FROM INTERPRETATION TO COMPILATION

INTERPRETER FOR $\mathcal{L}$: a program $\mathcal{I}$ in a language $\mathcal{L}'$ executing $\mathcal{L}$-programs

$$\mathcal{I}(P) = \text{result of } P$$

COMPILER FOR $\mathcal{L}$ IN $\mathcal{L}'$: a program $\mathcal{C}$ translating $\mathcal{L}$-programs to $\mathcal{L}'$-programs

Usually

$\mathcal{L}$: source language

$\mathcal{L}'$: machine language (executable by machine)

Focus on efficiency, energy consumption, code optimisation

INTENSIONAL PROGRAM BEHAVIOURS

What is compiler correctness?

↳ The generated code must meet the semantics of the source program

$$
\begin{tikzcd}
\text{Source Code} \arrow[r, "\text{eval}"] \arrow[d, "e"'] & \text{Source Code} \arrow[d, "e"] \\
\text{Machine Code} \arrow[r, "\text{execute}"'] & \text{Machine Code}
\end{tikzcd}
$$

$$\text{execute}\,(e\,(P)) \stackrel{?}{=} e\,(\text{eval}\,(P))$$

$\Rightarrow$ Need formal semantics of both
- source language
- machine language

Why compiler correctness?

"We tested thirteen production-quality C-compilers and, for each, found situations in which the compiler generated incorrect code [...]"

Eide & Regehr EMSOFT 2008

"To improve the quality of C compilers, we created Csmith [...] Every compiler we tested was found to crash and also to silently generate wrong code [..]"

Yang, Chen, Eide & Regehr
PLDI 2011

Our goal: Formal verification of a (non-optimising) compiler for IMP

Use techniques that scale to real world languages

How to compile IMP ?

IMP

virtual machine

Machine Code for processors (x86, ARM)

- similar to a real machine
  - program = sequences of instructions

  $\Longrightarrow$ no compound exp
  no control structure
- close to the source lang.
  (instructions reflect base operations in source lang.)

IMP virtual machine

4 components:

- Code C : list of instructions
- Code pointer pc : position of the instruction in C we are executing
- Store s : association variables - values
- Stack σ : list of integer values (to save intermediate results)

Instructions

`Inductive instr : Type :=`	
`| Iconst (n: Z)`	"push integer n"
`| Ivar (x: ident)`	"push value of x"
`| Isetvar (x: ident)`	"pop integer and assign it to x"
`| Iadd`	"pop two integers; push their sum"
`| Iopp`	"pop one integer; push its opposite"
`| Ibranch (d: Z)`	"skip "forward" of d instructions"
`| Ibeq (d1: Z) (d2: Z)`	"pop two integers; skip d1 instructions if equal; skip d2 instructions if not"
`| Ible (d1: Z) (d2. Z)`	...
`| Ihalt`	"stop"

NB. All instructions implement PC by 1; branching instructions of d+1

Ex.

$\langle$ code, pc, store, stack $\rangle$ } configurations : states of the machine

$\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(-5), 0, $x \mapsto 12$, [] $\rangle$

Ex.

$\langle code, pc, store, stack \rangle$ } configurations : states of the machine

$$\langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 0, x \mapsto 12, [\,] \rangle$$

$$\longrightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 1, x \mapsto 12, [12] \rangle$$

Ex.

$\langle code, pc, store, stack \rangle$ } configurations: states of the machine

$\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 0, $x \mapsto 12$, [] $\rangle$

$\longrightarrow$ $\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 1, $x \mapsto 12$, [12] $\rangle$

$\longrightarrow$ $\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 2, $x \mapsto 12$, [1, 12] $\rangle$

Ex.

⟨ code, pc, store, stack ⟩ } configurations : states of the machine

⟨ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 0, $x \mapsto 12$, [] ⟩

$\longrightarrow$ ⟨ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 1, $x \mapsto 12$, [12] ⟩

$\longrightarrow$ ⟨ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 2, $x \mapsto 12$, [1, 12] ⟩

$\longrightarrow$ ⟨ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 3, $x \mapsto 12$, [13] ⟩

Ex.

