Necro: Animating Skeletons

Nathanaël Courant Enzo Crance Alan Schmitt

June 22, 2020

Abstract

`necro` is a tool that generates an `OCaml` interpreter given a semantics in a specific format.

1 Introduction

Skeletal Semantics [1] is a recent framework designed to describe the semantics of programming languages. More precisely, it provides a syntax to describe the structural elements of a semantics (sequence, choice, and recursion). For instance, here is a rule for the `if` constructor.

$$\mathit{if}\ x_{t_1}\ x_{t_2}\ x_{t_3} \coloneqq \left[H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \begin{pmatrix} \texttt{isTrue}\left(x_{f_2}\right); H\left(x_\sigma, x_{t_2}, x_o\right) \\ \texttt{isFalse}\left(x_{f_2}\right); H\left(x_\sigma, x_{t_3}, x_o\right) \end{pmatrix} \right]$$

The `if` constructor takes three subterms, denoted by the term variables x_{t_1}, x_{t_2}, and x_{t_3}. To run it, one first recursively evaluate x_{t_1} in the starting environment, denoted by x_σ, and bind the result to x_{f_1}, as denoted by $H(x_\sigma, x_{t_1}, x_{f_1})$. Note that this is just a notation: H simply states that a recursive evaluation takes place, but it does not give the meaning of this evaluation. Then we reach a *choice* between two branches. The first branch starts with a *filter* `isTrue` that checks whether x_{f_1} is true, if so the second subterm is recursively evaluated. The second branch is similar, testing whether x_{f_1} is false, and if so evaluating the third subterm. In both cases, the result is bound to the special variable x_o that should hold the final result.

To formally define a skeletal semantics, one need to provide the following ingredients:

- *term sorts*, comprising *base sorts* that correspond to base terms (integers, identifiers, ...) and *program sorts* that correspond to constructed terms;

- *constructors*, with their *signature* that map list of term sorts to program sorts;
- *flow sorts*, that represent the values manipulated during an execution;
- *filters*, with their *signature* that map list of sorts to list of sorts;
- for each program sort, a pair of flow sorts indicating the sort of the state in which it is evaluated and the sort of the result of its evaluation;
- for each constructor, a *skeleton*, that describes the structure of its evaluation, as shown above for the `if` constructor.

The `necro` tool takes a semantics described in this format and generates an `OCaml` interpreter for it.

2 A Syntax for Skeletons

The first contribution of the `necro` tool is a syntax to describe skeletons. We describe it through simplified excerpts of an example for a While language. The full definition is available online.[1] We first declare the base sorts and flow sorts.

```
type ident                type state
type lit                  type value
```

We then declare conjointly the program sorts, their constructors, and their signatures.

```
type expr =                   type stat =
| Const of lit                | Skip
| Var of ident                | Assign of ident * expr
| Plus of expr * expr         | Seq of stat * stat
| Equal of expr * expr        | If of expr * stat * stat
| Not of expr                 | While of expr * stat
```

We next describe filters. Here are the ones we need for the `if` rule.

```
val isTrue : value -> unit
val isFalse : value -> unit
```

[1] `https://gitlab.inria.fr/skeletons/necro/blob/master/test/while_rules.txt`

We depart from the formalization of [1] by defining one hook per program sort. We express through annotations what state is expected for this program sort, and what output it produces. We initially used a single hook, but this made the generated code quite complex as we needed GADTs to be able to have different output types.

```
hook expr (st : state) : expr -> value =
...

hook stat (st : state) : stat -> state =
...
| If (cond, if_true, if_false) ->
  b <- expr st cond;
  r <- branch
    isTrue b;
    r <- stat st if_true
  or
    isFalse b;
    r <- stat st if_false
  end;
  r
...
```

In our syntax, a skeleton is a sequence of hooks and filters, of the form `output <- name input`, and branches, of the form `output <- branch ... or ... or ... end`. If a filter returns `unit`, then the `output <-` part is omitted. We no longer have special names such as x_σ or x_o, instead a hook takes an input that corresponds to the input environment (`st` in the example above), and every skeleton ends with the name of the variable holding the final result (`r` in the example above).

3 Generated Code

The general workflow of the `necro` tool is given in Figure 1. The boxes in yellow are written by the user, and the boxes in green are generated or instantiated. First, the user gives the skeletal definition to `necro`, which produces type definitions for the syntax of the language, a module type `FLOW` comprising the required base sorts, flow sorts, and the filters, and a functor `MakeInterpreter` that expects a module of type `FLOW` containing

the implementation of the filters and that returns evaluation functions.[2] To evaluate branching, we return the first branch that completes successfully, and raise an exception if no branch completes. The user can then use this module to obtain an interpreter for their language.[3]

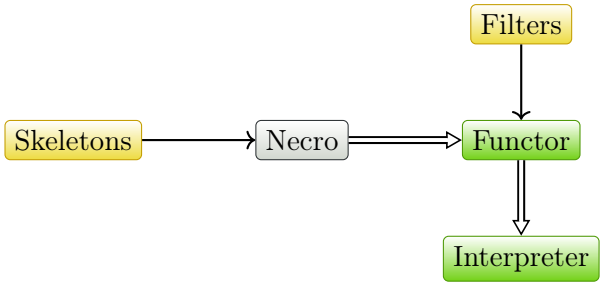

Figure 1: Workflow

4 Evaluation

We have tested `necro` with a While language, a call-by-value λ-calculus, PCF, mini-ML, and a subset of WebAssembly [2]. We are extending it to a k-CFA analyser: given a skeletal semantics and the definition of constraints for filters, we generate an `OCaml` tool that takes as input a program in the defined semantics and that generates and solves constraints corresponding to a k-CFA analysis. We plan to demo both tools during the presentation.

References

[1] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. Skeletal Semantics and their Interpretations. *Proceedings of the ACM on Programming Languages*, 44:1–31, 2019. Companion website with Coq development: `http://skeletons.inria.fr/popl2019/index.html`.

[2] Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. Bringing the web up to speed with webassembly. *Commun. ACM*, 61(12):107–115, 2018.

[2] `https://gitlab.inria.fr/skeletons/necro/blob/master/while.ml`

[3] `https://gitlab.inria.fr/skeletons/necro/blob/master/test/test_interpreter_while.ml`