The formal verification of compilers

Xavier Leroy

Inria Paris

DeepSpec Summer School 2017

Prologue

Mechanized semantics and its applications

Formal semantics of programming languages

Provide a mathematically-precise answer to the question

What does this program do, exactly?

What does this program do, exactly?

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l                ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

(Raymond Cheong, 2001)

What does this program do, exactly?

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l                ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

(Raymond Cheong, 2001)

(It computes arbitrary-precision square roots.)

What about this one?

```
#define crBegin static int state=0; switch(state) { case 0:
#define crReturn(x) do { state=__LINE__; return x; \
                         case __LINE__:; } while (0)
#define crFinish }


int decompressor(void) {
    static int c, len;
    crBegin;
    while (1) {
        c = getchar();
        if (c == EOF) break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--) crReturn(c);
        } else crReturn(c);
    }
    crReturn(EOF);
    crFinish;
}
```

(Simon Tatham, author of PuTTY)

What about this one?

```
#define crBegin static int state=0; switch(state) { case 0:
#define crReturn(x) do { state=__LINE__; return x; \
                         case __LINE__:; } while (0)
#define crFinish }
```

```
int decompressor(void) {
    static int c, len;
    crBegin;
    while (1) {
        c = getchar();
        if (c == EOF) break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--) crReturn(c);
        } else crReturn(c);
    }
    crReturn(EOF);
    crFinish;
}
```

(Simon Tatham, author of PuTTY)

(It's a co-routined version of a decompressor for run-length encoding.)

When is formal semantics necessary?

- When English prose is not enough.
  (e.g. language standardization documents.)
- A prerequisite to formal program verification.
  (Program proof, model checking, static analysis, etc.)
- A prerequisite to building reliable programming tools.
  (Programs that operate over programs: compilers, code generators, program verifiers, type-checkers, . . .)

Is this program transformation correct?

```
struct list { int head; struct list * tail; };

struct list * foo(struct list ** p)
{
  return ((*p)->tail = NULL);
}
```

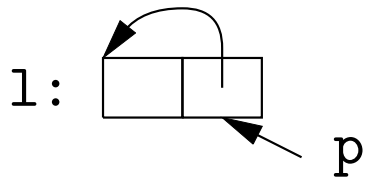
```
(*p)->tail = NULL;
return (*p)->tail;
```

Is this program transformation correct?

```
struct list { int head; struct list * tail; };

struct list * foo(struct list ** p)
{
  return ((*p)->tail = NULL);
}
```

```
(*p)->tail = NULL;
return (*p)->tail;
```

No, not if `p == &(l.tail)` and `l.tail == &l` (circular list).



What about this one?

```
double dotproduct(int n, double * a, double * b)
{
    double dp = 0.0;
    int i;
    for (i = 0; i < n; i++) dp += a[i] * b[i];
    return dp;
}
```

Compiled for the Alpha processor with all optimizations and manually decompiled back to C...

```
double dotproduct(int n, double * a, double * b)
{
     double dp, a0, a1, a2, a3, b0, b1, b2, b3;
     double s0, s1, s2, s3, t0, t1, t2, t3;
     int i, k;
     dp = 0.0;
     if (n <= 0) goto L5;
     s0 = s1 = s2 = s3 = 0.0;
     i = 0; k = n - 3;
     if (k <= 0 || k > n) goto L19;
     i = 4; if (k <= i) goto L14;
     a0 = a[0];  b0 = b[0];  a1 = a[1];  b1 = b[1];
     i = 8; if (k <= i) goto L16;
L17: a2 = a[2];  b2 = b[2];  t0 = a0 * b0;
     a3 = a[3];  b3 = b[3];  t1 = a1 * b1;
     a0 = a[4];  b0 = b[4];  t2 = a2 * b2;  t3 = a3 * b3;
     a1 = a[5];  b1 = b[5];
     s0 += t0; s1 += t1;  s2 += t2;  s3 += t3;
     a += 4; i += 4; b += 4;
     prefetch(a + 20); prefetch(b + 20);
     if (i < k) goto L17;
L16: s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
     a += 4; b += 4;
     a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1];
L18: s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
     a += 4; b += 4;
     dp = s0 + s1 + s2 + s3;
     if (i >= n) goto L5;
L19: dp += a[0] * b[0];
     i += 1; a += 1; b += 1;
     if (i < n) goto L19;
L5:  return dp;
L14: a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1]; goto L18;
}
```

```
      if (n <= 0) goto L5;
      s0 = s1 = s2 = s3 = 0.0;
      i = 0; k = n - 3;
      if (k <= 0 || k > n) goto L19;
      i = 4; if (k <= i) goto L14;
      a0 = a[0];  b0 = b[0];  a1 = a[1];  b1 = b[1];
      i = 8; if (k <= i) goto L16;
L17:  a2 = a[2];  b2 = b[2];  t0 = a0 * b0;
      a3 = a[3];  b3 = b[3];  t1 = a1 * b1;
      a0 = a[4];  b0 = b[4];  t2 = a2 * b2;  t3 = a3 * b3;
      a1 = a[5];  b1 = b[5];
      s0 += t0; s1 += t1;  s2 += t2;  s3 += t3;
      a += 4; i += 4; b += 4;
      prefetch(a + 20); prefetch(b + 20);
      if (i < k) goto L17;
L16:  s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3
      a += 4; b += 4;
      a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1];
L18:  s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3
      a += 4; b += 4;
      dp = s0 + s1 + s2 + s3;
```

This lecture

Using the Coq proof assistant, formalize some representative program transformations and static analyses, and prove their correctness.

In passing, introduce the semantic tools needed for this effort.

Lecture material

`http://gallium.inria.fr/~xleroy/courses/DSSS-2017/`

- The Coq development (source archive + HTML view).
- These slides.
- Further reading.

Course outline

1. Compiling IMP to a simple virtual machine; first compiler proofs; notions of semantic preservation.
2. More on semantics: big-step, small-step, small-step with continuations. Finishing the proof of the IMP $\rightarrow$ VM compiler.
3. Verification of optimizing program transformations (dead code elimination, register allocation) based on a static analysis (liveness analysis).
4. Compiler verification "in the large": a tour of CompCert.

Home work: the recommended exercises, plus
optional exercises,
or a mini-project (with much guidance),
or a full project (with little guidance).

Part I

Compiling IMP to virtual machine code

Compiling IMP to virtual machine code

1. Reminder: the IMP language
2. The IMP virtual machine
3. The compiler
4. Verifying the compiler: first results

Reminder: the IMP language

(From Benjamin Pierce's "Logical Foundations" course.)

A prototypical imperative language with structured control flow.

Arithmetic expressions:

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2$$

Boolean expressions:

$$b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \texttt{not } b \mid b_1 \texttt{ and } b_2$$

Commands (statements):

$$\begin{array}{lll} c ::= & \texttt{SKIP} & \text{(do nothing)} \\ \mid & x ::= a & \text{(assignment)} \\ \mid & c_1 ; ; c_2 & \text{(sequence)} \\ \mid & \texttt{IFB } b \texttt{ THEN } c_1 \texttt{ ELSE } c_2 \texttt{ FI} & \text{(conditional)} \\ \mid & \texttt{WHILE } b \texttt{ DO } c \texttt{ END} & \text{(loop)} \end{array}$$

Reminder: IMP's semantics

As defined in file `Imp.v` of "Logical Foundations":

- Evaluation function for arithmetic expressions

$$\texttt{aeval}\ st\ a : \texttt{nat}$$

- Evaluation function for boolean expressions

$$\texttt{beval}\ st\ b : \texttt{bool}$$

- Evaluation predicate for commands (in big-step operational style)

$$c / st \Rightarrow st'$$

(`st` ranges over variable states: $\texttt{id} \to \texttt{nat}$.)

Execution models for a programming language

1. Interpretation:
   the program is represented by its abstract syntax tree. The interpreter traverses this tree during execution.

Execution models for a programming language

1. Interpretation:
   the program is represented by its abstract syntax tree. The interpreter traverses this tree during execution.
2. Compilation to native code:
   before execution, the program is translated to a sequence of machine instructions, These instructions are those of a real microprocessor and are executed in hardware.

Execution models for a programming language

1. Interpretation:
   the program is represented by its abstract syntax tree. The interpreter traverses this tree during execution.
2. Compilation to native code:
   before execution, the program is translated to a sequence of machine instructions, These instructions are those of a real microprocessor and are executed in hardware.
3. Compilation to virtual machine code:
   before execution, the program is translated to a sequence of instructions, These instructions are those of a virtual machine. They do not correspond to that of an existing hardware processor, but are chosen close to the basic operations of the source language. Then,
   1. either the virtual machine instructions are interpreted (efficiently)
   2. or they are further translated to machine code (JIT).

Compiling IMP to virtual machine code

The IMP virtual machine

Components of the machine:

- The code C: a list of instructions.
- The program counter pc: an integer, giving the position of the currently-executing instruction in C.
- The store st: a mapping from variable names to integer values.
- The stack σ: a list of integer values
  (used to store intermediate results temporarily).

The instruction set

$i ::= \texttt{Iconst}(n)$	push n on stack
$\mid \texttt{Ivar}(x)$	push value of x
$\mid \texttt{Isetvar}(x)$	pop value and assign it to x
$\mid \texttt{Iadd}$	pop two values, push their sum
$\mid \texttt{Isub}$	pop two values, push their difference
$\mid \texttt{Imul}$	pop two values, push their product
$\mid \texttt{Ibranch\_forward}(\delta)$	unconditional jump forward
$\mid \texttt{Ibranch\_backward}(\delta)$	unconditional jump backward
$\mid \texttt{Ibeq}(\delta)$	pop two values, jump if $=$
$\mid \texttt{Ibne}(\delta)$	pop two values, jump if $\neq$
$\mid \texttt{Ible}(\delta)$	pop two values, jump if $\leq$
$\mid \texttt{Ibgt}(\delta)$	pop two values, jump if $>$
$\mid \texttt{Ihalt}$	end of program

By default, each instruction increments *pc* by 1. Exception: branch instructions increment it by $1 + \delta$ (forward) or $1 - \delta$ (backward).

(δ is a branch offset relative to the next instruction.)

Example

			1		
stack	ϵ	12	12	13	ϵ
store	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 13$
p.c.	0	1	2	3	4
code	`Ivar(`x`);`	`Iconst(1);`	`Iadd;`	`Isetvar(`x`);`	`Ibranch_backward(5)`

Semantics of the machine

Given by a transition relation (small-step), representing the execution of one instruction.

```
Definition code := list instruction.
Definition stack := list nat.
Definition configuration := (nat * stack * state)%type.

Inductive transition (C: code):
                  configuration -> configuration -> Prop :=
  ...
```

(See file `Compil.v`.)

Executing machine programs

By iterating the transition relation:

- Initial states: $pc = 0$, initial store, empty stack.
- Final states: pc points to a `halt` instruction, empty stack.

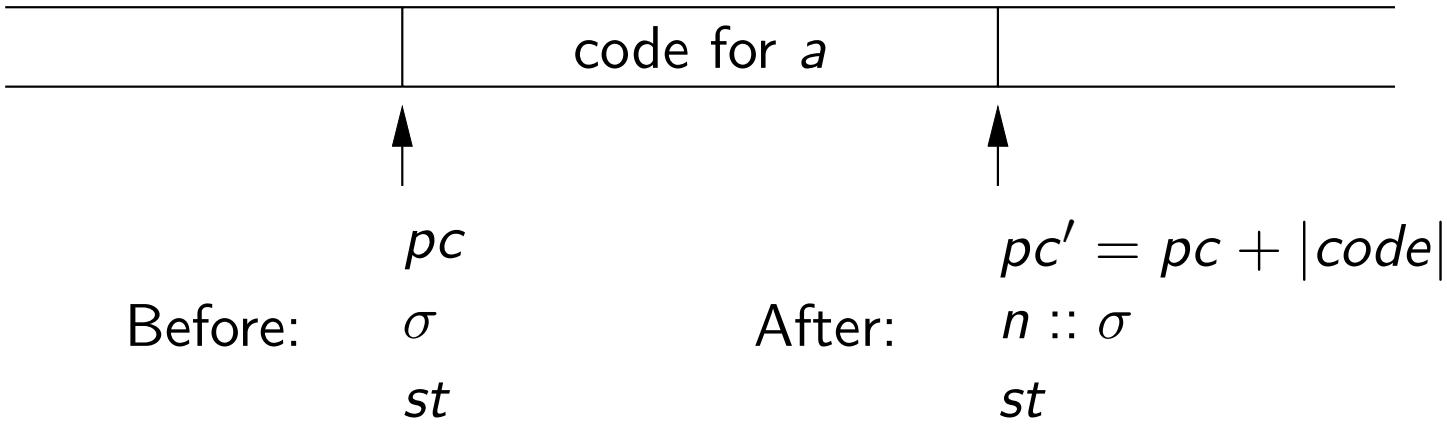
```
Definition mach_terminates (C: code) (s_init s_fin: state) :=
  exists pc,
  code_at C pc = Some Ihalt /\
  star (transition C) (0, nil, s_init) (pc, nil, s_fin).

Definition mach_diverges (C: code) (s_init: state) :=
  infseq (transition C) (0, nil, s_init).

Definition mach_goes_wrong (C: code) (s_init: state) :=
  (* otherwise *)
```

(`star` is reflexive transitive closure. See file `Sequences.v`.)

Compiling IMP to virtual machine code

Compilation of arithmetic expressions

General contract: if a evaluates to n in store st,

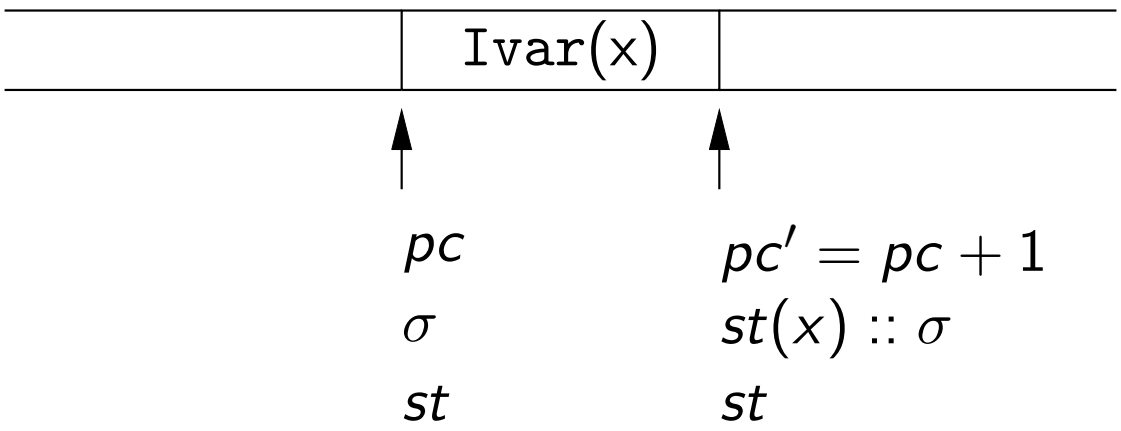


Compilation is just translation to "reverse Polish notation".

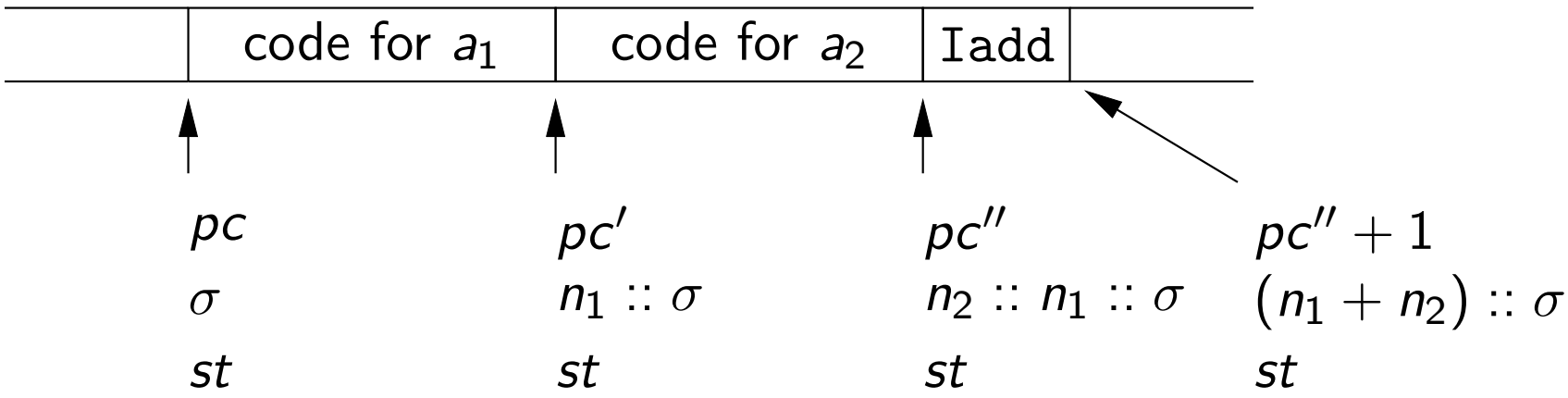
(See function `compile_aexpr` in `Compil.v`)

Compilation of arithmetic expressions

Base case: if $a = x$,



Recursive decomposition: if $a = a_1 + a_2$,

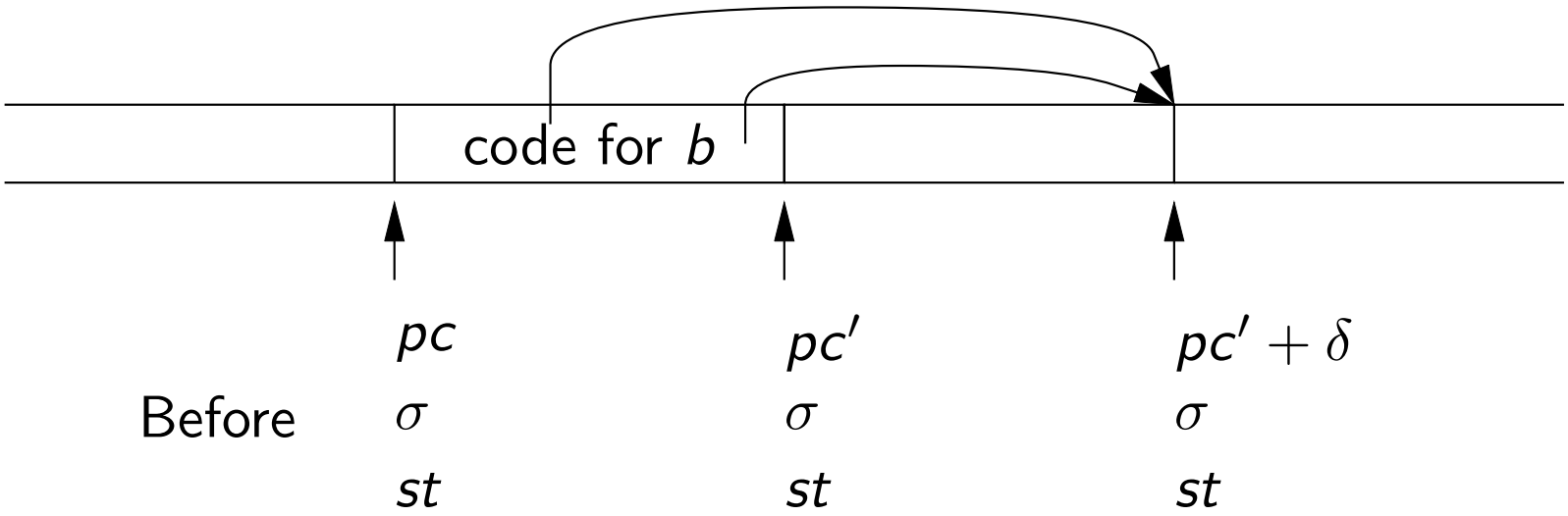


Compilation of boolean expressions

`compile_bexp` *b cond* δ:

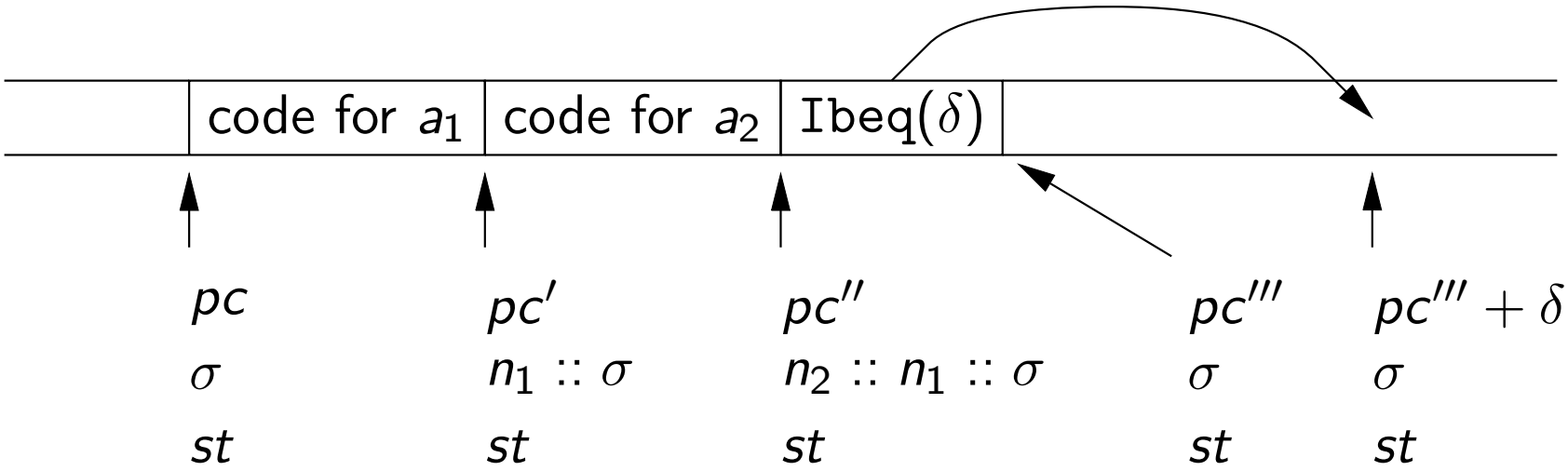
skip δ instructions forward if *b* evaluates to boolean *cond*

continue in sequence if *b* evaluates to boolean $\neg$*cond*



Compilation of boolean expressions

A base case: $b = (a_1 = a_2)$ and $cond = \texttt{true}$:



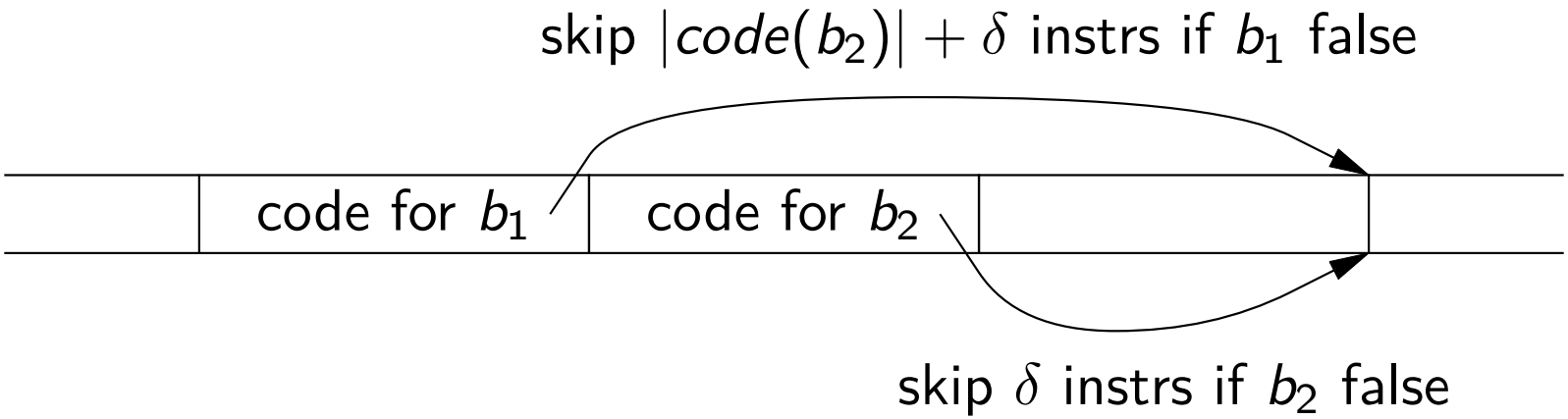
Short-circuiting "`and`" expressions

If b_1 evaluates to `false`, so does b_1 `and` b_2: no need to evaluate b_2!

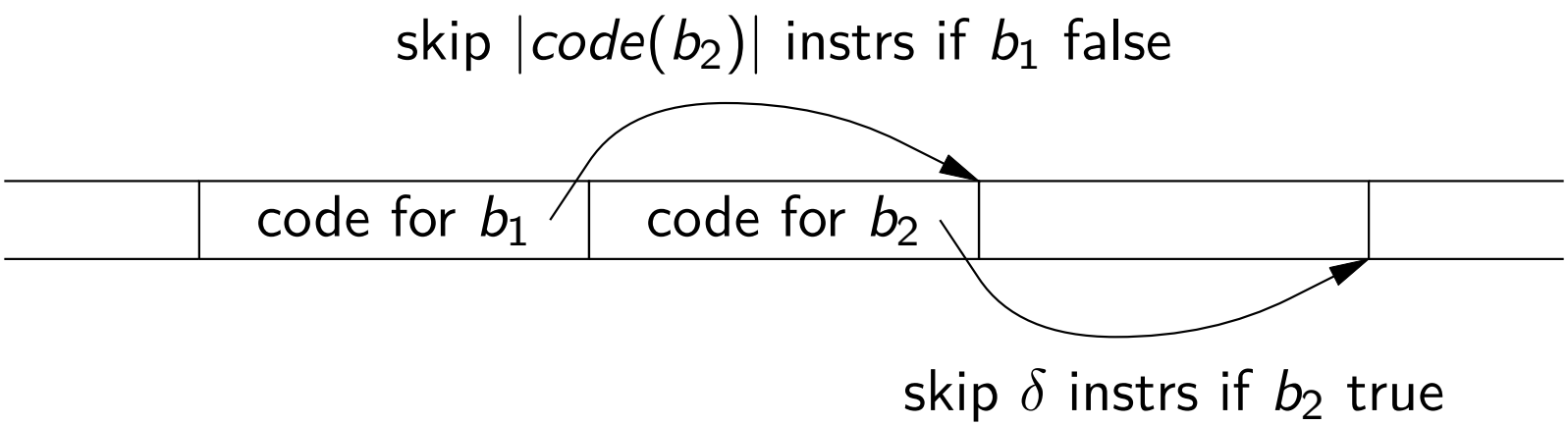
$\rightarrow$ In this case, the code generated for b_1 `and` b_2 should skip over the code for b_2 and branch directly to the correct destination.