$\langle \text{code}, \text{pc}, \text{store}, \text{stack} \rangle$ } configurations : states of the machine

$\langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 0, x \mapsto 12, [\,] \rangle$

$\to \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 1, x \mapsto 12, [12] \rangle$

$\to \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 2, x \mapsto 12, [1, 12] \rangle$

$\to \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 3, x \mapsto 12, [13] \rangle$

$\to \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 4, x \mapsto 13, [\,] \rangle$

Ex.

$\langle code, pc, store, stack \rangle$ } configurations : states of the machine

$\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 0, $x \mapsto 12$, [] $\rangle$

$\longrightarrow$ $\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 1, $x \mapsto 12$, [12] $\rangle$

$\longrightarrow$ $\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 2, $x \mapsto 12$, [1, 12] $\rangle$

$\longrightarrow$ $\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 3, $x \mapsto 12$, [13] $\rangle$

$\longrightarrow$ $\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 4, $x \mapsto 13$, [] $\rangle$

$\longrightarrow$ $\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 0, $x \mapsto 13$, [] $\rangle$

We specify the vm using operational semantics

The pc-th element of C is IConst(n)

$$\frac{C[pc] = \mathrm{IConst}(n)}{\langle C, pc, s, \sigma \rangle \rightarrow \langle C, pc+1, s, n :: \sigma \rangle}$$

$$\frac{C[pc] = \mathrm{IVar}(x)}{\langle C, pc, s, \sigma \rangle \rightarrow \langle C, pc+1, s, s(x) :: \sigma \rangle}$$

$$\frac{C[pc] = \mathrm{ISetvar}(x)}{\langle C, pc, s, n :: \sigma \rangle \rightarrow \langle C, pc+1, s[x \mapsto n], \sigma \rangle}$$

$$\frac{C[pc] = \mathrm{Iadd}}{\langle C, pc, s, n :: m :: \sigma \rangle \to \langle C, pc+1, s, (n+m) :: \sigma \rangle}$$

$$\vdots$$

$$\frac{C[pc] = \mathrm{Ibranch}(d)}{\langle C, pc, s, \sigma \rangle \to \langle C, pc+1+d, s, \sigma \rangle}$$

$$\vdots$$

How to encode op. sem in Coq?

```
Definition code := list instruction
Definition store := ident -> option Z
Definition stack := list Z
Definition config := Z * store * stack        (~> no code!)
```

Inductive Transition (c : code) : $\underbrace{\text{config} \to \text{config} \to \text{Prop}}_{\text{A relation on configuration}}$:=

| rule_iconst : ∀ pc s σ,
    if c[pc] = Some (Ivar (n))

⋮

Behaviours of the VM

- **Termination**

  $\langle c, pc, s, \sigma \rangle \to^* \langle c, pc', s', \sigma' \rangle$ and $c[pc'] = \mathrm{Ihalt}$

  Notation. $\langle c, pc, s, \sigma \rangle \Downarrow s'$

- **Divergence**

  Infinitely many transitions from $\langle c, pc, s, \sigma \rangle$

  Notation. $\langle c, pc, s, \sigma \rangle \Uparrow$

- **Going wrong** Otherwise

NB Machine programs can go wrong

E.g. $\langle \mathtt{Iadd} ; \mathtt{Ihalt}, pc, s, \cdot \rangle$ (empty stack)

Digression : What is a program behaviour ?

- Not a precisely defined notion ...
- Semantics usually induce notions of behaviour
- In case of an SOS $(\Delta, \to)$ behaviour defined thus

① Specify a collection $\Delta_F \subseteq \Delta$ of final states

coherence. $\forall s \in \Delta_F . \neg \exists s' \in \Delta . s \to s'$

② Termination

$s \Downarrow \quad :\text{iff} \quad \exists s' \in \Delta_F . s \to^* s'$

Recall that, for $R \subseteq A \times A$, R^* is the reflexive and transitive closure of R.

R^* = smallest $X \subseteq A \times A$ s.t.

