Certified Derivation of Small-Step From Big-Step Skeletal Semantics

Guillaume Ambal, Sergueï Lenglet, Alan Schmitt, Camille Noûs

September 22, 2022

Context

Different Operational Semantics

▷ Natural Semantics (Big-Step):

$$\frac{s, e_1 \Downarrow v_1 \qquad s, e_2 \Downarrow v_2 \qquad v_1 + v_2 = v}{s, \texttt{Plus}(e_1, e_2) \Downarrow v}$$

▷ Structural Operational Semantics (Small-Step):

$$\frac{s, e_1 \to s, e_1'}{s, \texttt{Plus}(e_1, e_2) \to s, \texttt{Plus}(e_1', e_2)} \qquad \ldots$$

▷ Context Based Reduction Semantics:

$$C ::= [\cdot] \mid \texttt{Plus}(C, e) \mid \texttt{Plus}(v, C)$$

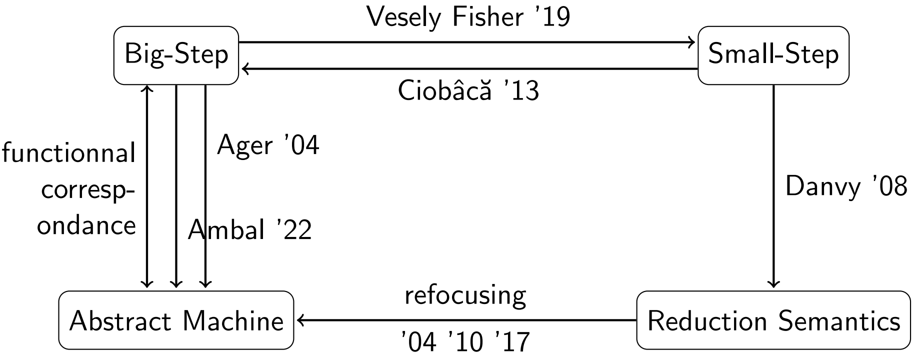
$$\frac{v_1 + v_2 = v}{s, \texttt{Plus}(v_1, v_2) \to v}$$

▷ Abstract Machine:

$$< \texttt{Plus}(e_1, e_2); s; \pi >_m \quad \to \quad < e_1; s; (\texttt{Plus}([\cdot], e_2), s) :: \pi >_m$$

Related Work

Interderiving Operational Semantics:

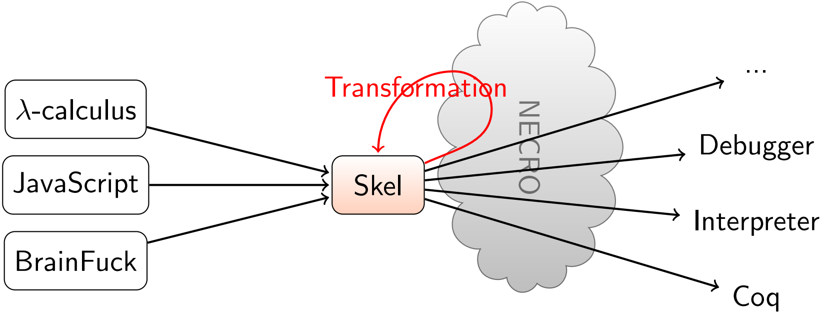


Mostly ad-hoc uncertified transformations

This Work

Certified generic automatic transformation from big-step to small-step skeletal semantics

Framework: Skeletal Semantics
With Necro: OCaml implementation of Skeletal Semantics



Example: IMP

```
type stmt =
| Skip
| Assign of ident * expr
| Seq of stmt * stmt
| If of expr * stmt * stmt
| While of expr * stmt

...

hook hstmt (s : state, t : stmt) matching t : state =
| Seq (t1, t2) ->
  let s' = hstmt (s, t1) in
  hstmt (s', t2)
| ...
```

Transformation

Big-Step vs Small-Step

Big-Step: fully evaluates the term

$$\frac{s, t_1 \Downarrow s' \qquad s', t_2 \Downarrow s''}{s, \texttt{Seq}(t_1, t_2) \Downarrow s''}$$

Small-Step: stops and reconstructs a term

$$\frac{s, t_1 \rightarrow s', t_1'}{s, \texttt{Seq}(t_1, t_2) \rightarrow s', \texttt{Seq}(t_1', t_2)} \qquad \frac{}{s, \texttt{Seq}(\texttt{Ret}(s'), t_2) \rightarrow s', t_2}$$