Short-circuiting "and" expressions

If *cond* = `false` (branch if b_1 `and` b_2 is false):

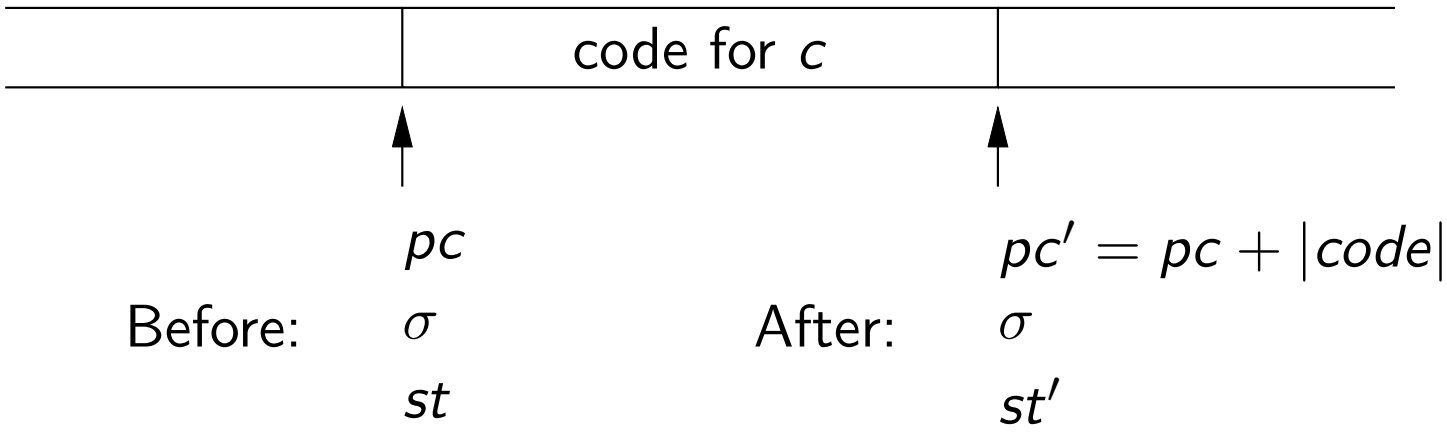


If *cond* = `true` (branch if b_1 `and` b_2 is true):



Compilation of commands

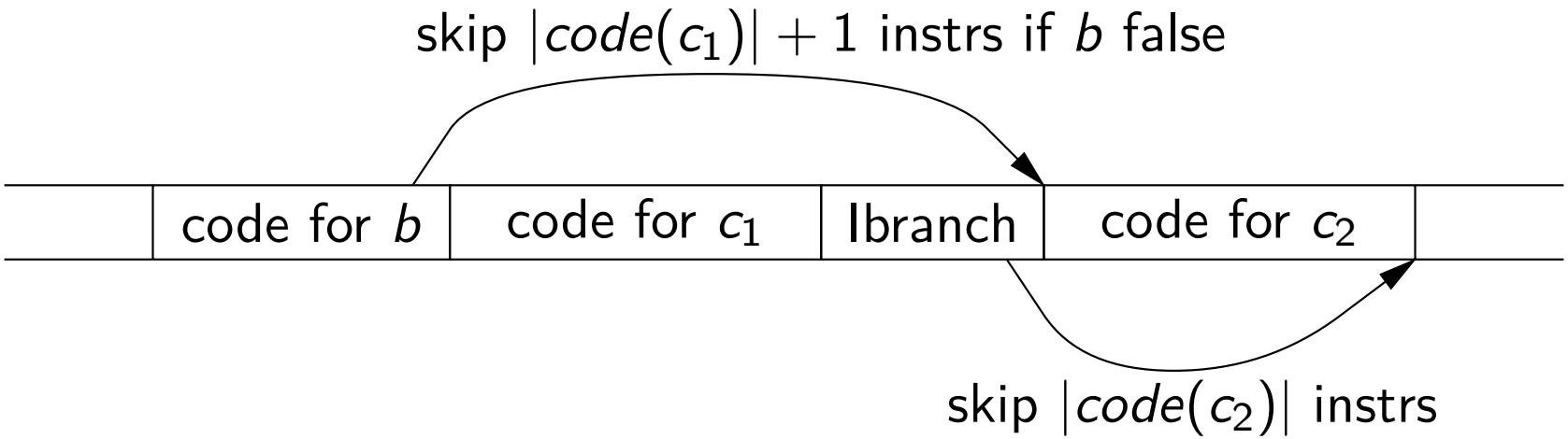
If the command c, started in initial state st, terminates in final state st',



(See function `compile_com` in `Compil.v`)

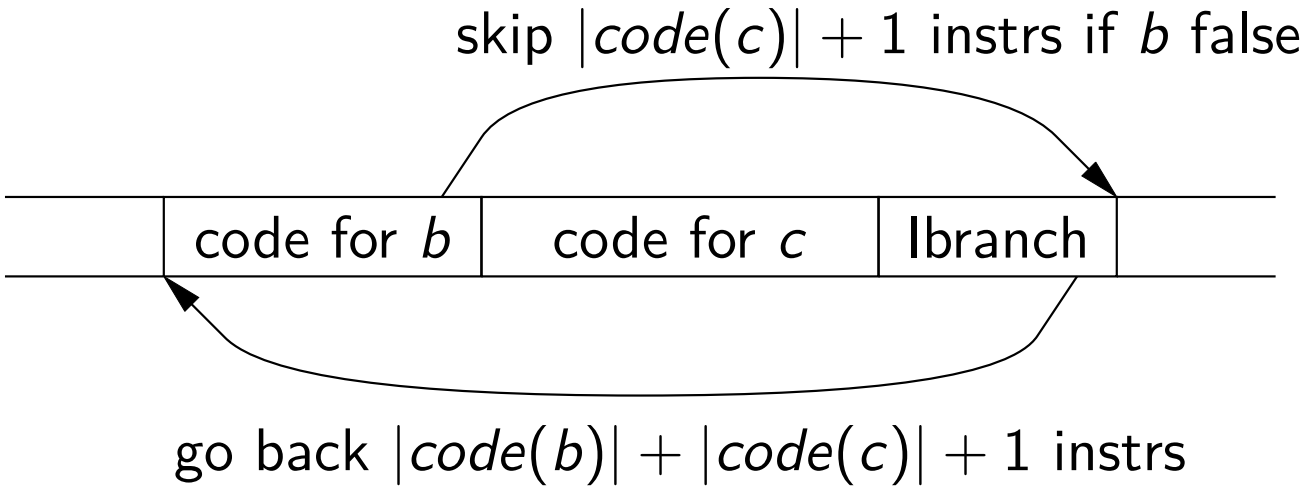
The mysterious offsets

Code for `IFB` b `THEN` c_1 `ELSE` c_2 `FI`:



The mysterious offsets

Code for `WHILE` b `DO` c `END`:



Compiling IMP to virtual machine code

Compiler verification

We now have two ways to run a program:

- Interpret it using e.g. the `ceval_step` function from `ImpCEvalFun.v`.
- Compile it, then run the generated virtual machine code.

Will we get the same results either way?

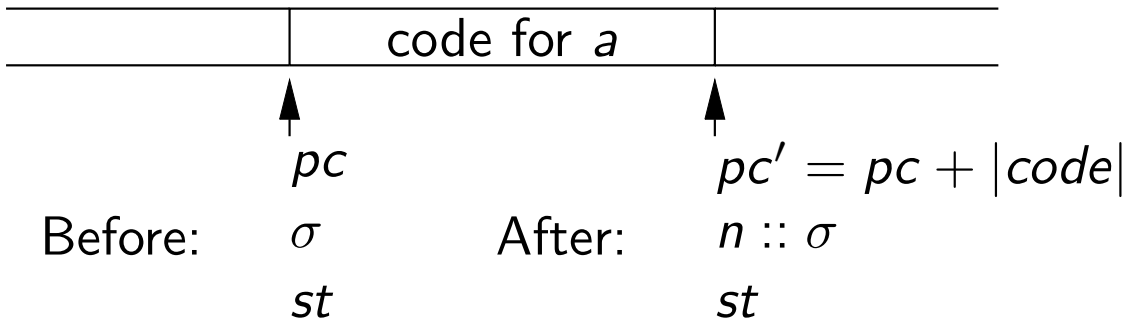
The compiler verification problem

Verify that a compiler is semantics-preserving:
the generated code behaves as prescribed by the semantics of the source program.

First verifications

Let's try to formalize and prove the intuitions we had when writing the compilation functions.

Intuition for arithmetic expressions: if a evaluates to n in store st,



A formal claim along these lines:

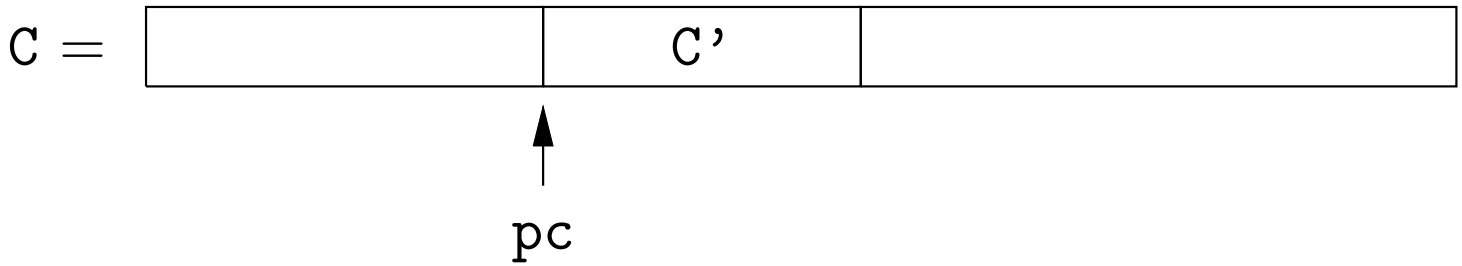
```
Lemma compile_aexp_correct:
  forall st a pc stk,
  star (transition (compile_aexp a))
       (0, stk, st)
       (length (compile_aexp a), aeval st a :: stk, st).
```

Verifying the compilation of expressions

For this statement to be provable by induction over the structure of the expression *a*, we need to generalize it so that

- the start PC is not necessarily 0;
- the code `compile_aexp` *a* appears as a fragment of a larger code *C*.

To this end, we define the predicate `codeseq_at C pc C'` capturing the following situation:



Verifying the compilation of expressions

```
Lemma compile_aexp_correct:
  forall C st a pc stk,
  codeseq_at C pc (compile_aexp a) ->
  star (transition C)
       (pc, stk, st)
       (pc + length (compile_aexp a), aeval st a :: stk, st).
```

Proof: a simple induction on the structure of a.

The base cases are trivial:

- $a = n$: a single `Iconst` transition.
- $a = x$: a single `Ivar`(x) transition.

An inductive case

Consider $a = a_1 + a_2$ and assume

$$\texttt{codeseq\_at}\ C\ pc\ (code(a_1) + +code(a_2) + +\texttt{Iadd} :: nil)$$

We have the following sequence of transitions:

$$(pc, \sigma, st)$$

$$\downarrow * \quad \text{ind. hyp. on } a_1$$

$$(pc + |code(a_1)|, \texttt{aeval}\ st\ a_1 :: \sigma, st)$$

$$\downarrow * \quad \text{ind. hyp. on } a_2$$

$$(pc + |code(a_1)| + |code(a_2)|, \texttt{aeval}\ st\ a_2 :: \texttt{aeval}\ st\ a_1 :: \sigma, st)$$

$$\downarrow \quad \texttt{Iadd transition}$$

$$(pc + |code(a_1)| + |code(a_2)| + 1, (\texttt{aeval}\ st\ a_1 + \texttt{aeval}\ st\ a_2) :: \sigma, st)$$

Historical note

As simple as this proof looks, it is of historical importance:

- First published proof of compiler correctness.
  (McCarthy and Painter, 1967).
- First mechanized proof of compiler correctness.
  (Milner and Weyrauch, 1972, using Stanford LCF).

John McCarthy
James Painter[1]

CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS[2]

1. **Introduction.** This paper contains a proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language.

The definition of correctness, the formalism used to express the description of source language, object language and compiler, and the methods of proof are all intended to serve as prototypes for the more complicated task of proving the correctness of usable compilers. The ultimate goal, as outlined in references [1], [2], [3] and [4] is to make it possible to use a computer to check proofs that compilers are correct.

Mathematical Aspects of Computer Science, 1967

3

Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch

Computer Science Department
Stanford University

Abstract

We discuss the task of machine-checking the proof of a simple compiling algorithm. The proof-checking program is LCF, an implementation of a logic for computable functions due to Dana Scott, in which the abstract syntax and extensional semantics of programming languages can be naturally expressed. The source language in our example is a simple ALGOL-like language with assignments, conditionals, whiles and compound statements. The target language is an assembly language for a machine with a pushdown store. Algebraic methods are used to give structure to the proof, which is presented only in outline. However, we present in full the expression-compiling part of the algorithm. More than half of the complete proof has been machine checked, and we anticipate no difficulty with the remainder. We discuss our experience in conducting the proof, which indicates that a large part of it may be automated to reduce the human contribution.

Machine Intelligence (7), 1972.

APPENDIX 2: command sequence for McCarthy-Painter lemma

```
GOAL ∀e sp,iswfse e;;MT(compe e,sp)≡svof(sp)∣((MSE(e,svof sp))&pdof(sp)),
     ∀e,iswfse e;;iswft(compe e)≡TT,
     ∀e,iswfse e;;(count(compe e)=Ø)≡TT;

TRY 1 INDUCT 56;
 TRY 1 SIMPL;
 LABEL INDHYP;
 TRY 2 ABSTR;
  TRY 1 CASES wfs_efun(f,e);
  LABEL TT;
   TRY 1 CASES type e=_N;
    TRY 1 SIMPL BY ,FMT1,,FMSE,,FCOMPE,,FISWFT1,,FCOUNT;
    TRY 2;SS-,TT;SIMPL,TT;QED;
    TRY 3 CASES typ_ e=_E;
     TRY 1 SUBST ,FCOMPE;
     SS-,TT;SIMPL,TT;USE BOTH3 -;SS+,TT;
     INCL-,1;SS+-;INCL--,2;SS+-;INCL---,3;SS+-;
      TRY 1 CONJ;
       TRY 1 SIMPL;
        TRY 1 USE COUNT1;
         TRY 1;
         APPL ,INDHYP+2,arg1of e;
         LABEL CARG1;
         SIMPL-;QED;
         TRY 2 USE COUNT1;
          TRY 1;
```

(Even the proof scripts look familiar!)

Verifying the compilation of expressions

Similar approach for boolean expressions:

```
Lemma compile_bexp_correct:
  forall C st b cond ofs pc stk,
  codeseq_at C pc (compile_bexp b cond ofs) ->
  star (transition C)
       (pc, stk, st)
       (pc + length (compile_bexp b cond ofs)
           + if eqb (beval st b) cond then ofs else 0,
        stk, st).
```

Proof: induction on the structure of `b`, plus copious case analysis.

Verifying the compilation of commands

```
Lemma compile_com_correct_terminating:
  forall C st c st',
  c / st ==> st' ->
  forall stk pc,
  codeseq_at C pc (compile_com c) ->
  star (transition C)
       (pc, stk, st)
       (pc + length (compile_com c), stk, st').
```

An induction on the structure of `c` fails because of the `WHILE` case. An induction on the derivation of `c / st ==> st'` works perfectly.

Summary so far

Piecing the lemmas together, and defining

```
compile_program c = compile_command c ++ Ihalt :: nil
```

we obtain a rather nice theorem:

```
Theorem compile_program_correct_terminating:
  forall c st st',
  c / st ==> st' ->
  mach_terminates (compile_program c) st st'.
```

But is this enough to conclude that our compiler is correct?

What could have we missed?

```
Theorem compile_program_correct_terminating:
  forall c st st',
  c / st ==> st' ->
  mach_terminates (compile_program c) st st'.
```

What if the generated VM code could terminate on a state other than `st'`? or loop? or go wrong?

What if the program `c` started in `st` diverges instead of terminating? What does the generated code do in this case?

Needed: more precise notions of semantic preservation + richer semantics (esp. for non-termination).

Part II

Notions of semantic preservation

Comparing the behaviors of two programs

Consider two programs P_1 and P_2, possibly in different languages.

(For example, P_1 is an IMP command and P_2 is virtual machine code generated by compiling P_1.)

The semantics of the two languages associate to P_1, P_2
sets $\mathcal{B}(P_1), \mathcal{B}(P_2)$ of observable behaviors.

$\mathrm{card}(\mathcal{B}(P)) = 1$ if P is deterministic, and $\mathrm{card}(\mathcal{B}(P)) > 1$ if it is not.

Observable behaviors

For an IMP-like language:

$$\textit{observable behavior} ::= \texttt{terminates}(st) \mid \texttt{diverges} \mid \texttt{goeswrong}$$

(Alternative: in the `terminates` case, observe not the full final state st but only the values of specific variables.)

For a functional language like lambda-calculus with constants:

$$\textit{observable behavior} ::= \texttt{terminates}(v) \mid \texttt{diverges} \mid \texttt{goeswrong}$$

where v is the value of the program.

Observable behaviors

For an imperative language with I/O: add a trace of input-output operations performed during execution.

```
x := 1;  x := 2;
```
(trace: ϵ)

$\approx$

```
x := 2;
```
(trace: ϵ)

```
print(1); print(2);
```
(trace: out(1).out(2))

$\not\approx$

```
print(2);
```
(trace: out(2))

Bisimulation (observational equivalence)

$$\mathcal{B}(P_1) = \mathcal{B}(P_2)$$

The source and transformed programs are completely undistinguishable.

A very strong notion of semantic preservation.

Often too strong in practice ...

Reducing non-determinism during compilation

Languages such as C leave evaluation order partially unspecified.

```
int x = 0;
int f(void) { x = x + 1; return x; }
int g(void) { x = x - 1; return x; }
```

The expression `f() + g()` can evaluate either

- to 1 if `f()` is evaluated first (returning 1), then `g()` (returning 0);
- to -1 if `g()` is evaluated first (returning -1), then `f()` (returning 0).

Every C compiler chooses one evaluation order at compile-time.

The compiled code therefore has fewer behaviors than the source program (1 instead of 2).

Reducing non-determinism during optimization

In a concurrent setting, classic optimizations often reduce non-determinism:

Original program:

```
a := x + 1; b := x + 1;      run in parallel with    x := 1;
```

Program after common subexpression elimination:

```
a := x + 1; b := a;          run in parallel with    x := 1;
```

Assuming $\mathtt{x} = 0$ initially, the final states for the original program are

$$(\mathtt{a}, \mathtt{b}) \in \{(1,1);\ (1,2);\ (2,2)\}$$

Those for the optimized program are

$$(\mathtt{a}, \mathtt{b}) \in \{(1,1);\ (2,2)\}$$

Backward simulation (refinement)

$$\mathcal{B}(P_1) \supseteq \mathcal{B}(P_2)$$

Every possible possible behavior of P_2 is justified as being one of the legal behaviors of P_1. However, P_2 can have fewer behaviors (e.g. because some behaviors were eliminated during compilation).

Let *Spec* (a set of behaviors) be the functional specification of a program. A program P satisfies *Spec* iff $\mathcal{B}(P) \subseteq Spec$.

Lemma

If "backward simulation" holds and P_1 satisfies Spec, then P_2 satisfies Spec.

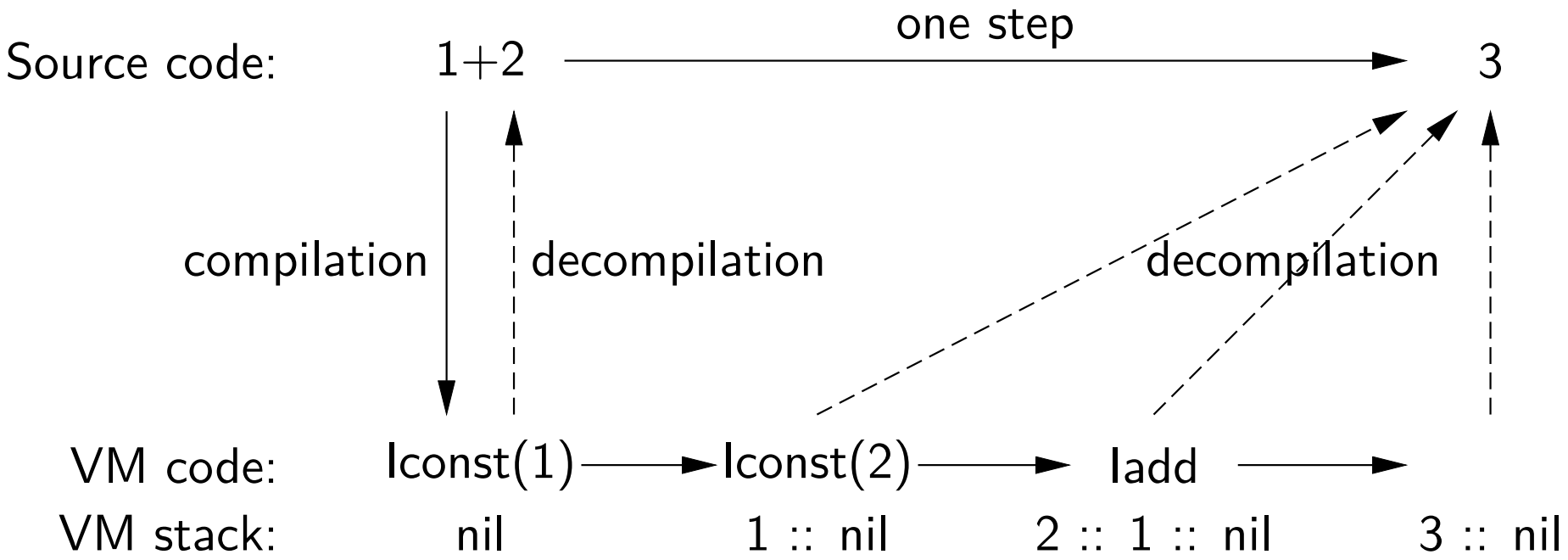
The pains of backward simulations

Backward simulation looks like "the" semantic preservation property we expect from a correct compiler.

It is however rather difficult to prove:

- We need to consider all steps that the compiled code can take, and trace them back to steps the source program can take.
- This is problematic if one source-level step is broken into several machine-level steps.
  (E.g. `x ::= a` is one step in IMP, but several instructions in the VM.)

General shape of a backward simulation proof



Intermediate VM code sequences like `Iconst(2); Iadd` or just `Iadd` do not correspond to the compilation of any source expression.

One solution: invent a decompilation function that is left-inverse of compilation. (Hard in general!)

Forward simulations

Forward simulation property:

$$\mathcal{B}(P_1) \subseteq \mathcal{B}(P_2)$$

Significantly easier to prove than backward simulations. (For example, our first correctness proof for IMP compilation is of this form: termination of IMP program P_1 implies termination of compiled VM code P_2.)

But is forward simulation a sufficient property? It shows that the compiled code P_2 has all the good behaviors of P_1, but P_2 could have other behaviors that are bad ...

Determinism to the rescue!

Lemma

If P_2 is deterministic (i.e. $\mathcal{B}(P_2)$ is a singleton), then "forward simulation" implies "backward simulation" and therefore "bisimulation".

Trivial result: follows from $\emptyset \subset X \subseteq \{y\} \Longrightarrow X = \{y\}$.

Should "going wrong" behaviors be preserved?

Compilers routinely "optimize away" going-wrong behaviors. For example:

`x := 1 / y; x := 42`	optimized to	`x := 42`
(goes wrong if $y = 0$)		(always terminates normally)

Justifications:

- We know that the program being compiled does not go wrong
  - because it was type-checked with a sound type system
  - or because it was formally verified.
- Or "it is the programmer's responsibility to avoid going-wrong behaviors, so the compiler can optimize under the assumption that there are none".

Simulations for safe programs

According to the ISO C standard, if one execution of the source program can go wrong ("invoke undefined behavior"), the code generated by the compiler can have any behavior whatsoever.

This corresponds to the weaker simulation properties below.

Safe backward simulation: $\texttt{goeswrong} \notin \mathcal{B}(P_1) \implies \mathcal{B}(P_1) \supseteq \mathcal{B}(P_2)$

Safe forward simulation: $\texttt{goeswrong} \notin \mathcal{B}(P_1) \implies \mathcal{B}(P_1) \subseteq \mathcal{B}(P_2)$

Should "going wrong" behaviors be preserved?

To minimize surprises and facilitate debugging, some C compilers (e.g. CompCert) preserve the behaviors of the source program up to the point where it goes wrong. Consider:

```
(1)     printf("dividing by zero"); return 1/0;

(2)     printf("dividing by zero"); return 0

(3)     tmp = 1/0; printf("dividing by zero"); return tmp;
```

According to ISO C, (1) can be compiled like (2) and like (3) too. Compiling like (2) preserves the output up to the crash; (3) does not.

Simulations up to improvement

Define a pre-order "improves on" between behaviors. E.g.

- ISO C interpretation: `goeswrong`(t) is improved by any behavior.
- CompCert interpretation: `goeswrong`(t) is improved by any behavior whose trace of observables starts with t.

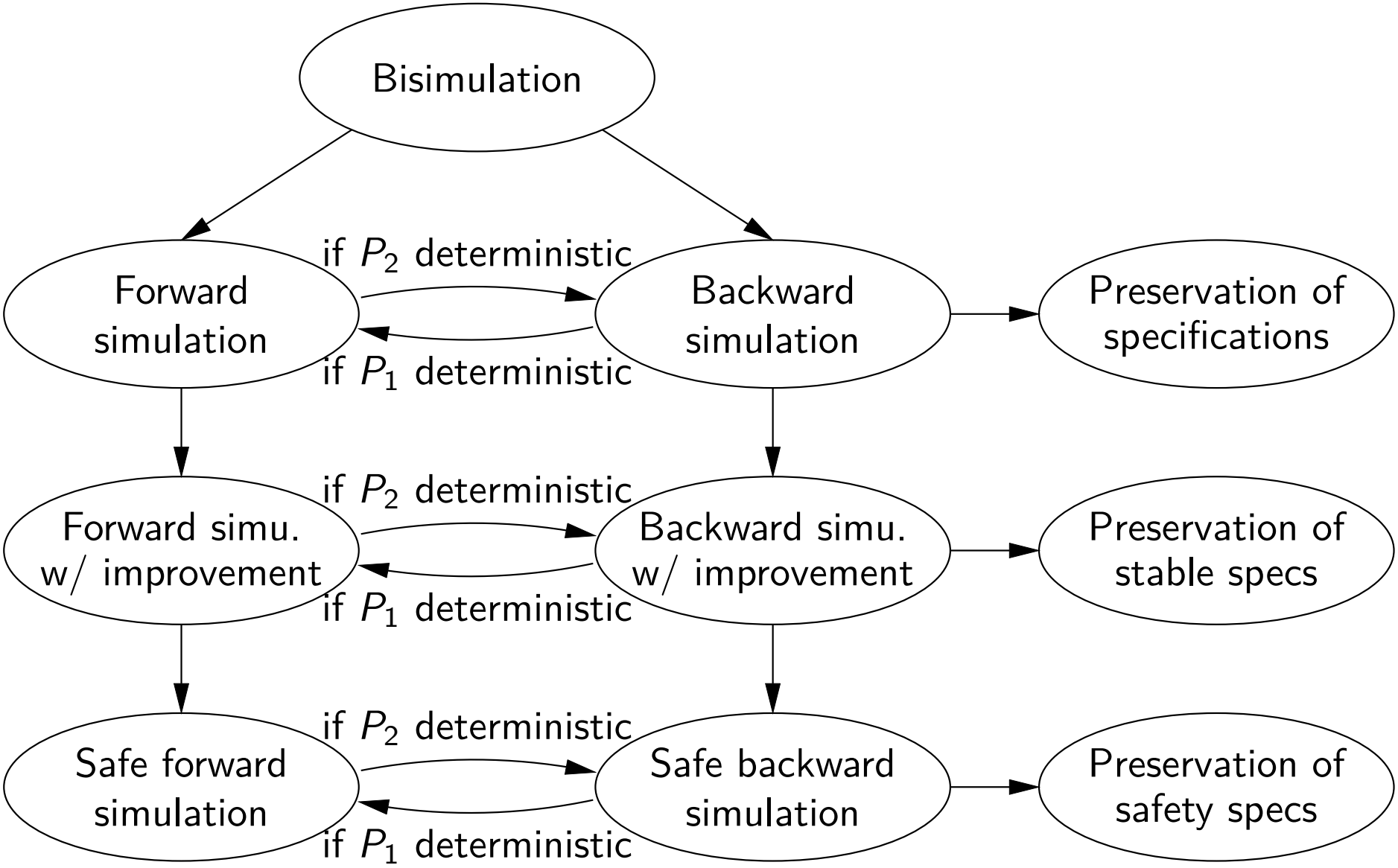
Backward simulation with improvement:

$\forall b_2 \in \mathcal{B}(P_2), \exists b_1 \in \mathcal{B}(P_1), b_2$ improves on b_1

Forward simulation with improvement:

$\forall b_1 \in \mathcal{B}(P_1), \exists b_2 \in \mathcal{B}(P_2), b_2$ improves on b_1

Relating preservation properties



Back to the IMP → VM compiler

We have already proved half of a forward simulation result:

```
Theorem compile_program_correct_terminating:
  forall c st st',
  c / st ==> st' ->
  mach_terminates (compile_program c) st st'.
```

It remains to show the other half:

If command c diverges when started in state st,
then the virtual machine, executing code `compile_program c`
from initial state st, makes infinitely many transitions.

(Note that IMP has no going-wrong behaviors.)

What we need: a formal characterization of divergence for IMP commands.

Part III

More on mechanized semantics

More on mechanized semantics

5. Reminder: big-step semantics for terminating programs

6. Small-step semantics

7. Small-step semantics with continuations

Big-step semantics

A predicate $c/s \Rightarrow s'$, meaning "started in state s, command c terminates and the final state is s'".

$$\texttt{SKIP}/s \Rightarrow s \qquad\qquad x := a/s \Rightarrow s[x \leftarrow \texttt{aeval}\ s\ a]$$

$$\frac{c_1/s \Rightarrow s_1 \qquad c_2/s_1 \Rightarrow s_2}{c_1; c_2/s \Rightarrow s_2}$$

$$\frac{\begin{array}{c} c_1/s \Rightarrow s' \text{ if } \texttt{beval}\ s\ b = \texttt{true} \\ c_2/s \Rightarrow s' \text{ if } \texttt{beval}\ s\ b = \texttt{false} \end{array}}{\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI}/s \Rightarrow s'}$$

$$\frac{\texttt{beval}\ s\ b = \texttt{false}}{\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}/s \Rightarrow s}$$

$$\frac{\texttt{beval}\ s\ b = \texttt{true} \qquad c/s \Rightarrow s_1 \qquad \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}/s_1 \Rightarrow s_2}{\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}/s \Rightarrow s_2}$$

Pros and cons of big-step semantics

Pros:

- Follows naturally the structure of programs.
  (Gilles Kahn called it "natural semantics").
- Close connection with interpreters.
- Powerful induction principle (on the structure of derivations).
- Easy to extend with various structured constructs
  (functions and procedures, other forms of loops)

Cons:

- Fails to characterize diverging executions.
  (More precisely: no distinction between divergence and going wrong.)
- Concurrency, unstructured control (`goto`) nearly impossible to handle.

