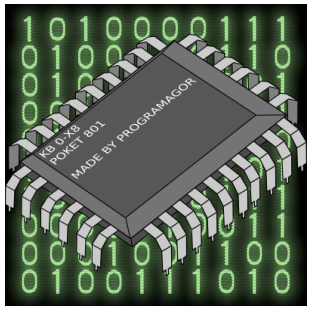
Skeletal Semantics Transformations

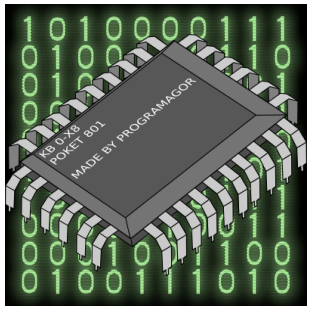
PhD Defense

Guillaume Ambal

PhD Supervisors: Alan Schmitt and Sergueï Lenglet

October 19, 2022

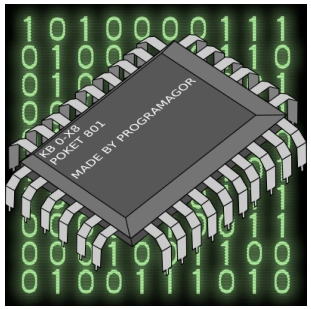

KB 0X8
POKET 801
MADE BY PROGRAMAGOR


KB 0x8
POKET 801
MADE BY PROGRAMAGOR

OCaml

C

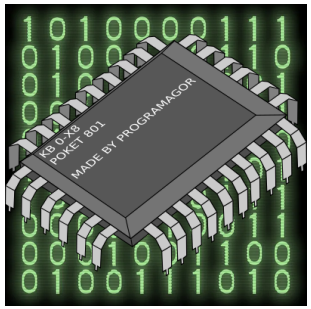
JavaScript

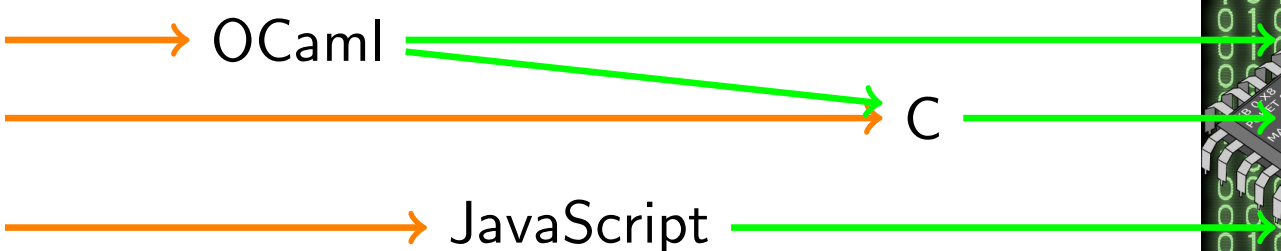


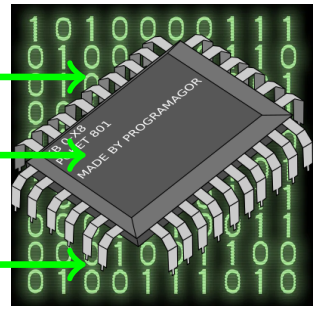
→ OCaml

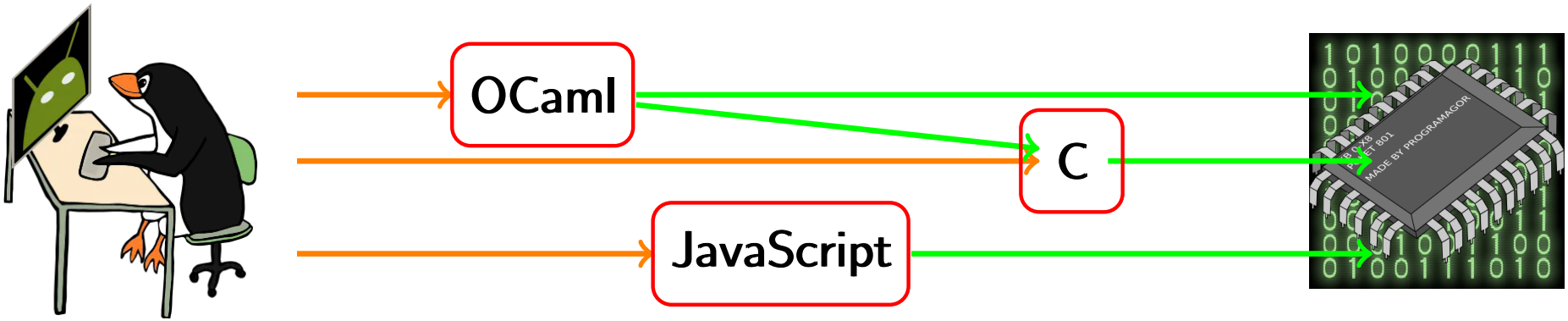
→ C

→ JavaScript




OCaml
C
JavaScript


MADE BY PROGRAMAGOR



Need to properly define **programming languages**

Different Semantics

- Denotational Semantics
  $[\![\texttt{fact}]\!] = \{(0,1);(1,1);\cdots;(5,120);\cdots\} = \{(n,n!) \mid n \in \mathbb{N}\}$
  mathematical approach, useful for equivalences

Different Semantics

- Denotational Semantics
  $[\![\texttt{fact}]\!] = \{(0,1);(1,1);\cdots;(5,120);\cdots\} = \{(n,n!) \mid n \in \mathbb{N}\}$
  mathematical approach, useful for equivalences

- Axiomatic Semantics
  $[x = 5]\{y \leftarrow \texttt{fact}\ x\}[x = 5 \wedge y = 120]$
  logical approach, useful for checking behaviors

Different Semantics

- Denotational Semantics
  $[\![\texttt{fact}]\!] = \{(0,1);(1,1);\cdots;(5,120);\cdots\} = \{(n,n!) \mid n \in \mathbb{N}\}$
  mathematical approach, useful for equivalences

- Axiomatic Semantics
  $[x = 5]\{y \leftarrow \texttt{fact}\ x\}[x = 5 \wedge y = 120]$
  logical approach, useful for checking behaviors

- Operational Semantics
  $\texttt{fact}\ 5 \Downarrow 120$
  computational approach, useful for running a program

Different Operational Semantics

▷ Natural Semantics (Big-Step):

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

e.g., $(1 \dot{+} 2) \dot{+} (4 \dot{+} 8) \Downarrow 15$

▷ Structural Operational Semantics (Small-Step):

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

e.g., $(1 \dot{+} 2) \dot{+} (4 \dot{+} 8) \to 3 \dot{+} (4 \dot{+} 8)$

Different Operational Semantics

▷ Context Based Reduction Semantics:

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

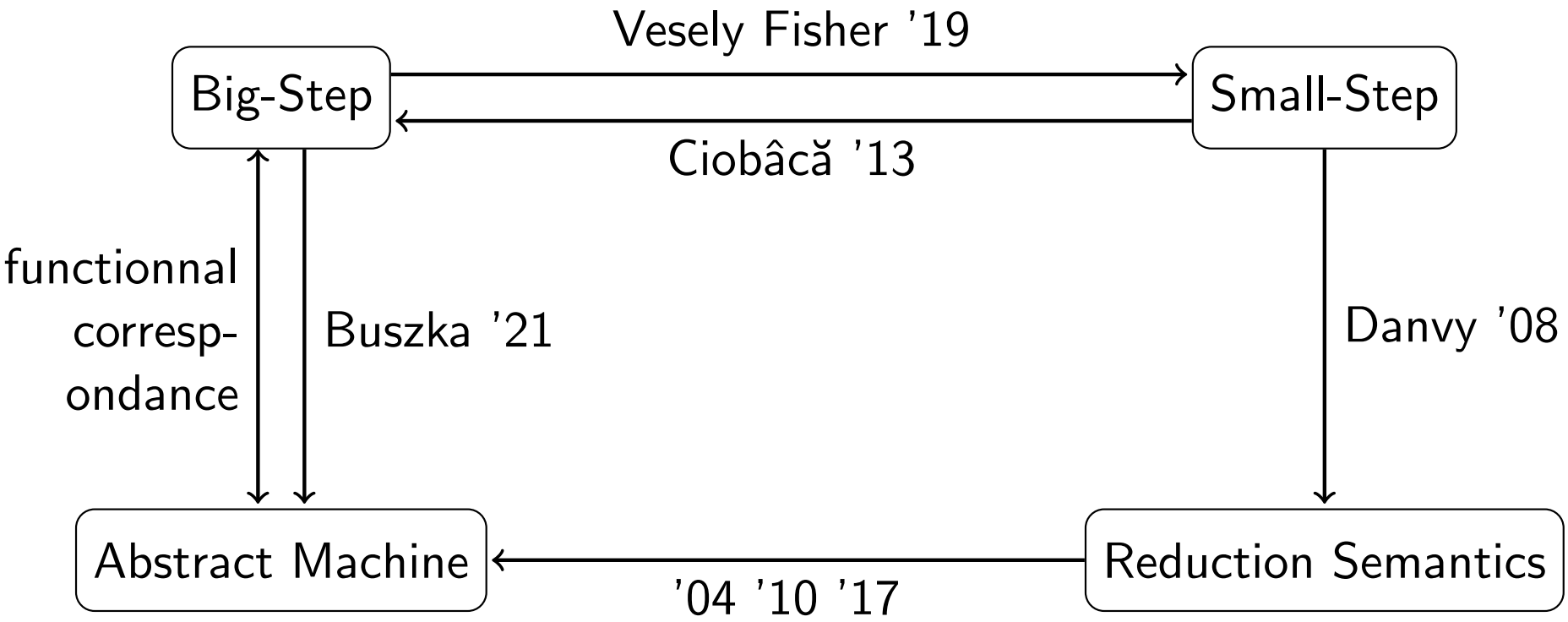
$$\frac{v_1 + v_2 = v}{\texttt{Plus}(v_1, v_2) \rightsquigarrow v} \qquad \frac{e \rightsquigarrow e'}{C[e] \rightarrow C[e']}$$

