), 302–323. https://doi.org/10.1016/j.jcss.2009.10.004

[13] Wojciech Jedynak, Malgorzata Biernacka, and Dariusz Biernacki. 2013. An operational foundation for the tactic language of Coq. In *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, Ricardo Peña and Tom Schrijvers (Eds.). ACM, 25–36. https://doi.org/10.1145/2505879.2505890

[14] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2021. *The OCaml system: Documentation and user's manual*. https://ocaml.org/manual/

[15] Pierre Letouzey. 2002. A New Extraction for Coq. In *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers (Lecture Notes in Computer Science, Vol. 2646)*, Herman Geuvers and Freek Wiedijk (Eds.). Springer, 200–219. https://doi.org/10.1007/3-540-39185-1_12

[16] Maciej Piróg and Dariusz Biernacki. 2010. A systematic derivation of the STG machine verified in Coq. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, Jeremy Gibbons (Ed.). ACM, 25–36. https://doi.org/10.1145/1863523.1863528

[17] Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

[18] John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, John J. Donovan and Rosemary Shields (Eds.). ACM, 717–740. https://doi.org/10.1145/800194.805852

[19] The Coq Development Team. 2021. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.4501022