Big-step semantics and divergence

For IMP, a negative characterization of divergence:

$$c/s \text{ diverges} \iff \neg(\exists s',\ c/s \Rightarrow s')$$

In general (e.g. STLC), executions can also go wrong (in addition to terminating or diverging). Big-step semantics fails to distinguish between divergence and going wrong:

$$c/s \text{ diverges} \vee c/s \text{ goes wrong} \iff \neg(\exists s',\ c/s \Rightarrow s')$$

Highly desirable: a positive characterization of divergence, distinguishing it from "going wrong".

More on mechanized semantics

Small-step semantics

Also called "structured operational semantics".

Like β-reduction in the λ-calculus: view computations as sequences of reductions

$$M \xrightarrow{\beta} M_1 \xrightarrow{\beta} M_2 \xrightarrow{\beta} \ldots$$

Each reduction $M \to M'$ represents an elementary computation.
M' represents the residual computations that remain to be done later.

Small-step semantics for IMP

Reduction relation: $c/s \to c'/s'$.

$$x := a/s \to \texttt{SKIP}/s[x \leftarrow \texttt{aeval}\ s\ a]$$

$$\frac{c_1/s \to c_1'/s'}{(c_1; c_2)/s \to (c_1'; c_2)/s'} \qquad (\texttt{SKIP}; c)/s \to c/s$$

$$\frac{\texttt{beval}\ s\ b = \texttt{true}}{\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI}/s \to c_1/s}$$

$$\frac{\texttt{beval}\ s\ b = \texttt{false}}{\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI}/s \to c_2/s}$$

$$\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}/s \to \texttt{IFB}\ b\ \texttt{THEN}\ c;\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}\ \texttt{ELSE}\ \texttt{SKIP}/s$$

Sequences of reductions

The behavior of a command c in an initial state s is obtained by forming sequences of reductions starting at c/s:

- Termination with final state s': finite sequence of reductions to `SKIP`.

$$c/s \to \cdots \to \texttt{SKIP}/s'$$

- Divergence: infinite sequence of reductions.

$$c/s \to c_1/s_1 \to \cdots \to c_n/s_n \to \cdots$$

- Going wrong: finite sequence of reductions to an irreducible command that is not `SKIP`.

$$(c, s) \to \cdots \to (c', s') \not\to \text{ with } c \neq \texttt{SKIP}$$

Equivalence small-step / big-step

A classic result:

$$c/s \Rightarrow s' \quad \Longleftrightarrow \quad c/s \stackrel{*}{\rightarrow} \texttt{SKIP}/s'$$

(See Coq file `Semantics.v`.)

Pros and cons of small-step semantics

Pros:

- Clean, unquestionable characterization of program behaviors (termination, divergence, going wrong).
- Extends even to unstructured constructs (`goto`, concurrency).
- De facto standard in the type systems community and in the concurrency community.

Cons:

- Does not follow the structure of programs; lack of a powerful induction principle.
- Syntax often needs to be extended with intermediate forms arising only during reductions.
- "Spontaneous generation" of terms.

Reasoning with or without structure

Reasoning, big-step style: by pre- and post-conditions

- Single program: if $c/s \Rightarrow s'$ and $P\ s$, then $Q\ s'$.
- Program transformation: if $c/s \Rightarrow s'$ and $T\ c\ c_1$ and $P\ s\ s_1$, there exists s_1' s.t. $c_1/s_1 \Rightarrow s_1'$ and $Q\ s'\ s_1'$.

Proofs: by induction on a derivation of $c/s \Rightarrow s'$.

Reasoning, small-step style: by invariants and simulations.

- Single program: if $c/s \rightarrow c'/s'$ and $I(c,s)$ then $I(c',s')$.
- Program transformation: a relation $I\ (c,s)\ (c_1,s_1)$ is a (bi)-simulation for the transitions of the two programs.

Proofs: by case analysis on each transition.

Intermediate forms extending the syntax

Many programming constructs require unnatural extensions of the syntax of terms so that we can give reduction rules for these constructs.

Example: the `break` statement (as in C, Java, ...).

Commands: $c ::= \ldots \mid \texttt{BREAK} \mid \texttt{INLOOP}\ c_1\ c_2$

Intuition: $\texttt{INLOOP}\ c_1\ c_2 \approx c_1; c_2$ but with special treatment of `BREAK` arising out of c_1.

$$\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}/s \rightarrow \texttt{IFB}\ b\ \texttt{THEN INLOOP}\ c\ (\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END})\ \texttt{ELSE SKIP FI}/s$$

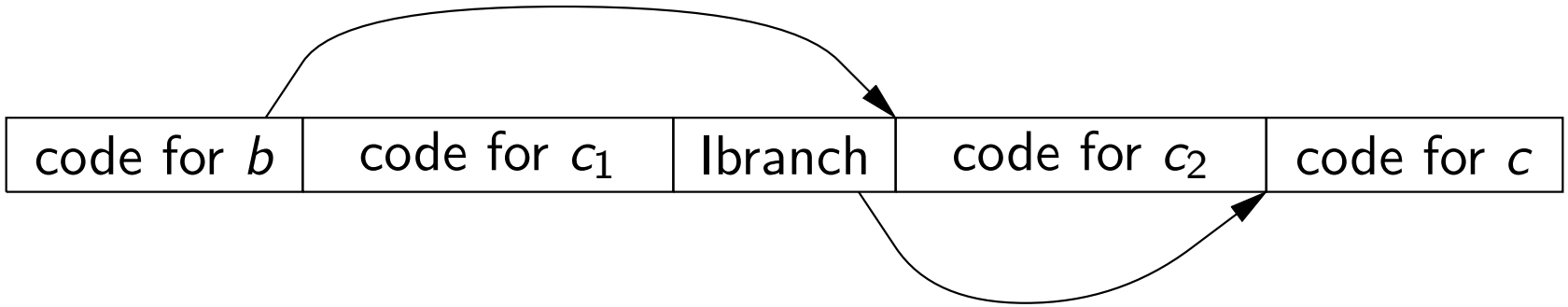
$$(\texttt{BREAK}; c)/s \rightarrow \texttt{BREAK}/s \qquad (\texttt{INLOOP SKIP}\ c)/s \rightarrow c/s$$

$$(\texttt{INLOOP BREAK}\ c)/s \rightarrow \texttt{SKIP}/s \qquad \frac{c_1/s \rightarrow c_1'/s'}{\texttt{INLOOP}\ c_1\ c_2/s \rightarrow \texttt{INLOOP}\ c_1'\ c_2/s'}$$

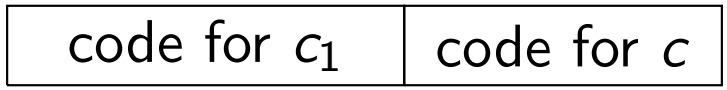
Spontaneous generation of terms

$$(\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI};c)/s \rightarrow (c_1;c)/s$$

Compiled code for initial command:



This code does not contain the compiled code for $c_1;c$, which is:



(Similar problem for
$\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}/s \rightarrow \texttt{IFB}\ b\ \texttt{THEN}\ \texttt{c};\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}\ \texttt{ELSE}\ \texttt{SKIP}/s$.)

More on mechanized semantics

Small-step semantics with continuations

A variant of standard small-step semantics that addresses issues #2 (no extensions of the syntax of commands) and #3 (no spontaneous generation of commands).

Idea: instead of rewriting whole commands:

$$c/s \to c'/s'$$

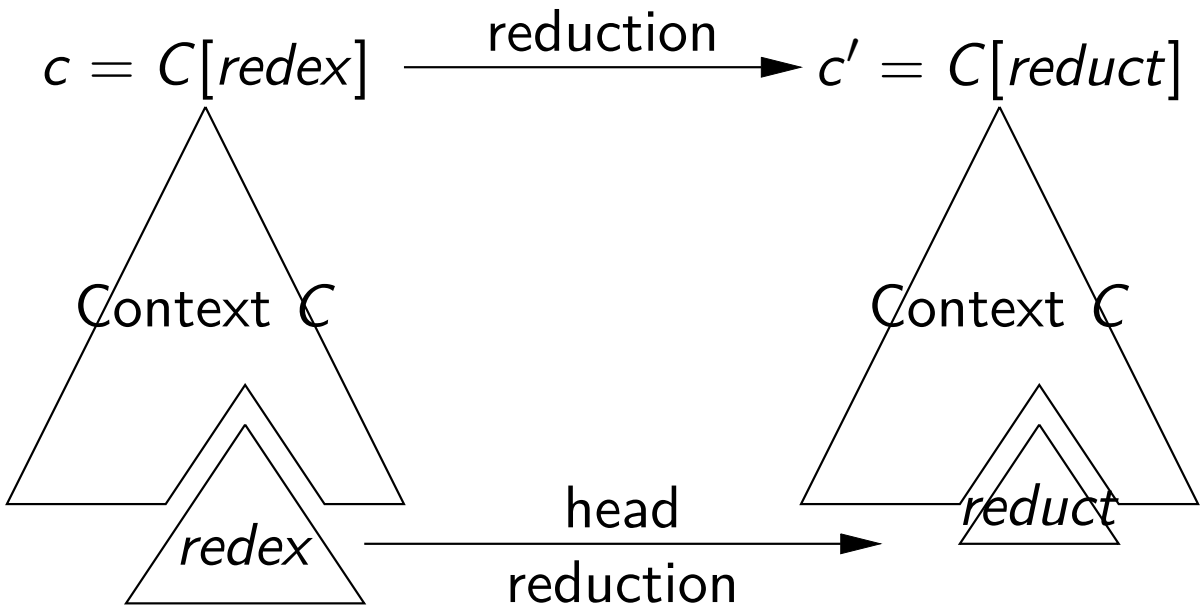
rewrite pairs of (subcommand under focus, remainder of command):

$$c/k/s \to c'/k'/s'$$

(Vaguely related to focusing in proof theory.)

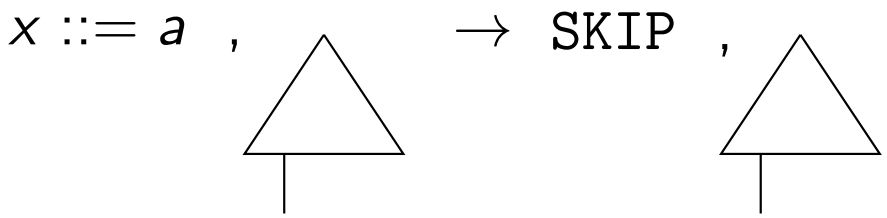
Standard small-step semantics

Rewrite whole commands, even though only a sub-command (the redex) changes.

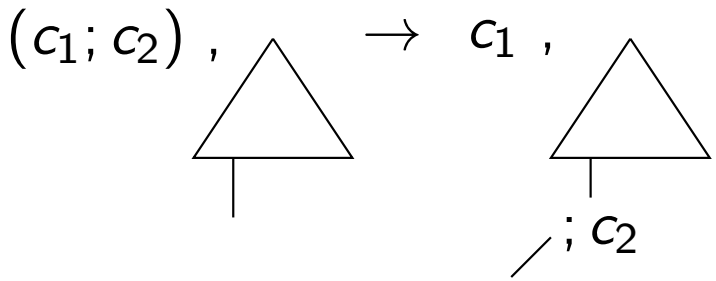


Focusing the small-step semantics

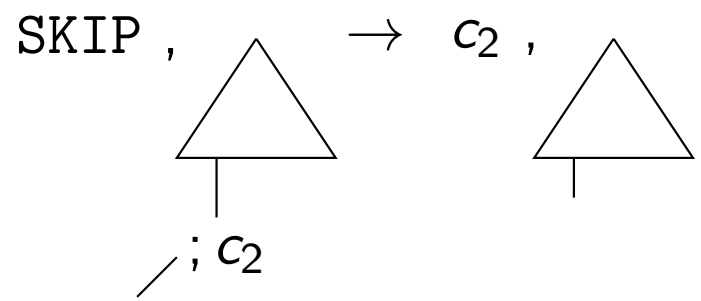
Rewrite pairs (subcommand, context in which it occurs).



The sub-command is not always the redex: add explicit focusing and resumption rules to move nodes between subcommand and context.



Focusing on the left of a sequence

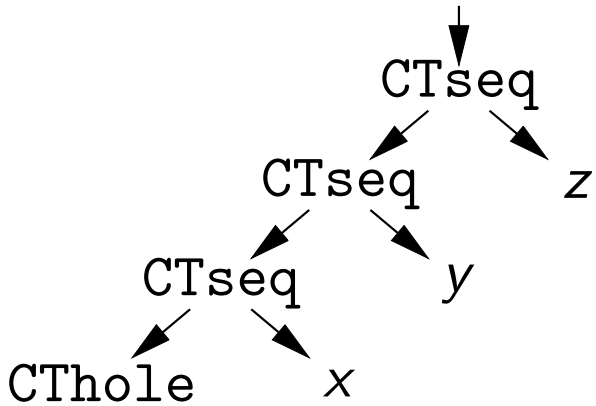


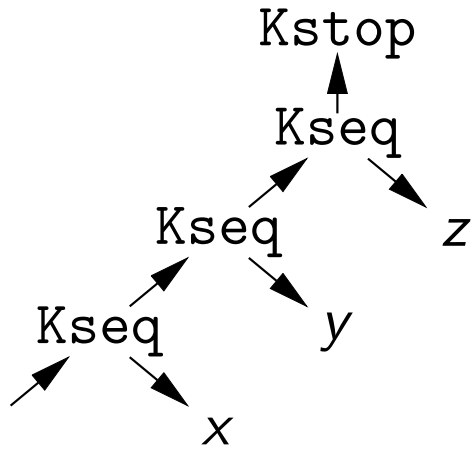
Resuming a sequence

Representing contexts "upside-down"

```
Inductive ctx :=
  | CThole: ctx
  | CTseq: com -> ctx -> ctx.
```

```
Inductive cont :=
| Kstop: cont
| Kseq: com -> cont -> cont.
```





`CTseq (CTseq (CTseq CThole x) y) z`

`Kseq x (Kseq y (Kseq z Kstop))`

Upside-down context $\approx$ continuation.
("Eventually, do x, then do y, then do z, then stop.")

Transition rules

$$
\begin{array}{rcll}
x := a/k/s & \to & \texttt{SKIP}/k/s[x \leftarrow \texttt{aeval}\ s\ a] & \\
(c_1; c_2)/k/s & \to & c_1/\texttt{Kseq}\ c_2\ k/s & \\
\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2/k/s & \to & c_1/k/s & \text{if } \texttt{beval}\ \ s\ b = \texttt{true} \\
\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2/k/s & \to & c_2/k/s & \text{if } \texttt{beval}\ \ s\ b = \texttt{false} \\
\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}/k/s & \to & c/\texttt{Kseq}\ (\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END})\ k/s & \\
 & & \text{if } \texttt{beval}\ \ s\ b = \texttt{true} & \\
\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}/k/s & \to & \texttt{SKIP}/c/k & \text{if } \texttt{beval}\ \ s\ b = \texttt{false} \\
\texttt{SKIP}/\texttt{Kseq}\ c\ k/s & \to & c/k/s &
\end{array}
$$

Note: no spontaneous generation of fresh commands.

Enriching the language

Let's add a `break` statement. We need a new form of continuations for loops, but no ad-hoc extension to the syntax of commands.

Commands: $c ::= \ldots \mid \texttt{BREAK}$

Continuations: $k ::= \texttt{Kstop} \mid \texttt{Kseq}\ c\ k \mid \texttt{Kwhile}\ b\ c\ k$

New or modified rules:

$$\begin{aligned}
\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}/k/s &\rightarrow c/\texttt{Kwhile}\ b\ c\ k/s \\
&\quad \text{if } \texttt{beval}\ \ s\ b = \texttt{true} \\
\texttt{SKIP}/\texttt{Kwhile}\ b\ c\ k/s &\rightarrow \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}/k/s \\
\texttt{BREAK}/\texttt{Kseq}\ c\ k/s &\rightarrow \texttt{BREAK}/k/s \\
\texttt{BREAK}/\texttt{Kwhile}\ b\ c\ k/s &\rightarrow \texttt{SKIP}/k/s
\end{aligned}$$

(Exercise: what about `continue`?)

Equivalence with the other semantics

$$c/\mathtt{Kstop}/s \stackrel{*}{\to} \mathtt{SKIP}/\mathtt{Kstop}/s' \quad \text{iff} \quad c/s \Rightarrow s' \quad \text{iff} \quad c/s \stackrel{*}{\to} \mathtt{SKIP}/s'$$

$$c/k/s \to \infty \quad \text{iff} \quad c/s \to \infty$$

(See Coq file `Semantics.v`)

Part IV

Compiling IMP to virtual machine code, continued

Finishing the proof of forward simulation

One half already proved: the terminating case.

```
Theorem compile_program_correct_terminating:
  forall c st st',
  c / st \\ st' ->
  mach_terminates (compile_program c) st st'.
```

One half to go: the diverging case.
(If c/st diverges, then `mach_diverges (compile_program c) st`.)

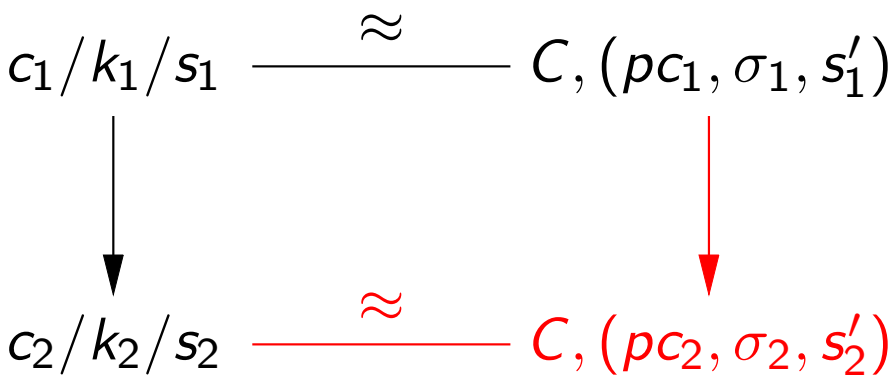
Forward simulations, small-step style

Show that every transition in the execution of the source program

- is simulated by some transitions in the compiled program
- while preserving a relation between the configurations of the two programs.

Lock-step simulation

Every transition of the source is simulated by exactly one transition in the compiled code.



(Black = hypotheses; red = conclusions.)

Lock-step simulation

Further show that initial configurations are related:

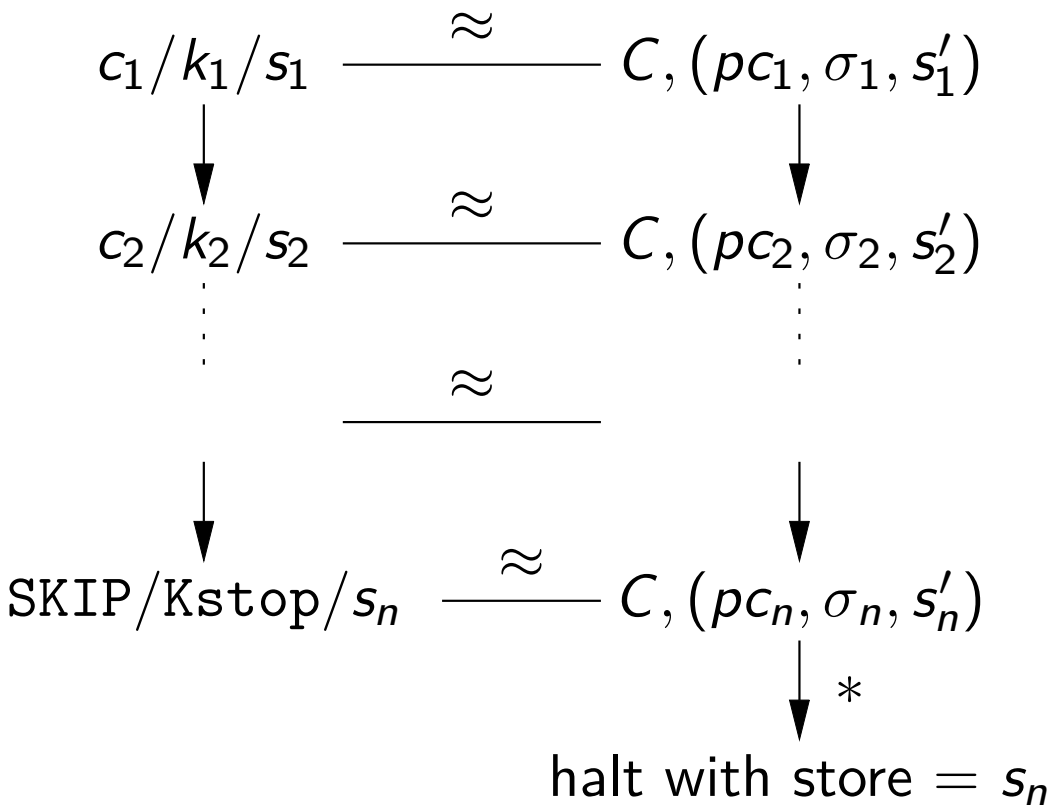
$$c/\texttt{Kstop}/s \approx (C, (0, \mathit{nil}, s)) \text{ with } C = \texttt{compile\_program}(c)$$

Further show that final configurations are quasi-related:

$$\begin{aligned} &\texttt{SKIP}/\texttt{Kstop}/s \approx (C, \mathit{mst}) \\ \Longrightarrow\quad &(C, \mathit{mst}) \stackrel{*}{\rightarrow} (C, (\mathit{pc}, \mathit{nil}, s)) \wedge C(\mathit{pc}) = \texttt{Ihalt} \end{aligned}$$

Lock-step simulation

Forward simulation follows easily:

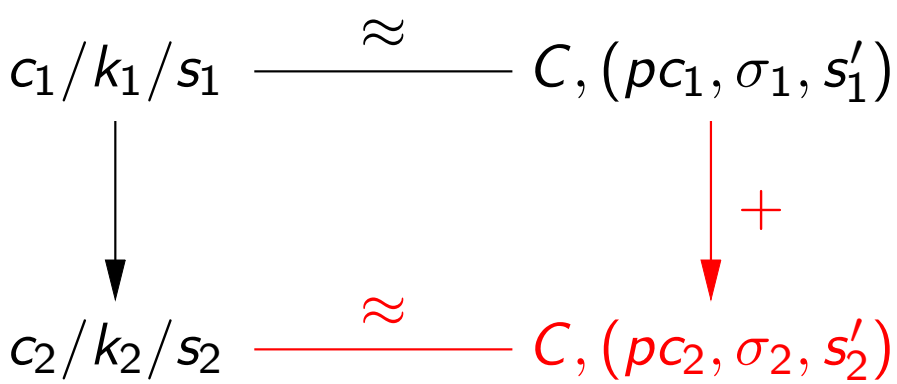


(Likewise if $c_1/k_1/s_1$ reduces infinitely.)

"Plus" simulation diagrams

In some cases, each transition in the source program is simulated by one or several transitions in the compiled code.

(Example: compiled code for $x ::= a$ consists of several instructions.)

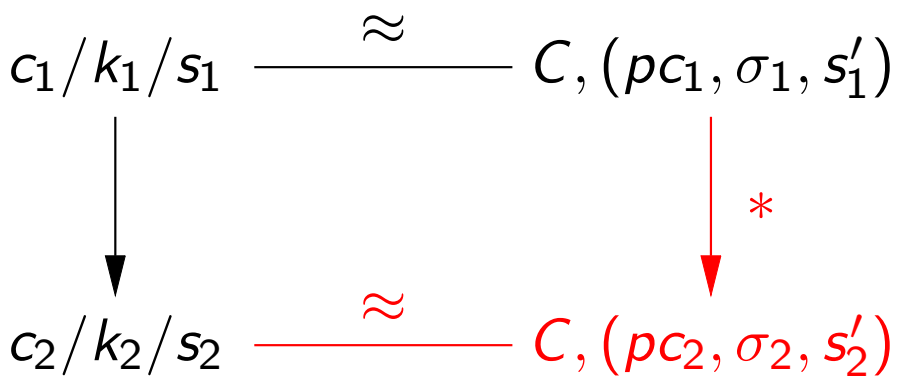


Forward simulation still holds.

"Star" simulation diagrams (incorrect)

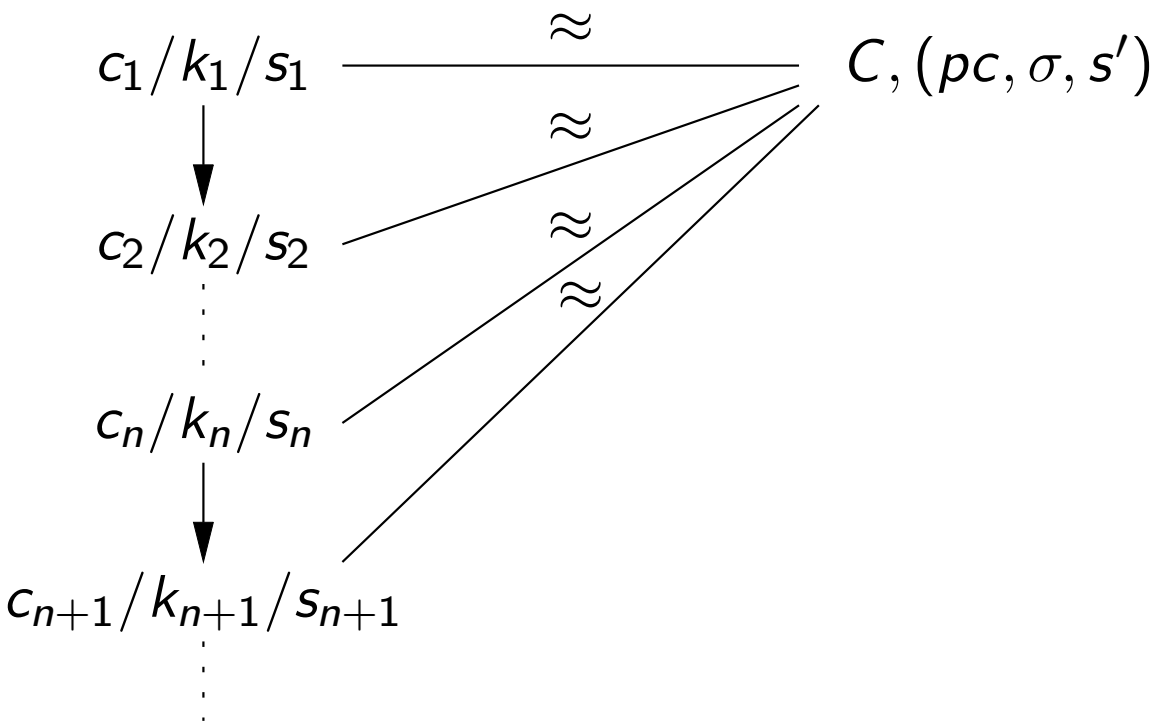
In other cases, each transition in the source program is simulated by zero, one or several transitions in the compiled code.

(Example: source reduction $(\texttt{SKIP}; c)/s \to c/s$ makes zero transitions in the machine code.)



Forward simulation is not guaranteed:
terminating executions are preserved;
but diverging executions may not be preserved.

The "infinite stuttering" problem



The source program diverges but the compiled code can terminate, normally or by going wrong.