▷ Abstract Machine:

$$\langle \texttt{Plus}(e_1, e_2); \kappa \rangle \quad \rightarrow \quad \langle e_1; \texttt{Plus}([\cdot], e_2) :: \kappa \rangle$$

Related Work

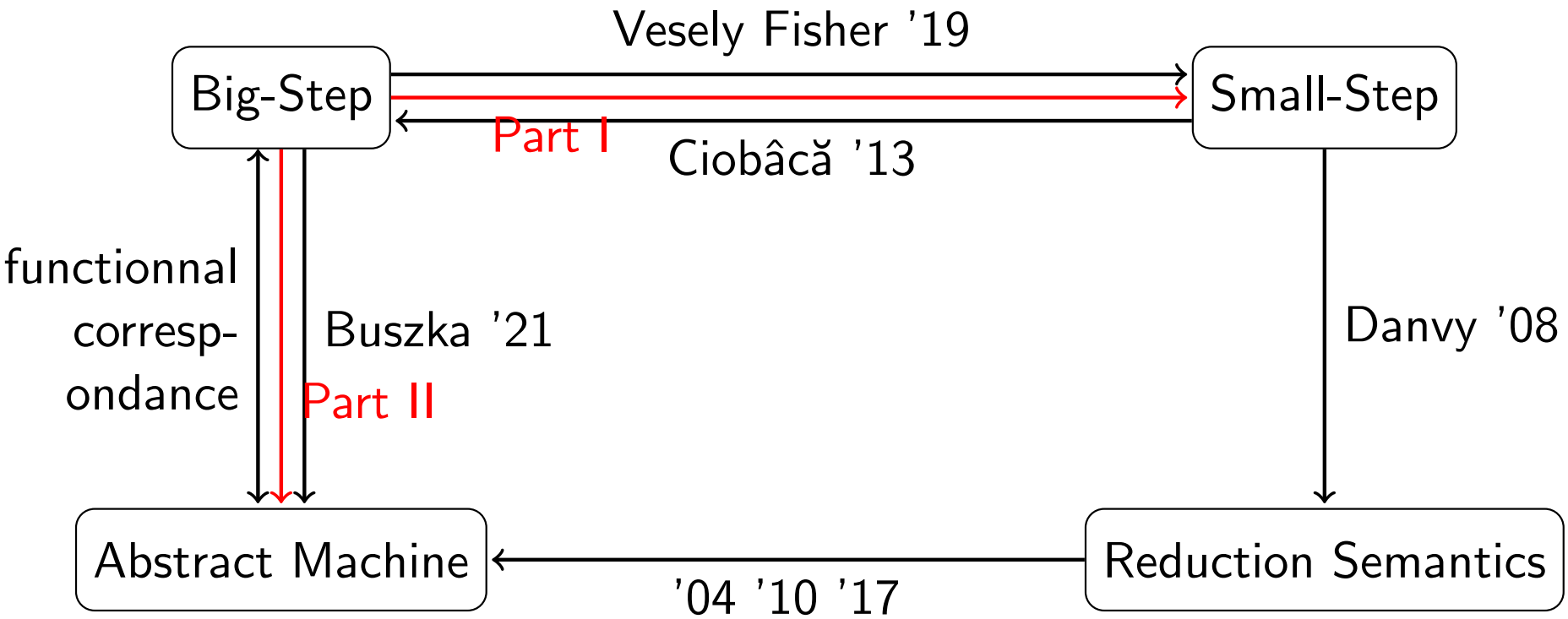
Interderiving Operational Semantics:



Mostly ad-hoc uncertified transformations

My PhD

Interderiving Operational Semantics:



Outline

1. Context
   - Operational Semantics
   - Skeletal Semantics
2. Part I: Transformation from Big-Step to Small-Step
   - Automatic Transformation
   - Correctness
3. Part II: Certified Interpreter and Abstract Machines
   - Abstract Machines
   - Certification and Interpreter
4. Conclusion

Defining a Language with a Computer

In a proof assistant, from scratch

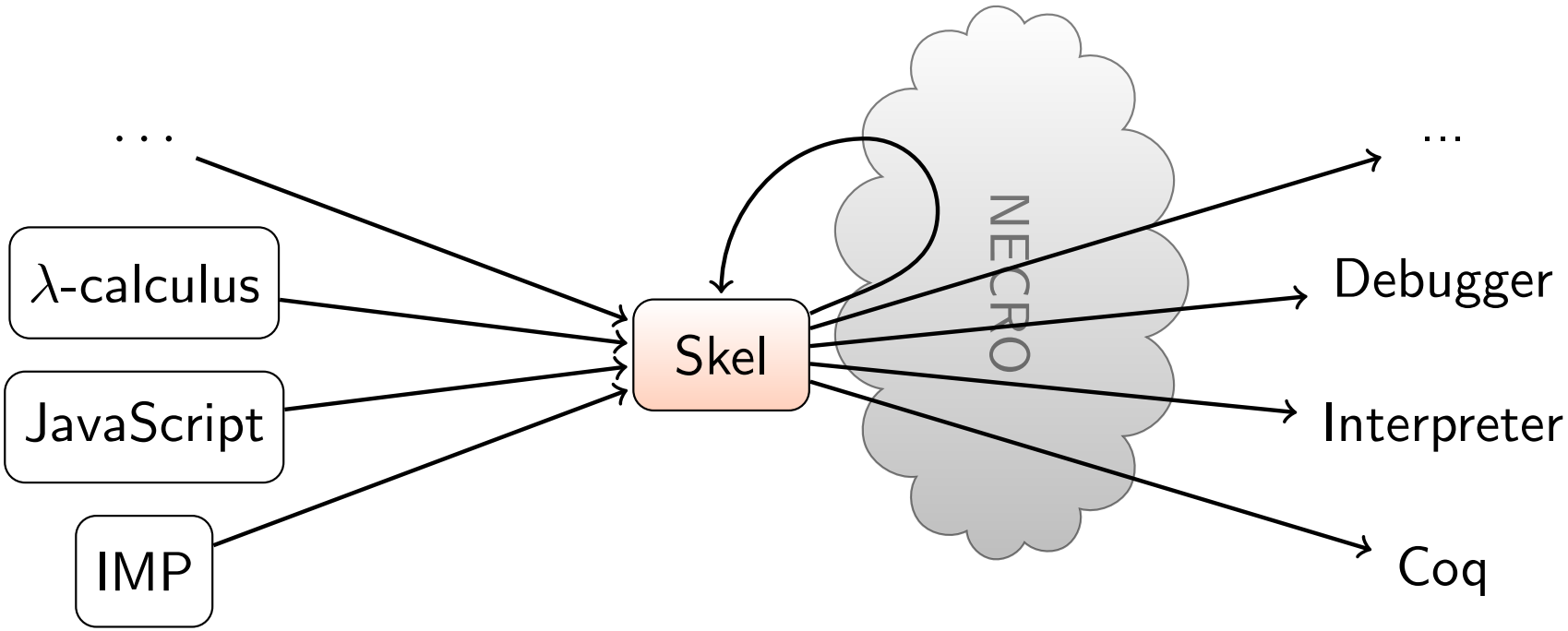
- Coq
- Isabelle/HOL
- ...

In a convenient framework

- Ott, Lem
- $\mathbb{K}$
- Skeletal Semantics

Skeletal Semantics

- Recent framework (first definition: POPL 2019)
- Meta-language (Skel) to define programming languages
- Toolbox to manipulate semantics: Necro.



