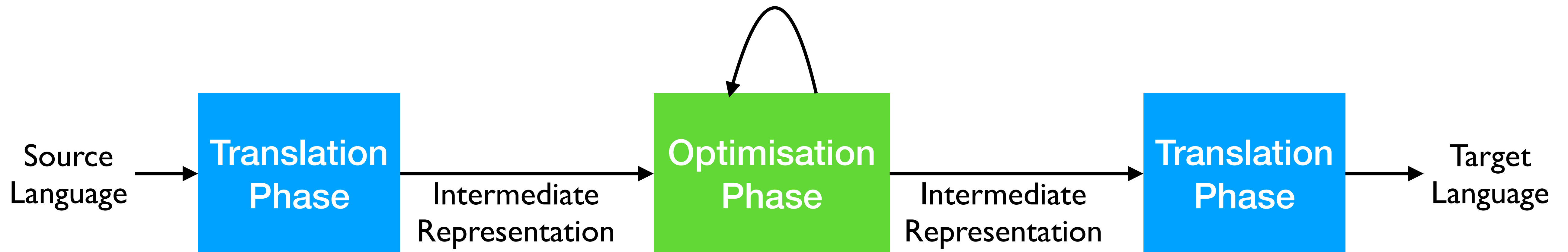
Compiler Correctness

KC Sivaramakrishnan

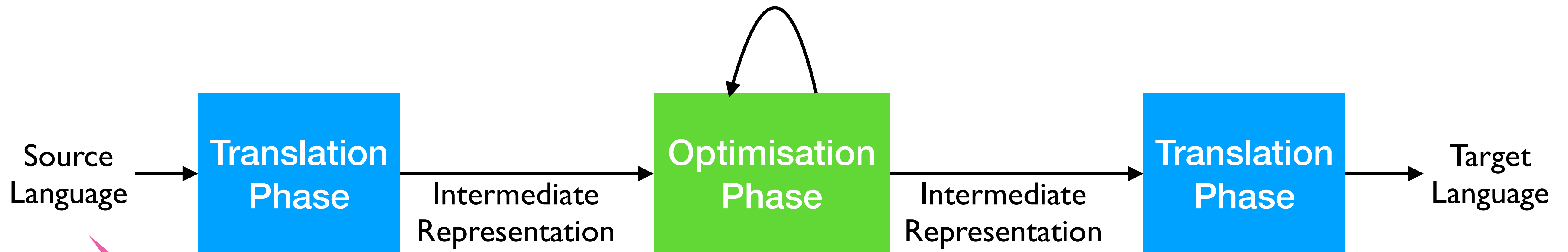
Spring 2021

IIT
MADRAS

Why prove compilers correct?

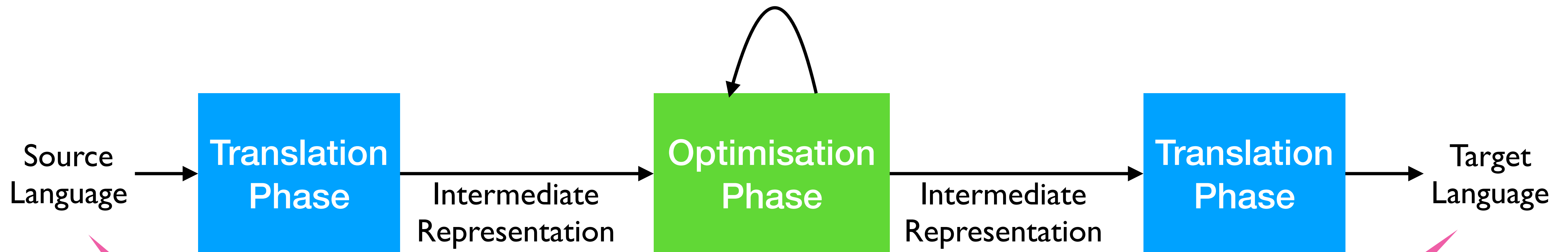


Why prove compilers correct?



