Certified Abstract Machines for Skeletal Semantics

Guillaume Ambal\textsuperscript{1}, Sergueï Lenglet\textsuperscript{2}, and Alan Schmitt\textsuperscript{3}

\textsuperscript{1} Univ Rennes, Rennes, France
guillaume.ambal@irisa.fr
\textsuperscript{2} Université de Lorraine, Nancy, France
serguei.lenglet@univ-lorraine.fr
\textsuperscript{3} Inria, Rennes, France
alan.schmitt@inria.fr

Skeletal Semantics. Skeletal semantics \cite{Skel} is a framework, relying on a meta-language named Skel, to formalize the operational semantics of programming languages. It is based on a general and systematic way to break down the semantics of a language. 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.

The OCaml \cite{OCaml} implementation benefits from a toolbox called Necro. It can generate an OCaml interpreter \cite{OCaml interp} or a Coq \cite{Coq} mechanization \cite{Skel} of a skeletal semantics given as input.

Skeletal semantics is a framework generic enough to express any semantics which can be written with inductive rules. For instance, consider a call-by-value \(\lambda\)-calculus with closures. The syntax \(t ::= \lambda x.t | x | t t\) and rules of the form \(s,\lambda x.t \Downarrow (x,t,s)\) can be represented as follows.

```ocaml
val eval : env \* lterm \rightarrow clos

type ident

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

val eval : env \* lterm \rightarrow clos

let Lam (x, t) = l in
Clos (x, t, s)
```

The syntax is represented by unspecified types (e.g., \texttt{ident}) and algebraic data types (e.g., \texttt{lterm}). The semantics is represented by evaluation functions (e.g., \texttt{eval}) defined using \textit{skeletons}.

To make sense of this representation, we attribute a meaning to skeletons, called an interpretation. The main one, called concrete interpretation, is written in a non-deterministic big-step style and formalized in Coq. While useful to prove some properties of a language or of programs, the concrete interpretation cannot reason about non-terminating programs and it is quite far from an actual implementation. We thus propose alternate interpretations, in the form of non-deterministic and deterministic abstract machines, derived using functional correspondence \cite{Functional correspondence}.

These new interpretations are proved sound in relation to the concrete interpretation, and we use the deterministic version to generate a certified generic OCaml interpreter.

This work summarizes a paper recently accepted at CPP 2022 \cite{CPP}, and our results are outlined in Figure 1.

Functional Correspondence. Functional correspondence \cite{Functional correspondence} is a systematic strategy for transforming functional evaluators (i.e., big-step interpreters) into equivalent abstract machines. This approach combines several known transformations. The main phases of the derivation are a CPS-transformation \cite{CPS}, a phase of defunctionalization \cite{Defunctionalization}, and then the proper creation of the abstract machine and its evaluation modes.
Previous works | This work
---|---
Meta-language (Skel) | funct. | funct. | NDAM
concrete interpretation | corresp. | extraction | generic certified interpreter
User language (e.g., λ-calculus) | funct. | corresp. | AM
skeletal semantics | | extraction | OCaml module
language definition | | | import | certified interpreter

**Figure 1: Summary of our work and comparison with related work**

The technique of functional correspondence has been manually applied to many languages with many different features [5, 2, 16, 7, 6, 3, 12, 13], showing its robustness and usefulness. Recently, a tool was also developed for the automatic application of the technique [9].

**Abstract Machines.** We apply by hand the standard strategy of functional correspondence on the concrete interpretation of skeletal semantics (i.e., the big-step semantics of the meta-language Skel). Since the input semantics is non-deterministic, we obtain a non-deterministic abstract machine (NDAM) for Skel. While the transformation is classic, a novelty of our approach is to use it at the meta-level. This yields a generic abstract machine that can be proved sound once and for all, independently of the input language.

To create a deterministic executable version (AM), we proceed similarly but use a more involved CPS-transformation with two continuations [11], allowing for checkpoints and backtracking during a computation.

The concrete interpretation was already defined in the Coq proof assistant. Our two new abstract machine interpretations are formalized in Coq, and we certify them independently from the skeletal semantics (language) we are interested in. First, we prove that the NDAM is equivalent to the standard concrete interpretation. Second, the AM is proved sound with respect to the NDAM, i.e., if the AM finds a result, then the NDAM can also find the same result. The AM does not necessarily find a result, as it can get stuck in an infinite computation. By transitivity, the AM is also sound w.r.t. the concrete interpretation.

**Certified Interpreter.** Using the Coq extraction mechanism [15] on the deterministic 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.

The advantage of working at the meta-level, i.e., proving correctness once and for all languages, has a drawback: the execution happens in the meta-language, namely Skel. In contrast, the previous tool [10] produces a more efficient OCaml interpreter and allows the user to work at the level of the language, e.g., λ-terms, but without any guarantees.
References