Example: Call-by-Value λ-calculus

$$
\begin{array}{rrl}
\text{Variables} & x \in \mathcal{V} & \\
\text{Term} & t ::= & \lambda x.t \mid x \mid t\ t \\
\text{Closure} & c ::= & (x, t, s) \\
\text{Environment} & s ::= & [(x_1 \mapsto c_1), \ldots, (x_n \mapsto c_n)]
\end{array}
$$

$$
\frac{}{s, \lambda x.t \Downarrow (x, t, s)} \qquad \frac{s(x) = c}{s, x \Downarrow c}
$$

$$
\frac{s, t_1 \Downarrow (x, t, s') \qquad s, t_2 \Downarrow c' \qquad (s' + \{x \mapsto c'\}), t \Downarrow c}{s, (t_1\ t_2) \Downarrow c}
$$

Skeletal Semantics for CbV λ-calculus

```
type ident

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

type clos =
| Clos (ident, lterm, env)

type env

val extEnv: (env,ident,clos) → env
val getEnv: (ident,env) → clos
```

```
val eval (s:env, l:lterm): clos =
branch
  let Lam (x, t) = l in
  Clos (x, t, s)
or
  let Var x = l in
  getEnv (x, s)
or
  let App (t1, t2) = l in
  let Clos (x, t, s') = eval s t1 in
  let w = eval s t2 in
  let s'' = extEnv (s', x, w) in
  eval s'' t
end
```

Skeletal Semantics for CbV λ-calculus

```
type ident

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

type clos =
| Clos (ident, lterm, env)

type env

val extEnv: (env,ident,clos) → env
val getEnv: (ident,env) → clos
```

Variables $x \in \mathcal{V}$

Term $t ::= \lambda x.t \mid x \mid t\ t$

Skeletal Semantics for CbV λ-calculus

```
type ident

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

type clos =
| Clos (ident, lterm, env)

type env

val extEnv: (env,ident,clos) → env
val getEnv: (ident,env) → clos
```

Can leave types/terms unspecified

$$\text{Closure} \quad c ::= (x, t, s)$$

$$\text{Environment} \quad s ::= [(x_1 \mapsto c_1), \ldots, (x_n \mapsto c_n)]$$

Skeletal Semantics for CbV λ-calculus

$$\frac{}{s, \lambda x.t \Downarrow (x, t, s)}$$

$$\frac{s(x) = c}{s, x \Downarrow c}$$

$$\frac{s, t_1 \Downarrow (x, t, s') \qquad \ldots}{s, (t_1\ t_2) \Downarrow c}$$

```
val eval (s:env, l:lterm): clos =
branch
  let Lam (x, t) = l in
  Clos (x, t, s)
or
  let Var x = l in
  getEnv (x, s)
or
  let App (t1, t2) = l in
  let Clos (x, t, s') = eval s t1 in
  let w = eval s t2 in
  let s'' = extEnv (s', x, w) in
  eval s'' t
end
```

Skeletal Semantics for CbV λ-calculus

```
type ident

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

type clos =
| Clos (ident, lterm, env)

type env

val extEnv: (env,ident,clos) → env
val getEnv: (ident,env) → clos
```

```
val eval (s:env, l:lterm): clos =
branch
  let Lam (x, t) = l in
  Clos (x, t, s)
or
  let Var x = l in
  getEnv (x, s)
or
  let App (t1, t2) = l in
  let Clos (x, t, s') = eval s t1 in
  let w = eval s t2 in
  let s'' = extEnv (s', x, w) in
  eval s'' t
end
```

Skeletal Semantics for IMP

```
type state
type value
...
type stmt =
| Skip
| Seq (stmt, stmt)
| While (expr, stmt)
| ...

val isTrue: value → unit
val isFalse: value → unit
...
```

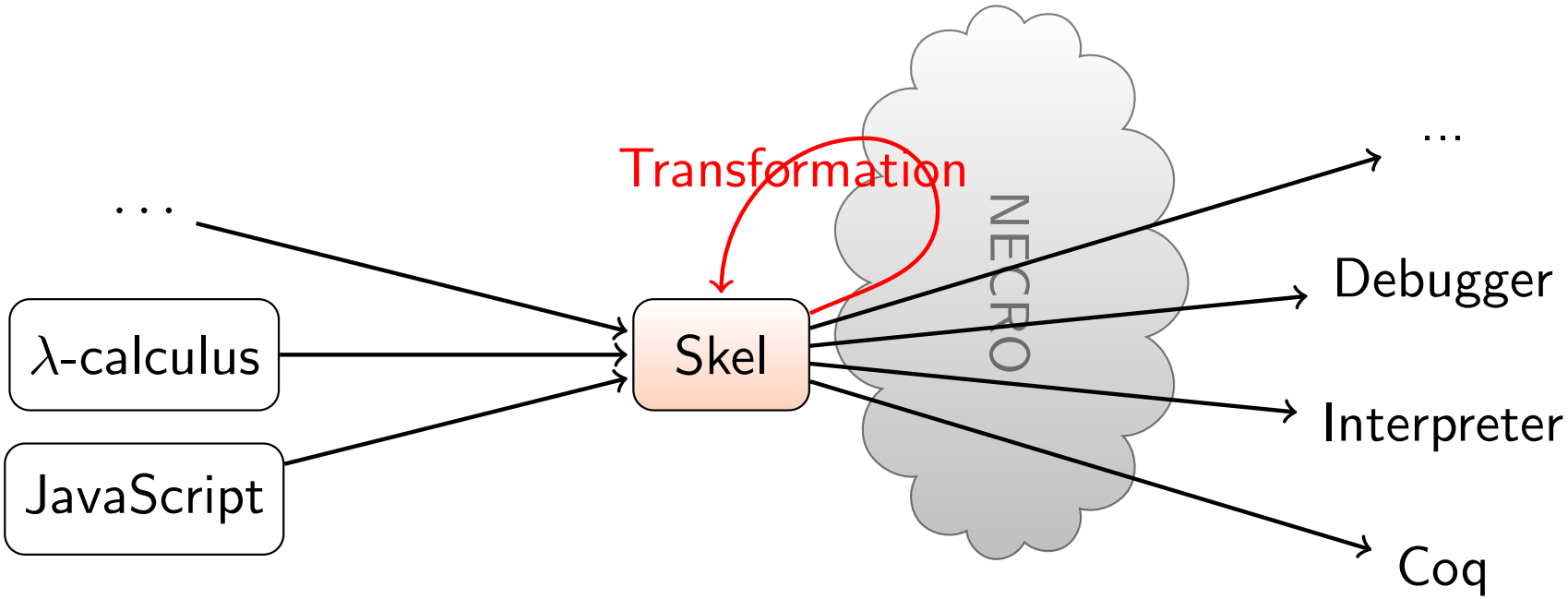
```
val hstmt (s:state, t:stmt) : state =
branch
  let Seq (t1, t2) = t in
  let s1 = hstmt (s, t1) in
  hstmt (s1, t2)
or
  let While (e1, t2) = t in
  let (s1, v) = hexpr (s, e1) in
  branch
    let () = isTrue (v) in
    let s2 = hstmt (s1, t2) in
    hstmt (s2, While (e1, t2))
  or
    let () = isFalse (v) in
    s1
  end
or ... end
```

Outline

1. Context
   - Operational Semantics
   - Skeletal Semantics
2. Part I: Transformation from Big-Step to Small-Step
   - Automatic Transformation
   - Correctness
3. Part II: Certified Interpreter and Abstract Machines
   - Abstract Machines
   - Certification and Interpreter
4. Conclusion

Transformation

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



Big-Step vs Small-Step

Big-Step: fully evaluates the term

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

Small-Step: stops and reconstructs a term

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

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:

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

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:

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

Small-Step:

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

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:

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

$\Rightarrow$ 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 cluttered result...

Smarter way: analyze skeletons

Main reason to flag a premise as problematic:
not enough slots to store arguments

Found by a simple local heuristic

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:

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

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

2. New Constructors

One for each 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:

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

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

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

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

```
val hstmt (s : state, t : stmt) : state * stmt =
branch
  let While (e1, t2) = t in
  (s, While1 (s, e1, e1, t2))
or ... end
```

▷ Good evaluation calls are replaced by a branching

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

```
val hstmt (s : state, t : stmt) : state * stmt =
branch
  let Seq (t1, t2) = t in
  let (s', t1') = hstmt (s, t1) in
  (s', Seq (t1', t2))
or
  let Seq (Ret(s'), t2) = t in
  (s', t2)
or ... end
```

Transformation: Summary

- 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

Outline

1. Context
   - Operational Semantics
   - Skeletal Semantics
2. Part I: Transformation from Big-Step to Small-Step
   - Automatic Transformation
   - Correctness
3. Part II: Certified Interpreter and Abstract Machines
   - Abstract Machines
   - Certification and Interpreter
4. Conclusion

Certification

Theorem

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

$\Downarrow$: given Big-Step semantics

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

- A generic pen-and-paper proof
- An automatic Coq certificate generation

Pen-and-paper Proof

Full transformation seems too complex
Instead, we prove a simplified version without analysis/heuristics

23 pages of results:

- Freshness conditions for variables
- New constructors working as intended

Theorem: diverging behaviors

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

Coq Certificate Generation

Second certification method: Coq proof script

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

Coq: BS $\Rightarrow$ SS

Big-Step:

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

Coq: BS ⇒ 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

Coq: SS $\Rightarrow$ BS

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

Instead, we use a concatenation lemma:

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

- Easy for Coq (case analysis)
- Problem: requires same syntax

Three Semantics

Initial (BS)

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

val hstmt ... : state =
branch
  let Seq (t1, t2) = t in
  let s' = hstmt (s, t1) in
  hstmt (s', t2)
or ... end
```

$\Longleftrightarrow$

Intermediate

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

val hstmt ... : state =
branch
  let Seq (t1, t2) = t in
  let s' = hstmt (s, t1) in
  hstmt (s', t2)
or ... or
  let While2 (s0,t0,e1,t2)
      = t in
  let s' = hstmt (s0,t0) in
  hstmt (s', While (e1,t2))
or
  let Ret (s') = t in
  s'
end
```