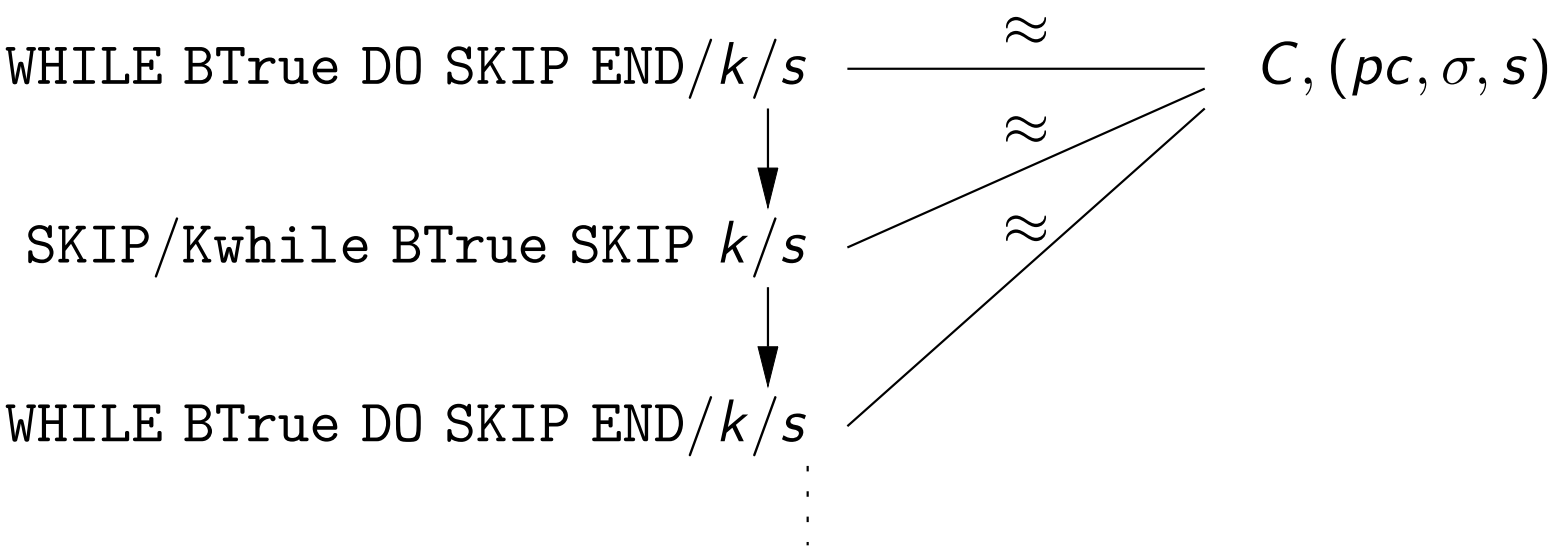
An incorrect optimization that exhibits infinite stuttering

Add special cases to `compile_com` so that the following trivially infinite loop gets compiled to no instructions at all:

```
compile_com (WHILE BTrue DO SKIP END) = nil
```

Infinite stuttering

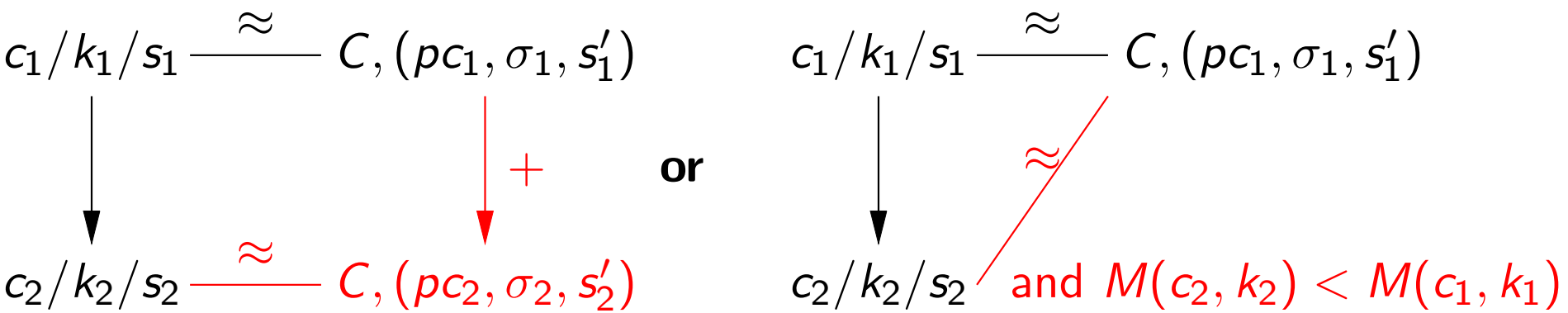
Adding special cases to the $\approx$ relation, we can prove the following naive "star" simulation diagram:



Conclusion: a naive "star" simulation diagram does not prove that a compiler is correct.

"Star" simulation diagrams (corrected)

Find a measure $M(c, k) : \texttt{nat}$ over source terms that decreases strictly when a stuttering step is taken. Then show:



Forward simulation, terminating case: OK (as before).

Forward simulation, diverging case: OK.
(If $c/k/s$ diverges, it must perform infinitely many non-stuttering steps, so the machine executes infinitely many transitions.)

Application to the IMP $\rightarrow$ VM compiler

Let's try to prove a "star" simulation diagram for our compiler.

Two difficulties:

1. Rule out infinite stuttering.
2. Match the current command-continuation c, k (which changes during reductions) with the compiled code C (which is fixed throughout execution).

Anti-stuttering measure

Stuttering reduction = no machine instruction executed. These include:

$$
\begin{array}{rcl}
(c_1; c_2)/k/s & \to & c_1/\texttt{Kseq}\ c_2\ k/s \\
\texttt{SKIP}/\texttt{Kseq}\ c\ k/s & \to & c/k/s \\
(\texttt{IFB BTrue THEN}\ c_1\ \texttt{ELSE}\ c_2)/k/s & \to & c_1/k/s \\
(\texttt{WHILE BTrue DO}\ c\ \texttt{END})/k/s & \to & c/\texttt{Kwhile BTrue}\ c\ k/s
\end{array}
$$

No measure M on the command c can rule out stuttering: for M to decrease in the second case above, we should have

$$M(\texttt{SKIP}) > M(c) \quad \text{for all commands } c\text{, including } c = \texttt{SKIP}$$

$\to$ We must measure (c, k) pairs.

Anti-stuttering measure

After some trial and error, an appropriate measure is:

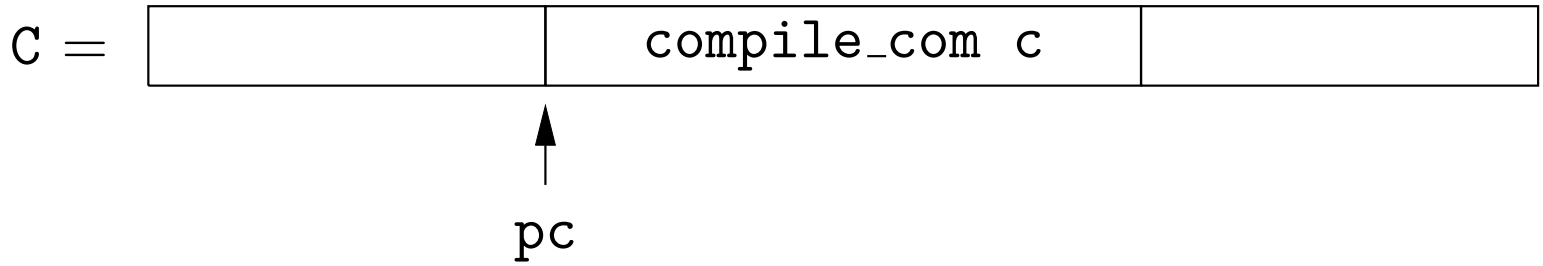
$$M(c, k) = size(c) + \sum_{c' \text{ appears in } k} size(c')$$

In other words, every constructor of `com` counts for 1, and every constructor of `cont` counts for 0.

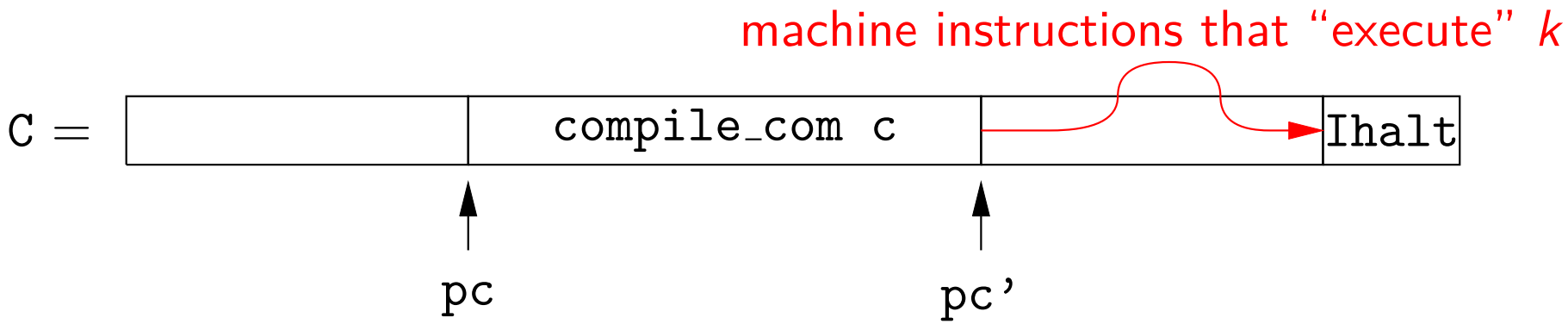
$$\begin{aligned}
M((c_1; c_2), k) &= M(c_1, \texttt{Kseq}\ c_2\ k) + 1 \\
M(\texttt{SKIP}, \texttt{Kseq}\ c\ k) &= M(c, k) + 1 \\
M(\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI}, k) &\geq M(c_1, k) + 1 \\
M(\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}, k) &= M(c, \texttt{Kwhile}\ b\ c\ k) + 1
\end{aligned}$$

Relating commands and continuations with compiled code

In the big-step proof: `codeseq_at C pc (compile_com c)`.



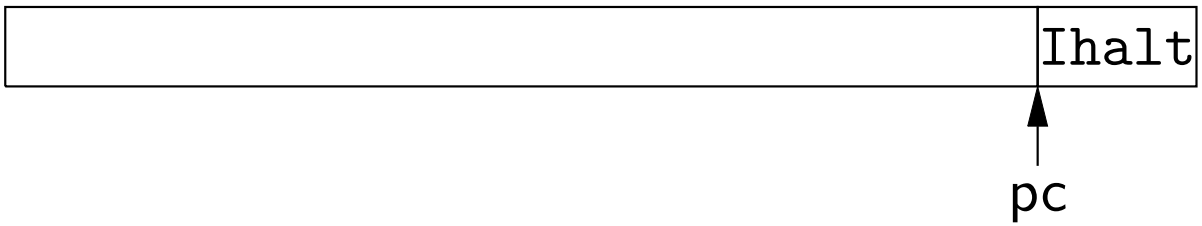
In a proof based on the small-step continuation semantics: we must also relate continuations k with the compiled code:



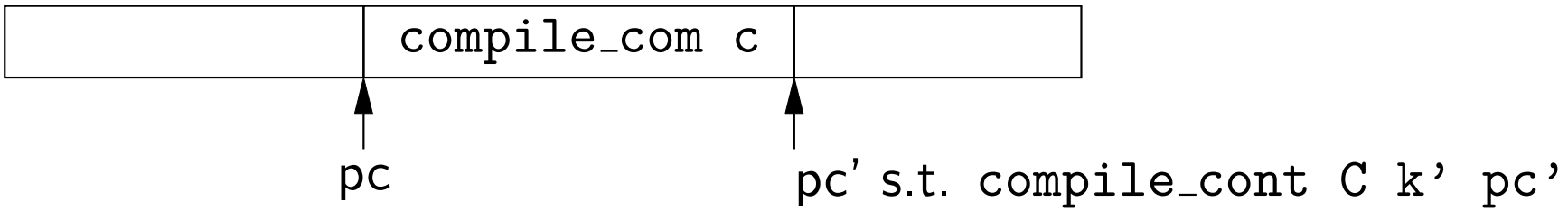
Relating continuations with compiled code

A predicate `compile_cont` C k pc, meaning "there exists a code path in C from pc to a `Ihalt` instruction that executes the pending computations described by k".

Base case $k =$ `Kstop`:

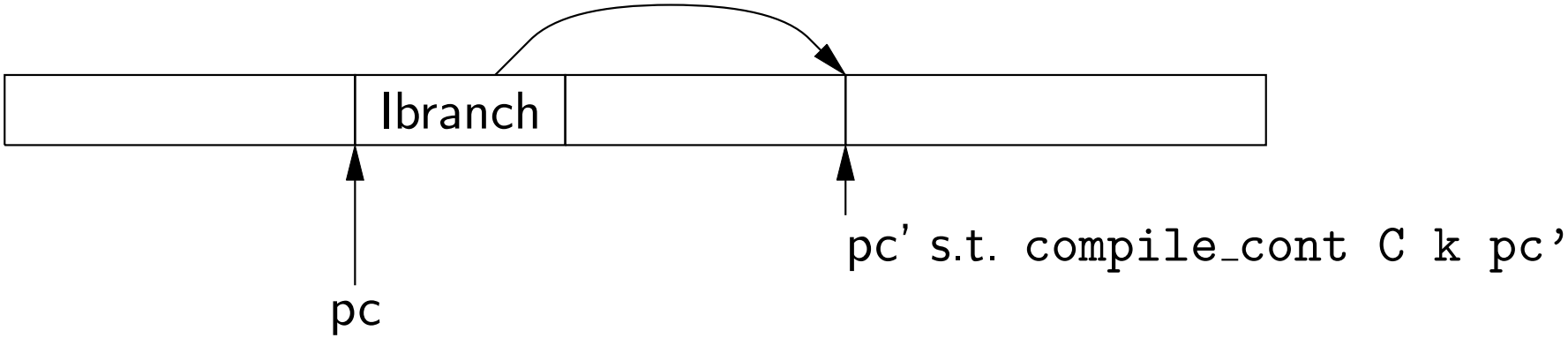


Sequence case $k =$ `Kseq` c k':

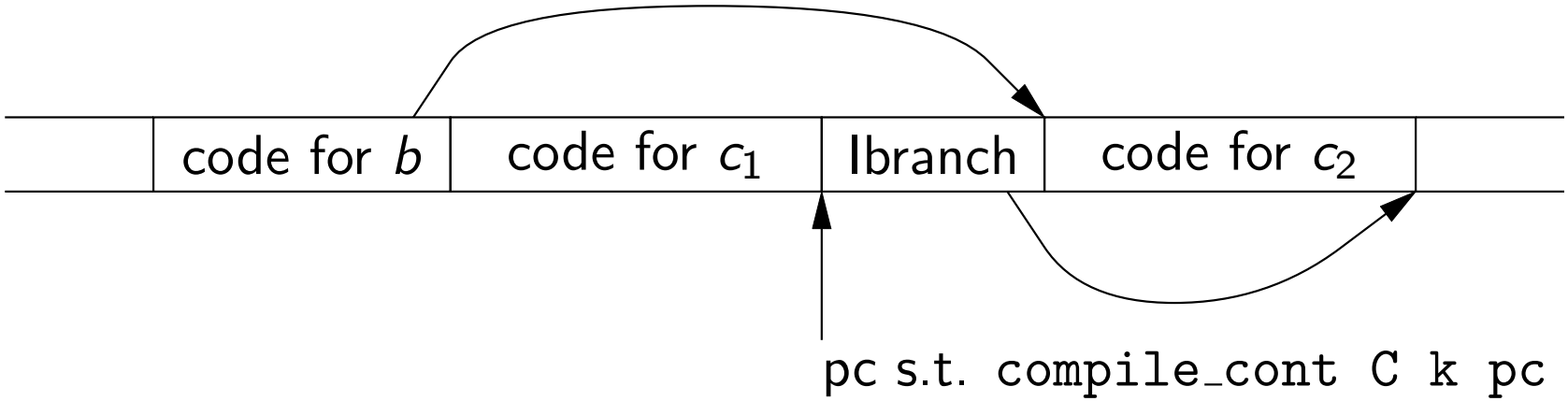


Relating continuations with compiled code

A "non-structural" case allowing us to insert branches at will:



Useful to handle continuations arising out of `IFB` b `THEN` c_1`ELSE` c_2:

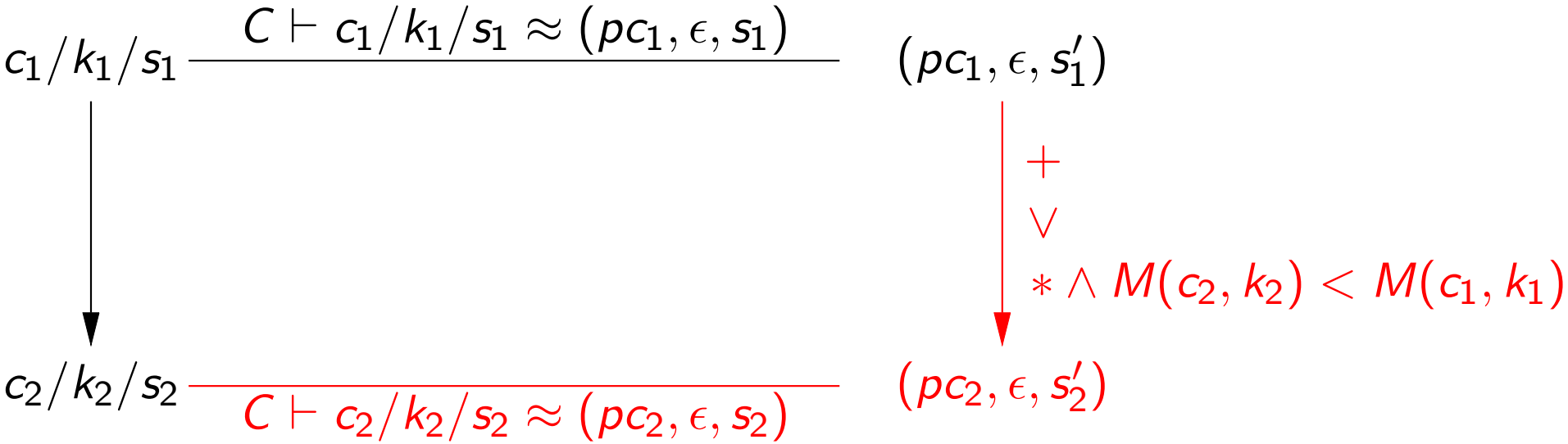


The simulation invariant

A source-level configuration (c, k, s) is related to a machine configuration $C, (pc, \sigma, s')$ iff:

- the memory states are identical: $s' = s$
- the stack is empty: $\sigma = \epsilon$
- C contains the compiled code for command c starting at pc
- C contains compiled code matching continuation k starting at $pc + |code(c)\}$.

The simulation diagram



Proof: by case analysis on the source transition on the left.

Wrapping up

As a corollary of this simulation diagram, we obtain both:

- An alternate proof of compiler correctness for terminating programs:
  if $c/\texttt{Kstop}/s \stackrel{*}{\rightarrow} \texttt{SKIP}/\texttt{Kstop}/s'$
  then $\texttt{mach\_terminates}\ (\texttt{compile\_program}\ c)\ s\ s'$
- A proof of compiler correctness for diverging programs:
  if $c/\texttt{Kstop}/s$ reduces infinitely,
  then $\texttt{mach\_diverges}\ (\texttt{compile\_program}\ c)\ s$

Mission complete!

Part V

Optimizations based on liveness analysis

Compiler optimizations

Automatically transform the programmer-supplied code into equivalent code that

- Runs faster
  - Removes redundant or useless computations.
  - Use cheaper computations (e.g. `x * 5` $\rightarrow$ `(x << 2) + x`)
  - Exhibits more parallelism (instruction-level, thread-level).
- Is smaller
  (For cheap embedded systems.)
- Consumes less energy
  (For battery-powered systems.)
- Is more resistant to attacks
  (For smart cards and other secure systems.)

Dozens of compiler optimizations are known, each targeting a particular class of inefficiencies.

Compiler optimization and static analysis

Some optimizations are unconditionally valid, e.g.:

$$\texttt{x} * 2 \;\rightarrow\; \texttt{x} + \texttt{x}$$
$$\texttt{x} * 4 \;\rightarrow\; \texttt{x} << 2$$

Most others apply only if some conditions are met:

$$\texttt{x} \,/\, 4 \;\rightarrow\; \texttt{x} >> 2 \quad \text{only if } \texttt{x} \geq 0$$
$$\texttt{x} + 1 \;\rightarrow\; 1 \quad \text{only if } \texttt{x} = 0$$
$$\texttt{if x} < \texttt{y then } c_1 \texttt{ else } c_2 \;\rightarrow\; c_1 \quad \text{only if } \texttt{x} < \texttt{y}$$
$$\texttt{x} := \texttt{y} + 1 \;\rightarrow\; \texttt{skip} \quad \text{only if x unused later}$$

$\rightarrow$ need a static analysis prior to the actual code transformation.

Static analysis

Determine some properties of all concrete executions of a program.

Often, these are properties of the values of variables at a given program point:

$$\mathtt{x} = n \qquad \mathtt{x} \in [n, m] \qquad \mathtt{x} = expr \qquad a.\mathtt{x} + b.\mathtt{y} \leq n$$

Requirements:

- The inputs to the program are unknown.
- The analysis must terminate.
- The analysis must run in reasonable time and space.

Running example: dead code elimination via liveness analysis

Remove assignments $x := e$, turning them into `skip`, whenever the variable x is never used later in the program execution.

Example

Consider: `x := 1; y := y + 1; x := 2`

The assignment `x := 1` can always be eliminated since `x` is not used before being redefined by `x := 2`.

Builds on a static analysis called liveness analysis.

Optimizations based on liveness analysis

8. Liveness analysis

9. Dead code elimination

10. Advanced topic: register allocation

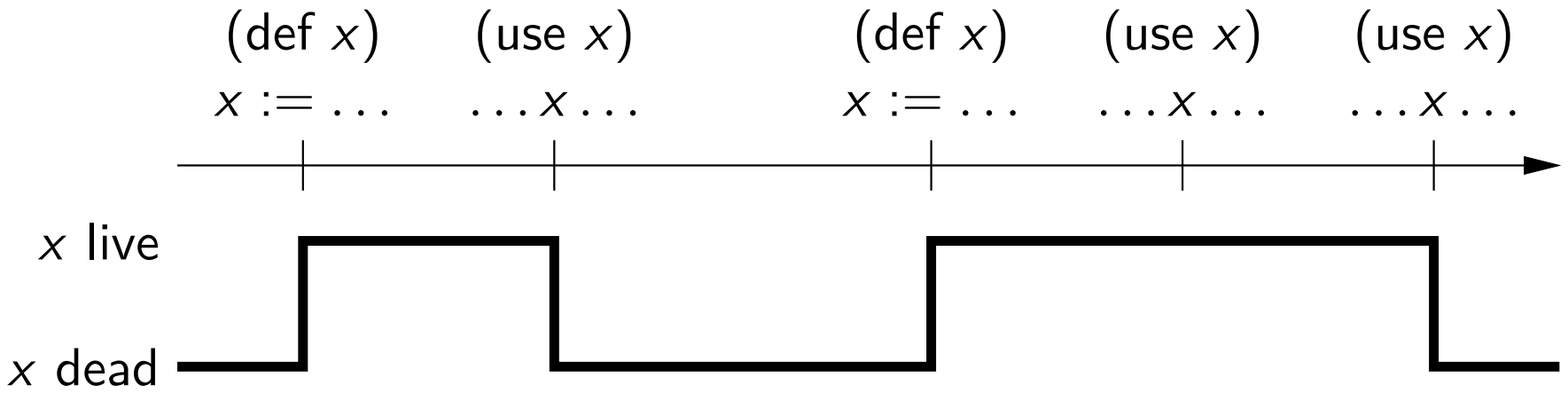
Notions of liveness

A variable is dead at a program point if its value is not used later in any execution of the program:

- either the variable is not mentioned again before going out of scope
- or it is always redefined before further use.

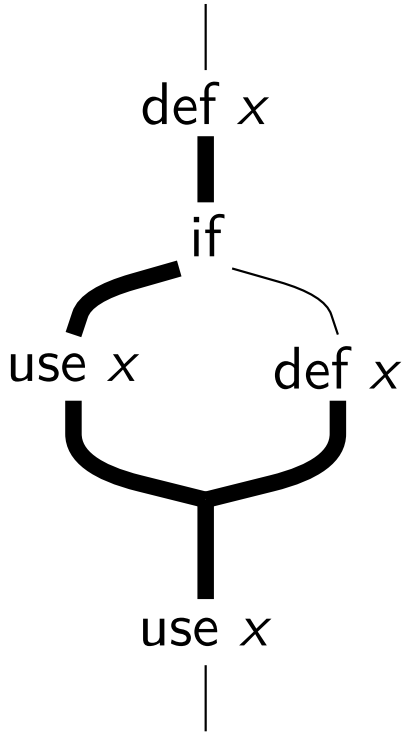
A variable is live if it is not dead.

Easy to compute for straight-line programs (sequences of assignments):



Notions of liveness

Liveness information is more delicate to compute in the presence of conditionals and loops:



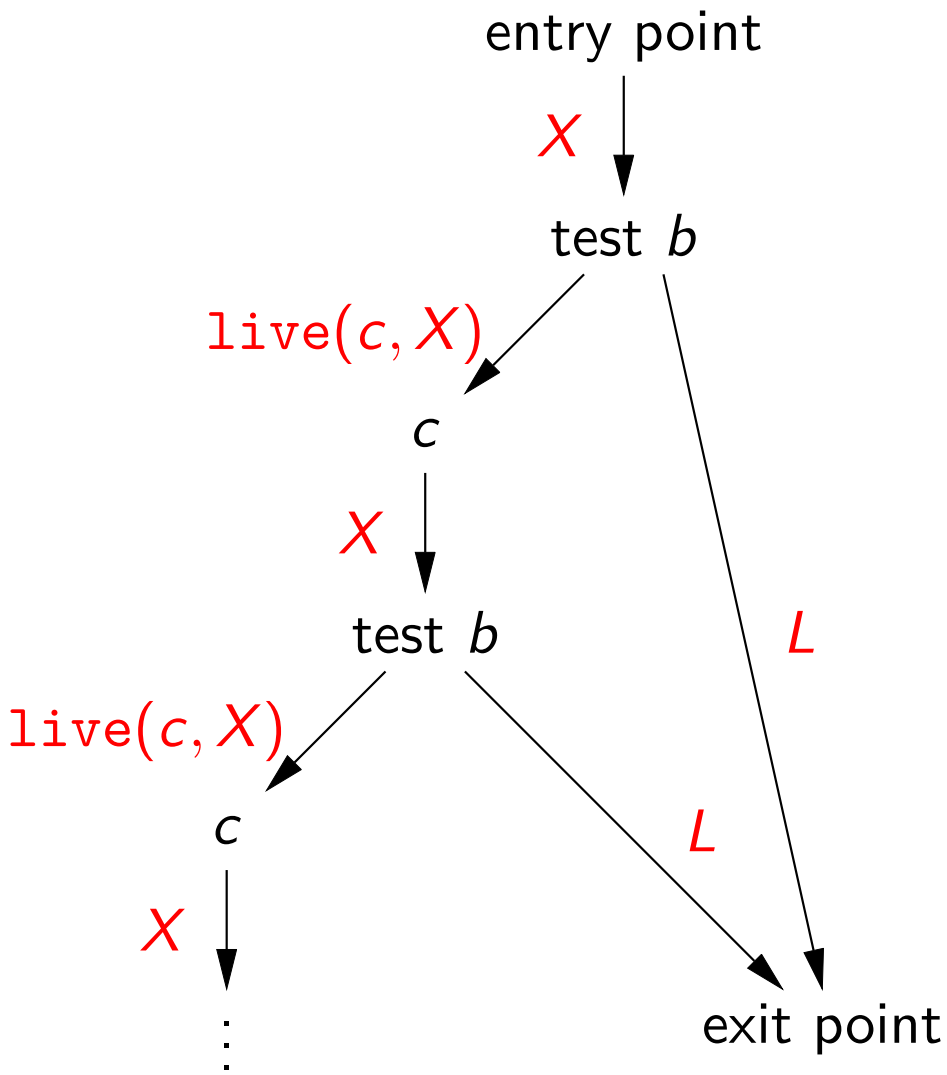
Conservatively over-approximate liveness, assuming all `if` conditionals can be true or false, and all `while` loops are taken 0 or several times.

Liveness equations

Given a set L of variables live "after" a command c, write $\texttt{live}(c, L)$ for the set of variables live "before" the command.

$$\begin{aligned}
\texttt{live}(\texttt{SKIP}, L) &= L \\
\texttt{live}(x := a,\ L) &= \begin{cases} (L \setminus \{x\}) \cup FV(a) & \text{if } x \in L; \\ L & \text{if } x \notin L. \end{cases} \\
\texttt{live}((c_1; c_2),\ L) &= \texttt{live}(c_1, \texttt{live}(c_2, L)) \\
\texttt{live}((\texttt{IFB } b \texttt{ THEN } c_1 \texttt{ ELSE } c_2),\ L) &= FV(b) \cup \texttt{live}(c_1, L) \cup \texttt{live}(c_2, L) \\
\texttt{live}((\texttt{WHILE } b \texttt{ DO } c \texttt{ END}),\ L) &= X \text{ such that} \\
& \quad X \supseteq L \cup FV(b) \cup \texttt{live}(c, X)
\end{aligned}$$

Liveness for loops



We must have:

- $FV(b) \subseteq X$
  (evaluation of b)
- $L \subseteq X$
  (if b is false)
- $\texttt{live}(c, X) \subseteq X$
  (if b is true and c is executed)

Fixpoints, a.k.a "the recurring problem"

Consider $F = \lambda X.\ L \cup FV(b) \cup \texttt{live}(c, X)$.