I have some semantics for the source language

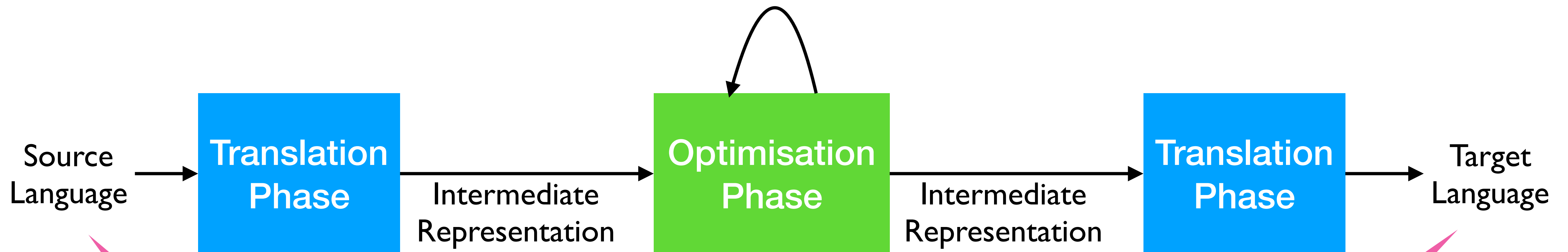
Why prove compilers correct?



I have some semantics for the source language

Does the target language have ***equivalent*** semantics?

Why prove compilers correct?



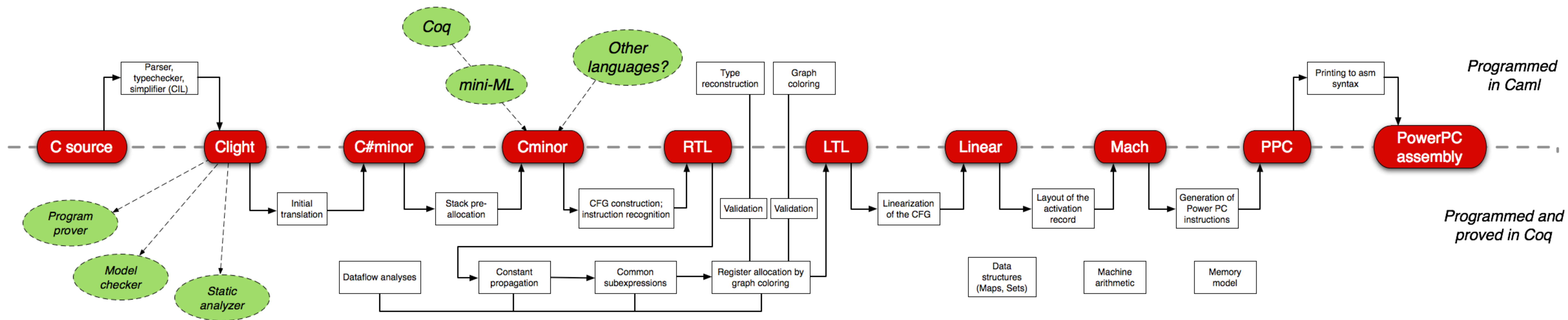
I have some semantics for the source language

Does the target language have ***equivalent*** semantics?

Idea: Prove each step of the translation and optimisation correct through *Simulation*

Compcert Verified C Compiler

https://compcert.org/



Language

$$
\begin{array}{rccl}
\text{Numbers} & n & \in & \mathbb{N} \\
\text{Variables} & x & \in & \mathsf{Strings} \\
\text{Expressions} & e & ::= & n \mid x \mid e + e \mid e - e \mid e \times e \\
\text{Commands} & c & ::= & \mathsf{skip} \mid x \leftarrow e \mid c; c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \underline{\mathsf{out}(e)}
\end{array}
$$

Language

Numbers	n	$\in$	$\mathbb{N}$
Variables	x	$\in$	Strings
Expressions	e	$::=$	$n \mid x \mid e + e \mid e - e \mid e \times e$
Commands	c	$::=$	$\mathsf{skip} \mid x \leftarrow e \mid c; c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \underline{\mathsf{out}(e)}$

- An optimising compiler should preserve the behaviour of the program in terms of the output *traces* generated.