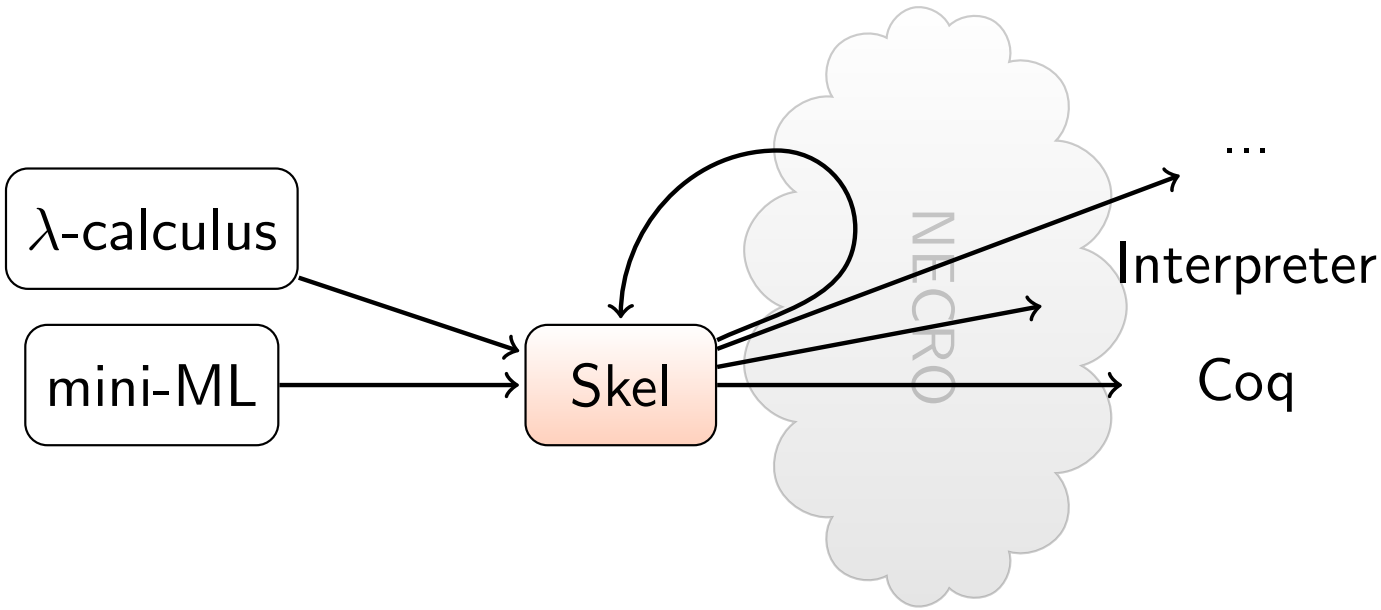
$\Longleftrightarrow$

Output (SS)

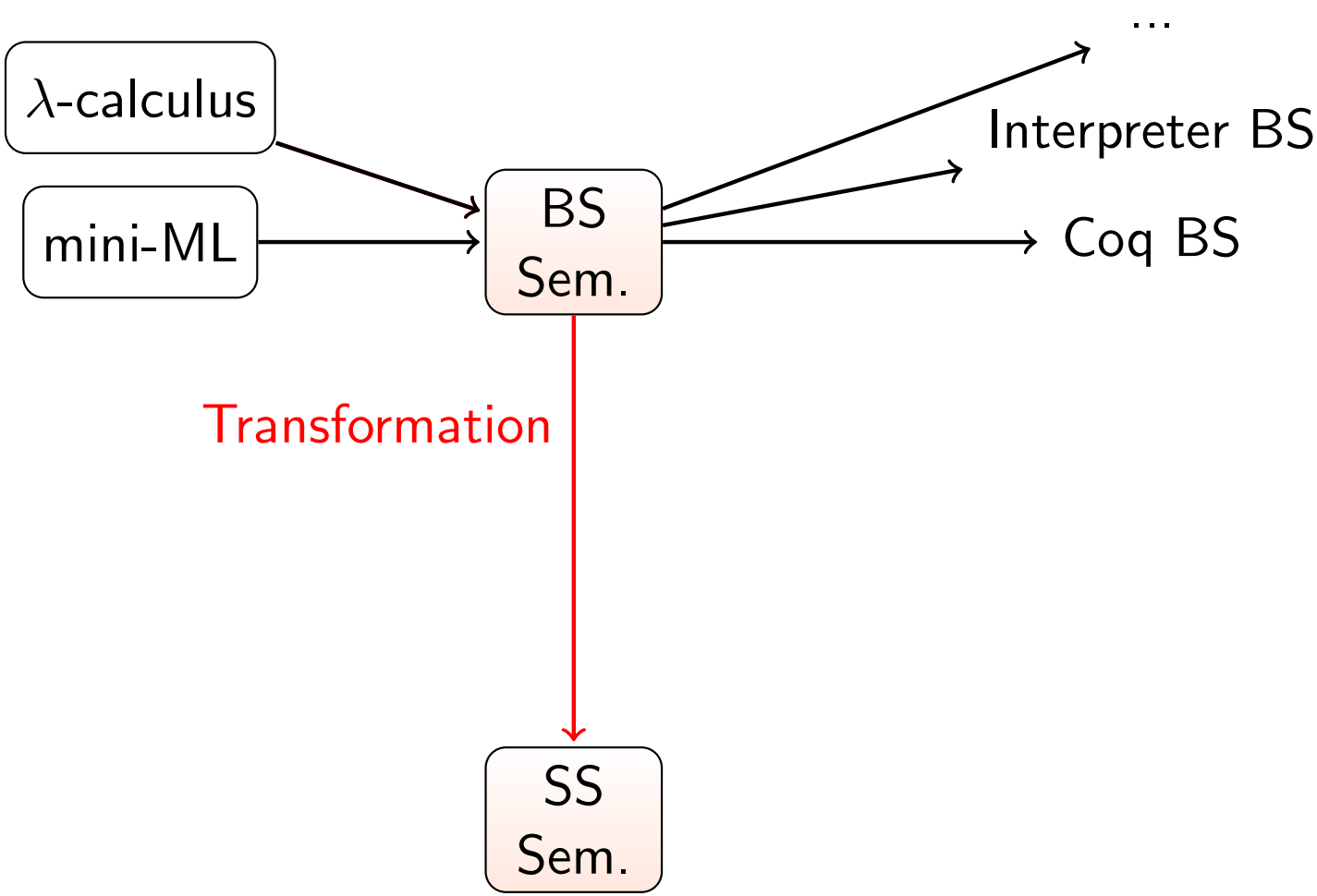
```
type stmt =
| ...
| While1 of ...
| While2 of ...
| Ret of state

val ... : state * stmt =
branch
  let Seq (t1, t2) = t in
  let (s', t1')
      = hstmt (s, t1) in
  (s', Seq (t1', t2))
or
  let Seq (t1, t2) = t in
  let Ret s' = t1 in
  (s', t2)
or ... end
```