- $R \subseteq X$
- $Id \subseteq X$
- $X ; X \subseteq X$

$$\Longrightarrow \quad \frac{R \subseteq Q \qquad Id \subseteq Q \qquad Q;Q \subseteq Q}{R^* \subseteq Q} \quad (*\text{-induction})$$

$$\frac{a \, R \, a'}{a \, R^* \, a'} \qquad \frac{}{a \, R^* \, a}$$

$$\frac{a \, R^* \, b \qquad b \, R^* \, c}{a \, R^* \, c}$$

$a R b$

Constructively: $R^* = \bigcup_{n \geq 0} R^{(n)}$

where $R^{(0)} = Id$

$R^{(n+1)} = R ; R^{(n)}$

Termination

$s \Downarrow$ iff $\exists s'.\ s' \in \Delta_F \ \wedge\ s \to^* s'$

Divergence

$s \Uparrow$ iff $\forall s'.\ s \to^* s' \implies \exists s''.\ s' \to s''$

Goes Wrong

$s \not\Downarrow$ iff $\exists s'.\ \underbrace{s' \notin \Delta_F \ \wedge\ s' \not\to}_{\text{stuck states}} \ \wedge\ s \to^* s'$

$(\neg \exists s''.\ s' \to s'')$

NB. These behaviours are exhaustive

In case of VM

$$\mathcal{S} = \{ \langle c, pc, s, \sigma \rangle \mid c \text{ code}, \ pc \in \mathbb{Z}, \ \ldots \}$$

$$\mathcal{S}_F = \{ \langle c, pc, s, \sigma \rangle \mid \ldots \quad c[pc] = \text{I halt} \}$$

In case of IMP

$$\mathcal{S} = \{ (c, s) \mid c \in \text{Comm} \ \& \ s : \text{Var} \to \text{option}(\mathbb{Z}) \}$$

$$\mathcal{S}_F = \{ (\text{skip}, s) \mid s : \text{Var} \to \text{option}(\mathbb{Z}) \}$$

<u>Prop.</u> Programs do not go wrong:

$$\forall (c, s). \quad (c, s) \Downarrow \ \vee \ (c, s) \Uparrow$$

} safety properties

Compilation

We translate IMP programs into VM code

$comp_A : AExp \to Code$

$comp_A(x) = Ivar(x)$

$comp_A(n) = Iconst(n)$

$comp_A(e_1 + e_2) = \underbrace{comp_A(e_1) :: comp_A(e_2) :: Iadd}_{\text{reverse polish notation}}$

$comp_B : BExp \times \mathbb{Z} \times \mathbb{Z} \to Code$

$comp_B\,(e_1 = e_2, d_0, d_1) = comp_A\,(e_1) :: comp_A\,(e_2) :: Ieq\,(d_0, d_1)$

$comp_B\,(e_1 \leq e_2, d_0, d_1) = comp_A\,(e_1) :: comp_A\,(e_2) :: Ileq\,(d_0, d_1)$

