Formal Transformations of Operational Semantics

Guillaume Ambal [1]

September 29, 2020

[1] PhD Supervisors: Alan Schmitt and Sergueï Lenglet

Context

Different Operational Semantics

Big-Step

Small-Step

Abstract Machine

Reduction 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):

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

- Reduction Semantics:

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

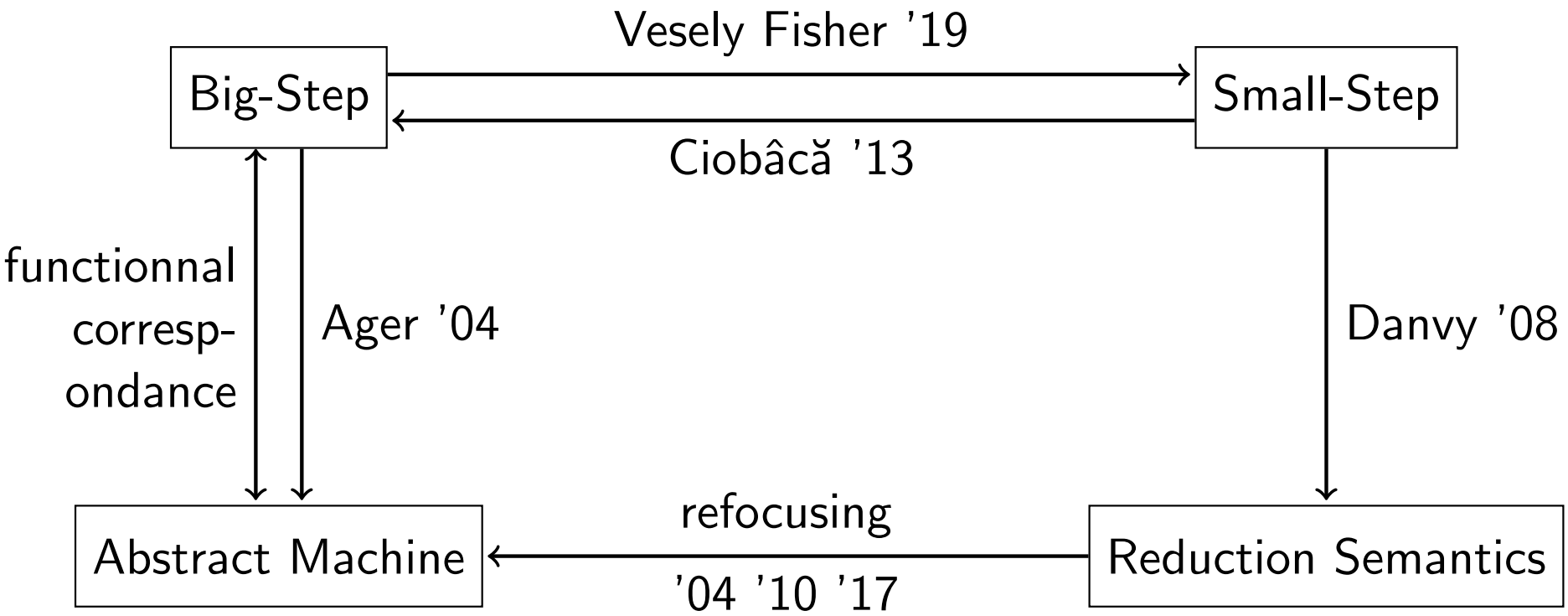
$$\frac{v_1 + v_2 = v}{s, \texttt{Plus}(v_1, v_2) \rightarrow v}$$

- Abstract Machine:

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

Related Work

Interderiving Operational Semantics:



Mostly ad-hoc uncertified transformations

My Thesis

Goal: Formalize translations between different operational semantics

Framework: Skeletal Semantics
meta-language to define the specification of a language.
with Necro: tool implemented in OCaml to manipulate skeletal semantics

First Year:

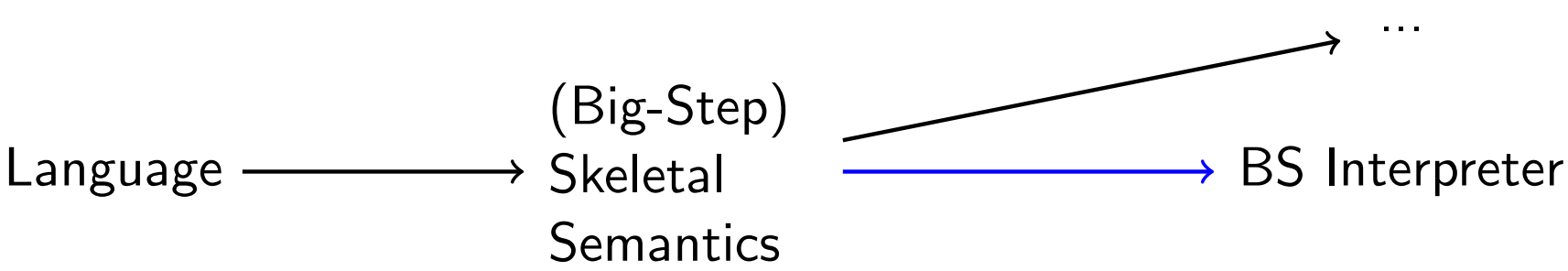
- Automatic translation of a Big-Step skeletal semantics into an equivalent Small-Step semantics
- A-posteriori Coq certification of the result

Related Work

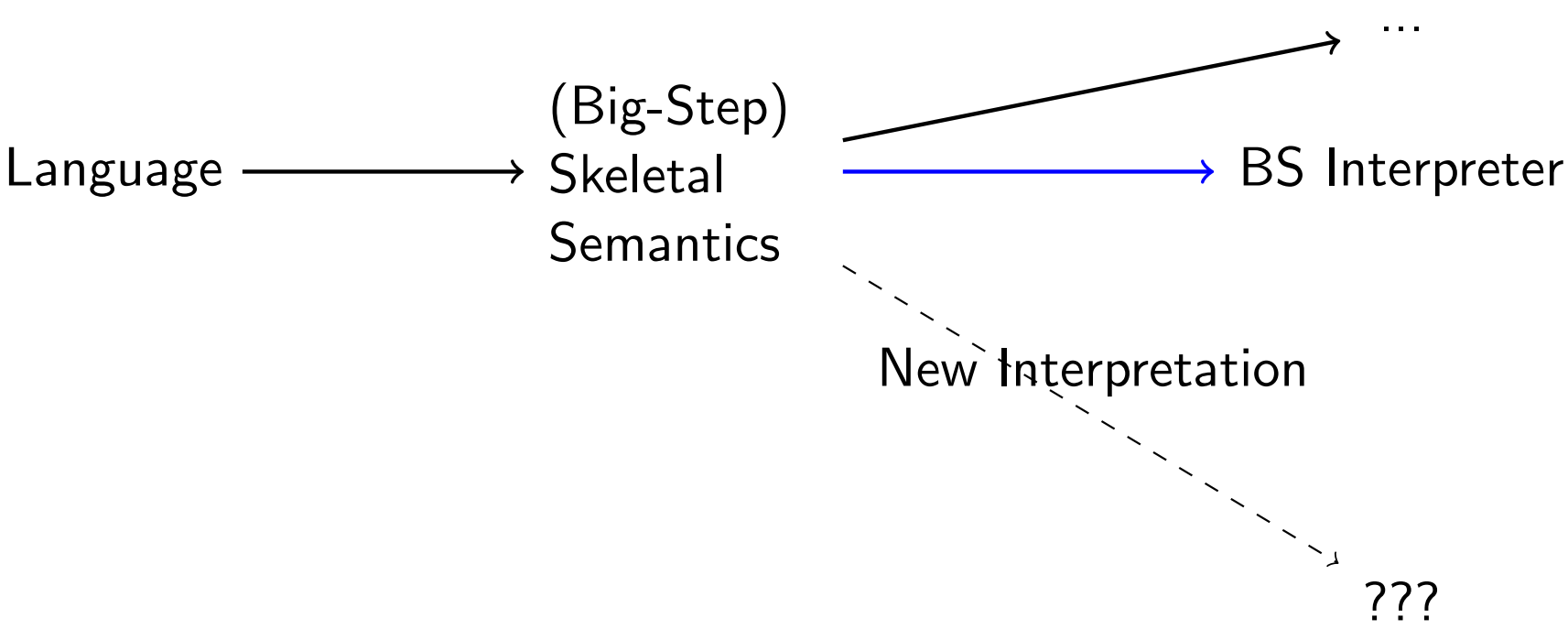
Frameworks to formalize a language:

K, Ott, Lem, ...