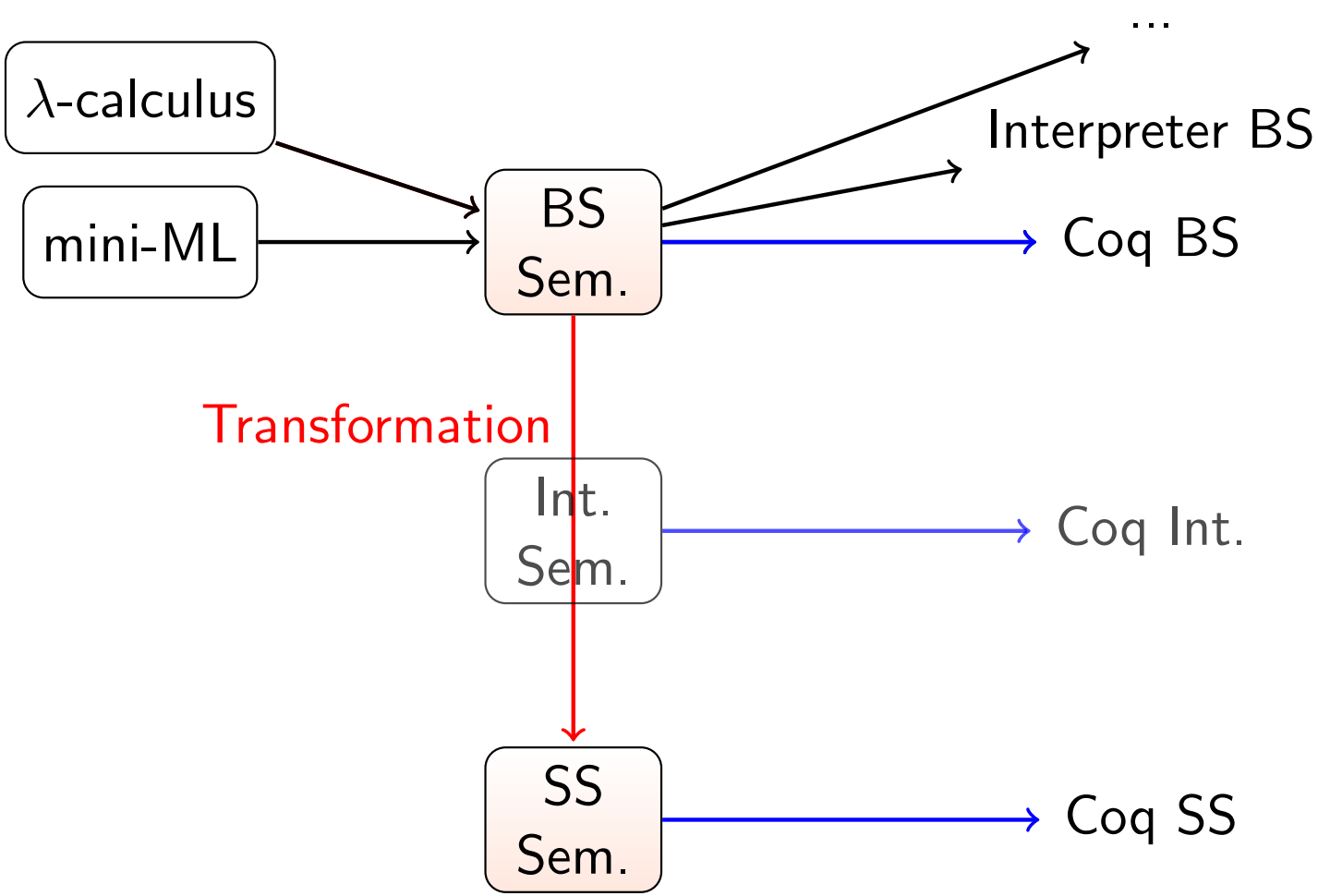
Conclusion



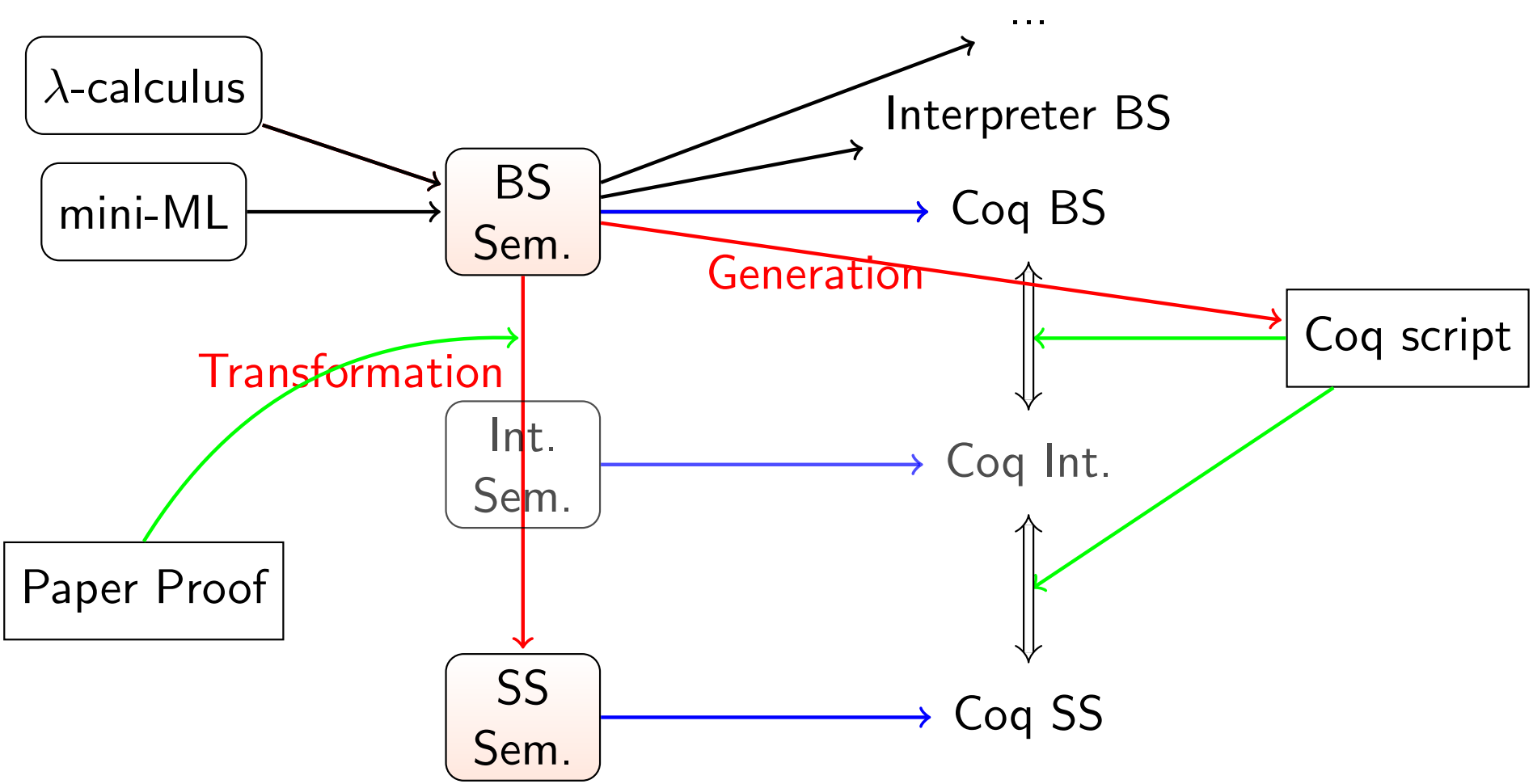
Conclusion


λ-calculus
mini-ML
BS
Sem.
...
Interpreter BS
Coq BS
Transformation
SS
Sem.

Conclusion


λ-calculus
mini-ML
BS
Sem.
...
Interpreter BS
Coq BS
Transformation
Int.
Sem.
Coq Int.
SS
Sem.
Coq SS

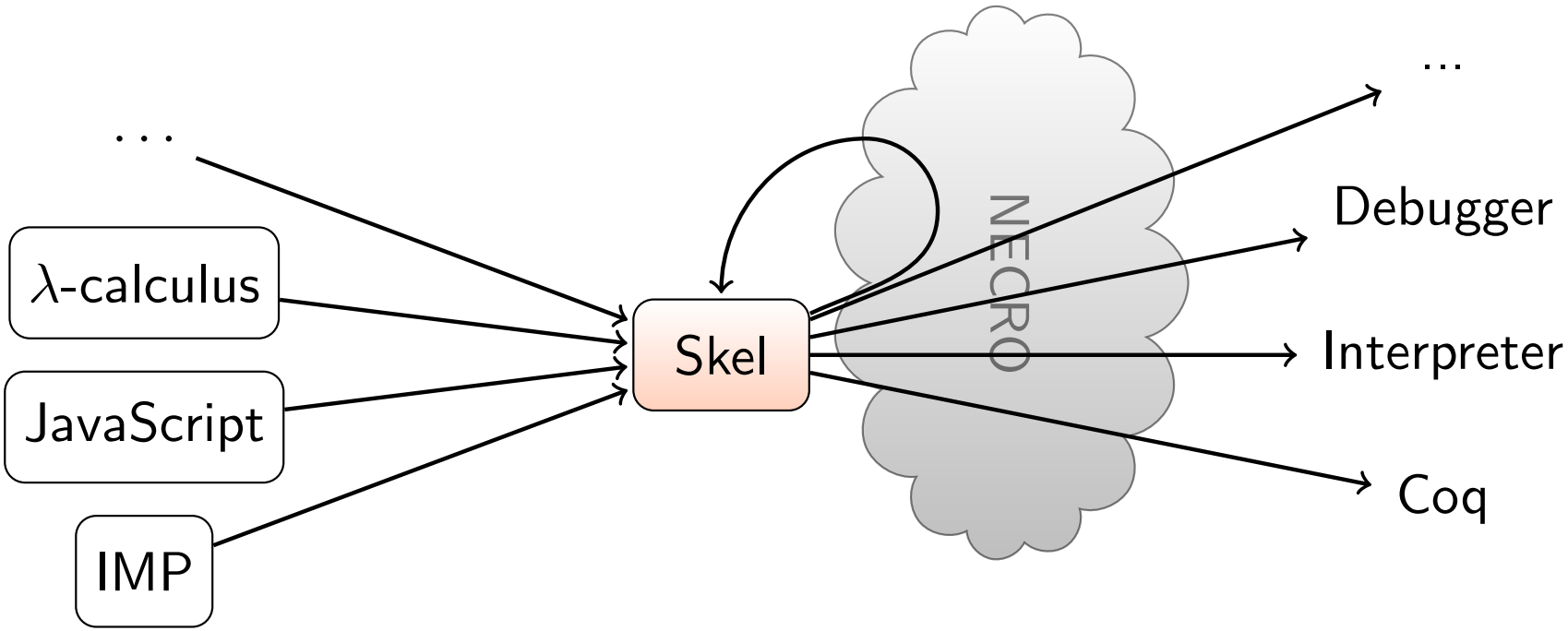
Conclusion


...
λ-calculus
mini-ML
BS Sem.
Interpreter BS
Coq BS
Generation
Coq script
Transformation
Int. Sem.
Coq Int.
Paper Proof
SS Sem.
Coq SS