Language

$$\begin{array}{rccl} \text{Numbers} & n & \in & \mathbb{N} \\ \text{Variables} & x & \in & \textsf{Strings} \\ \text{Expressions} & e & ::= & n \mid x \mid e+e \mid e-e \mid e\times e \\ \text{Commands} & c & ::= & \textsf{skip} \mid x \leftarrow e \mid c;c \mid \textsf{if}\ e\ \textsf{then}\ c\ \textsf{else}\ c \mid \textsf{while}\ e\ \textsf{do}\ c \mid \underline{\textsf{out}(e)} \end{array}$$

- An optimising compiler should preserve the behaviour of the program in terms of the output *traces* generated.
- Equivalence of traces is fundamental correctness property
  - ✦ Invariants — safety
  - ✦ Trace equivalence — liveness (not only for terminating programs)

Labelled transition semantics

$$\frac{}{(v, \mathsf{out}(e)) \xrightarrow{[\![e]\!]v}_0 (v, \mathsf{skip})}$$

$$\frac{}{(v, x \leftarrow e) \xrightarrow{\epsilon}_0 (v[x \mapsto [\![e]\!]v], \mathsf{skip})} \qquad \frac{}{(v, \mathsf{skip}; c_2) \xrightarrow{\epsilon}_0 (v, c_2)}$$

$$\frac{[\![e]\!]v \neq 0}{(v, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \xrightarrow{\epsilon}_0 (v, c_1)} \qquad \frac{[\![e]\!]v = 0}{(v, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \xrightarrow{\epsilon}_0 (v, c_2)}$$

$$\frac{[\![e]\!]v \neq 0}{(v, \mathsf{while}\ e\ \mathsf{do}\ c_1) \xrightarrow{\epsilon}_0 (v, c_1; \mathsf{while}\ e\ \mathsf{do}\ c_1)} \qquad \frac{[\![e]\!]v = 0}{(v, \mathsf{while}\ e\ \mathsf{do}\ c_1) \xrightarrow{\epsilon}_0 (v, \mathsf{skip})}$$

$$\frac{(v, c) \xrightarrow{\ell}_0 (v', c')}{(v, C[c]) \xrightarrow{\ell}_{\mathsf{c}} (v', C[c'])}$$

Traces

- Finite sequences of outputs and termination events

$$\frac{}{\cdot \in \mathsf{Tr}(s)} \qquad \frac{}{\mathsf{terminate} \in \mathsf{Tr}((v, \mathsf{skip}))} \qquad \frac{s \xrightarrow{\epsilon}_{\mathsf{c}} s' \quad t \in \mathsf{Tr}(s')}{t \in \mathsf{Tr}(s)} \qquad \frac{s \xrightarrow{n}_{\mathsf{c}} s' \quad t \in \mathsf{Tr}(s')}{\mathsf{out}(n) \bowtie t \in \mathsf{Tr}(s)}$$

- A trace is allowed to end even if the program hasn't terminated

Traces

- Finite sequences of outputs and termination events

$$\frac{}{\cdot \in \mathsf{Tr}(s)} \qquad \frac{}{\mathsf{terminate} \in \mathsf{Tr}((v, \mathsf{skip}))} \qquad \frac{s \xrightarrow{\epsilon}_{\mathsf{c}} s' \quad t \in \mathsf{Tr}(s')}{t \in \mathsf{Tr}(s)} \qquad \frac{s \xrightarrow{n}_{\mathsf{c}} s' \quad t \in \mathsf{Tr}(s')}{\mathsf{out}(n) \bowtie t \in \mathsf{Tr}(s)}$$

- A trace is allowed to end even if the program hasn't terminated

DEFINITION 9.1 (Trace inclusion). For commands c_1 and c_2, let $c_1 \preceq c_2$ iff $\mathsf{Tr}(c_1) \subseteq \mathsf{Tr}(c_2)$.

DEFINITION 9.2 (Trace equivalence). For commands c_1 and c_2, let $c_1 \simeq c_2$ iff $\mathsf{Tr}(c_1) = \mathsf{Tr}(c_2)$.

Traces

- Finite sequences of outputs and termination events

$$\frac{}{\cdot \in \mathsf{Tr}(s)} \quad \frac{}{\mathsf{terminate} \in \mathsf{Tr}((v, \mathsf{skip}))} \quad \frac{s \xrightarrow{\epsilon}_{\mathsf{c}} s' \quad t \in \mathsf{Tr}(s')}{t \in \mathsf{Tr}(s)} \quad \frac{s \xrightarrow{n}_{\mathsf{c}} s' \quad t \in \mathsf{Tr}(s')}{\mathsf{out}(n) \bowtie t \in \mathsf{Tr}(s)}$$

- A trace is allowed to end even if the program hasn't terminated

DEFINITION 9.1 (Trace inclusion). For commands c_1 and c_2, let $c_1 \preceq c_2$ iff $\mathsf{Tr}(c_1) \subseteq \mathsf{Tr}(c_2)$.