$comp : Com \to Code$

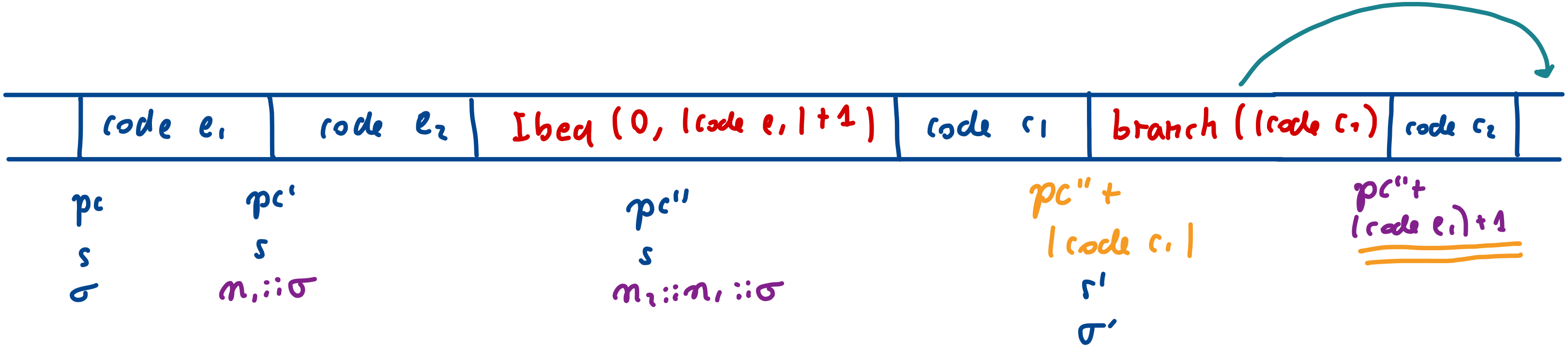
$comp\,(skip) = [\,]$

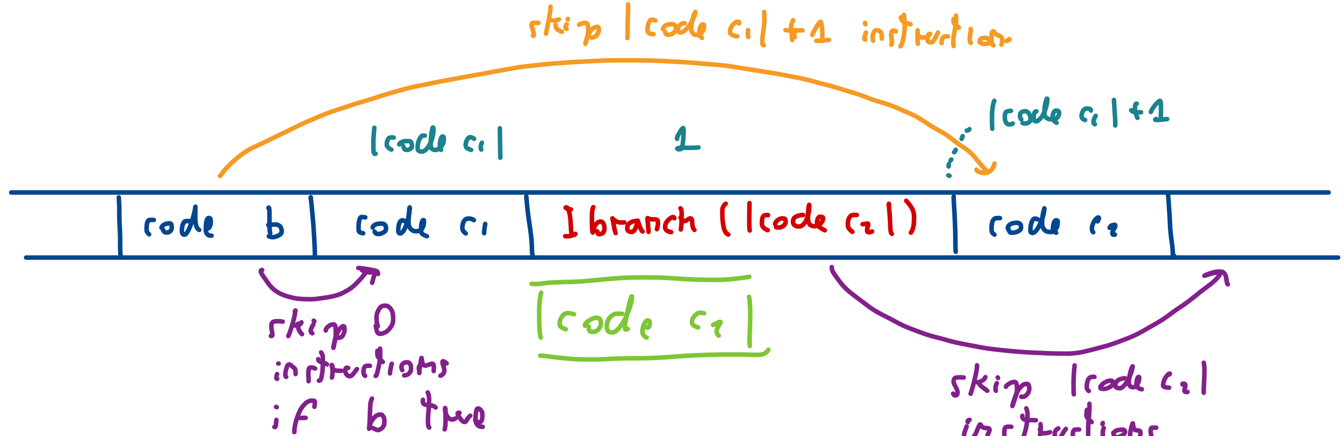
$comp\,(x := e) = comp(e) :: Isetvar\,(x)$

$comp\,(c_1 ; c_2) = comp(c_1) :: comp(c_2)$

$comp\,(\text{if } b \text{ then } c_1 \text{ else } c_2) = comp_B(b, 0, |comp(c_1)| + 1)$
$:: comp\,(c_1) :: branch\,(\,|comp(c_2)|\,)$
$:: comp\,(c_2)$

?