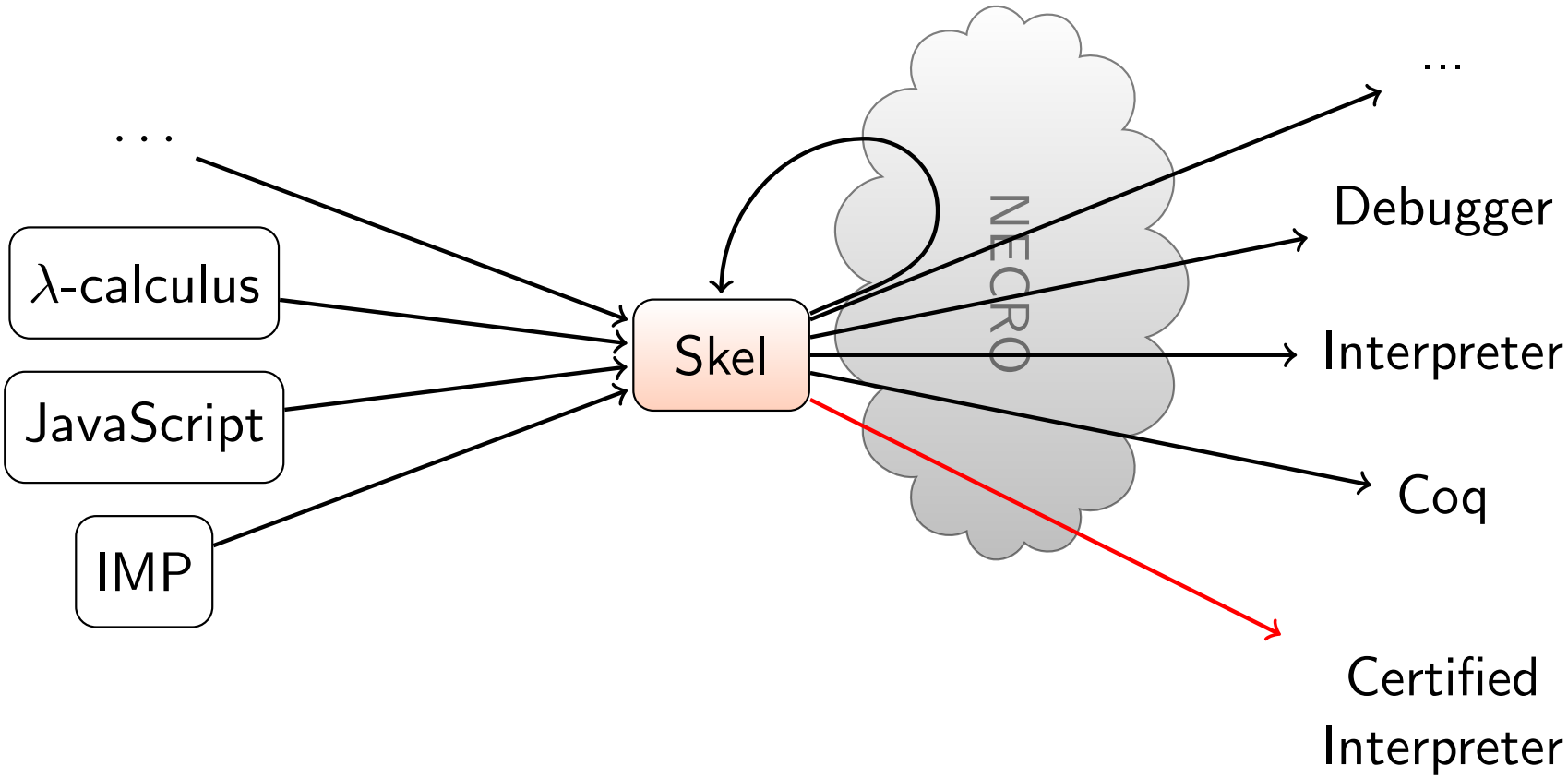
Outline

1. Context
   - Operational Semantics
   - Skeletal Semantics
2. Part I: Transformation from Big-Step to Small-Step
   - Automatic Transformation
   - Correctness
3. Part II: Certified Interpreter and Abstract Machines
   - Abstract Machines
   - Certification and Interpreter
4. Conclusion

Certified Interpreter



Certified Interpreter



Goal: certified execution of programs

New (meta-)semantics for the meta-language Skel

Syntax of Skel

$$
\begin{array}{rl}
\text{Identifier} & x \in \mathcal{V} \\
\text{Term} & t ::= x \mid C\ t \mid (t, \ldots, t) \mid \lambda p.S \\
\text{Skeleton} & S ::= t_0\ t_1 \ldots t_n \mid \texttt{let}\ \ p = S_1\ \ \texttt{in}\ \ S_2 \\
 & \quad \mid \texttt{Branch}(S, \ldots, S) \mid t \\
\text{Pattern} & p ::= _ \mid x \mid C\ p \mid (p, \ldots, p)
\end{array}
$$

Meta-Semantics of Skel

Main meta-semantics of Skel is big-step.

$$\frac{\Sigma \vdash S_1 \Downarrow_s r' \qquad \Sigma + \{p \mapsto r'\} \Downarrow_p \Sigma' \qquad \Sigma' \vdash S_2 \Downarrow_s r}{\Sigma \vdash \texttt{let}\ \ p = S_1\ \ \texttt{in}\ \ S_2 \Downarrow_s r}$$

Want a different format of meta-semantics: Abstract Machines.

Known technique by Danvy et al.:

- CPS Transform
- Defunctionalization

Non-Deterministic Abstract Machine

$$\langle \texttt{let}\ \ p = S_1\ \ \texttt{in}\ \ S_2, \kappa \rangle_{\mathsf{sk}} \rightarrow \langle S_1, \lceil \texttt{let}\ \ p = \square\ \ \texttt{in}\ \ S_2 \rfloor :: \kappa \rangle_{\mathsf{sk}}$$
$$\langle \texttt{Branch}(l), \kappa \rangle_{\mathsf{sk}} \rightarrow \langle S, \kappa \rangle_{\mathsf{sk}} \quad \texttt{for}\ \ (S \in l)$$
$$\ldots \rightarrow \ldots$$

Non-Deterministic Abstract Machine

$$\langle \texttt{let}\ \ p = S_1\ \ \texttt{in}\ \ S_2, \kappa \rangle_{\mathsf{sk}} \rightarrow \langle S_1, \lceil \texttt{let}\ \ p = \square\ \ \texttt{in}\ \ S_2 \rfloor :: \kappa \rangle_{\mathsf{sk}}$$
$$\langle \texttt{Branch}(l), \kappa \rangle_{\mathsf{sk}} \rightarrow \langle S, \kappa \rangle_{\mathsf{sk}} \quad \texttt{for}\ \ (S \in l)$$
$$\ldots \rightarrow \ldots$$

Problem: still non-deterministic, so not really computable...
Next: deterministic AM, with backtracking.

Deterministic Abstract Machine

$$\langle \texttt{let}\ \ p = S_1\ \ \texttt{in}\ \ S_2, \kappa, f \rangle_{\mathsf{sk}} \rightarrow \langle S_1, \lceil \texttt{let}\ \ p = \square\ \ \texttt{in}\ \ S_2 \rfloor :: \kappa, f \rangle_{\mathsf{sk}}$$

$$\langle \texttt{Branch}(S :: l), \kappa, f \rangle_{\mathsf{sk}} \rightarrow \langle S, \kappa, [\![\texttt{Branch}(l), \kappa]\!] :: f \rangle_{\mathsf{sk}}$$

$$\langle \texttt{Branch}([]), \kappa, f \rangle_{\mathsf{sk}} \rightarrow \langle f \rangle_{\mathsf{fc}}$$

$$\ldots \rightarrow \ldots$$

$$\langle [\![S, \kappa]\!] :: f \rangle_{\mathsf{fc}} \rightarrow \langle S, \kappa, f \rangle_{\mathsf{sk}}$$

Outline

1. Context
   - Operational Semantics
   - Skeletal Semantics
2. Part I: Transformation from Big-Step to Small-Step
   - Automatic Transformation
   - Correctness
3. Part II: Certified Interpreter and Abstract Machines
   - Abstract Machines
   - Certification and Interpreter
4. Conclusion

Equivalence Certification

Definitions in Coq:

Equivalence Certification

Definitions in Coq:

Certification:

- We prove Big-Step and NDAM are equivalent (standard proof)
- We prove AM is sound w.r.t. NDAM (cut backtracks)

BS $\Leftrightarrow$ NDAM

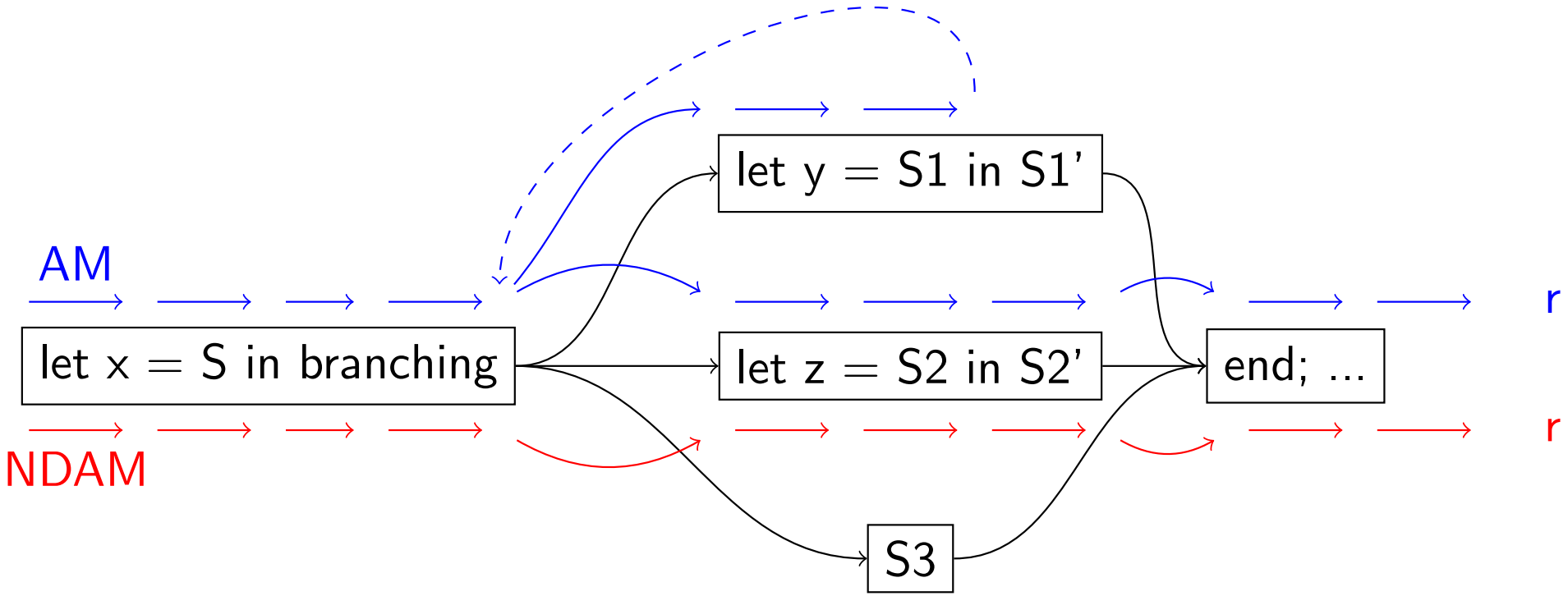
Big-Step:

$$\frac{\Sigma \vdash S_1 \Downarrow_s r' \qquad \Sigma + \{p \mapsto r'\} \Downarrow_p \Sigma' \qquad \Sigma' \vdash S_2 \Downarrow_s r}{\Sigma \vdash \texttt{let}\ \ p = S_1\ \ \texttt{in}\ \ S_2 \Downarrow_s r}$$