DEFINITION 9.2 (Trace equivalence). For commands c_1 and c_2, let $c_1 \simeq c_2$ iff $\mathsf{Tr}(c_1) = \mathsf{Tr}(c_2)$.

$$\begin{aligned} & c_1 \simeq c2 \\ \iff & \mathrm{Tr}(c_1) = \mathrm{Tr}(c_2) \\ \iff & \mathrm{Tr}(c_1) \subseteq \mathrm{Tr}(c_2) \wedge \mathrm{Tr}(c_2) \subseteq \mathrm{Tr}(c_1) \\ \iff & c_1 \preceq c_2 \wedge c_2 \preceq c_1 \end{aligned}$$

Constant-folding

- Optimisation is
  - ✦ Find all maximal program subexpressions that don't contain variables
  - ✦ Replace each subexpression with its known constant value
- The optimisation only changes variable free-expressions
  - ✦ The original and the optimised program match at each *step*

Basic Simulation Relation

Definition 9.3 (Simulation relation). We say that binary relation R over states of our object language is a *simulation relation* iff:

(1) Whenever $(v_1, \mathsf{skip})\ R\ (v_2, c_2)$, it follows that $c_2 = \mathsf{skip}$.

(2) Whenever $s_1\ R\ s_2$ and $s_1 \xrightarrow{\ell}_{\mathsf{c}} s_1'$, there exists s_2' such that $s_2 \xrightarrow{\ell}_{\mathsf{c}} s_2'$ and $s_1'\ R\ s_2'$.

$$\begin{array}{ccc} s_1 & \xrightarrow{R} & s_2 \\ \downarrow \forall \xrightarrow{\ell}_{\mathsf{c}} & & \downarrow \exists \xrightarrow{\ell}_{\mathsf{c}} \\ s_1' & \xleftarrow[R^{-1}]{} & s_2' \end{array}$$

Theorem 9.4. *If there exists a simulation R such that $s_1\ R\ s_2$, then $s_1 \simeq s_2$.*

Simulation for Constant Folding

THEOREM 9.5. *For any v and c,* $(v, c) \simeq (v, \mathsf{cfold}_1(c))$.

PROOF. By a simulation argument using this relation:

$$(v_1, c_1)\ R\ (v_2, c_2) \quad = \quad v_1 = v_2 \land c_2 = \mathsf{cfold}_1(c_1)$$

Basic Simulation Relation

THEOREM 9.4. *If there exists a simulation R such that $s_1 \; R \; s_2$, then $s_1 \simeq s_2$.*

$$\begin{array}{ccc} s_1 & \xrightarrow{R} & s_2 \\ \Big\downarrow \forall \xrightarrow{\ell}_{\mathsf{c}} & & \Big\downarrow \exists \xrightarrow{\ell}_{\mathsf{c}} \\ s_1' & \xleftarrow[R^{-1}]{} & s_2' \end{array}$$

- We prove two trace inclusion directions separately
- Left-to-right Tr(s1) ⊆ Tr(s2) proved by induction on traces on the left
- Right-to-left Tr(s2) ⊆ Tr(s1) proved similarly
  - ✦ Depends on operational semantics being total and deterministic

Simulation with skipping

- Consider extension of constant folding to conditionals

$$\frac{\mathrm{cfoldArith}(e_1) = 1}{\mathrm{cfold}(\mathrm{if}\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3) = \mathrm{cfold}(e_2)}$$

$$\frac{\mathrm{cfoldArith}(e_1) = 0}{\mathrm{cfold}(\mathrm{if}\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3) = \mathrm{cfold}(e_3)}$$

- We can no longer use our basic simulation
  - ✦ Steps are intentionally skipped in the optimised program