- Intuitive readable definitions
- Offers tools: certifications, interpreters, LaTeX, ...
- Can't manipulate different semantics at the same time

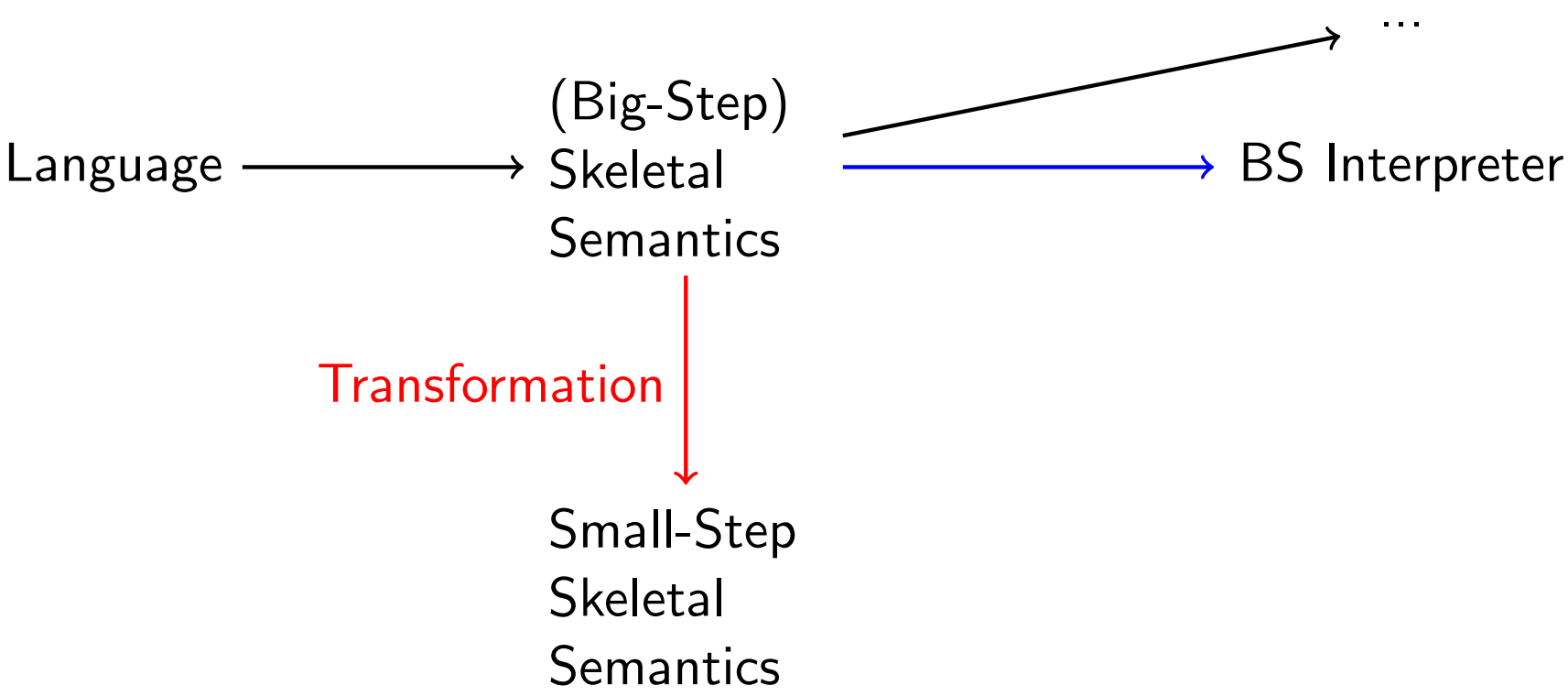


Current structure of Necro

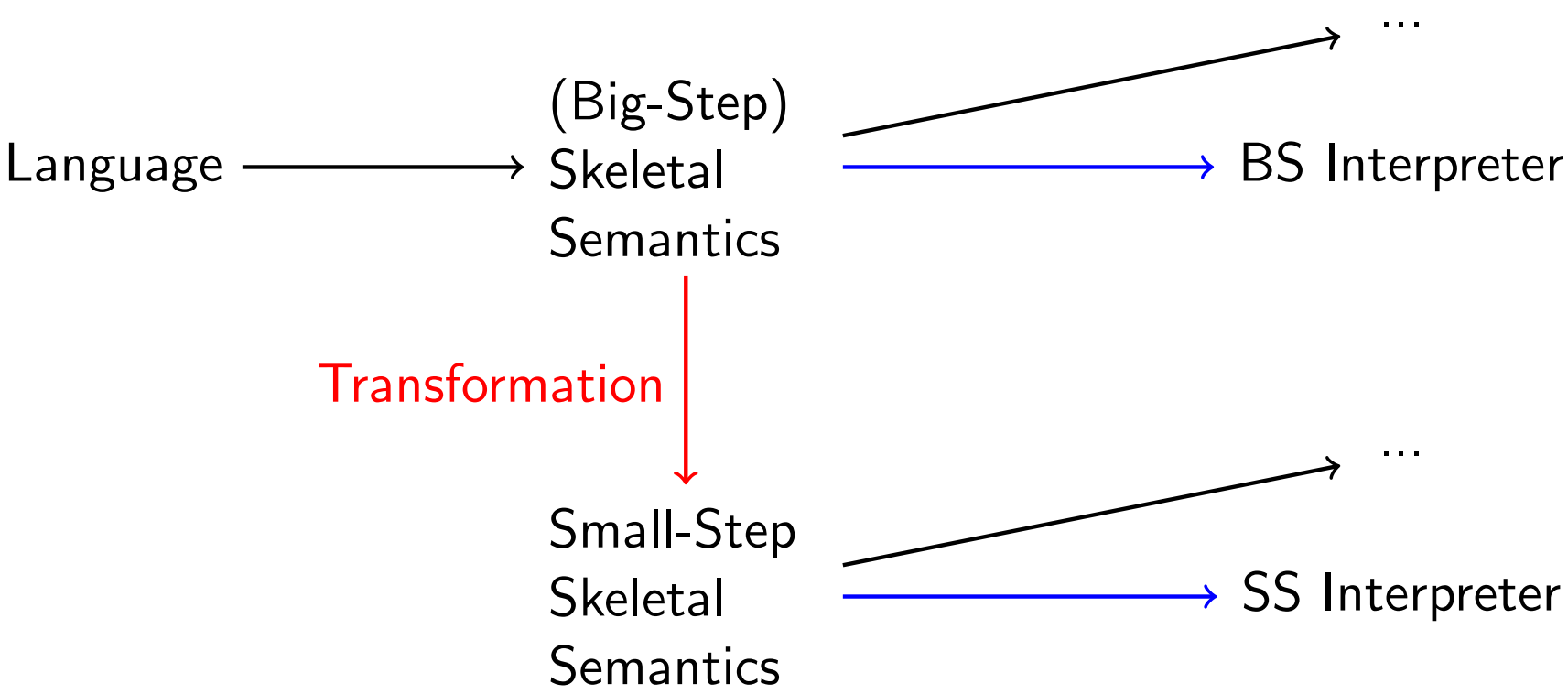


New interpretation of the same semantics ?