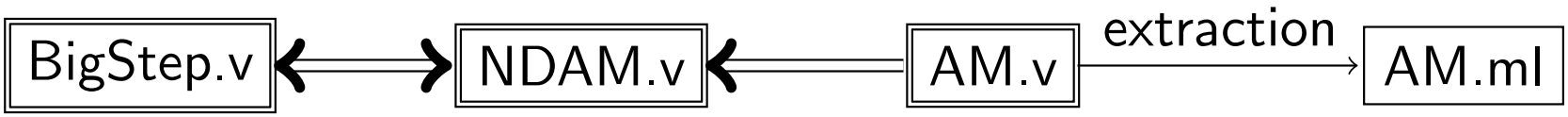
Abstract Machine:

$$\begin{aligned}
& \langle \texttt{let}\ \ p = S_1\ \ \texttt{in}\ \ S_2, \Sigma, \epsilon \rangle_{\mathsf{sk}} \rightarrow \langle S_1, \Sigma, \lceil \texttt{let}\ \ p = \square\ \ \texttt{in}\ \ S_2, \Sigma \rfloor :: \epsilon \rangle_{\mathsf{sk}} \\
\rightarrow^* & \langle \lceil \texttt{let}\ \ p = \square\ \ \texttt{in}\ \ S_2, \Sigma \rfloor :: \epsilon, r' \rangle_{\mathsf{kr}} \rightarrow \langle p, r', \Sigma, \lceil S_2, \square \rfloor :: \epsilon \rangle_{\mathsf{pat}} \\
\rightarrow^* & \langle \lceil S_2, \square \rfloor :: \epsilon, \Sigma' \rangle_{\mathsf{ke}} \rightarrow \langle S_2, \Sigma', \epsilon \rangle_{\mathsf{sk}} \\
\rightarrow^* & \langle \epsilon, r \rangle_{\mathsf{kr}}
\end{aligned}$$