To analyze `while` loops, we need to compute a pre-fixpoint of F, i.e. an X such that $F(X) \subseteq X$.

For maximal precision, X would preferably be the smallest fixpoint $F(X) = X$; but for soundness, any pre-fixpoint suffices.

The mathematician's approach to fixpoints

Let $A, \leq$ be a partially ordered type. Consider $F : A \to A$.

Theorem (Knaster-Tarski)

The sequence

$$\bot,\ F(\bot),\ F(F(\bot)),\ \ldots,\ F^n(\bot), \ldots$$

converges to the smallest fixpoint of F, provided that

- *F is increasing: $x \leq y \Rightarrow F(x) \leq F(y)$.*
- *$\bot$ is a smallest element.*
- *All strictly ascending chains $x_0 < x_1 < \ldots < x_n$ are finite.*

This provides an effective way to compute fixpoints.
(See Coq file `Fixpoint.v`).

Problems with Knaster-Tarski

1. Formalizing and exploiting the ascending chain property
   $\rightarrow$ well-founded orderings and Noetherian induction.

2. In our case (liveness analysis), the ordering $\subset$ has infinite ascending chains: $\emptyset \subset \{x_1\} \subset \{x_1, x_2\} \subset \cdots$
   Need to restrict ourselves to subsets of a given, finite universe of variables (= all variables free in the program).
   $\rightarrow$ dependent types.

Time for plan B...

The engineer's approach to fixpoints

$$F = \lambda X.\ L \cup FV(b) \cup \texttt{live}(c, X)$$

- Compute $F(\emptyset), F(F(\emptyset)), \ldots, F^N(\emptyset)$ up to some fixed N.
- Stop as soon as a pre-fixpoint is found ($F^{i+1}(\emptyset) \subseteq F^i(\emptyset)$).
- Otherwise, return a safe over-approximation (in our case, $a \cup FV(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done})$).

A compromise between analysis time and analysis precision.

(Coq implementation: see file `Deadcode.v`)

Optimizations based on liveness analysis

Dead code elimination

The program transformation eliminates assignments to dead variables:

$x := a$ becomes `SKIP` if x is not live "after" the assignment

Presented as a function `dce` : `com` $\rightarrow$ `VS.t` $\rightarrow$ `com`
taking the set of variables live "after" as second parameter
and maintaining it during its traversal of the command.

(Implementation & examples in file `Deadcode.v`)

The semantic meaning of liveness

What does it mean, semantically, for a variable x to be live at some program point?

Hmmm...

The semantic meaning of liveness

What does it mean, semantically, for a variable x to be live at some program point?

Hmmm...

What does it mean, semantically, for a variable x to be dead at some program point?

That its precise value has no impact on the rest of the program execution!

Liveness as an information flow property

Consider two executions of the same command c in different initial states:

$$\begin{array}{lcl} c/s_1 & \Rightarrow & s_2 \\ c/s'_1 & \Rightarrow & s'_2 \end{array}$$

Assume that the initial states agree on the variables $\texttt{live}(c, L)$ that are live "before" c:

$$\forall x \in \texttt{live}(c, L), \quad s_1(x) = s'_1(x)$$

Then, the two executions terminate on final states that agree on the variables L live "after" c:

$$\forall x \in L, \quad s_2(x) = s'_2(x)$$

The proof of semantic preservation for dead-code elimination follows this pattern, relating executions of c and $\texttt{dce}\ c\ L$ instead.

Agreement and its properties

```
Definition agree (L: VS.t) (s1 s2: state) : Prop :=
  forall x, VS.In x L -> s1 x = s2 x.
```

Agreement is monotonic w.r.t. the set of variables L:

```
Lemma agree_mon:
  forall L L' s1 s2,
  agree L' s1 s2 -> VS.Subset L L' -> agree L s1 s2.
```

Expressions evaluate identically in states that agree on their free variables:

```
Lemma aeval_agree:
  forall L s1 s2, agree L s1 s2 ->
  forall a, VS.Subset (fv_aexp a) L -> aeval s1 a = aeval s2 a.
Lemma beval_agree:
  forall L s1 s2, agree L s1 s2 ->
  forall b, VS.Subset (fv_bexp b) L -> beval s1 b = beval s2 b.
```

Agreement and its properties

Agreement is preserved by parallel assignment to a variable:

```
Lemma agree_update_live:
  forall s1 s2 L x v,
  agree (VS.remove x L) s1 s2 ->
  agree L (update s1 x v) (update s2 x v).
```

Agreement is also preserved by unilateral assignment to a variable that is dead "after":

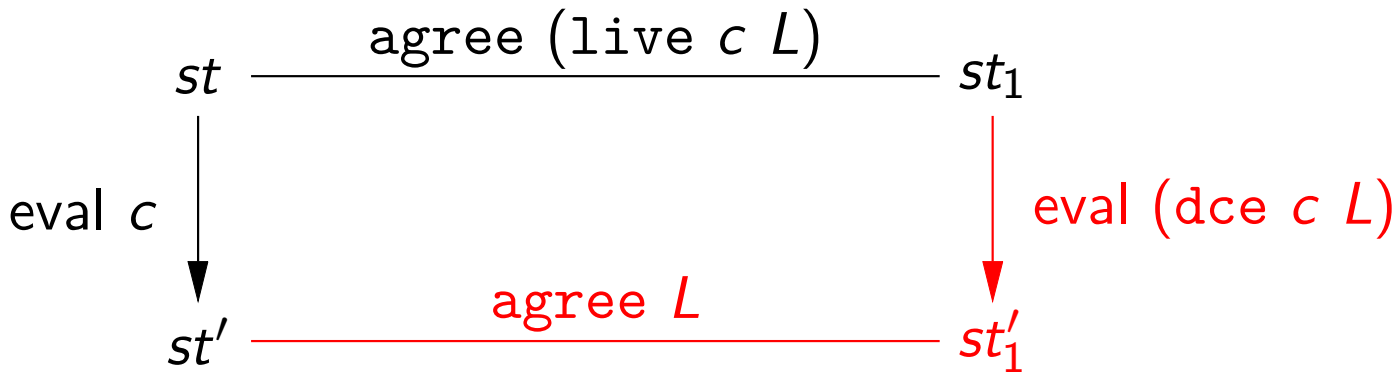
```
Lemma agree_update_dead:
  forall s1 s2 L x v,
  agree L s1 s2 -> ~VS.In x L ->
  agree L (update s1 x v) s2.
```

Forward simulation for dead code elimination

For terminating source programs:

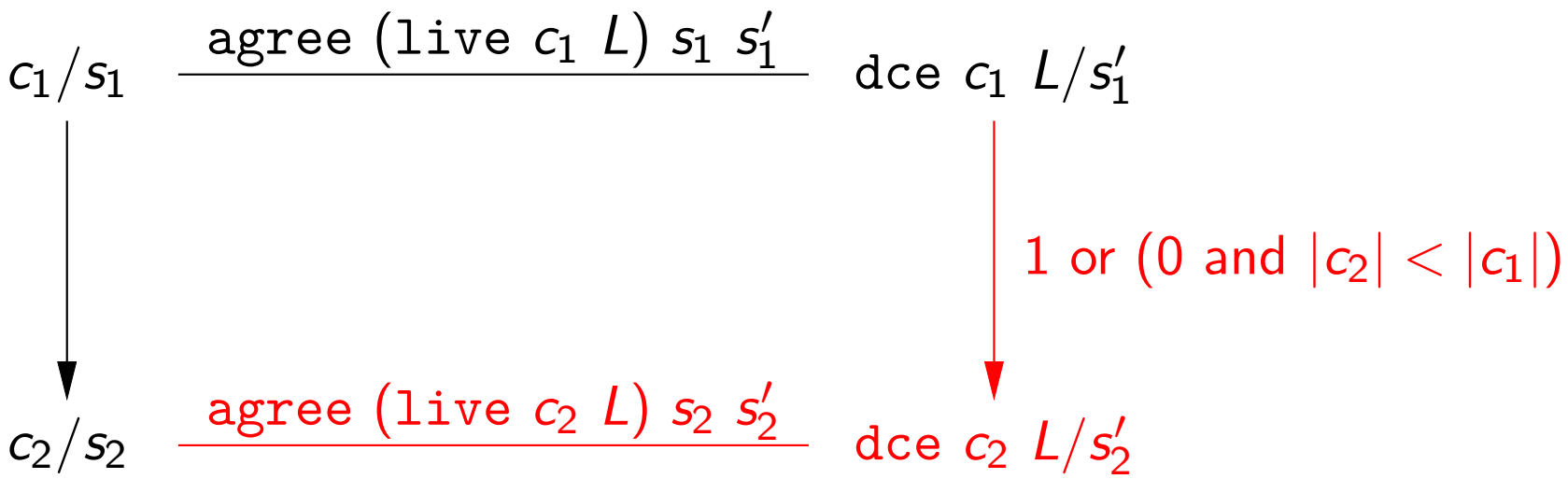
```
Theorem dce_correct_terminating:
  forall st c st', c / st \\ st' ->
  forall L st1,
  agree (live c L) st st1 ->
  exists st1', dce c L / st1 \\ st1' /\ agree L st' st1'.
```

(Proof: an induction on the derivation of `c / st ==> st'`.)



Forward simulation for dead code elimination

Exercise: extend the result to diverging programs by proving a simulation diagram for the transitions of the small-step semantics of IMP (no need for continuations):



Optimizations based on liveness analysis

8 Liveness analysis

9 Dead code elimination

10 Advanced topic: register allocation

The register allocation problem

Place the variables used by the program (in unbounded number) into:

- either hardware registers
  (very fast access, but available in small quantity)
- or memory locations (generally allocated on the stack)
  (available in unbounded quantity, but slower access)

Try to maximize the use of hardware registers.

A crucial step for the generation of efficient machine code.

Approaches to register allocation

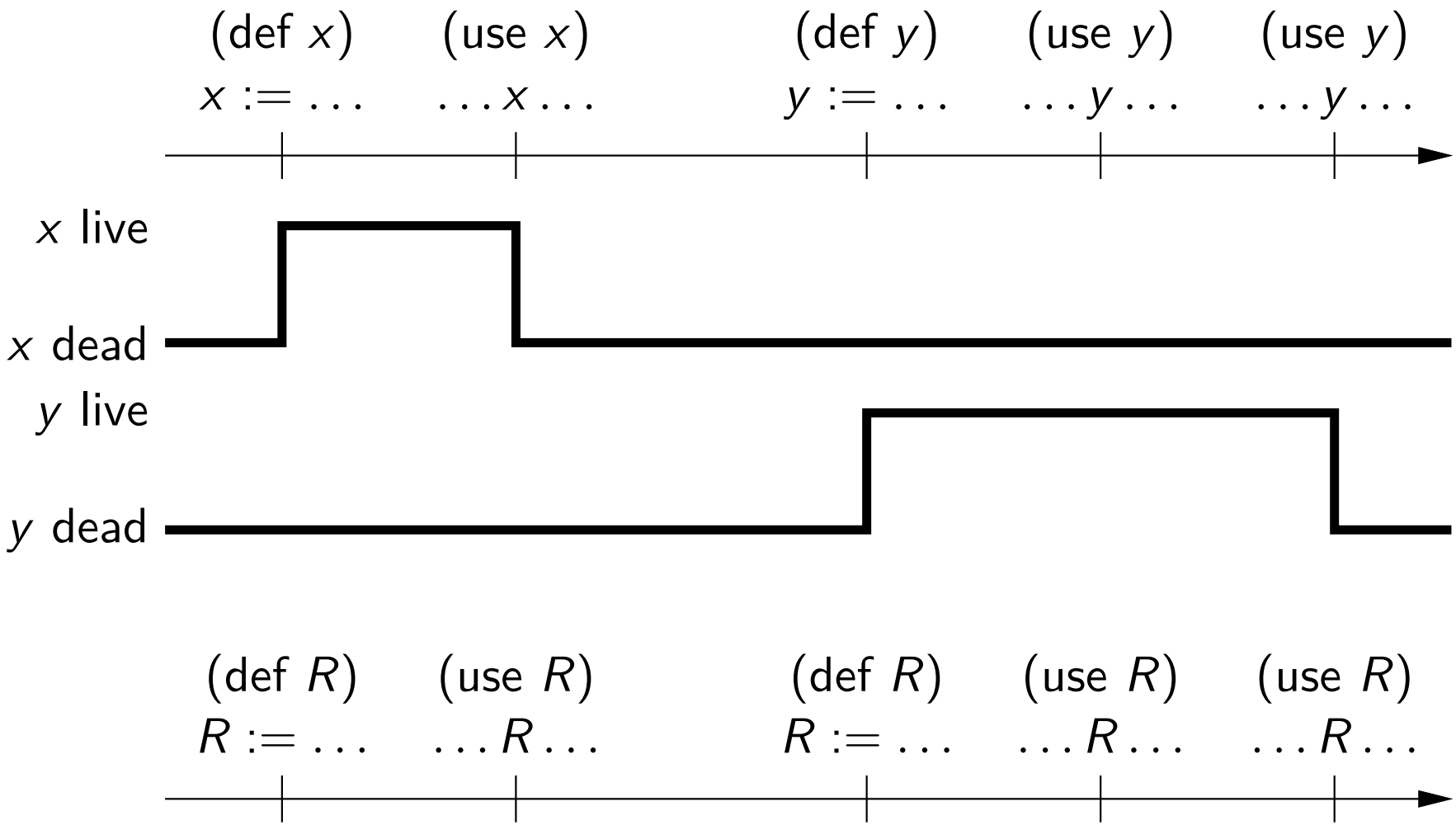
Naive approach (injective allocation):

- Assign the N most used variables to the N available registers.
- Assign the remaining variables to memory locations.

Optimized approach (non-injective allocation):

- Notice that two variables can share a register
  as long as they are not simultaneously live.

Example of register sharing



Register allocation for IMP

Properly done:

1. Break complex expressions by introducing temporaries.
   (E.g. `x = (a + b) * y` becomes `tmp = a + b; x = tmp * y`.)
2. Translate IMP to a variant IMP′ that uses registers ∪ memory locations instead of variables.

Simplified as follows in this lecture:

1. Do not break expressions.
2. Translate from IMP to IMP, by renaming identifiers.
   (Convention: low-numbered identifiers ≈ hardware registers.)

The program transformation

Assume given a "register assignment" $f : \texttt{id} \to \texttt{id}$.

The program transformation consists of:

- Renaming variables: all occurrences of x become $f\ x$.
- Dead code elimination:

$$x ::= a \quad \longrightarrow \quad \texttt{SKIP} \qquad \text{if } x \text{ is dead "after"}$$

- Coalescing:

$$x ::= y \quad \longrightarrow \quad \texttt{SKIP} \qquad \text{if } f\ x = f\ y$$

Correctness conditions on the register assignment

Clearly, not all register assignments f preserve semantics.

Example: assume $f\ x = f\ y = f\ z = R$

```
x ::= 1;                        R ::= 1;
y ::= 2;            ---->       R ::= 2;
z ::= x + y;                    R ::= R + R;
```

Computes 4 instead of 3 . . .

What are sufficient conditions over f? Let's discover them by reworking the proof of dead code elimination.

Agreement, revisited

```
Definition agree (L: VS.t) (s1 s2: state) : Prop :=
  forall x, VS.In x L -> s1 x = s2 (f x).
```

An expression and its renaming evaluate identically in states that agree on their free variables:

```
Lemma aeval_agree:
  forall L s1 s2, agree L s1 s2 ->
  forall a, VS.Subset (fv_aexp a) L ->
  aeval s1 a = aeval s2 (rename_aexp a).
Lemma beval_agree:
  forall L s1 s2, agree L s1 s2 ->
  forall b, VS.Subset (fv_bexp b) L ->
  beval s1 b = beval s2 (rename_bexp b).
```

Agreement, revisited

As before, agreement is monotonic w.r.t. the set of variables L:

```
Lemma agree_mon:
  forall L L' s1 s2,
  agree L' s1 s2 -> VS.Subset L L' -> agree L s1 s2.
```

As before, agreement is preserved by unilateral assignment to a variable that is dead "after":

```
Lemma agree_update_dead:
  forall s1 s2 L x v,
  agree L s1 s2 -> ~VS.In x L ->
  agree L (update s1 x v) s2.
```

Agreement, revisited

Agreement is preserved by parallel assignment to a variable x and its renaming $f\ x$, but only if f satisfies a non-interference condition (in red below):

```
Lemma agree_update_live:
  forall s1 s2 L x v,
  agree (VS.remove x L) s1 s2 ->
  (forall z, VS.In z L -> z <> x -> f z <> f x) ->
  agree L (update s1 x v) (update s2 (f x) v).
```

Counter-example: assume $f\ x = f\ y = R$.
agree $\{y\}$ $(x = 0, y = 0)$ $(R = 0)$ holds, but
agree $\{x; y\}$ $(x = 1, y = 0)$ $(R = 1)$ does not.

A special case for moves

Consider a variable-to-variable copy $x ::= y$.
In this case, the value v assigned to x is not arbitrary, but known to be `s1` y. We can, therefore, weaken the non-interference criterion:

```
Lemma agree_update_move:
  forall s1 s2 L x y,
  agree (VS.union (VS.remove x L) (VS.singleton y)) s1 s2 ->
  (forall z, VS.In z L -> z <> x -> z <> y -> f z <> f x) ->
  agree L (update s1 x (s1 y)) (update s2 (f x) (s2 (f y))).
```

This makes it possible to assign x and y to the same location, even if x and y are simultaneously live.

The interference graph

The various non-interference constraints $f\ x \neq f\ y$ can be represented as an interference graph:

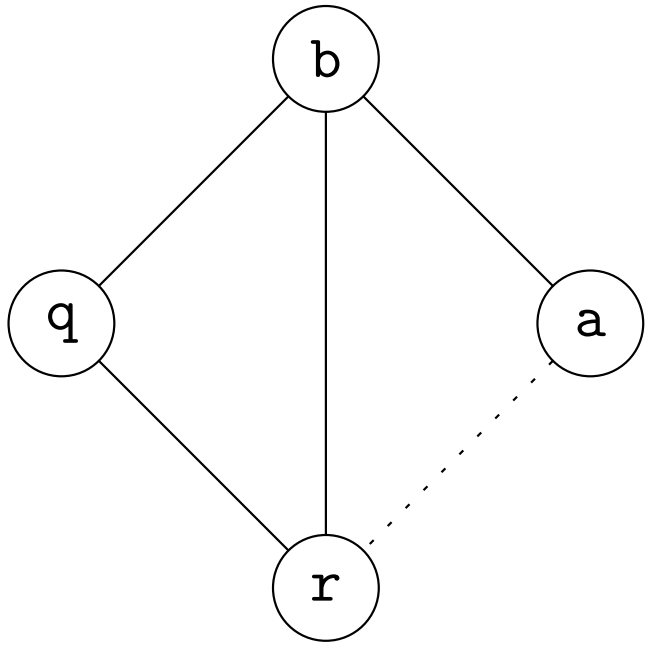
- Nodes = program variables.
- Undirected edge between x and y = x and y cannot be assigned the same location.

Chaitin's algorithm to construct this graph:

- For each move $x ::= y$, add edges between x and every variable z live "after" except x and y.
- For each other assignment $x ::= a$, add edges between x and every variable z live "after" except x.

Example of an interference graph

```
r := a;
q := 0;
WHILE b <= r DO
  r := r - b;
  q := q + 1
END
```



(Full edge = interference; dotted edge = preference.)

Register allocation as a graph coloring problem

(G. Chaitin, 1981; P. Briggs, 1987)

Color the interference graph, assigning a register or memory location to every node;

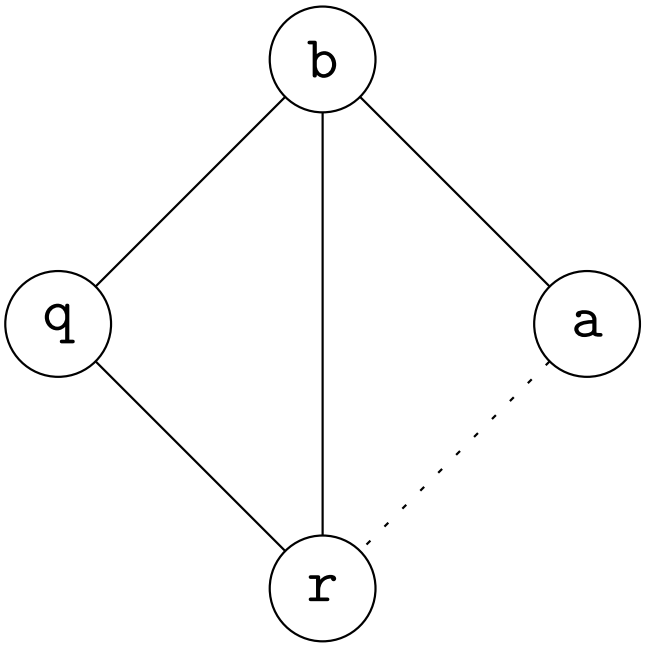
under the constraint that the two ends of an interference edge have different colors;

with the objective to

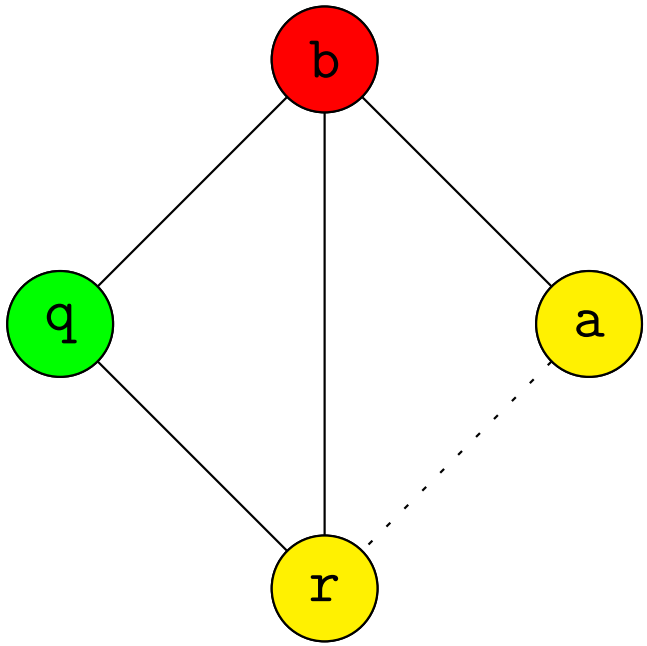
- minimize the number (or total weight) of nodes that are colored by a memory location
- maximize the number of preference edges whose ends have the same color.

(A NP-complete problem in general, but good linear-time heuristics exist.)

Example of coloring



Example of coloring




```
yellow := yellow;
green := 0;
WHILE red <= yellow DO
  yellow := yellow - red;
  green := green + 1
END
```

What needs to be proved in Coq?

Full compiler proof:

formalize and prove correct a good graph coloring heuristic.

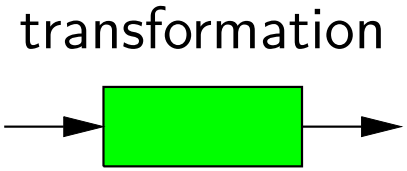
George and Appel's *Iterated Register Coalescing* $\approx$ 6 000 lines of Coq.

Validation a posteriori:

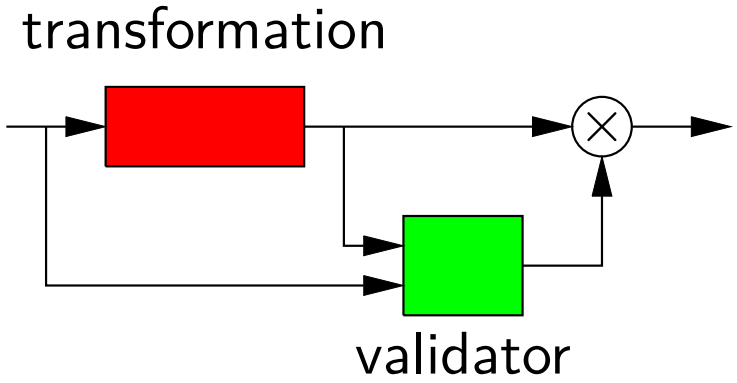
invoke an external, unproven oracle to compute a candidate allocation;
check that it satisfies the non-interference conditions;
abort compilation if the checker says `false`.

The verified transformation–verified validation spectrum

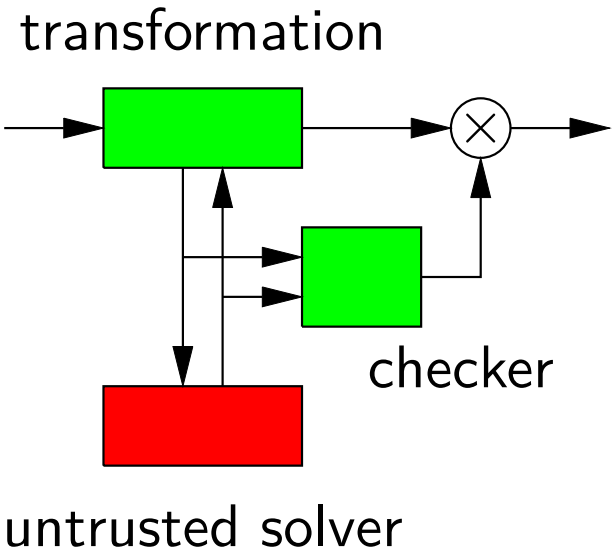
Verified transformation



Verified translation validation



External solver with verified validation



= formally verified

= not verified

Validating candidate allocations in Coq

It is easy to write a Coq boolean-valued function

```
correct_allocation: (id -> id) -> com -> VS.t -> bool
```

that returns `true` only if the expected non-interference properties are satisfied.

(See file `Regalloc.v`.)

Semantic preservation

The proofs of forward simulation that we did for dead code elimination then extend easily, under the assumption that `correct_allocation` returns true:

```
Theorem transf_correct_terminating:
  forall st c st', c / st
 st' ->
  forall L st1, agree (live c L) st st1 ->
  correct_allocation c L = true ->
  exists st1', transf_com c L / st1
 st1' / agree L st' st1'.
```

Part VI

Compiler verification in the large: CompCert

Compiler verification in the large: CompCert

11. Compiler issues in critical software
12. The CompCert project
13. Tower of Babel: the languages of CompCert
14. The CompCert memory model
15. Observables, non-determinism, and semantic preservation
16. Concluding remarks

The CompCert project

(X.Leroy, S.Blazy, et al — `http://compcert.inria.fr/`)

Develop and prove correct a realistic compiler, usable for critical embedded software.

- Source language: a subset of C.
- Target language: PowerPC, ARM and x86 assembly.
- Generates reasonably compact and fast code
  $\Rightarrow$ some optimizations.

This is "software-proof codesign": the compiler and its proof are written from scratch; not trying to prove an existing compiler.

Uses Coq to mechanize the proof of semantic preservation and also to implement most of the compiler.

The subset of C supported

Supported:

- Types: integers, floats, arrays, pointers, `struct`, `union`.
- Operators: arithmetic, pointer arithmetic.
- Control: `if`/`then`/`else`, loops, simple `switch`, `goto`.
- Functions, recursive functions, function pointers.