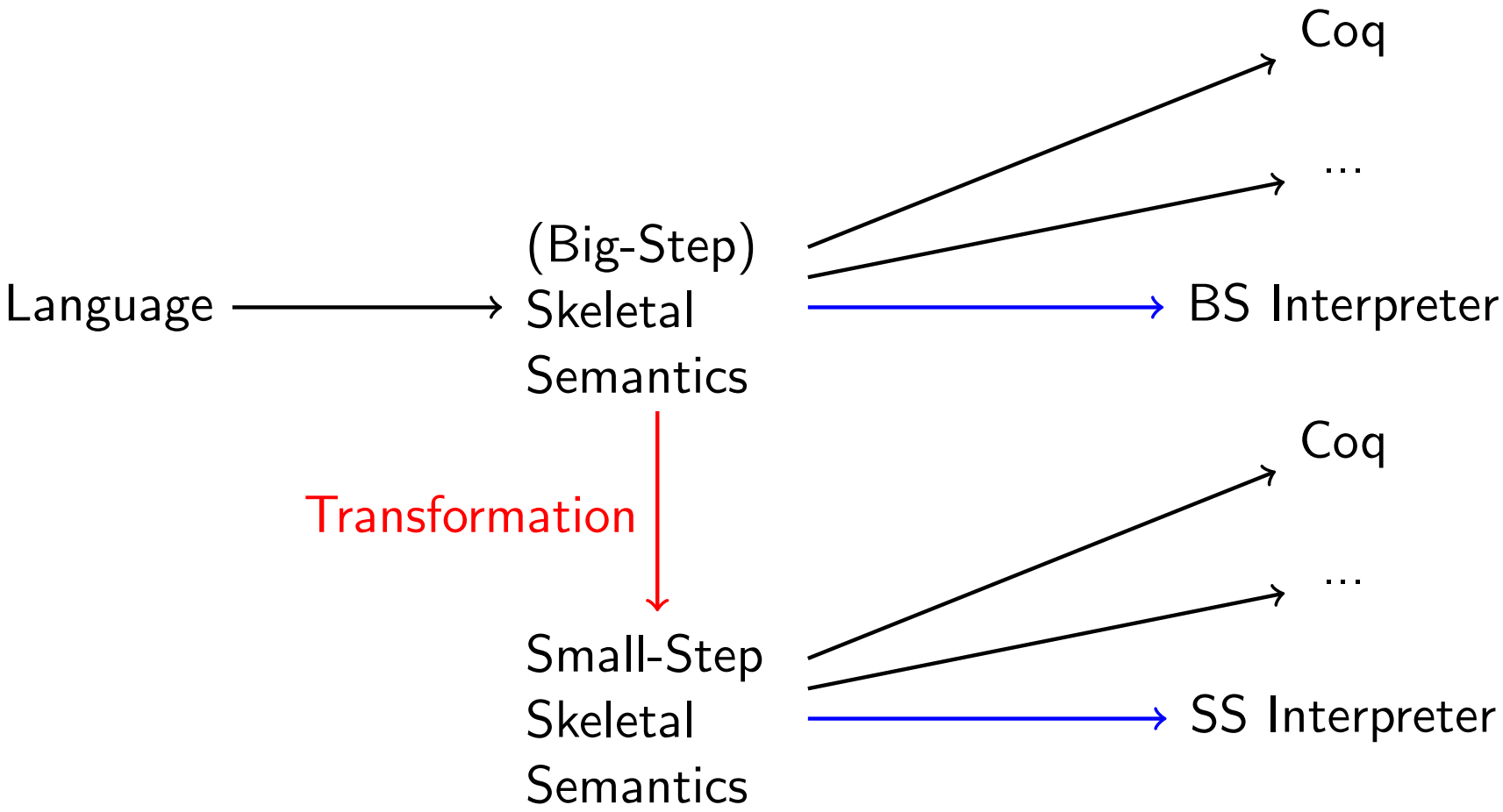
Does not reconstruct terms...