Simulation with Skipping: Faulty

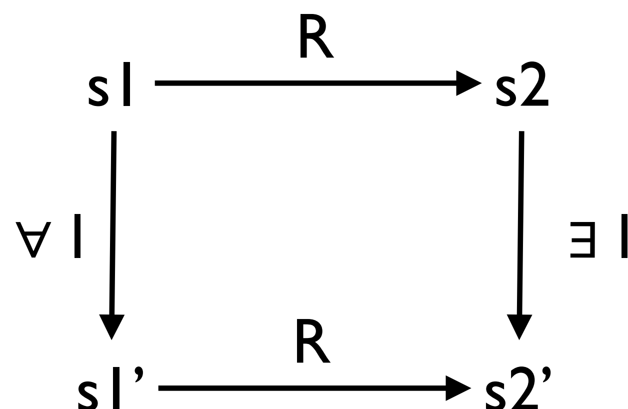
DEFINITION 9.6 (Simulation relation with skipping (*faulty* version!)). We say that binary relation R over states of our object language is a *simulation relation with skipping* iff:

(1) Whenever $(v_1, \mathsf{skip})\ R\ (v_2, c_2)$, it follows that $c_2 = \mathsf{skip}$.

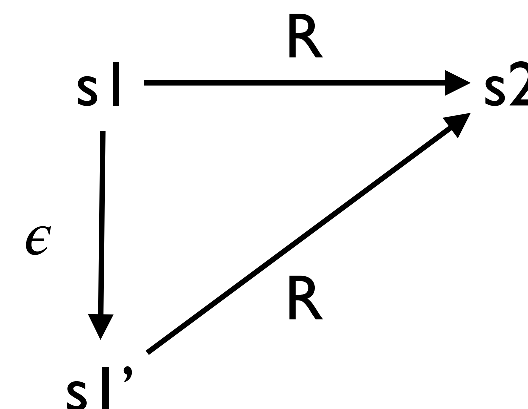
(2) Whenever $s_1\ R\ s_2$ and $s_1 \stackrel{\ell}{\rightarrow}_{\mathsf{c}} s_1'$, then either:

  (a) there exists s_2' such that $s_2 \stackrel{\ell}{\rightarrow}_{\mathsf{c}} s_2'$ and $s_1'\ R\ s_2'$,

  (b) or $\ell = \epsilon$ and $s_1'\ R\ s_2$.



(a)



(b)

Faulty Optimisation: Advertising

$$\begin{aligned} \mathrm{withAds}(\mathrm{while}\ 1\ \mathrm{do}\ \mathrm{Skip}) &= \mathrm{while}\ 1\ \mathrm{do}\ \mathrm{Out}(0) \\ \mathrm{withAds}(\mathrm{c}) &= \mathrm{c} \end{aligned}$$

Faulty Optimisation: Advertising

$$\begin{aligned} \text{withAds}(\text{while } 1 \text{ do Skip}) &= \text{while } 1 \text{ do Out}(0) \\ \text{withAds}(\text{c}) &= \text{c} \end{aligned}$$

- We can use the candidate simulation relation

Faulty Optimisation: Advertising

$$\begin{aligned} \text{withAds}(\text{while 1 do Skip}) &= \text{while 1 do Out}(0) \\ \text{withAds}(\text{c}) &= \text{c} \end{aligned}$$

- We can use the candidate simulation relation

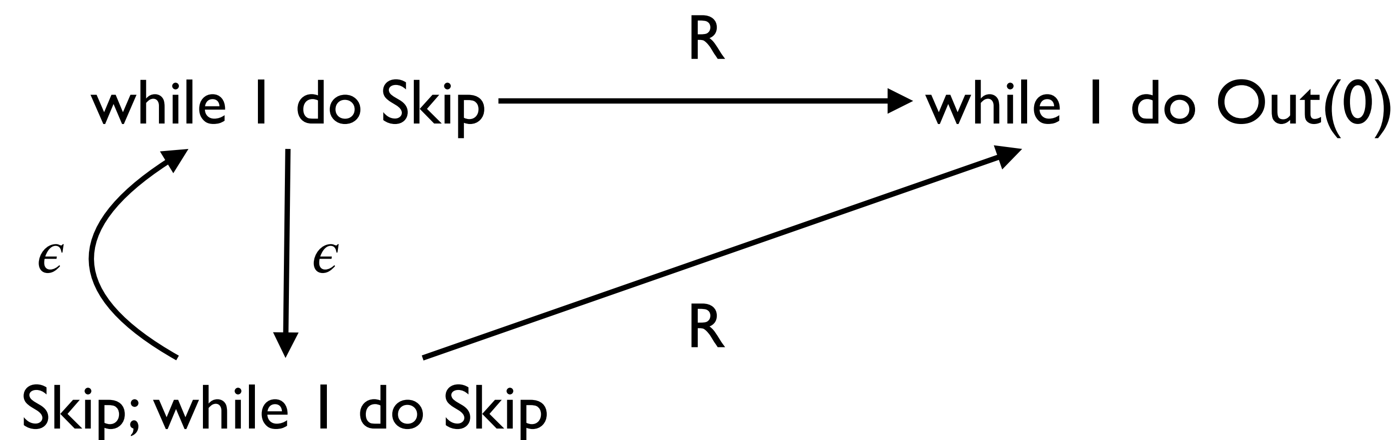
$$(v_1, c_1)\ R\ (v_2, c_2) \quad = \quad c_1 \in \{\mathsf{while}\ 1\ \mathsf{do}\ \mathsf{skip}, (\mathsf{skip}; \mathsf{while}\ 1\ \mathsf{do}\ \mathsf{skip})\}$$