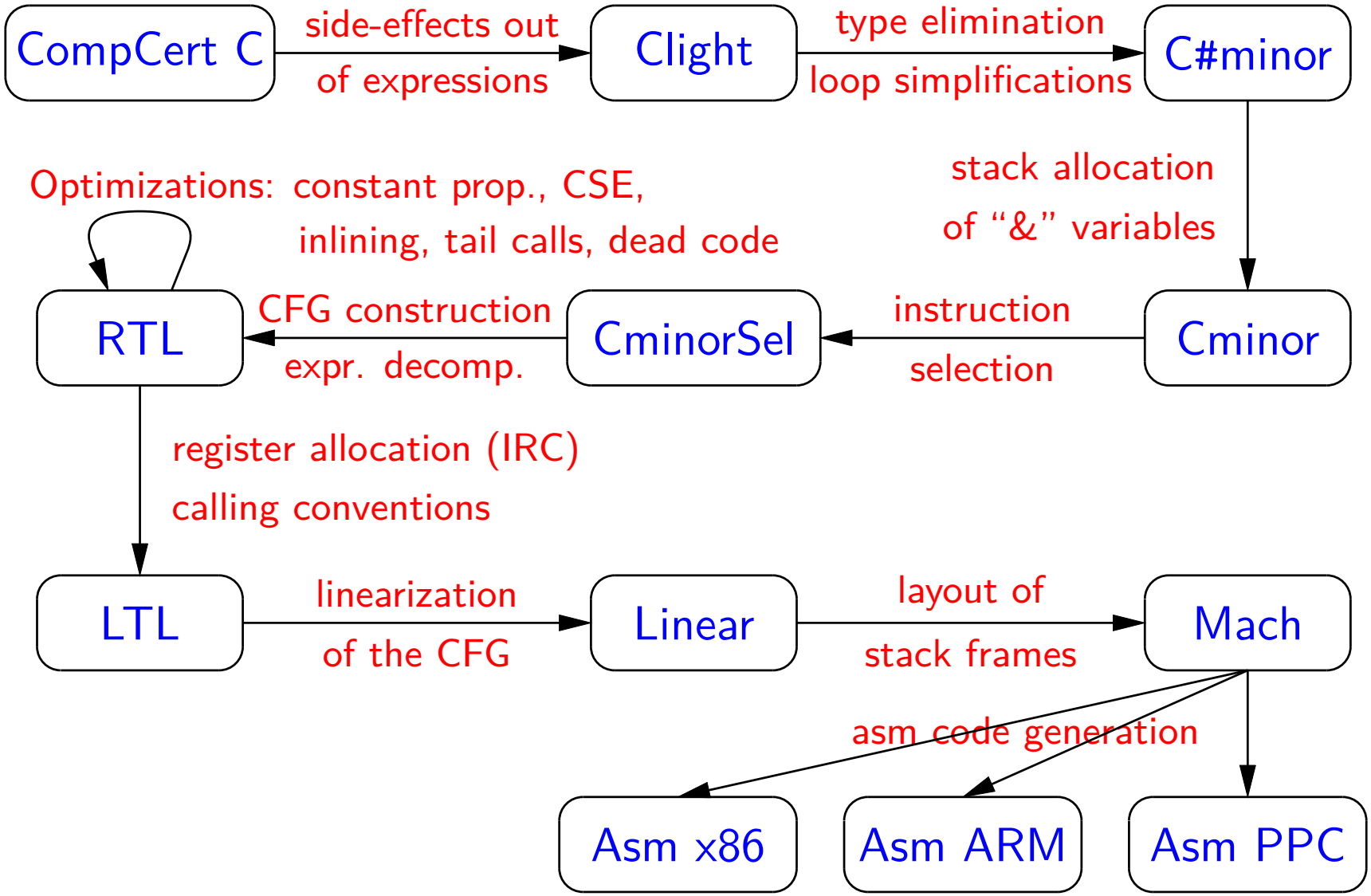
Not supported:

- The `long double` type.
- Unstructured `switch`, `longjmp`/`setjmp`.

Supported via de-sugaring (not proved!):

- Block-scoped variables.
- Returning `struct` and `union` by value from functions
- Bit-fields.
- Variable-arity functions.

The formally verified part of the compiler



Formally verified using Coq

The correctness proof (semantic preservation) for the compiler is entirely machine-checked, using the Coq proof assistant.

```
Theorem transf_c_program_correct:
  forall (p: Csyntax.program) (tp: Asm.program)
         (b: behavior),
  transf_c_program p = OK tp ->
  program_behaves (Asm.semantics tp) b ->
  exists b', program_behaves (Csem.semantics p) b'
          /\ behavior_improves b' b.
```

A fairly large proof: 120 000 lines, 8 person.years, low automation.

Programmed (mostly) in Coq

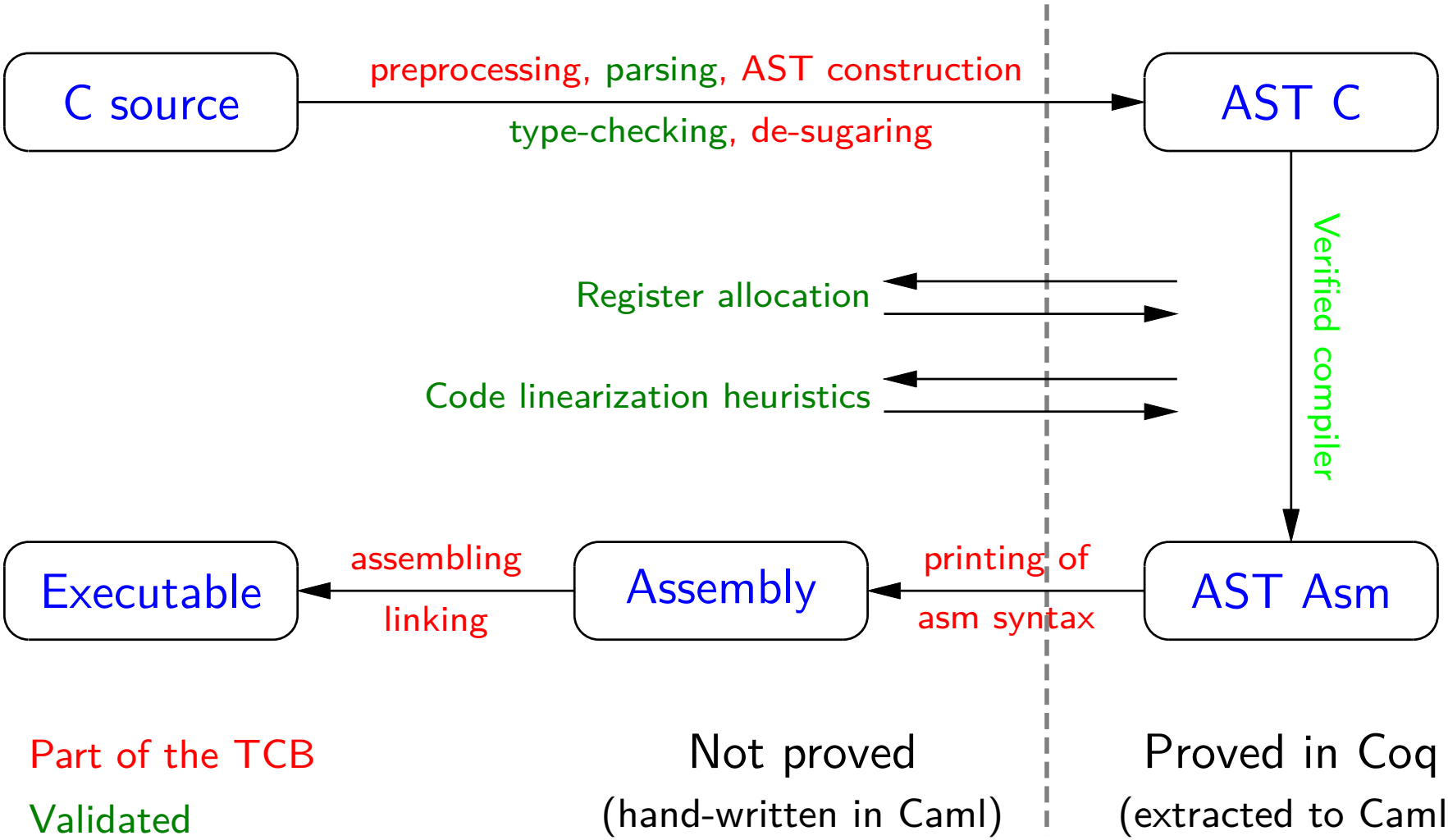
All the verified parts of the compiler are programmed directly in Coq's specification language, using pure functional style.

- Monads to handle errors and mutable state.
- Purely functional data structures.

Coq's extraction mechanism produces executable Caml code from these specifications.

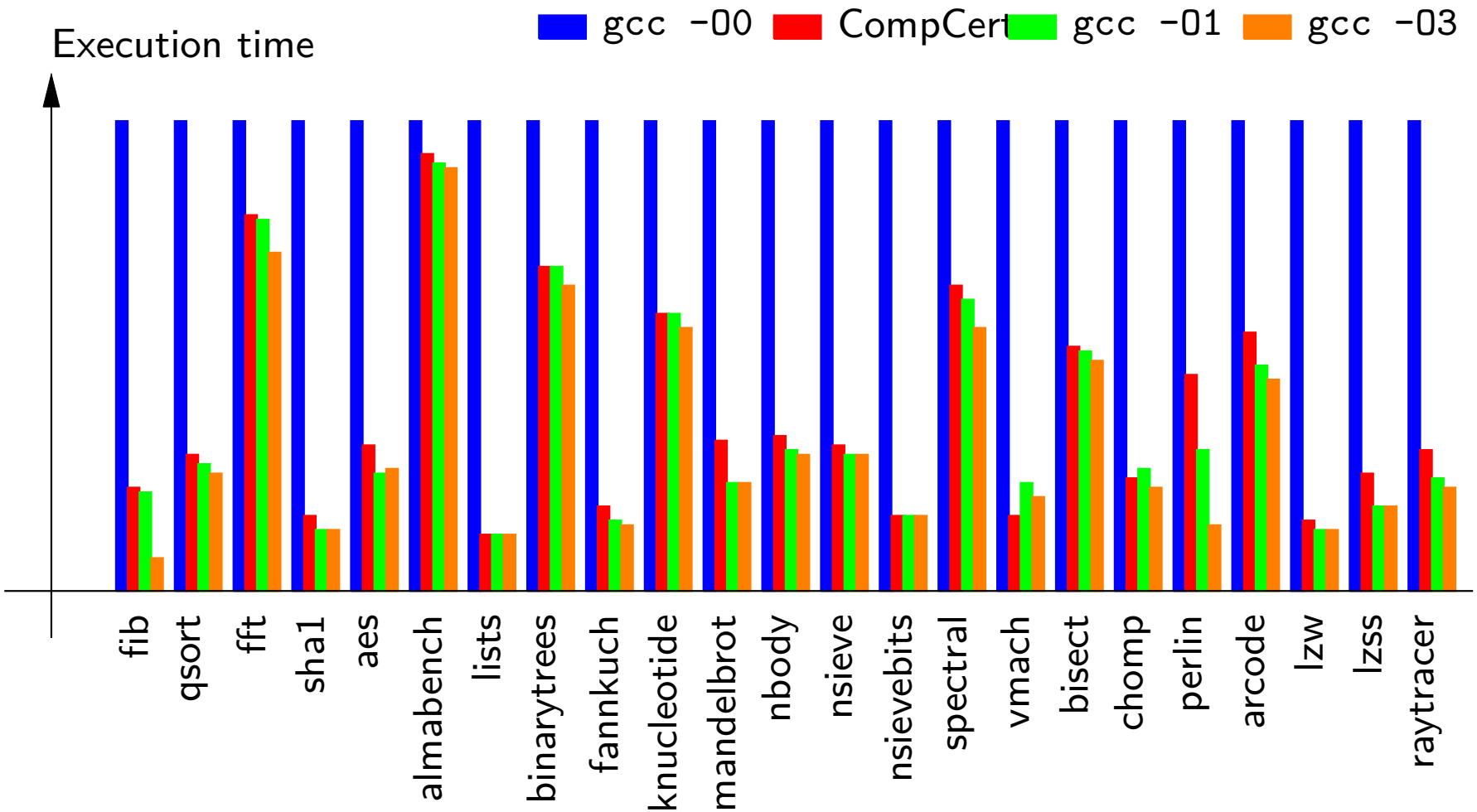
Claim: purely functional programming is the shortest path to writing and proving a program.

The whole Compcert compiler



Performance of generated code

(On a Power 7 processor)



A tangible increase in quality

The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

X. Yang, Y. Chen, E. Eide, J. Regehr, PLDI 2011

Availability

`http://compcert.inria.fr/`

Source distribution, documentation, papers, commented Coq.

For non-commercial use (evaluation, research, teaching).

AbsInt (`http://absint.com`) sells a commercial version of CompCert with technical support.

Compiler verification in the large: CompCert

The languages of CompCert

Source language: CompCert C

Target language: Asm (three variants: ARM, PowerPC, x86)

8 intermediate languages that bridge the gap.

Some intermediate languages are reused in other projects:

- Clight: VST program logic; Velus Lustre compiler
- C#minor: Verasco static analyzer by abstract interpretation
- Cminor: MiniML compiler, GHC Core compiler.

Machine (in-)dependence

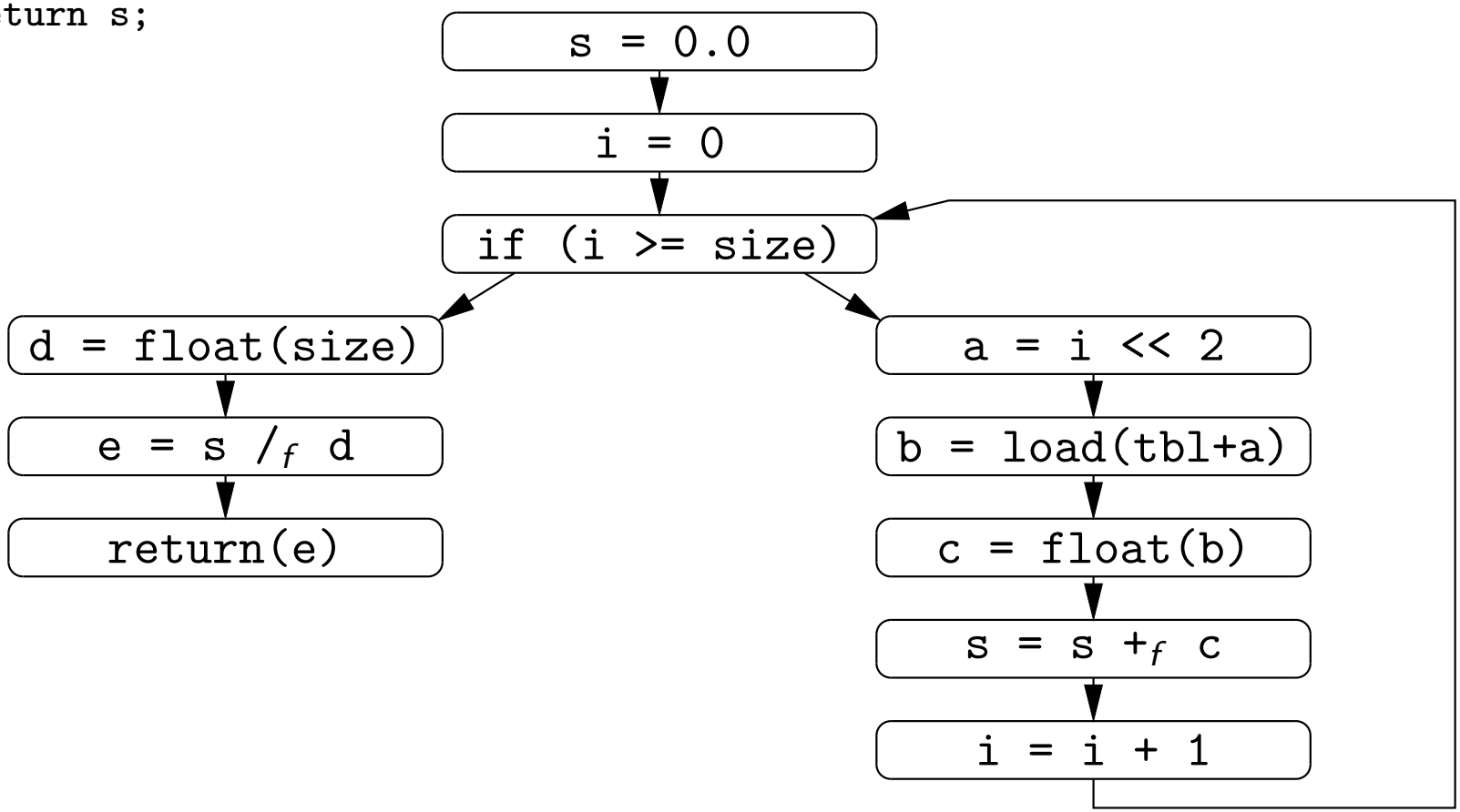
CompCert C Clight C#minor Cminor	Independent of the target processor
CminorSel RTL LTL Linear Mach	Parameterized by the target processor – Operations, conditions, addressing modes (module `Op`) – Machine registers (`Machregs`) – Calling conventions (`Conventions0`)
Asm	Specific to the target processor

Representing control flow

CompCert C	Expressions with side-effects; statements (structured, goto)
Clight	Pure expressions; statements (structured, goto)
C#minor	
Cminor	
CminorSel	
RTL	Control-Flow Graph of instructions
LTL	Control-Flow Graph of basic blocks
Linear	List of instructions w/ labels and jumps
Mach	
Asm	List of instructions indexed by PC

Control-flow graphs

```
double average(int * tbl, int size)
{
    double s = 0.0;
    int i;
    for (i = 0; i < size; i++)
        s = s + tbl[i];
    return s;
}
```



Representing local variables

(Global variables and dynamically-allocated memory: unchanged throughout compilation.)

CompCert C	Local addressable variables
Clight C#minor	Local addressable variables + temporaries (local, non-a var)
Cminor CminorSel RTL	Stack frame + temporaries
LTL Linear	Stack frame + locations (machine registers or stack slots)
Mach Asm	Stack frame + machine registers

A look at abstract syntax trees

Asm

Note: flat structure, many instructions, some pseudo.

RTL

Note: few instructions, parameterization by operations/conditions/addressing modes of the target.

Clight

Note: expressions are pure, assignments and function calls are statements.

CompCert C

Note: complex expressions including assignments (several forms) and function calls.

Programs and compilation units

Common to all languages, a compilation unit is a name for a `main` entry point plus a list of (name, definition/declaration):

- Internal function definition (language-dependent)
- External function declaration
- Variable definition/declaration, including
  size and initialization data
  language-dependent information (e.g. the C type)

See module `AST`.

Module `Linking` defines a general framework for linking compilation units together, connecting identically-named definitions of one unit with external declarations of another unit [KKH+16].

Styles of semantics used (as a function of time)

	Clight . . . Cminor	RTL . . . Mach	Asm
1st gen.	big-step	"mixed-step" (b.s. for calls, (s.s. otherwise)	small-step
2nd gen. (+ divergence)	big-step (coinductive)	small-step (w/ call stacks)	small-step
3rd gen. (+ `goto` & tailcalls)	small-step (w/ continuations)	small-step (w/ call stacks)	small-step

CompCert's semantics, today

All languages use Labeled Transition Systems to describe their operational semantics in small-step style.

$$genv \vdash state_1 \xrightarrow{t} state_2$$

- *genv*: global environment, maps global identifiers (functions, variables) to memory addresses, and those addresses to the definitions/declarations. (See module `Globalenvs`.)
- $state_1$, $state_2$: execution states "before" and "after" the transition. Contain at least a memory state (see later) and a way to tell the current control point.
- t trace of observable events performed during the transition, e.g. system calls and volatile memory accesses.
  The trace is empty for internal transitions.

Generic shape of small-step semantics

```
Record semantics : Type := Semantics_gen {
  state: Type;
  genvtype: Type;
  step : genvtype -> state -> trace -> state -> Prop;
  initial_state: state -> Prop;
  final_state: state -> int -> Prop;
  globalenv: genvtype;
  symbolenv: Senv.t
}.
```

See module `Smallstep` for generic definitions and theorems about those semantics.

The Asm semantics

States: memory state $\times$ register state (register $\mapsto$ value).

One transition = execute the instruction pointed by register PC.

Broadly similar to the instruction set manuals of the target architecture and to processor formalizations such as [FM10, KBJD13].

More abstract in some respects:

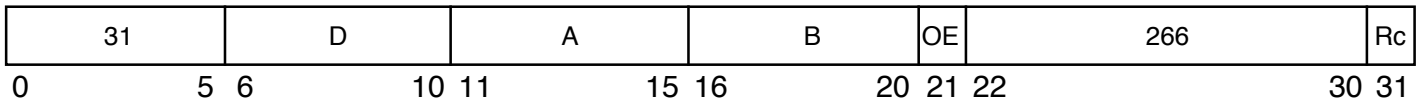
- The code is immutable and not stored in memory.
- No bit-level encoding of instructions.

addx

addx

Add (x'7C00 0214')

add	**r**D,**r**A,**r**B	(OE = 0 Rc = 0)
add.	**r**D,**r**A,**r**B	(OE = 0 Rc = 1)
addo	**r**D,**r**A,**r**B	(OE = 1 Rc = 0)
addo.	**r**D,**r**A,**r**B	(OE = 1 Rc = 1)



rD ← (**r**A) + (**r**B)

The sum (**r**A) + (**r**B) is placed into **r**D.

The **add** instruction is preferred for addition because it sets few status bits.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (If Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (see next bullet item.

- XER:

  Affected: SO, OV (If OE = 1)

  NOTE: For more information on condition codes see Section 2.1.3, "Condition Register," and Section 2.1.5, "XER Register."

The Asm semantics

For most instructions there are no observable events (trace is empty) and the state after is a partial function `Asm.exec_instr` of the state before.

Execution of builtin functions (e.g. volatile memory accesses) and external functions (e.g. system calls) can produce observables and use a language-independent semantics partially defined, partially axiomatized in module `Events`.

The RTL semantics

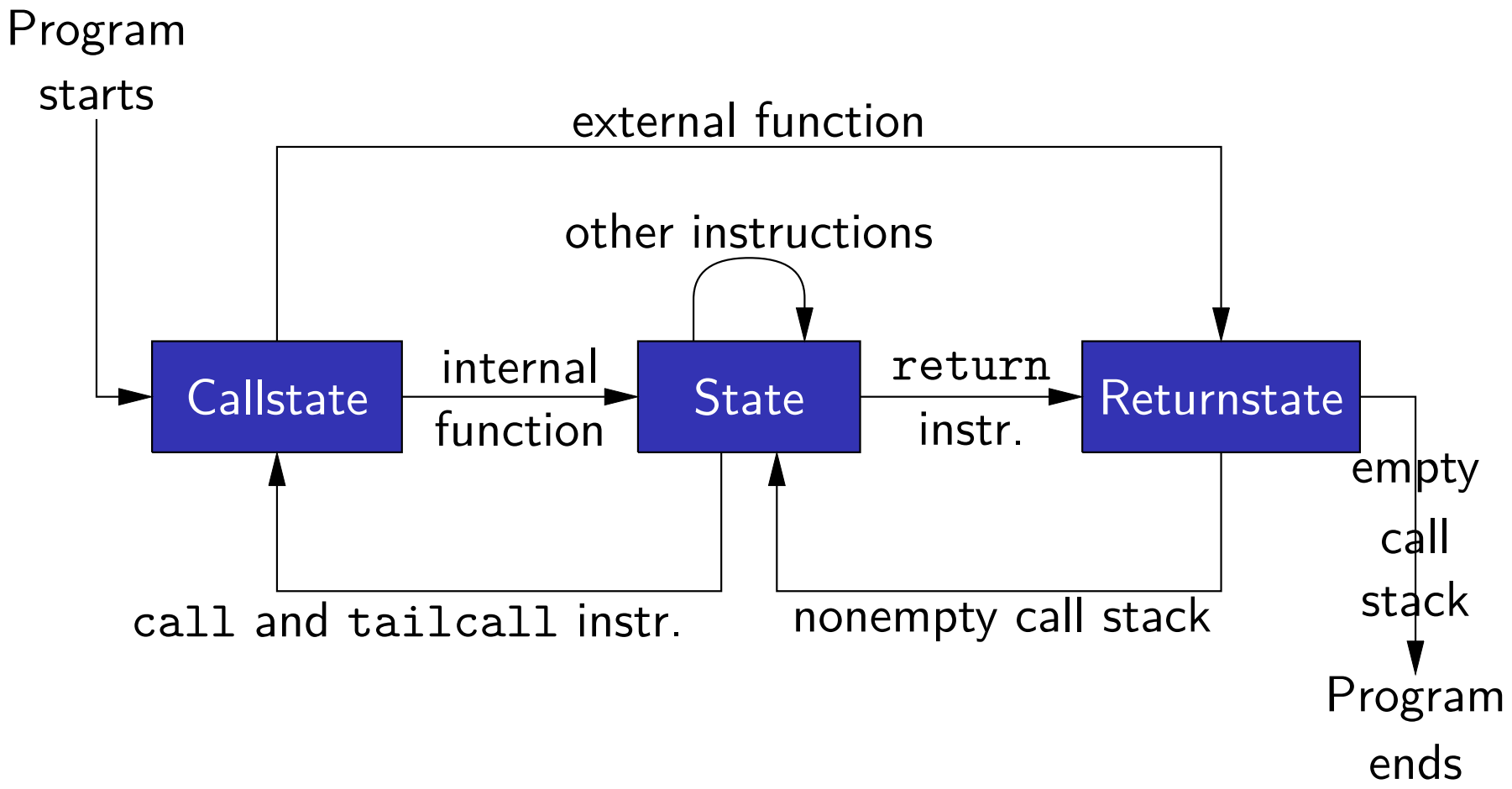
Execution states have more components and more structure:

- `f`: RTL function currently executing
- `sp`: pointer to its stack frame
- `pc`: node of the CFG to be executed next
- `rs`: maps temporaries (pseudoregisters) to their values
- `m`: memory state
- `stk`: call stack (list of pending function calls, with `f`, `sp`, `pc` and `rs`)

Most transitions execute the instruction at node `pc` of the CFG of the current function `f`.

Handling of function calls and returns

Function calls and function returns are slightly different and use special intermediate states `Callstate` and `Returnstate`.



The Clight semantics

Like in IMP, execution points are pairs (s, k) of a statement under focus s and a continuation k.

The continuation k also encodes the call stack (pending function calls).

Two local environments:

- e: name of addressable local variable $\mapsto$ memory address
- le: name of temporary $\mapsto$ current value

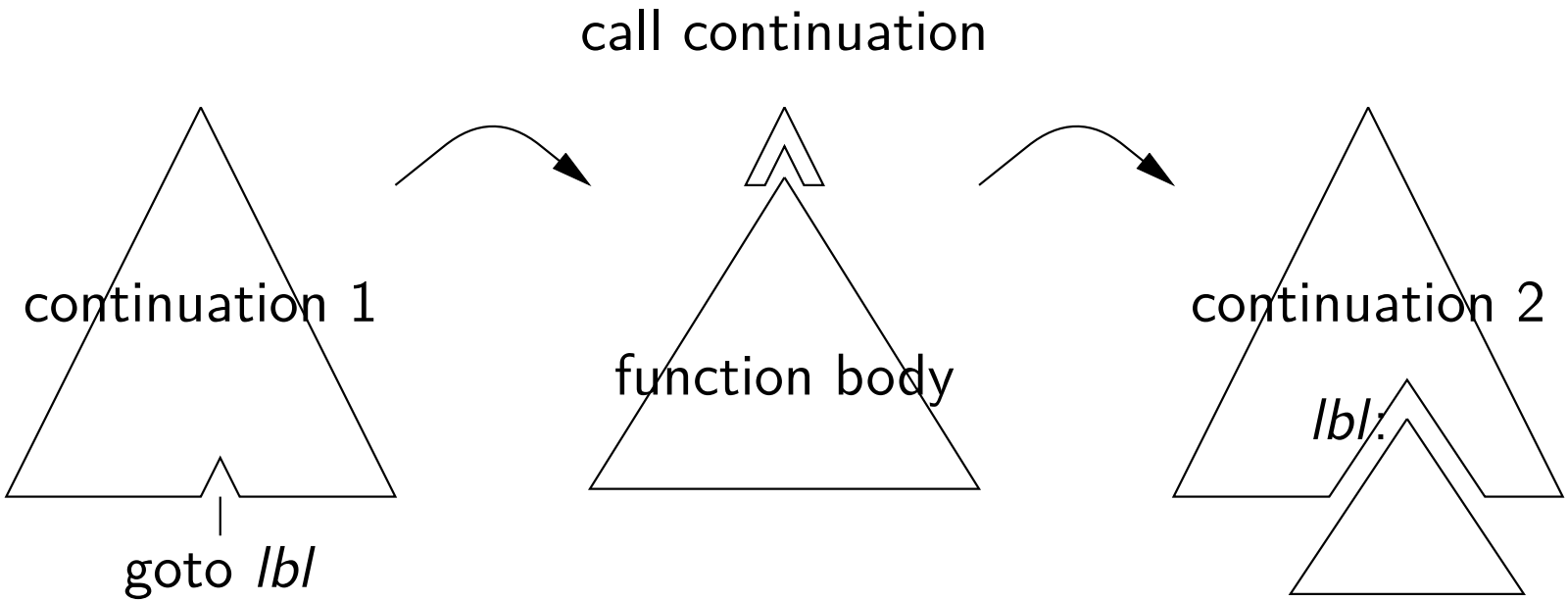
Like in IMP, expressions are evaluated in big-step style. Complications: overloading of operators; lvalues and rvalues.