Intuition: Seq

Big-Step:

$$\frac{s, t_1 \Downarrow s' \qquad s', t_2 \Downarrow s''}{s, \texttt{Seq}(t_1, t_2) \Downarrow s''}$$

Intuition: Seq

Big-Step:

$$\frac{s, t_1 \Downarrow s' \qquad s', t_2 \Downarrow s''}{s, \texttt{Seq}(t_1, t_2) \Downarrow s''}$$

Small-Step:

$$\frac{\ldots}{s, \texttt{Seq}(t_1, t_2) \rightarrow \ldots}$$

Intuition: Seq

Big-Step:

$$\frac{s, t_1 \Downarrow s' \qquad s', t_2 \Downarrow s''}{s, \texttt{Seq}(t_1, t_2) \Downarrow s''}$$

Small-Step:

$$\frac{s, t_1 \rightarrow s', t_1'}{s, \texttt{Seq}(t_1, t_2) \rightarrow \ldots}$$

Intuition: Seq

Big-Step:

$$\frac{s, t_1 \Downarrow s' \qquad s', t_2 \Downarrow s''}{s, \texttt{Seq}(t_1, t_2) \Downarrow s''}$$

Small-Step:

$$\frac{s, t_1 \rightarrow s', t_1'}{s, \texttt{Seq}(t_1, t_2) \rightarrow s', \texttt{Seq}(t_1', t_2)}$$

Intuition: While

Big-Step:

$$\frac{s, e_1 \Downarrow s', v \qquad \texttt{isTrue}(v) \qquad s', t_2 \Downarrow s'' \qquad s'', \texttt{While}(e_1, t_2) \Downarrow s'''}{s, \texttt{While}(e_1, t_2) \Downarrow s'''}$$

Intuition: While

Big-Step:

$$\frac{s, e_1 \Downarrow s', v \qquad \texttt{isTrue}(v) \qquad s', t_2 \Downarrow s'' \qquad s'', \texttt{While}(e_1, t_2) \Downarrow s'''}{s, \texttt{While}(e_1, t_2) \Downarrow s'''}$$

Small-Step:

$$\frac{\ldots}{s, \texttt{While}(e_1, t_2) \rightarrow \ldots}$$

Intuition: While

Big-Step:

$$\frac{s, e_1 \Downarrow s', v \qquad \texttt{isTrue}(v) \qquad s', t_2 \Downarrow s'' \qquad s'', \texttt{While}(e_1, t_2) \Downarrow s'''}{s, \texttt{While}(e_1, t_2) \Downarrow s'''}$$

Small-Step:

$$\frac{s, e_1 \rightarrow s', e_1'}{s, \texttt{While}(e_1, t_2) \rightarrow \ldots}$$

Intuition: While

Big-Step:

$$\frac{s, \color{red}{e_1} \Downarrow s', v \qquad \texttt{isTrue}(v) \qquad s', t_2 \Downarrow s'' \qquad s'', \texttt{While}(\color{red}{e_1}, t_2) \Downarrow s'''}{s, \texttt{While}(e_1, t_2) \Downarrow s'''}$$

Small-Step:

$$\frac{s, e_1 \rightarrow s', e_1'}{s, \texttt{While}(e_1, t_2) \rightarrow \cancel{s', \texttt{While}(e_1', t_2)}}$$

Intuition: While

Big-Step:

$$\frac{s, e_1 \Downarrow s', v \qquad \texttt{isTrue}(v) \qquad s', t_2 \Downarrow s'' \qquad s'', \texttt{While}(e_1, t_2) \Downarrow s'''}{s, \texttt{While}(e_1, t_2) \Downarrow s'''}$$

Small-Step:

$$\frac{s, e_1 \rightarrow s', e_1'}{s, \texttt{While}(e_1, t_2) \rightarrow \xcancel{s', \texttt{While}(e_1', t_2)}}$$

$\Rightarrow$ Need new constructor to remember both e_1 and e_1'

Transformation Phases

Three main phases of the transformation:

- Find the problematic premises
- Create new constructors for them
- Turn everything small-step

1. Problematic cases

Could simply flag everything as problematic!
It works, but ugly results...