Create a new independant semantics



Can use other Necro tools



Including Coq formalization

Transformation

Big-Step vs Small-Step

Big-Step:

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

Small-Step:

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

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)
| ...
```

Example: IMP

```
hook hstmt (s : state, t : stmt) matching t : state =
| ...
| While (e1, t2) ->
  let (s', v) = hexpr (s, e1) in
  branch
    isTrue (v);
    let s'' = hstmt (s', t2) in
    hstmt (s'', While (e1, t2))
  or
    isFalse (v);
    s'
  end
```

Intuition: Seq

Big-Step:

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

Intuition: Seq

Big-Step:

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

Small-Step:

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

Intuition: Seq

Big-Step:

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

Small-Step:

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

Intuition: Seq

Big-Step:

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

Small-Step:

$$\frac{s, t_1 \to s', t_1'}{s, \mathtt{Seq}(t_1, t_2) \to s', \mathtt{Seq}(t_1', t_2)}$$

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:

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

Small-Step:

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

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:

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

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

Three main phases of the transformation:

- Find the "bad hook calls"
- Create new constructors for them
- Turn everything small-step

1. Bad Hook Calls

Main case of bad hook call: not enough memory space.

```
hook hstmt (s : state, t : stmt) matching t : state =
| ...
| While (e1, t2) ->
  let (s', v) = hexpr (s, e1) in (* BAD! *)
  branch
    isTrue (v);
    let s'' = hstmt (s', t2) in
    hstmt (s'', While (e1, t2)) (* because we need e1 *)
  or ... end
```

Bad cases are found by a simple local analysis.

2. New Constructors

One for each "bad hook call".
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:

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

2. New Constructors

One for each "bad hook call".
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:

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

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

2. New Constructors

One for each "bad hook call".
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:

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

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

2. New Constructors

One for each "bad hook call".
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:

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

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

With skeletons

```
| ...
| While1 (s0, e0, e1, t2) ->
  let (s', v) = hexpr (s0, e0) in (* GOOD! *)
  branch
    isTrue (v);
    let s'' = hstmt (s', t2) in
    hstmt (s'', While (e1, t2))
  or ... end
```

3. Small-Stepify

▷ Bad hook calls are replaced by the new constructor

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

```
hook hstmt (s : state, t : stmt) matching t : state =
| While (e1, t2) ->
  let (s', v) = hexpr (s, e1) in
  ...
```

Becomes:

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

▷ Good hook calls are replaced by a branching

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

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

Becomes:

```
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
```

Example: While1 in small-step

```
hook hstmt (s : state, t : stmt) matching t : state * stmt =
| While1 (s0, e0, e1, t2) ->
  branch
    let (s0', e0') = hexpr (s0, e0) in
    (s, While1 (s0', e0', e1, t2))
  or
    let Ret (s', v) = e0 in
    branch
      isTrue (v);
      (s, (While2 (s', t2, e1, t2)))
    or
      isFalse (v);
      (s, Ret s')
    end
  end
```

Transformation: Conclusion

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

Certification

Certification

Theorem we want (for every hook)

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

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

▷ Certification of the full transformation looks hard...

▷ Instead, we automatically generate a Coq proof script alongside the transformation (self-certification).

BS $\Longrightarrow$ SS

Big-Step:

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

BS $\Longrightarrow$ SS

Big-Step:

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

Small-Step:

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

Easy to automate in Coq

SS $\Longrightarrow$ BS

Small-Step:

$$\texttt{Plus}(e_1, e_2) \rightarrow^* v$$

SS $\Longrightarrow$ BS

Small-Step:

$$\texttt{Plus}(e_1, e_2) \rightarrow^* v$$

Standard strategy: Splitting Lemma

$$\texttt{Plus}(e_1, e_2) \rightarrow \cdots \rightarrow$$
$$\texttt{Plus}(v_1, e_2) \rightarrow \cdots \rightarrow$$
$$\texttt{Plus}(v_1, v_2) \rightarrow v$$

SS $\Longrightarrow$ BS

Small-Step:

$$\texttt{Plus}(e_1, e_2) \rightarrow^* v$$

Standard strategy: Splitting Lemma

$$\begin{array}{l} \texttt{Plus}(e_1, e_2) \rightarrow \cdots \rightarrow \\ \texttt{Plus}(v_1, e_2) \rightarrow \cdots \rightarrow \\ \texttt{Plus}(v_1, v_2) \rightarrow v \end{array}$$

Big-Step:

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

Hard to automate for any language in Coq

SS $\Longrightarrow$ BS

Instead, we use a simple concatenation lemma[2]:

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

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: $s, \texttt{While}(e_1, t_2) \to s, \texttt{While1}(\ldots) \Downarrow v$

[2] from "Flag-Based Big-Step Semantics", by Poulsen and Mosses

Three Semantics

Initial (BS)

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

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

$\Longleftrightarrow$

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'
```