Faulty Optimisation: Advertising

$$\begin{aligned}\text{withAds(while 1 do Skip)} &= \text{while 1 do Out(0)} \\ \text{withAds(c)} &= \text{c}\end{aligned}$$

- We can use the candidate simulation relation

$$(v_1, c_1)\ R\ (v_2, c_2) \quad = \quad c_1 \in \{\text{while 1 do skip, (skip; while 1 do skip)}\}$$



Simulation with skipping

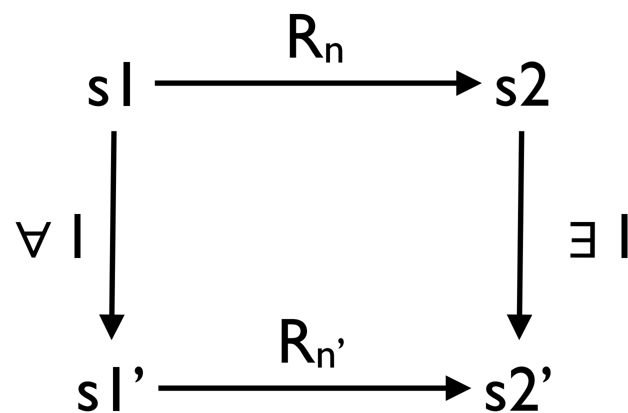
Definition 9.7 (Simulation relation with skipping). We say that an $\mathbb{N}$-indexed family of binary relations R_n over states of our object language is a *simulation relation with skipping* iff:

(1) Whenever $(v_1, \mathsf{skip})\ R_n\ (v_2, c_2)$, it follows that $c_2 = \mathsf{skip}$.

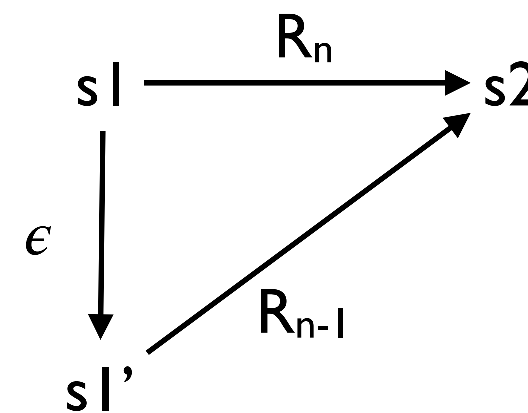
(2) Whenever $s_1\ R_n\ s_2$ and $s_1 \xrightarrow{\ell}_{\mathsf{c}} s_1'$, then either:

  (a) there exist n' and s_2' such that $s_2 \xrightarrow{\ell}_{\mathsf{c}} s_2'$ and $s_1'\ R_{n'}\ s_2'$,

  (b) or $n > 0$, $\ell = \epsilon$, and $s_1'\ R_{n-1}\ s_2$.



(a)



(b)

Trace Equivalence for Simulation with Skipping

THEOREM 9.8. *If there exists a simulation with skipping R such that $s_1 \; R_n \; s_2$, then $s_1 \simeq s_2$.*

THEOREM 9.9. *For any v and c, $(v, c) \simeq (v, \mathsf{cfold}_2(c))$.*

PROOF. By a simulation argument (with skipping) using this relation:

$$(v_1, c_1) \; R_n \; (v_2, c_2) \quad = \quad v_1 = v_2 \wedge c_2 = \mathsf{cfold}_2(c_1) \wedge \mathsf{countIfs}(c_1) < n$$