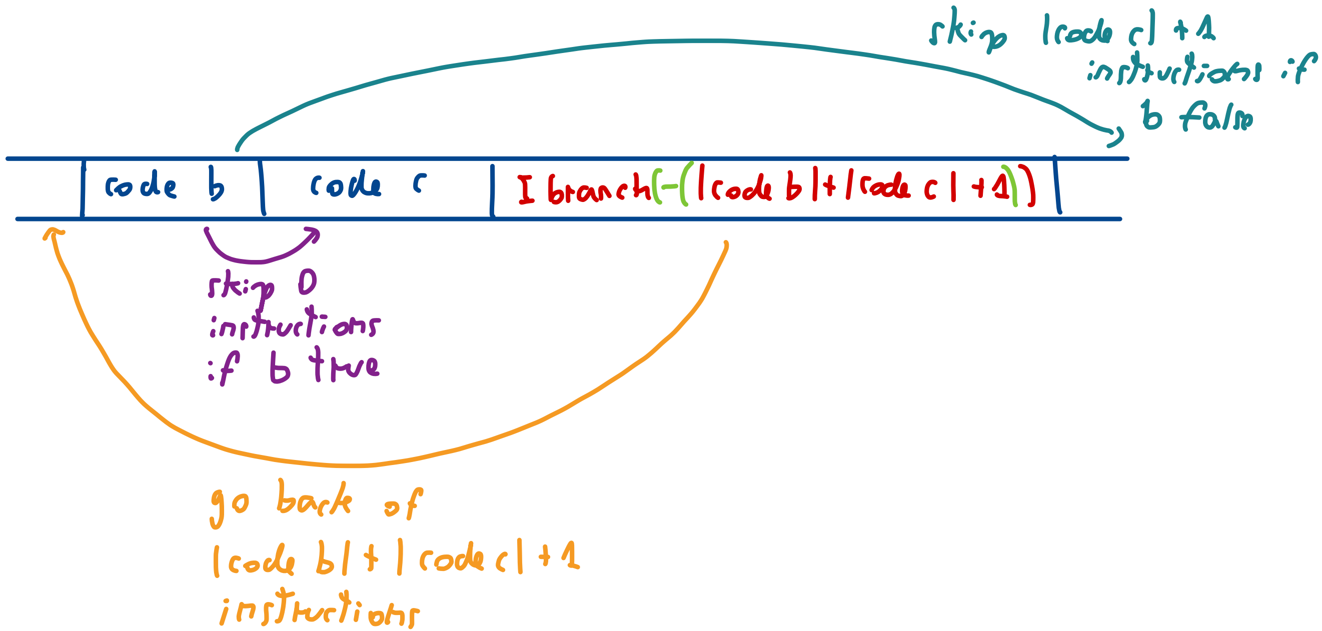




if b then c_1 else c_2

$|\text{code } c_1| = n_1$ $\quad |\text{code } c_2| = n_2$

$$\begin{aligned}\mathrm{comp}(\texttt{while } b \texttt{ do } c) = {} & \mathrm{comp}(b, 0, |\mathrm{code}\ c| + 1) \\ & :: \mathrm{comp}(c) \\ & :: \mathrm{Ibranch}(|\mathrm{code}\ b| + |\mathrm{code}\ c| + 1)\end{aligned}$$



Finally compilation

$$\text{compile}(c) = \text{comp}(c) :: \text{Ihalt}$$

New way to execute IMP programs

① IMP operational semantics ⇝ interpreter
② VM operational semantics ⇝ compiler

$\Big\}$ Are they the same ??

$$\begin{tikzcd}
\text{IMP} & (c, s) \arrow[rr, dotted] \arrow[d, "\text{compile}"] & & n \\
\text{VM} & \langle \text{compile}(c), 0, s, \cdot \rangle \arrow[r, "*"] & \langle \text{compile}(c), pc, s, n :: \sigma \rangle &
\end{tikzcd}$$

Compiler Correctness

Intuition

$\forall c \in Com, \forall s \in Store. \left(\langle c, s \rangle \Downarrow s' \implies \langle compile(c), 0, s, \cdot \rangle \Downarrow s' \right)$

Q. • What about divergence?

• What if we require $\iff$?

• What about going wrong?

Def. Given languages $\mathcal{L}_1, \mathcal{L}_2$ and programs $P_1 \in \mathcal{L}_1$, $P_2 \in \mathcal{L}_2$,
let $beh(P_i)$ the collection of observable behaviours of P_i.
For us, possible behaviours are: $\Downarrow, \Uparrow, \lightning$.
Then:

- $P_1 \approx P_2$ (bisimulation) if $beh(P_1) = beh(P_2)$
- $P_1 \lesssim P_2$ (simulation) if $beh(P_1) \subseteq beh(P_2)$
    - Forward simulation $\lesssim$
    - Backward simulation $\gtrsim$
- $P_1 \sqsubseteq^f P_2$ (correct-only forward sim) if
  $\lightning \notin beh(P_1) \Longrightarrow beh(P_1) \subseteq beh(P_2)$
- $P_1 \sqsupseteq^b P_2$ (correct-only backward sim) if
  $\lightning \notin beh(P_1) \Longrightarrow beh(P_1) \supseteq beh(P_2)$

Which notion of correctness should we pick?