The semantics is parameterized and instantiated twice to give:

- Clight1 semantics: function parameters are addressable local variables
- Clight2 semantics: function parameters are temporaries.

Handling `goto` by continuation surgery

A search function `find_label` that finds a statement labeled *lbl* while manufacturing the corresponding continuation:



Implements the transition `goto` $lbl/k_1 \rightarrow lbl : c/k_2$.

The CompCert C semantics

The semantics of statements is similar to that of Clight, using focusing and continuations.

Expressions can contain function calls
$\Rightarrow$ small-step evaluation of expressions

Several evaluation orders are allowed for expressions
$\Rightarrow$ non-determinism
$\Rightarrow$ reductions under a context (Felleisen-style).

A reference interpreter (`ccomp -interp`) is provided to explore the semantics.

Evaluation orders in C expressions

```
int a(void) { printf("a"); return 1; }
int b(void) { printf("b"); return 2; }
int c(void) { printf("c"); return 3; }
int main(void) { return a() + b() + c(); }
```

The subexpressions `a()` and `b()` and `c()` can be evaluated in any order that the compiler chooses.

$\Rightarrow$ any of the 6 permutations `abc`, `acb`, `bac`, `bca`, `cab`, `cba` is a valid output for this program.

(This is different from saying that $e_1 + e_2$ either evaluates e_1 fully then e_2, or e_2 then e_1, as in Scheme or OCaml.)

(The ISO C standards have a notion of *sequence points* that declare as illegal some expressions where different evaluation orders give different results. Function calls as in the above are not illegal.)

Reductions under contexts

(Felleisen and Hieb, 1989; Wright and Felleisen, 1992)

One or several head reduction rules, e.g. for the call-by-value λ-calculus,

$$(\lambda x.a)\ v \quad \xrightarrow{\epsilon} \quad a\{x \leftarrow v\} \qquad\qquad (\beta_v)$$

A single reduction under context rule:

$$\frac{a \xrightarrow{\epsilon} b}{\Gamma[a] \rightarrow \Gamma[b]}$$

A definition of valid reduction contexts Γ, often as a grammar:

Left-to-right evaluation:	Γ	$::=$	$[\,] \mid \Gamma\ b \mid v\ \Gamma$
Right-to-left evaluation:	Γ	$::=$	$[\,] \mid \Gamma\ v \mid a\ \Gamma$
Nondeterministic evaluation:	Γ	$::=$	$[\,] \mid \Gamma\ b \mid a\ \Gamma$

Nondeterminism and going-wrong behaviors

Depending on evaluation order, an expression can evaluate normally, or diverge, or go wrong:

```
(x = 0) + (x = 1) + (10 / x)

loop() + (10 / 0)
```

(Assuming `loop` is a function that never returns.)

Common sense and the ISO C standard say that an expression is undefined as soon as one evaluation order goes wrong. ("Demonic nondeterminism".)

The usual convention for head reduction rules is "no reduction if going wrong", as opposed to "reduce to `ERROR` if going wrong". To get demonic nondeterminism, we need one "reduce to `ERROR`" rule:

$$\frac{a \text{ is not a value and doesn't head-reduce}}{\Gamma[a] \to \texttt{ERROR}}$$

Quick tour of the Coq specification

Module `Csem`: the nondeterministic semantics.

- Head reductions: `callred` for function calls, `rred` for other r-value expressions, `lred` for l-value expressions.
- Contexts: functions C : `expr` $\rightarrow$ `expr` constrained by the predicate `context` k_1 k_2 C, where k_1 is the kind (l-value or r-value) of the hole and k_2 the kind of the result.
- Reduction of expressions: relation `estep` between states.
- Other transitions between states: relation `sstep`.

Module `Cstrategy`: the deterministic evaluation strategy implemented by CompCert. (Roughly: evaluate effectful subexpressions first, then finish.)

- Big-step evaluation of simple, pure subexpressions.
- Reductions under contexts for other expressions.
- Contexts (predicate `lcontext`) impose strict left-to-right evaluation.
- Reuses the `sstep` transitions of `Csem`.

Compiler verification in the large: CompCert

What is a memory model?

Semantics for imperative languages are expressed in terms of evolution of the state.

A memory model defines:

- What a state contains.
- The basic operations over states:
  at least `read` and `write`; also `alloc` and `free`.
- The properties of these operations
  $\Rightarrow$ reasoning over programs
  $\Rightarrow$ reasoning over program transformations.

Not treated here: issues of weak consistency in the presence of parallelism with shared memory (threads).

High-level memory models

For IMP:

a memory state = a mapping from variable names to values.

For Java, OCaml, etc:

a memory state = a mapping from abstract references to values.

(References are identifiers, like variable names; fresh references are created at dynamic allocation points.)

In both cases: the good variables properties:

$$\begin{aligned} \texttt{load}(\texttt{store}(m, r, v), r) &= v \\ \texttt{load}(\texttt{store}(m, r, v), r') &= \texttt{load}(m, r') \text{ if } r \neq r' \end{aligned}$$

High-level memory models

For (a well-behaved fragment of) C and C++, we can use references to designate objects and paths to designate their sub-objects (record fields, array elements) [RF95]:

Path:	$p ::= r$		reference to top-level object
		$\mid p.\mathrm{field}$	access to a field of a struct or union
		$\mid p[i]$	access to the i-th element of an array

The good variables properties become:

$$\begin{aligned} \texttt{load}(\texttt{store}(m, p, v), p) &= v \\ \texttt{load}(\texttt{store}(m, p, v), p') &= \texttt{load}(m, p') \text{ if } p \text{ and } p' \text{ are disjoint} \end{aligned}$$

Problems with low-level C programming idioms

High-level models have difficulties accounting for low-level C programming idioms such as byte-per-byte access to the representation of data:

```
void memcpy(void * dst, void * src, int n)
{
  int i;
  for (i = 0; i < n; i++) ((char *)dst)[i] = ((char *)src)[i];
}

double a[10], b[10];
memcpy(a, b, 10 * sizeof(double));
```

Problems with low-level C programming idioms

Another low-level idiom that works only with IEEE floating-point numbers and a little-endian processor:

```
double copysign(double x, double y)
{
  union { double d, uint64_t i } ux, uy;
  ux.d = x; uy.d = y;
  ux.i &= 0x7FFFFFFFFFFFFFFF;
  ux.i |= (uy.i & 0x8000000000000000);
  return ux.d;
}
```

(This is not standard-conformant C but widely used in the field.)

Low-level memory models

A machine-oriented view of memory as a flat array of bytes [TKN07]

- Memory state = mapping $int \mapsto byte$
- Pointer value = machine integer.

For each data type τ of the language, define two functions

$$\begin{aligned} \texttt{encode}_\tau &: \tau \to \texttt{list byte} \\ \texttt{decode}_\tau &: \texttt{list byte} \to \tau \end{aligned}$$

such that (at least)

$$\begin{aligned} \texttt{decode}_\tau(\texttt{encode}_\tau(x)) &= x \\ \texttt{length}(\texttt{encode}_\tau(x)) &= \texttt{sizeof}(\tau) \end{aligned}$$

Low-level memory models

Definition of memory operations:

$$\begin{aligned} \mathtt{load}_\tau(m, a) &= \mathtt{decode}_\tau(m[a], \ldots, m[a + \mathtt{sizeof}(\tau) - 1]) \\ \mathtt{store}_\tau(m, a, x) &= m[a, \ldots, a + \mathtt{sizeof}(\tau) - 1 \leftarrow \mathtt{encode}_\tau(x)] \end{aligned}$$

Weak "good variables" properties:

$$\begin{aligned} \mathtt{load}_\tau(\mathtt{store}_\tau(m, a, x), a) &= x \\ \mathtt{load}_{\tau'}(\mathtt{store}_\tau(m, a, x), a') &= \mathtt{load}_{\tau'}(m, a') \\ &\quad \text{if } a' + \mathtt{sizeof}(\tau') \leq a \text{ or } a + \mathtt{sizeof}(\tau) \leq a' \end{aligned}$$

Low-level memory models

Also need to account for:

- Alignment constraints, e.g. "for a 32-bit memory access, the address must be a multiple of 4".
  $\Rightarrow$ define the minimal alignment of a type $\texttt{alignof}(\tau)$
  $\Rightarrow$ enforce $(\texttt{alignof}(\tau) \mid a)$ for a τ access at a.

- Permissions / access rights. Some addresses do not correspond to working memory. Others correspond to readonly memory.
  $\Rightarrow$ memory states include a mapping address $\rightarrow$ permission

  The CompCert permissions:
  $\texttt{Empty} < \texttt{Nonempty} < \texttt{Readable} < \texttt{Writable} < \texttt{Freeable}$.

The good, the bad, and the ugly

Good: all low-level C programming idioms can be accounted for.

Bad: C programs can observe too many things about memory:

- Adjacency of data in memory:

```
int a[2], b[2];   a[3] = 42;  return b[1];  // returns 42
```

- Relative order of data in memory:

```
int a, b;  return &a < &b;
```

- Absolute addresses:

```
int a;  return &a == (int *) 4;
```

The good, the bad, and the ugly

Ugly: program transformations in general and compilers in particular do not preserve these extra observations:

- Insertion of padding ⇒ invalidates adjacency properties

```
int a[2], b[2];   a[3] = 42;  return b[1];
```

-->

```
int a[2], padding[2], b[2];  a[3] = 42; return b[1];
```

- Permutation of allocations ⇒ invalidates adjacency and relative order

```
int a, b;  return &a < &b;    // returns true
```

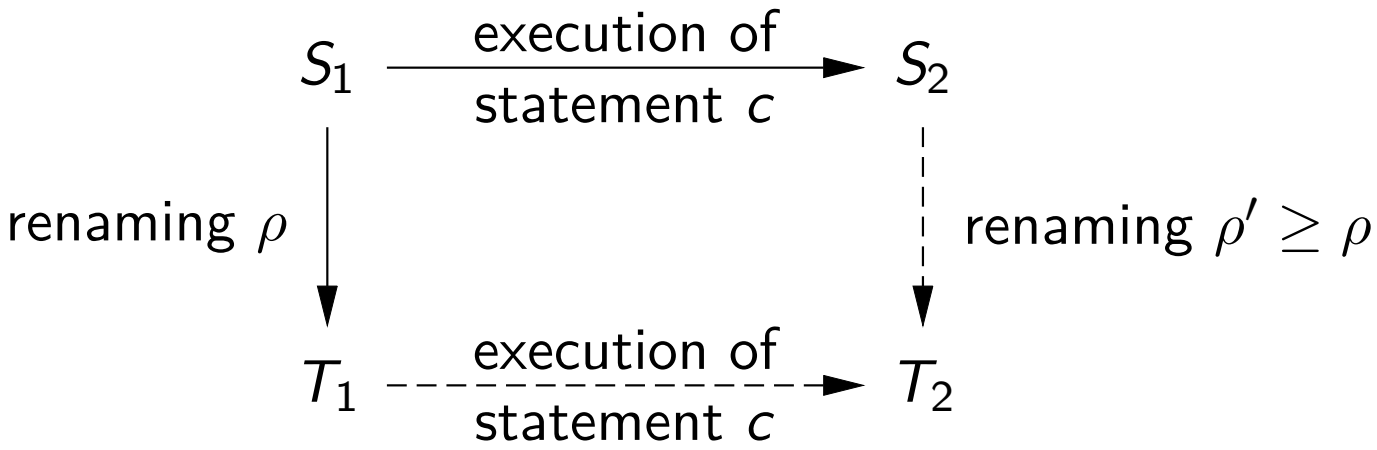
-->

```
int b, a;  return &a < &b;    // returns false
```

- Late (link-time or run-time) placement of data
  ⇒ invalidates absolute addresses.

A sanity condition on semantics

Semantics for imperative languages should be invariant under renaming of references/addresses



High-level models: OK. Low-level models: no.

The CompCert memory model [LABS14]

A "medium-level" memory model:

- States are composed of memory blocks (similar to top-level objects in the abstract C/C++ view), disjoint by construction, designated by abstract references b.
- The contents of a memory block are accessed by byte offsets, with alignment constraints and per-byte permissions, just like in a low-level model.
- Pointer values are pairs (b, δ) of a block identifier b and a byte offset δ (unsigned 32 or 64-bit integer, depending on the target processor).

Used for all the languages of CompCert, from C to Asm.

The CompCert value model

(Module `Values`)

Values are either pointers or numbers (different kinds), plus a special `Vundef` value meaning "an unknown bit pattern".

```
Inductive val: Type :=
  | Vundef: val
  | Vint: int -> val
  | Vlong: int64 -> val
  | Vfloat: float -> val
  | Vsingle: float32 -> val
  | Vptr: block -> ptrofs -> val.
```

In-memory encoding of values

(Module `Memdata`)

```
Inductive memval: Type :=
  | Undef: memval
  | Byte: byte -> memval
  | Fragment: val -> quantity -> nat -> memval.
```

The concrete encoding:

As a list of bytes (8-bit integers).

Used for numbers (integers and FP).

Exposes bit-level representation (2's complement, IEEE754 FP) and processor endianness.

The abstract encoding:

As a list of symbolic fragments "the n-th byte of value v".

Used for pointers.

Main properties of the model

(Module `Memory`; see also [LABS14])

- Weak good variables properties:

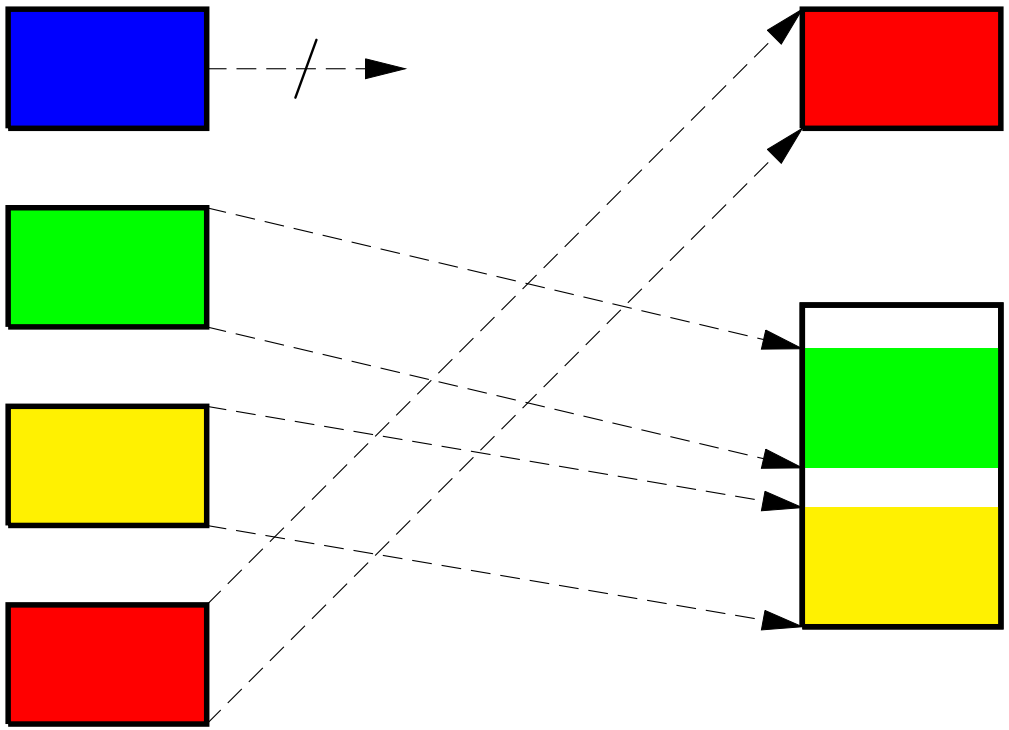
$$\begin{aligned} \texttt{load}_\tau(\texttt{store}_\tau(m,p,x),p) &= x \\ \texttt{load}_{\tau'}(\texttt{store}_\tau(m,p,x),p') &= \texttt{load}_{\tau'}(m,p') \end{aligned}$$

  if p, p' point to different blocks
  or to disjoint areas of the same block

- Pointer values cannot be forged: the only way for a `load` to return a pointer value is for this pointer value to have been stored earlier at the same address.
- Compatibility with renamings of block identifiers.
- Compatibility with generalized renamings: memory injections.

Memory injections

$j : \texttt{block} \to \texttt{option}(\texttt{block} \times Z)$
$j(b) = \texttt{None}$: block b disappears (e.g. contents pulled into a register)
$j(b) = \texttt{Some}(b', \delta)$: block b becomes a sub-block of b' at offset δ.



Memory injections

A memory injection $j : \mathtt{block} \to \mathtt{option}(\mathtt{block} \times Z)$ induces a correspondence $j \vdash v \approx v'$ between values:

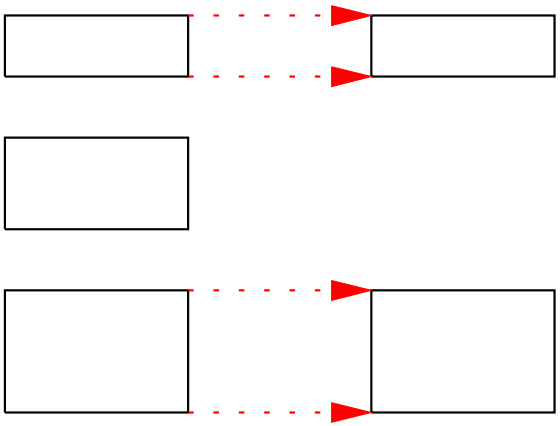
$$j \vdash \mathtt{Vundef} \approx v \qquad\qquad j \vdash \mathtt{Vint}(i) \approx \mathtt{Vint}(i)$$

$$\frac{j(b) = \mathtt{Some}(b', \delta) \qquad o' = o + \delta \pmod{2^{wordsize}}}{j \vdash \mathtt{Vptr}(b, o) \approx \mathtt{Vptr}(b', o')}$$

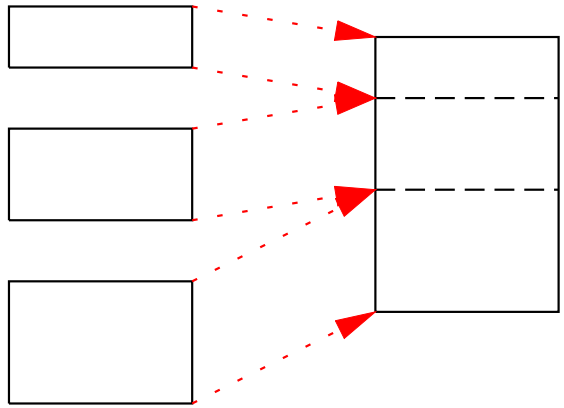
It also induces a correspondence between memory states $j \vdash M \approx M'$:

- No overlap between block images
- Block contents match.

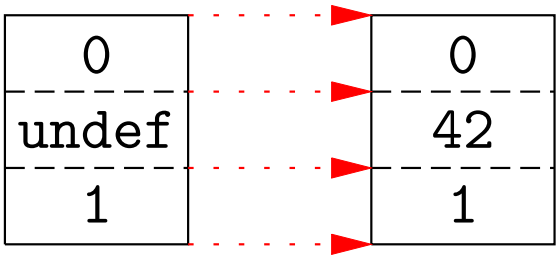
Uses of memory injections in CompCert's proofs



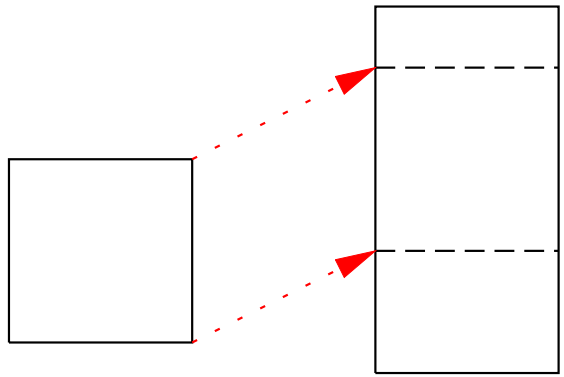
Pulling data out of memory
(Clight1 to Clight2)



Merging blocks
(C#minor to Cminor; inlining)



Refinement of stored values
(many passes)



Extending stack frames
(spilling)

Compiler verification in the large: CompCert

Semantic preservation in early CompCert

The first versions of CompCert were similar to big-step IMP, in that:

- The semantics only let us observe termination with a given exit status (return value of the `main` function).
- The semantic preservation result proved was of the form

  If the source Cminor program terminates with status N,
  the generated assembly code also terminates with status N.

- The assembly language had deterministic semantics, hence no other behavior of the compiled code was possible.

The need for stronger preservation properties

Most real programs produce more output than just an integer exit status, e.g. write results to files.

Many real programs are interactive and input data in addition to outputting it.

Many real programs are intended to run forever; termination is an error. This is the case of servers but also of control-command codes.

every δt seconds do
    acquire *current-state* from sensors
    compute *action* = *control-law*(*desired-state* − *current-state*)
    send *action* to actuators
done

Observing more behaviors

1- Associate observable events to certain actions of the program:

- Calls to the OS or to a standard I/O library,
  e.g. `putchar`, `getchar`.
- Reads from and writes to `volatile` global variables,
  because these variables can correspond to hardware I/O devices.

2- Describe diverging (non-terminating) executions in addition to terminating executions.

Labeled Transition Systems

$$genv \vdash state_1 \xrightarrow{t} state_2$$

Classically, labels t are either τ (for internal, non-observable computations), $c!v$ for output operations, and $c?v$ for input operations.

In CompCert, t is a list of observable events (see module `Events`). The list is empty for internal computations.

Events come in several kinds and can combine output and input, e.g. "call system function F with argument *out* and get result *in*".

From transitions to whole-program behaviors

Normal termination with trace $a_1 \ldots a_k$:

$$\texttt{initial} \ni s \xrightarrow{\tau} s_1 \xrightarrow{a_1} s_2 \xrightarrow{\tau} \cdots \xrightarrow{a_k} s_n \in \text{final}$$

"Going wrong" with trace $a_1 \ldots a_k$:

$$\texttt{initial} \ni s \xrightarrow{\tau} s_1 \xrightarrow{a_1} s_2 \xrightarrow{\tau} \cdots \xrightarrow{a_k} s_n \in \text{error}$$

Reactive divergence with infinite trace $a_1 \ldots a_k \ldots$:

$$\texttt{initial} \ni s \xrightarrow{\tau} \cdots \xrightarrow{a_i} \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} \cdots \xrightarrow{a_j} \xrightarrow{\tau}\xrightarrow{\tau} \cdots$$

Silent divergence with trace $a_1 \ldots a_k$:

$$\texttt{initial} \ni s \xrightarrow{\tau} \cdots \xrightarrow{a_k} s_n \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} \cdots$$

Simulations

Using these notions of whole-program behaviors, we can reuse the framework from part 2 of this lecture to define forward simulation and backward simulation between an original program P_1 and a compiled program P_2:

Backward simulation with improvement:

> Any observable behavior b_2 of P_2 is identical to or improves upon a possible behavior b_1 of P_1.

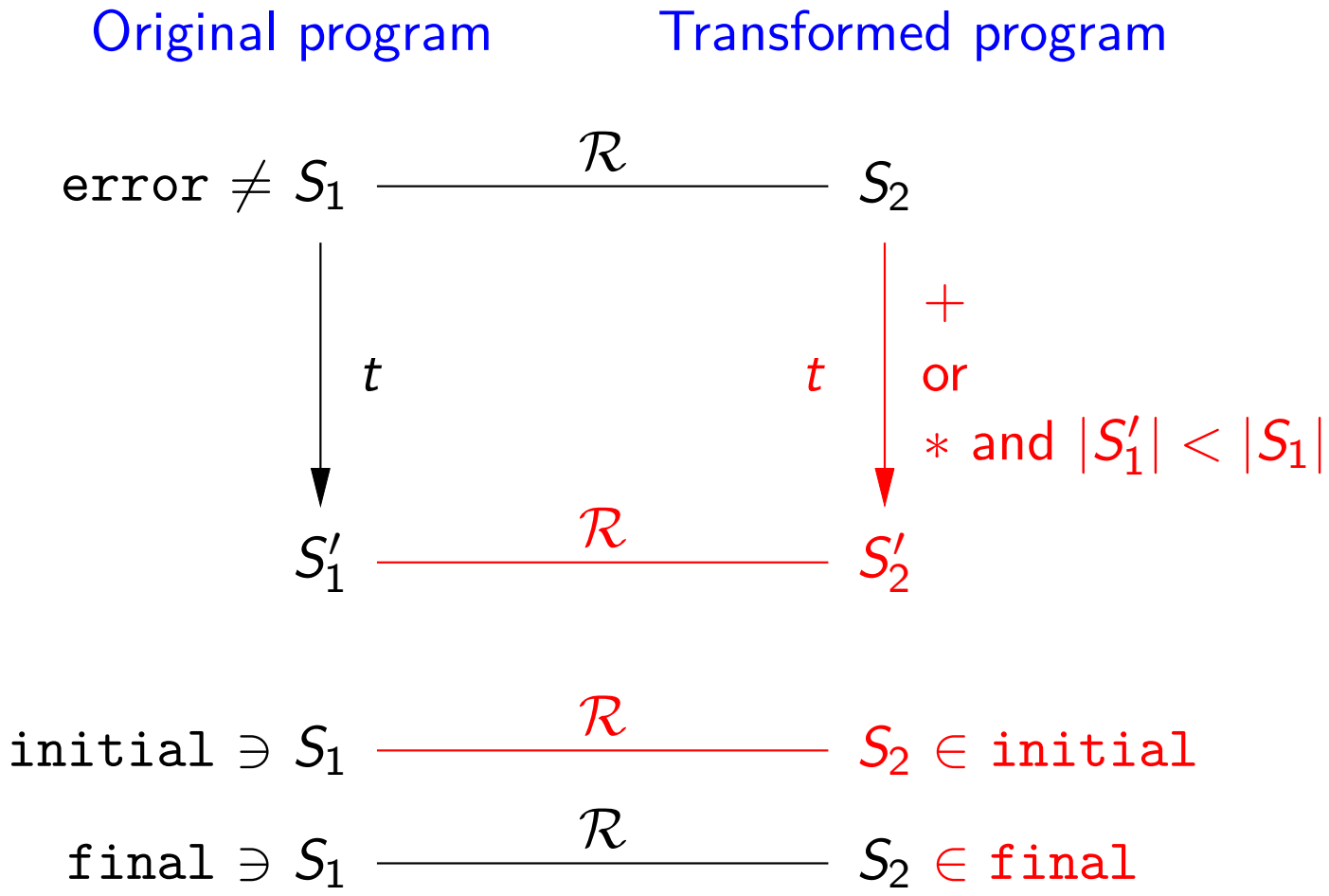
Forward simulation with improvement:

> Any possible behavior b_1 of P_1 is identical to or is improved by a possible behavior b_2 of P_2.

However, these simulations can be defined directly in terms of the labeled transition systems for P_1 and P_2, as commutative diagrams.

Forward simulation diagram

$\mathcal{R}$ is a relation between execution states of the two programs.
Black = hypothesis; red = conclusion.



Early CompCert semantic preservation proofs

(See [Ler09b])

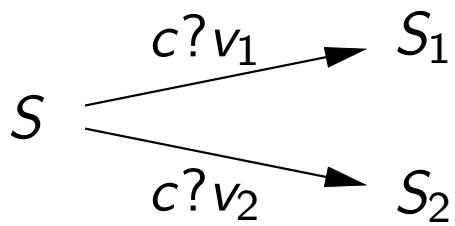
1. Show safe forward simulation for every compilation pass.
   (Using simulation diagrams and small-step semantics, mostly.)
2. Show safe forward simulation for the whole compiler
   by composing the per-pass simulations.
3. Argue that the target language (Asm) is deterministic.
4. Conclude that we have safe backward simulation.

Issues with external nondeterminism

Input operations ("$c?v$" events in the trace) receive values v that are not determined by the program, but instead chosen by the environment.

For example, reading from a global volatile variable returns any value of its type.

Hence the Asm semantics is not deterministic:



Workaround: determinize the environment by making input values partial functions of an initial environment state and of the previous outputs of the program.

Issues with internal nondeterminism

A compiler pass that reduces internal nondeterminism
(e.g. by selecting one evaluation order among several permitted orders)
cannot enjoy a forward simulation property.

(Forward simulation: every behavior of the source is matched by a behavior of the compiled.)