Smarter way: analyze skeletons
Main reasons to flag a premise as problematic:

- Not enough memory space
- After a choice we do not want to cancel

These bad cases are found by a simple local analysis

2. New Constructors

One for each problematic premise

Big-Step:

$$\frac{s, e_1 \Downarrow s', v \qquad \texttt{isTrue}(v) \qquad s', t_2 \Downarrow s'' \qquad s'', \texttt{While}(e_1, t_2) \Downarrow s'''}{s, \texttt{While}(e_1, t_2) \Downarrow s'''}$$

2. New Constructors

One for each problematic premise

Big-Step:

$$\frac{s, e_1 \Downarrow s', v \qquad \texttt{isTrue}(v) \qquad s', t_2 \Downarrow s'' \qquad s'', \texttt{While}(e_1, t_2) \Downarrow s'''}{s, \texttt{While}(e_1, t_2) \Downarrow s'''}$$

End goal:

$$\frac{}{s, \texttt{While}(e_1, t_2) \to s, \texttt{While1}(...)}$$

$$\frac{... \Downarrow s', v \qquad \texttt{isTrue}(v) \qquad s', t_2 \Downarrow s'' \qquad s'', \texttt{While}(e_1, t_2) \Downarrow s'''}{s, \texttt{While1}(...) \Downarrow s'''}$$

2. New Constructors

One for each problematic premise
Big-Step:

$$\frac{s, e_1 \Downarrow s', v \qquad \texttt{isTrue}(v) \qquad s', t_2 \Downarrow s'' \qquad s'', \texttt{While}(e_1, t_2) \Downarrow s'''}{s, \texttt{While}(e_1, t_2) \Downarrow s'''}$$

End goal:

$$\frac{}{s, \texttt{While}(e_1, t_2) \to s, \texttt{While1}(s, e_1, \ldots)}$$

$$\frac{s_0, e_0 \Downarrow s', v \qquad \texttt{isTrue}(v) \qquad s', t_2 \Downarrow s'' \qquad s'', \texttt{While}(e_1, t_2) \Downarrow s'''}{s, \texttt{While1}(s_0, e_0, \ldots) \Downarrow s'''}$$

2. New Constructors

One for each problematic premise

Big-Step:

$$\frac{s, e_1 \Downarrow s', v \qquad \texttt{isTrue}(v) \qquad s', t_2 \Downarrow s'' \qquad s'', \texttt{While}(e_1, t_2) \Downarrow s'''}{s, \texttt{While}(e_1, t_2) \Downarrow s'''}$$

End goal:

$$\frac{}{s, \texttt{While}(e_1, t_2) \rightarrow s, \texttt{While1}(s, e_1, e_1, t_2)}$$

$$\frac{s_0, e_0 \Downarrow s', v \qquad \texttt{isTrue}(v) \qquad s', t_2 \Downarrow s'' \qquad s'', \texttt{While}(e_1, t_2) \Downarrow s'''}{s, \texttt{While1}(s_0, e_0, e_1, t_2) \Downarrow s'''}$$

3. Small-Stepify

▷ Problematic evaluation calls are replaced by the new constructor

For instance for while:

$$\frac{}{s, \texttt{While}(e_1, t_2) \rightarrow s, \texttt{While1}(s, e_1, e_1, t_2)}$$

With skeletons:

```
hook hstmt (s : state, t : stmt) matching t : state * stmt =
| ...
| While (e1, t2) ->
  (s, While1 (s, e1, e1, t2))
```

▷ Good evaluation calls are replaced by a branching

For instance for sequences:

$$\frac{s, t_1 \to s', t_1'}{s, \texttt{Seq}(t_1, t_2) \to s', \texttt{Seq}(t_1', t_2)} \qquad \frac{}{s, \texttt{Seq}(\texttt{Ret}(s'), t_2) \to s', t_2}$$

With skeletons:

```
hook hstmt (s : state, t : stmt) matching t : state * stmt =
| Seq (t1, t2) ->
  branch
    let (s', t1') = hstmt (s, t1) in
    (s', Seq (t1', t2))
  or
    let Ret s' = t1 in
    (s', t2)
  end
```

Transformation: Conclusion

- Automatic translation of a Big-Step skeletal semantics into an equivalent Small-Step semantics
- Works on any language expressible with inference rules
- Reuses constructors as much as possible
- Implemented in Necro

Certification

Certification

Theorem we want (for every evaluation function)