① Bisimulation is the strongest notion ($\simeq \; = \; \lesssim \cap \gtrsim \cap \equiv^f \cap \equiv^b$)

$\leadsto$ Usually too strong

E.g. C has nondeterministic semantics (evaluation order of some expressions is <u>not</u> specified), but compilers choose an order while generating deterministic machine code

$$\mathrm{beh}(P_{source}) \supsetneq \mathrm{beh}(P_{target})$$

$$P_{source} \gtrsim P_{target}$$

(<u>Lemma</u>. If $(S, \to)$ is such that $\to$ is deterministic (i.e. $\forall s, s_1, s_2 \in S.\ (s \to s_1 \wedge s \to s_2) \Longrightarrow s_1 = s_2$), then $|\mathrm{beh}(s)| = 1$, for any s)

② Compilers optimise away 'going wrong' behaviours

P_{source} =

$\vdots$
$z := x / y$
$\vdots$
never use z

$\xrightarrow{\text{compile}}$

$\vdots$
$\cancel{z := x / y}$
$\vdots$
never use z

$\underbrace{\overbrace{z := x / y}\,;\, skip}$

$(P_{source}, \sigma[y \mapsto 0])$ ↯

$\underbrace{(P_{target}, \ldots)}$ ✓

$\rightsquigarrow$ backward correct-only simulation

$$↯ \notin beh(P_{source}) \implies beh(P_{source}) \supseteq beh(P_{target})$$

$$P_{source} \sqsupseteq^{b} P_{target}$$

In particular: $P_{source} \not↯ \implies P_{source} \sqsupseteq^{b} P_{target}$

③ $\sqsupseteq^b$ seems the right notion for IMP and VM.

Working with $\sqsupseteq^b$ usually not easy.

What about $\sqsubseteq^f$?

(3.1) $\sqsubseteq^f$ easier than $\sqsupseteq^b$

(3.2) $\sqsubseteq^f$ weaker than $\sqsupseteq^b$: P_{target} can have more behaviours than P_{source}

but...

Lemma. If $(\Delta, \rightarrow)$ deterministic, then $\sqsubseteq^f \subseteq \sqsupseteq^b$

We can thus rely on $\sqsubseteq^f$ for IMP and VM

To prove correctness of compilation in VM, we thus have to prove forward simulation

① $\forall c, s, s'. \left(\langle c, s \rangle \Downarrow s' \rightarrow \langle \text{compile}(c), 0, s, \cdot \rangle \Downarrow s' \right)$

Sketch of the proof. Main techniques used

- induction on op. semantics or syntax
- case analysis
- inversion

IH

?

syntactic shape → conclusion rule op. sem. → premises rule op sem

1.1 Arithmetic

Lemma. $\forall e_1, e_2 : \text{code}, \forall s, \forall \sigma.$

$$\langle e_1 ; \text{comp}(e) ; e_2 , |e_1| , s , \sigma \rangle \rightarrow^* \langle e_1 ; \text{comp}(e) ; e_2 , |e_1| + |\text{comp}(e)| , s , [\![e]\!](s) :: \sigma \rangle$$

Corollary.

$$\forall e, s, \sigma. \quad \langle comp(e), 0, s, \cdot \rangle \to^* \langle comp(e), |comp(e)|+1, s, [\![e]\!](s) \rangle$$

denote e s

Proof. By induction on e

1.2 Boolean Expressions

Lemma $\forall e_1, e_2.$

$$\langle e_1; comp(b, d); e_2, |e_1|, s, \sigma \rangle \to^* \langle e_1; comp(b, d); e_2, pc, s, \sigma \rangle$$

where

$$pc = \begin{cases} |e_1| + |comp(b)| & \text{if } [\![b]\!]s = true \\ |e_1| + |comp(b)| + \delta & \text{otherwise} \end{cases}$$

Theorem

$\forall c, s, s', e_1, e_2, \sigma$. implies

$$\langle c, s \rangle \Downarrow s' \longrightarrow \langle e_1; \mathit{comp}(c); e_2, |e_1|, s, \sigma \rangle$$
$$\to^* \langle e_1; \mathit{comp}(c); e_2, |e_1| + |\mathit{comp}(c)|, s', \sigma \rangle$$

Proof: Induction on $\langle c, s \rangle \Downarrow s'$

To prove $\sqsubseteq^f$, we still need to handle divergence...

How to do that?