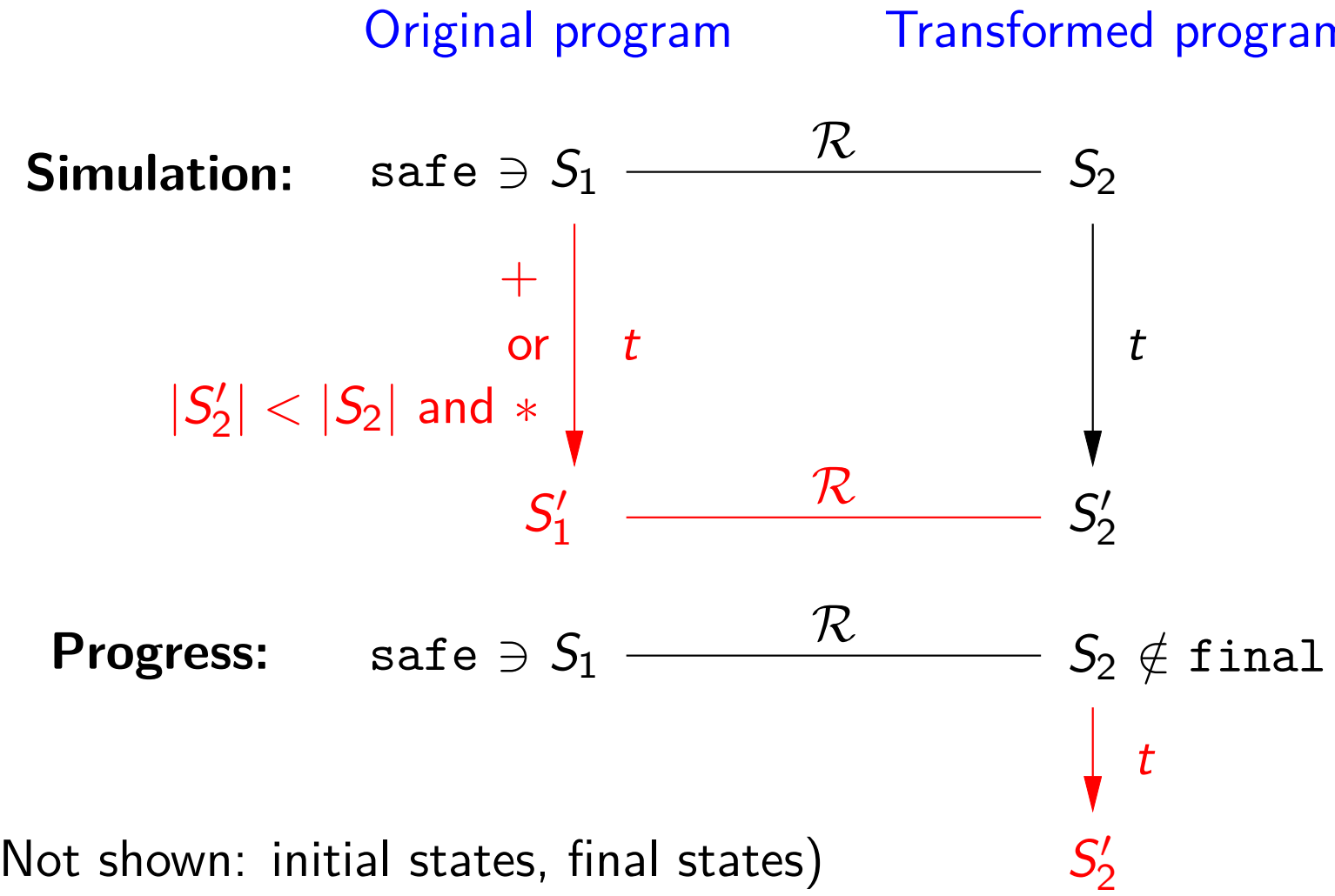
Early CompCert: problem not apparent because the C→Clight pass that chooses an evaluation order was not formally verified yet.

Later CompCert: use a backward simulation diagram to relate

- CompCert C with multiple evaluation orders
- CompCert C with the fixed order from module `Cstrategy`

then a forward simulation from the latter to Clight.

Backward simulation diagrams



(Not shown: initial states, final states)

$S \in \texttt{safe}$ means $\neg\ S \xrightarrow[\tau]{*} \texttt{error}$ (cannot crash silently).

From determinism to determinacy and receptiveness

In their CompCertTSO work [SVN+11], Sevcík, Vafeiadis, Zappa Nardelli, Jagannathan and Sewell show how to get rid of the "deterministic external world" hypothesis, by reasoning purely over diagrams and forgetting about whole-program behaviors.

Theorem (Sevcík et al)

Assume a forward simulation diagram from LTS L_1 to LTS L_2.
If L_1 is receptive and L_2 is determinate, there exists a backward simulation diagram from L_1 to L_2.

Proof: highly nontrivial; see module `Smallstep`.

Determinacy and receptiveness

Two labels are compatible $\ell_1 \asymp \ell_2$ if they differ only by input values. (I.e. $\ell_1 = \ell_2 = \tau$ or $\ell_1 = \ell_2 = c!v$ or $\ell_1 = c?v_1, \ell_2 = c?v_2$.)

A language is determinate if:

- $s \xrightarrow{\ell_1} s_1$ and $s \xrightarrow{\ell_2} s_2$ imply $\ell_1 \asymp \ell_2$.
- $s \xrightarrow{\ell} s_1$ and $s \xrightarrow{\ell} s_2$ imply $s_1 = s_2$.

In other words: the only nondeterminism comes from the input values in labels. This is the case for Asm and all other CompCert language except CompCert C (because of its internal nondeterminism).

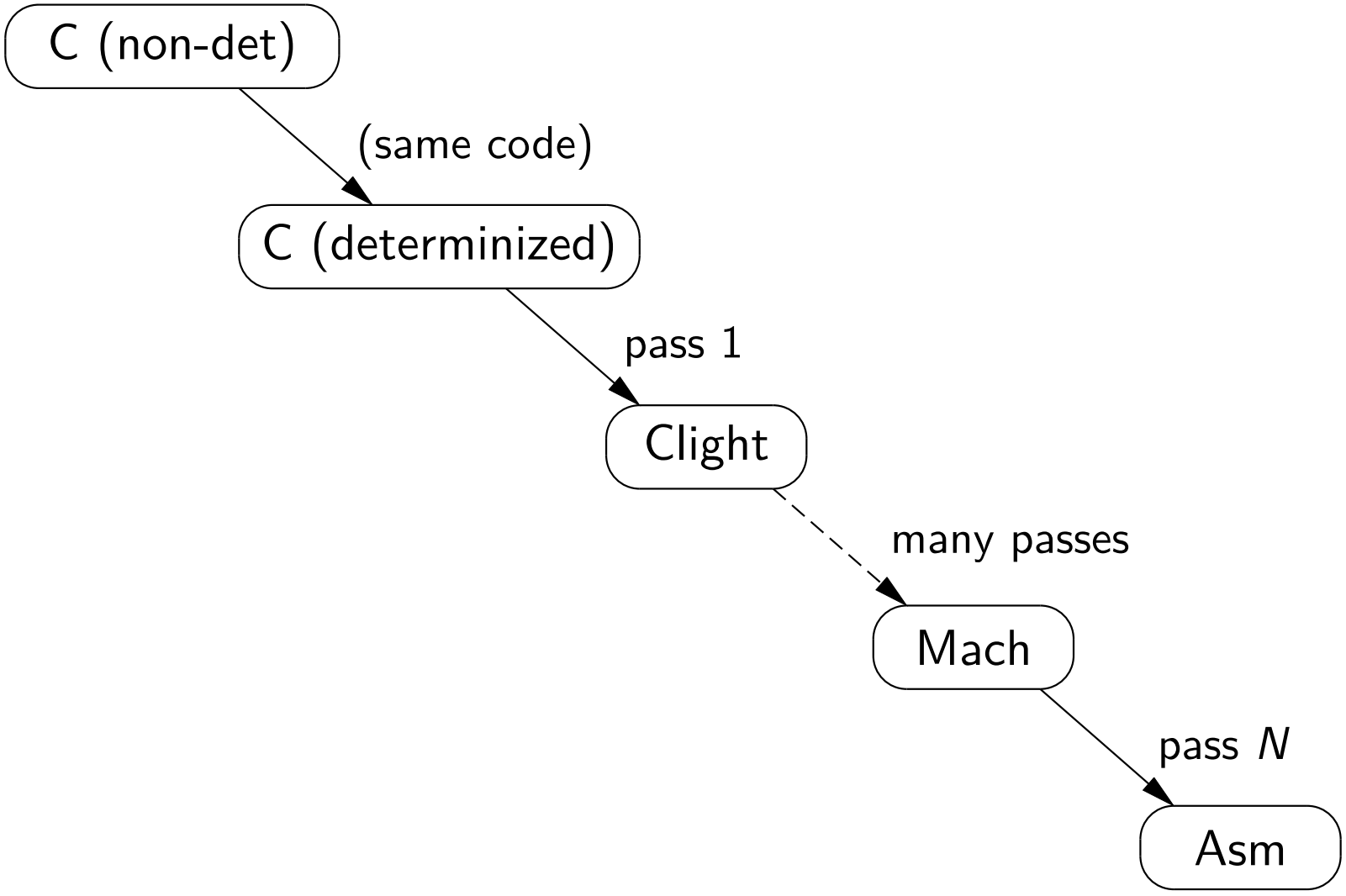
A language is receptive if:

- $s \xrightarrow{\ell_1} s_1$ and $\ell_1 \asymp \ell_2$ implies $\exists s_2, s \xrightarrow{\ell_2} s_2$.

In other words: a transition with an input is possible regardless of the value of the input. This is the case for all CompCert languages.

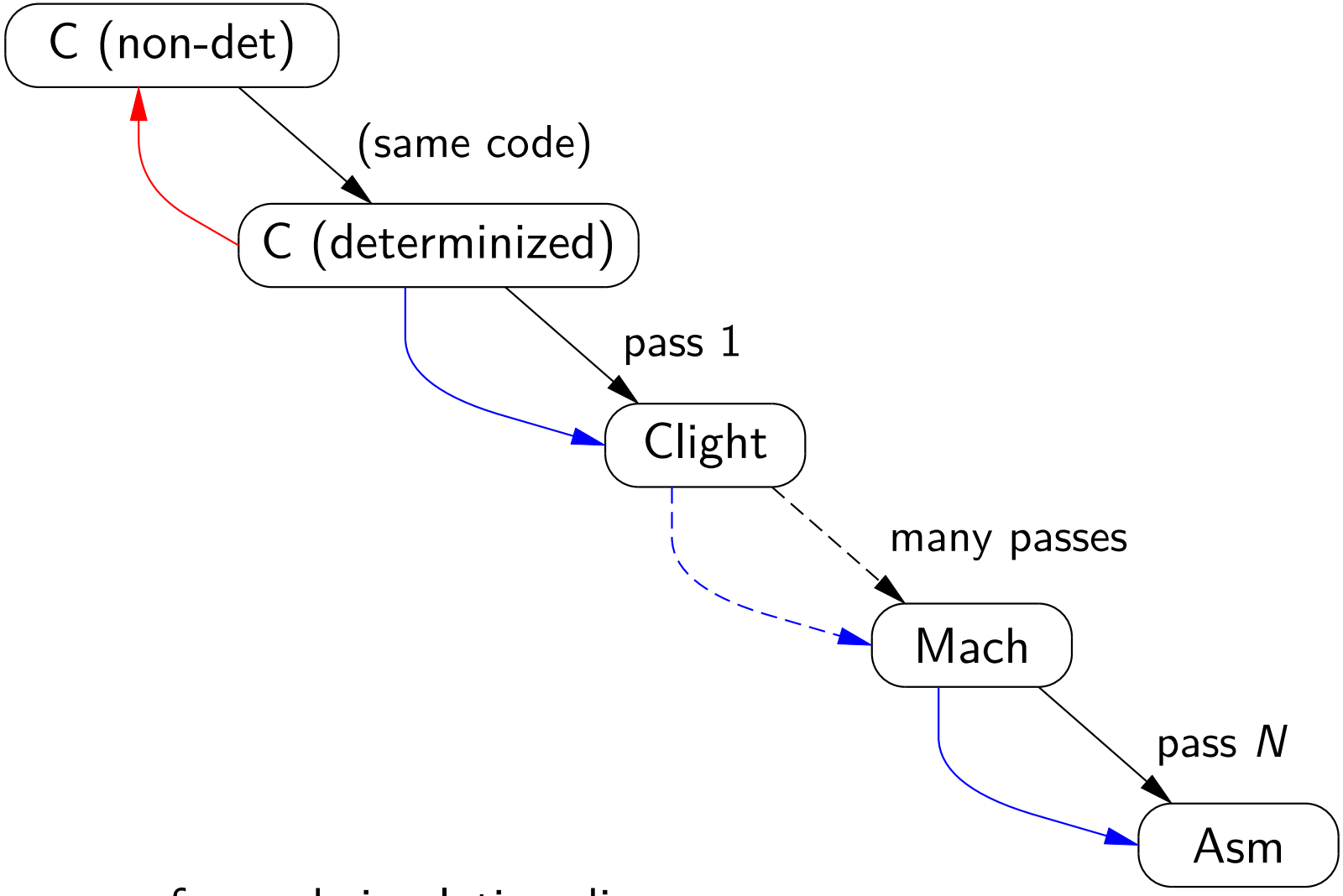
Putting it all together

(Module `Compiler`)



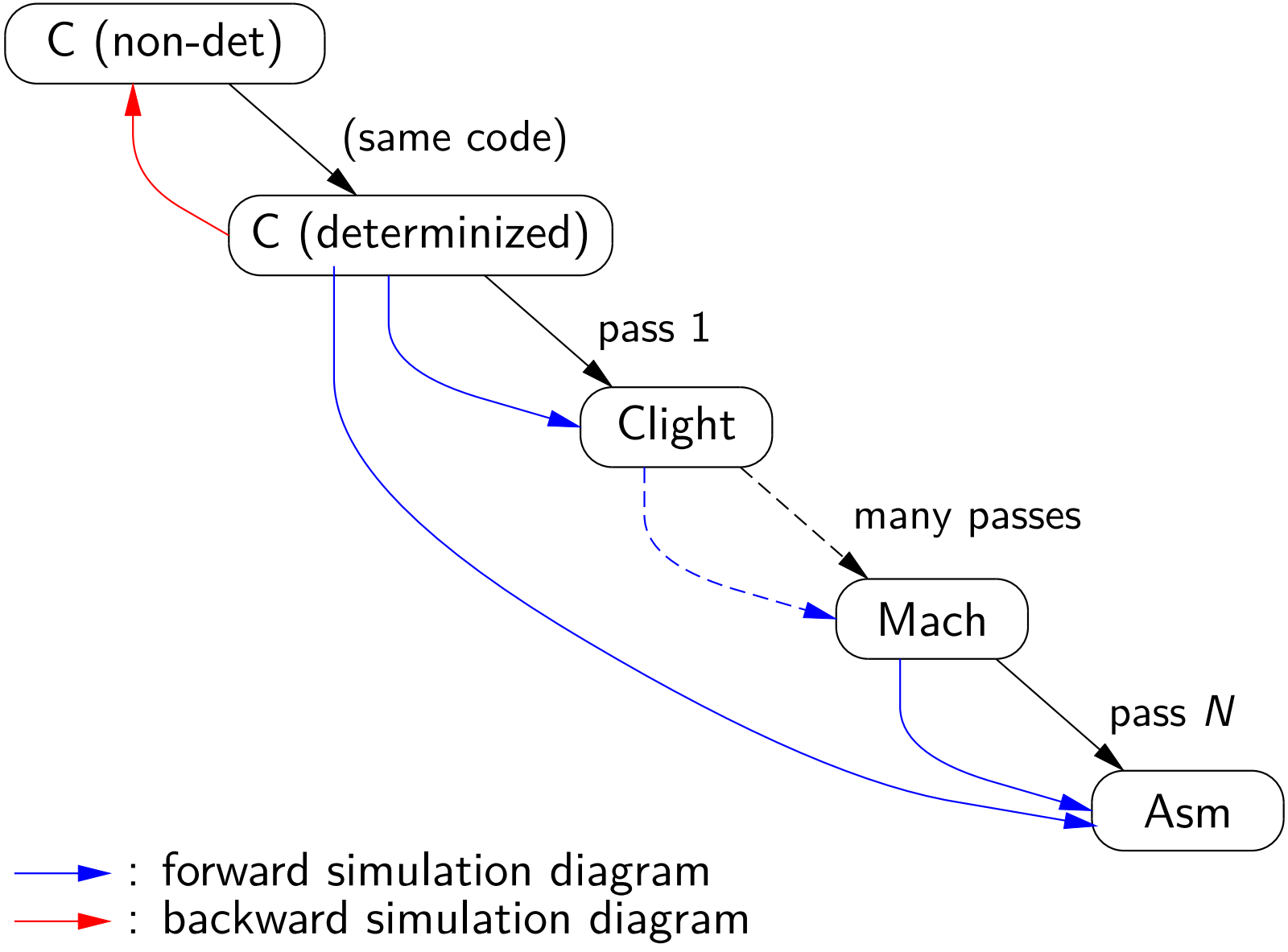
Putting it all together

(Module `Compiler`)



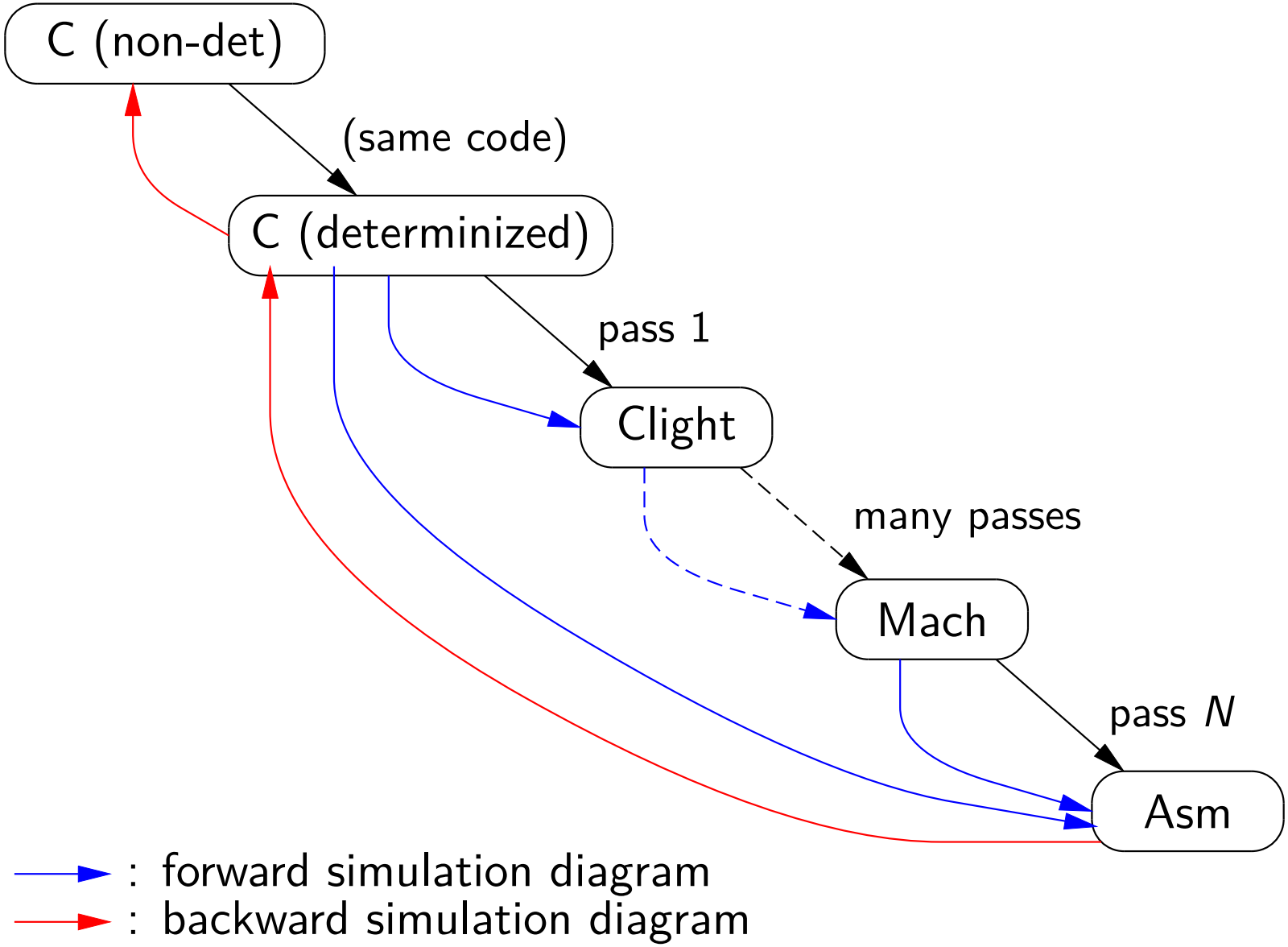
Putting it all together

(Module `Compiler`)



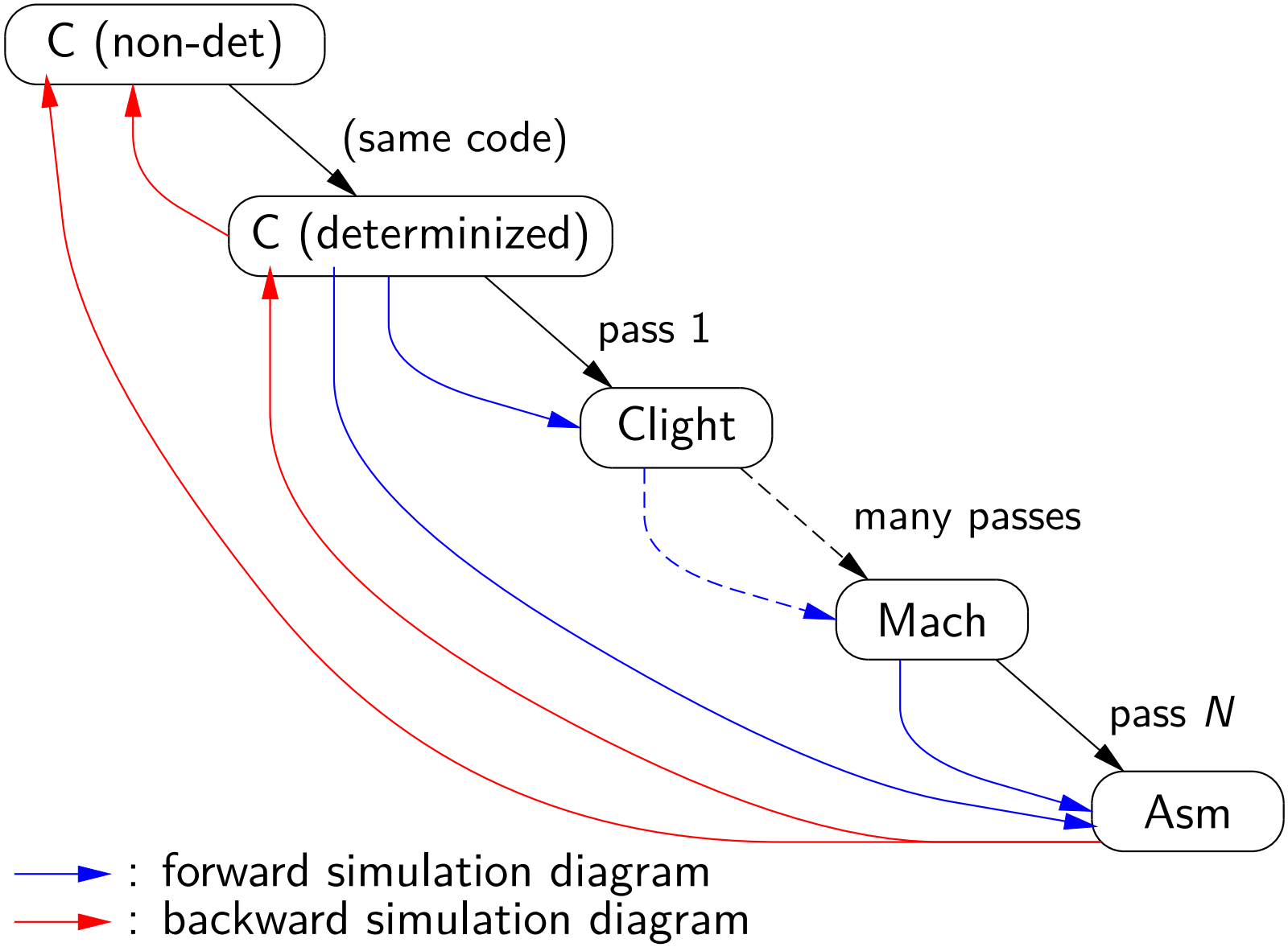
Putting it all together

(Module `Compiler`)



Putting it all together

(Module `Compiler`)



Compiler verification in the large: CompCert

On tool verification

CompCert is still an ongoing project, but it demonstrates that the formal verification of realistic compilers is feasible (within the limitations of today's proof assistants).

Some related tool verification projects:

- CakeML (compiler for core ML)
- Velus (Lustre→C code generator)
- Verasco (C static analyzer based on abstract interpretation)
- The Verified Software Toolchain at Princeton

Future directions for CompCert

- Increase confidence further by reducing the number of unverified components.
- Re-engineer and model differently to be more easily extensible, e.g. to new target processors and ABIs.
- Add more optimizations, esp. loop optimizations.
- Extend the verification "up" to connect with more abstract semantics of C such as Krebbers's [Kre15].
- Extend the verification "down" towards instruction set architectures (machine language, e.g. [FM10, KBJD13]) and hardware implementations (e.g. the CLI stack [Moo96]).
- Support shared-memory concurrency.
  Early attempt: CompCertTSO [SVN+11]. Ongoing work at Princeton.

On trusting the specifications

A difficult problem, faced by all kinds of formal verifications.

A small simplification in the specifications is worth a large increase in proof effort.

In the DeepSpec project: exercise specifications by connecting them to multiple verifications.

Executable specifications (e.g. reference interpreters) can also help:

- For testing the specifications.
- To discuss with standard committees.

On mechanized semantics

A need shared by many verification efforts, not just verified compilers.

A difficult task, especially for realistic programming languages (i.e. Java and the JVM; C; Javascript; C++).

Machine assistance is a necessity to scale up to realistic programming languages.

The sensitivity is disturbingly high: adding one language feature can deeply impact the whole semantics.

The unreasonable effectiveness of Labeled Transition Systems (despite looking more like abstract machines than high-level specs).

Part VII

Appendix

References I

Anthony C. J. Fox and Magnus O. Myreen.
A trustworthy monadic formalization of the ARMv7 instruction set architecture.
In *ITP 2010: Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 243–258. Springer, 2010.
https://www.cl.cam.ac.uk/~mom22/itp10-armv7.pdf.

Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand.
Coq: the world's best macro assembler?
In *PPDP'13: Principles and Practice of Declarative Programming*, pages 13–24. ACM, 2013.
https://www.microsoft.com/en-us/research/publication/coq-worlds-best-macro-assembler/.

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis.
Lightweight verification of separate compilation.
In *POPL 2016: Symposium on Principles of Programming Languages*, pages 178–190. ACM, 2016.
https://people.mpi-sws.org/~viktor/papers/sepcompcert.pdf.

Robbert Krebbers.
The C standard formalized in Coq.
PhD thesis, Radboud University Nijmegen, 2015.
http://robbertkrebbers.nl/thesis.html.

References II

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart.
The CompCert memory model.
In Andrew W. Appel, editor, *Program Logics for Certified Compilers*, pages 237–271.
Cambridge University Press, March 2014.
Preliminary version as Inria report RR-7987, `https://hal.inria.fr/hal-00703441`.

Xavier Leroy.
Formal verification of a realistic compiler.
Communications of the ACM, 52(7):107–115, 2009.
`http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf`.

Xavier Leroy.
A formally verified compiler back-end.
Journal of Automated Reasoning, 43(4):363–446, 2009.
`http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf`.

J. S. Moore.
Piton: a mechanically verified assembly-language.
Kluwer, 1996.

Michael Norrish.
C formalized in HOL.
PhD thesis, University of Cambridge, 1998.
`https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-453.pdf`.

References III

Jonathan G. Rossie and Daniel P. Friedman.
An algebraic semantics of subobjects.
In *OOPSLA'95: Object-Oriented Programming, Systems, Languages, and Applications*, pages 187–199. ACM, 1995.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.57.

Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell.
Relaxed-memory concurrency and verified compilation.
In *POPL 2011: Symposium on Principles of Programming Languages*, pages 43–54. ACM, 2011.
http://www.cl.cam.ac.uk/~pes20/CompCertTSO/doc/paper.pdf.

Harvey Tuch, Gerwin Klein, and Michael Norrish.
Types, bytes, and separation logic.
In *POPL 2007: Symposium on Principles of Programming Languages*, pages 97–108. ACM, 2007.
https://ts.data61.csiro.au/publications/nicta_full_text/134.pdf.