$t \Downarrow v \quad \Longleftrightarrow \quad t \rightarrow^* v$

$\Downarrow$: given Big-Step semantics

$\rightarrow^*$: transitive closure of the resulting Small-Step semantics

Pen-and-paper Proof

Full transformation seems too complex
Instead, we prove a simplified version without analysis
(i.e., assume every premise is a problematic case)

Pages of lemmas about:

- Freshness conditions for variables
- Showing new constructors work as intended when going small-step

The paper proof also covers diverging behaviors:

$$t \Uparrow^{\infty} \quad \Longleftrightarrow \quad t \rightarrow^{\infty}$$

Coq Certification

Second certification method: Coq proof script

- Fully automatic
- Language specific
- Handles constructor reuse
- Makes use of Necro-Coq

Coq: BS $\Rightarrow$ SS

Big-Step:

$$\dfrac{\dfrac{\vdots}{e_1 \Downarrow v_1} \qquad \dfrac{\vdots}{e_2 \Downarrow v_2} \qquad v_1 + v_2 = v}{\texttt{Plus}(e_1, e_2) \Downarrow v}$$

Coq: BS $\Rightarrow$ SS

Big-Step:

$$\dfrac{\dfrac{\vdots}{e_1 \Downarrow v_1} \qquad \dfrac{\vdots}{e_2 \Downarrow v_2} \qquad v_1 + v_2 = v}{\texttt{Plus}(e_1, e_2) \Downarrow v}$$

Small-Step:

$$\begin{aligned}
&\texttt{Plus}(e_1, e_2) \rightarrow \texttt{Plus}(e_1', e_2) \rightarrow \cdots \rightarrow \\
&\texttt{Plus}(v_1, e_2) \rightarrow \texttt{Plus}(v_1, e_2') \rightarrow \cdots \rightarrow \\
&\texttt{Plus}(v_1, v_2) \rightarrow v
\end{aligned}$$

Easy to automate in Coq

Coq: SS $\Rightarrow$ BS

Same strategy backwards is hard...
(e.g., splitting $\texttt{Plus}(e_1, e_2) \to^* v$)

Instead, we use a simple concatenation lemma:

$$t \to t' \Downarrow v \quad \Longrightarrow \quad t \Downarrow v$$

Then, iterating the lemma gives us:

$$t \to^* v \quad \Longrightarrow \quad t \Downarrow v$$

- Easy for Coq to bruteforce (no sequences or transitive closure)
- Only works if the big-step semantics is defined on the same constructors, for cases like: $s, \texttt{While}(e_1, t_2) \to s, \texttt{While1}(\ldots) \Downarrow v$

Three Semantics

Initial (BS) ⟺ Intermediate ⟺ Output (SS)

Initial (BS)

```
type stmt =
| ...
| While of expr * stmt

hook hstmt ... : state =
| Seq (t1, t2) ->
  let s' = hstmt (s, t1) in
  hstmt (s', t2)
```

Intermediate

```
type stmt =
| ...
| While1 of ...
| While2 of ...
| Ret of state

hook hstmt ... : state =
| Seq (t1, t2) ->
  let s' = hstmt (s, t1) in
  hstmt (s', t2)
| ...
| While2 (s0, t0, e1, t2) ->
  let s' = hstmt (s0,t0) in
  hstmt (s', While (e1,t2))
| Ret (s') -> s'
```

Output (SS)

```
type stmt =
| ...
| While1 of ...
| While2 of ...
| Ret of state

hook ... : state * stmt =
| Seq (t1, t2) ->
  branch
   let (s', t1')
      = hstmt (s, t1) in
   (s', Seq (t1', t2))
  or
   let Ret s' = t1 in
   (s', t2)
  end
```

Evaluation

Language	Constructors		
	Big-Step	Small-Step	No Reuse
Call-by-Name	3	4	5
Call-by-Value	3	4	5
CBV, choice	4	5	6
CBV, fail	5	6	7
Arithmetic	5	5	13
IMP	11	13	21
IMP, write in exp	12	14	23
IMP, LetIn	12	16	24
IMP, try/catch	15	17	26
MiniML	18	28	33

Table: Size of the Generated Semantics

Conclusion

Conclusion

- Fully automated generic transformation, implemented in Necro
- Generic proof for a simplified version without constructor reuse
- For any language, generation of an equivalence proof script

$$t \Downarrow v \iff t \rightarrow^{*} v$$

- Tested on several languages (including mini-ML)
- Does not work (yet?) with recent Skel