$\Longleftrightarrow$

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
```

Conclusion

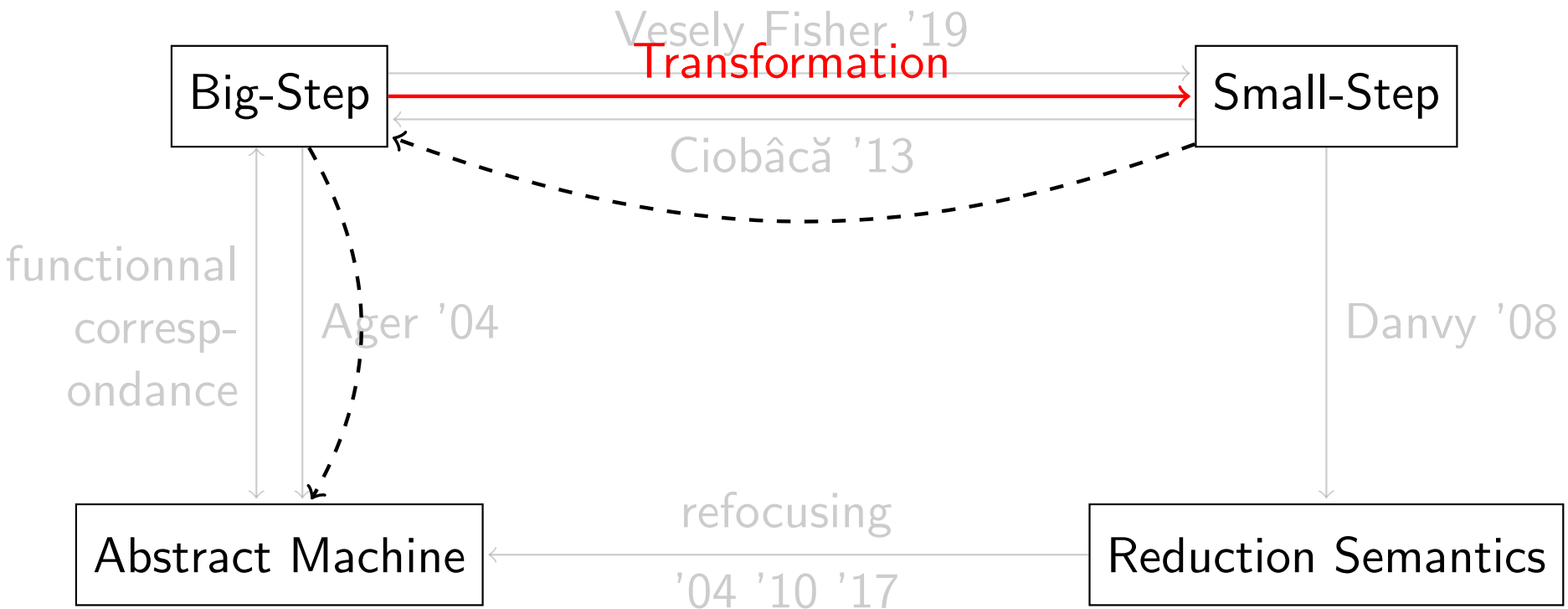
Conclusion

- Fully automated generic transformation, implemented in Necro
- Term reconstruction is complex
- Tested on several languages (including mini-ML)
- For any language, generation of an equivalence proof script

$$t \Downarrow v \iff t \to^* v$$

- No generic certification...
- Paper submitted at POPL (and rejected)

Future Work


Vesely Fisher '19
Transformation
Big-Step
Small-Step
Ciobâcă '13
functionnal
corresp-
ondance
Ager '04
Danvy '08
refocusing
'04 '10 '17
Abstract Machine
Reduction Semantics