Certified Abstract Machines for Skeletal Semantics

Guillaume Ambal[1], Sergueï Lenglet[2], and Alan Schmitt[3]

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

Skeletal Semantics. Skeletal semantics [8] 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 [14] implementation benefits from a toolbox called Necro. It can generate an OCaml interpreter [10] or a Coq [19] mechanization [8] 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 λ-calculus with closures. The syntax $t ::= \lambda x.t \mid x \mid t\ t$ and rules of the form $s, \lambda x.t \Downarrow (x, t, s)$ can be represented as follows.

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

```
val eval (s : env) (l : lterm) : clos =
  branch
    let Lam (x, t) = l in
    Clos (x, t, s)
  or ...
  end
```

The syntax is represented by unspecified types (e.g., `ident`) and algebraic data types (e.g., `lterm`). The semantics is represented by evaluation functions (e.g., `eval`) defined using *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 [1]. 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 [4], and our results are outlined in Figure 1.

Functional Correspondence. Functional correspondence [1] 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 [17], a phase of defunctionalization [18], and then the proper creation of the abstract machine and its evaluation modes.

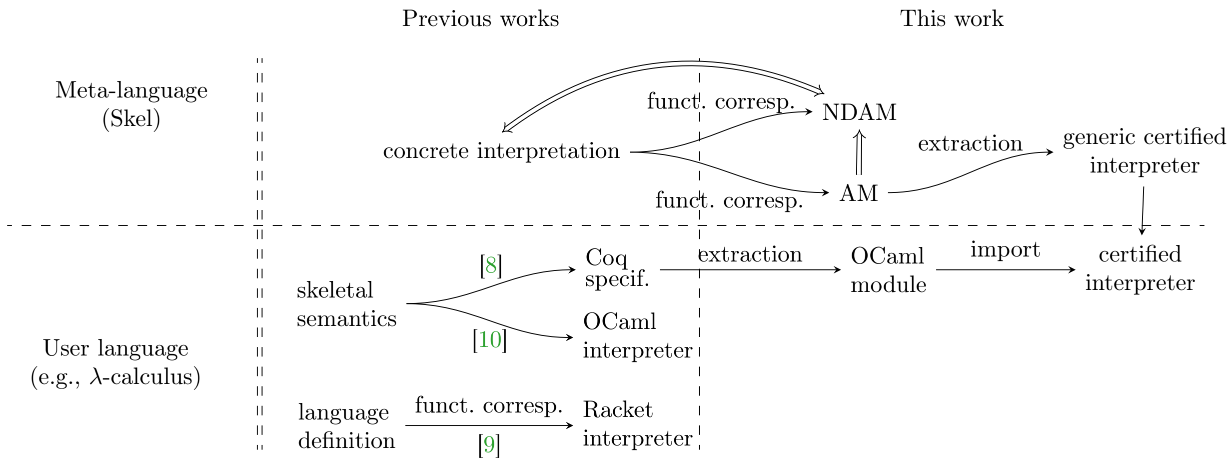

Figure 1: Summary of our work and comparison with related work

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

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 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*, pages 8–19. ACM, 2003.

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

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

[4] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt. Certified abstract machines for skeletal semantics. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 55–67. ACM, 2022.

[5] Malgorzata Biernacka, Dariusz Biernacki, Witold Charatonik, and Tomasz Drab. An abstract machine for strong call by value. In Bruno C. d. S. Oliveira, editor, *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*, volume 12470 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 2020.

[6] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Log. Methods Comput. Sci.*, 1(2), 2005.

[7] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers*, volume 3018 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2003.

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

[9] Maciej Buszka and Dariusz Biernacki. Automating the functional correspondence between higher-order evaluators and abstract machines. In *LOPSTR 2021*, volume abs/2108.07132, 2021.

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

[11] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, pages 151–160. ACM, 1990.

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

[13] Wojciech Jedynak, Malgorzata Biernacka, and Dariusz Biernacki. An operational foundation for the tactic language of coq. In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 25–36. ACM, 2013.

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

[15] Pierre Letouzey. A new extraction for coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2002.

[16] Maciej Piróg and Dariusz Biernacki. A systematic derivation of the STG machine verified in coq. In Jeremy Gibbons, editor, *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, pages 25–36. ACM, 2010.

[17] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[18] John C. Reynolds. Definitional interpreters for higher-order programming languages. In John J. Donovan and Rosemary Shields, editors, *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, pages 717–740. ACM, 1972.

[19] The Coq Development Team. The coq proof assistant, January 2021.