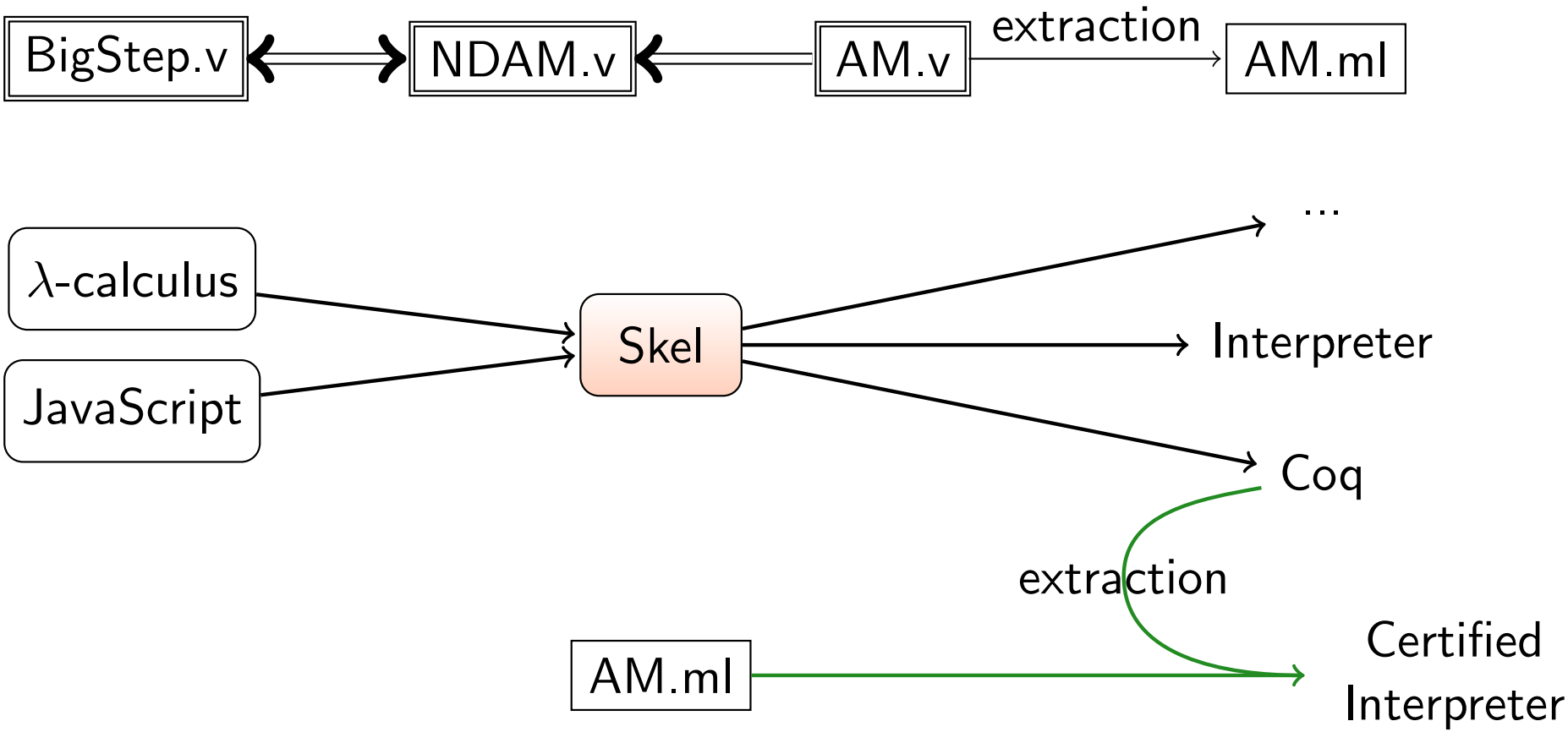
AM ⇒ NDAM



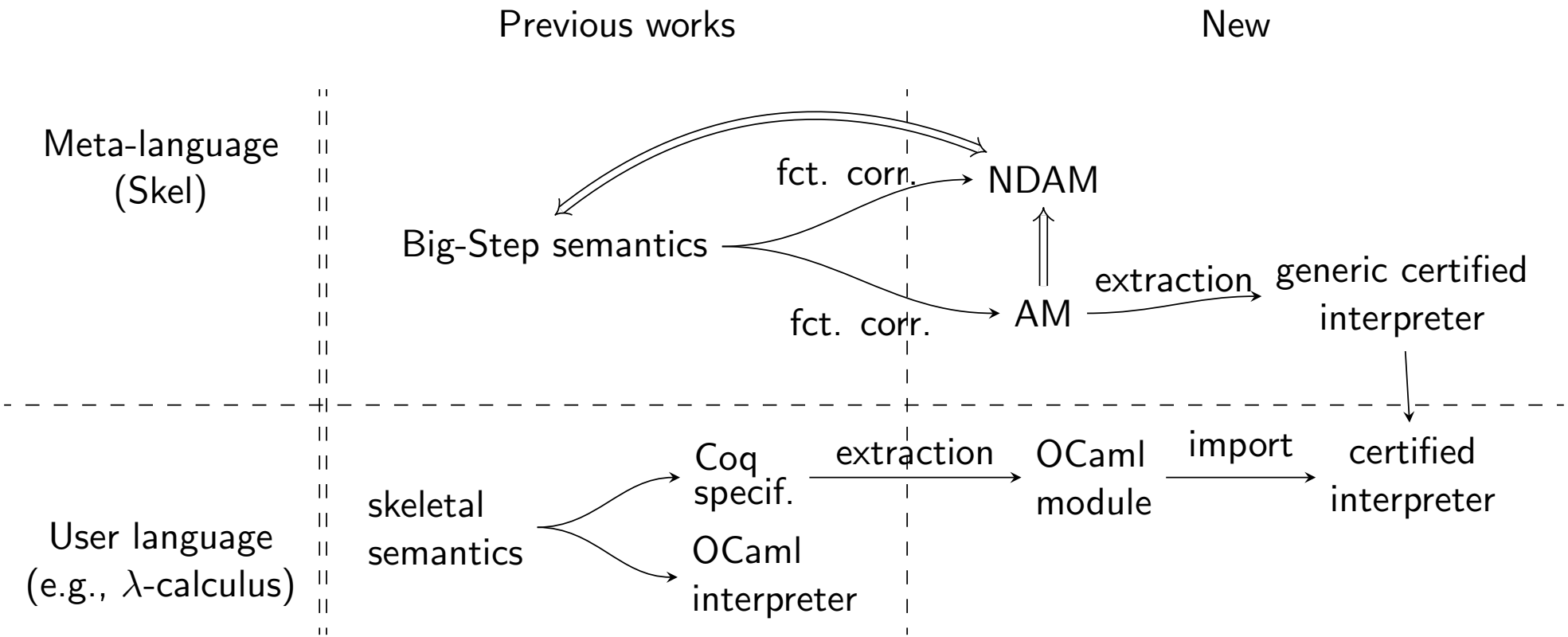
Certified Interpreter



Certified Interpreter



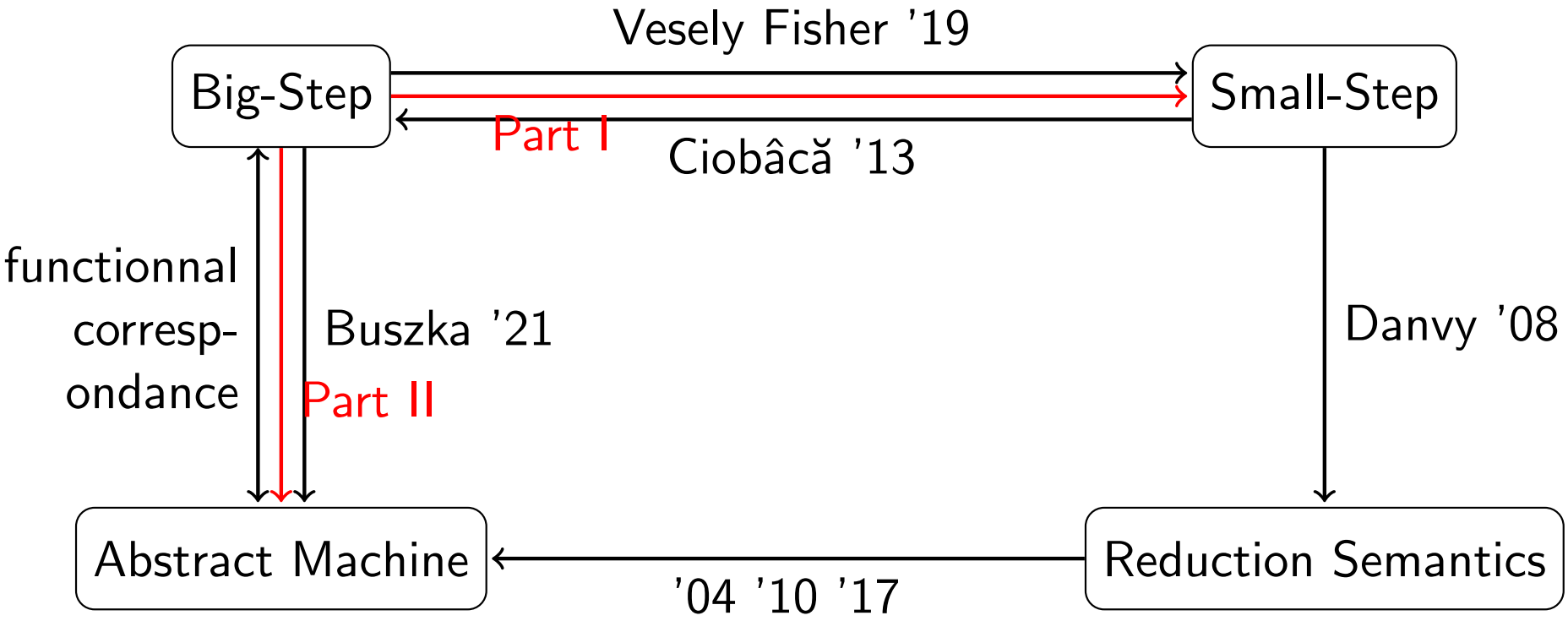
Conclusion


Previous works
New
Meta-language
(Skel)
Big-Step semantics
fct. corr.
NDAM
AM
fct. corr.
extraction
generic certified
interpreter
User language
(e.g., λ-calculus)
skeletal
semantics
Coq
specif.
OCaml
interpreter
extraction
OCaml
module
import
certified
interpreter

Outline

1. Context
   - Operational Semantics
   - Skeletal Semantics
2. Part I: Transformation from Big-Step to Small-Step
   - Automatic Transformation
   - Correctness
3. Part II: Certified Interpreter and Abstract Machines
   - Abstract Machines
   - Certification and Interpreter
4. Conclusion

Future Work



Future Work:

- Update part I for monads
- Other directions

Publications

Appendices
(Bonus Slides)

EBS $\Leftrightarrow$ SS for arithmetic expressions

```
Theorem ebs_implies_ss_hexpr : forall s e s' v,
  hexpr [s; e] [s'; v] ->
  exists s2, ehexpr_tclo [s; e] [s2; Ret [s'; v]].
```

$$\forall s, e, s', v,\ (s, e) \Downarrow_{\texttt{hexpr}} (s', v) \implies \exists s_2,\ (s, e) \rightarrow^*_{\texttt{ehexpr}} (s_2, \texttt{Ret}(s', v))$$

```
Theorem ss_implies_ebs_hexpr : forall s e s2 s' v,
  ehexpr_tclo [s; e] [s2; Ret [s'; v]] ->
  hexpr [s; e] [s'; v].
```

$$\forall s, e, s_2, s', v,\ (s, e) \rightarrow^*_{\texttt{ehexpr}} (s_2, \texttt{Ret}(s', v)) \implies (s, e) \Downarrow_{\texttt{hexpr}} (s', v)$$

Monadic Small-Step (1/2)

```
type m<a> = ...
val ret<a> (v : a) : m<a> = ...
val bind<a,b> (w : m<a>) (f : a → m<b>) : m<b> = ...
binder @ := bind
```

```
type value =
  | Int int
  | ...
```

```
type term =
  | Plus (term, term)
  | ...
```

```
val add : int -> int -> int
val eval (t : term) : m<value> =
branch
  let Plus (t1, t2) = t in
  let Int n1 =@ eval t1 in
  let Int n2 =@ eval t2 in
  let n = add n1 n2 in
  ret<value> (Int n)
or ... end
```

Monadic Small-Step (2/2)

```
val evalss (t : term) : m<term> =
branch
  let Plus (t1, t2) = t in
  branch
    let t1' =@ evalss t1 in
    ret<term> (Plus(t1', t2))
  or
    let Value (Int n1) = t1 in
    branch
      let t2' =@ evalss t2 in
      ret<term> (Plus(Value (Int n1), t2'))
    or
      let Value (Int n2) = t2 in
      let n = add n1 n2 in
      ret<term> (Value (Int n))
    end
  end
or ... end
```

```
type term =
  | Plus (term, term)
  | ...
  | Value value
```

Limits: Useless states (1/2)

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

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

$$\frac{}{s, \texttt{While1}(s_0, \texttt{Ret}(s', \top), e_1, t_2) \rightarrow s, \texttt{While2}(s', t_2, e_1, t_2)}$$

$$\frac{s_0, t_0 \Downarrow s'' \qquad s'', \texttt{While}(e_1, t_2) \Downarrow s'''}{s, \texttt{While2}(s_0, t_0, e_1, t_2) \Downarrow s'''}$$

Limits: Useless states (2/2)

```
val hexpr (s : state, e : expr) : value =
branch
  let Plus (e1, e2) = e in
  let v1 = hexpr (s, e1) in
  let v2 = hexpr (s, e2) in
  add (v1, v2)
or ... end
```

Produces new constructor `Plus1`.

If state is never modified, need global understanding to simplify the results.

Deterministic Abstract Machine

$$\langle \texttt{Branch}(S :: l), \Sigma, \kappa, f \rangle_{\mathsf{sk}} \rightarrow \langle S, \Sigma, \kappa, \lceil\!\lceil \texttt{Branch}(l), \Sigma, \kappa \rceil\!\rceil :: f \rangle_{\mathsf{sk}}$$

$$\langle \texttt{Branch}([]), \Sigma, \kappa, f \rangle_{\mathsf{sk}} \rightarrow \langle f \rangle_{\mathsf{fk}}$$

$$\langle \texttt{let } p = S_1 \texttt{ in } S_2, \Sigma, \kappa, f \rangle_{\mathsf{sk}} \rightarrow \langle S_1, \Sigma, \lceil \texttt{let } p = \Box \texttt{ in } S_2, \Sigma \rfloor :: \kappa, f \rangle_{\mathsf{sk}}$$

$$\ldots \rightarrow \ldots$$

$$\langle \lceil \texttt{let } p = \Box \texttt{ in } S, \Sigma \rfloor :: \kappa, r, f \rangle_{\mathsf{kr}} \rightarrow \langle p, r, \Sigma, \lceil S, \Box \rfloor :: \kappa, f \rangle_{\mathsf{pat}}$$

$$\langle \lceil S, \Box \rfloor :: \kappa, \Sigma, f \rangle_{\mathsf{ke}} \rightarrow \langle S, \Sigma, \kappa, f \rangle_{\mathsf{sk}}$$

$$\ldots \rightarrow \ldots$$

$$\langle \lceil\!\lceil S, \Sigma, \kappa \rceil\!\rceil :: f \rangle_{\mathsf{fk}} \rightarrow \langle S, \Sigma, \kappa, f \rangle_{\mathsf{sk}}$$