Mechanized semantics for compiler verification

or: CompCert's tortuous path through semantics land

Xavier Leroy

INRIA Paris-Rocquencourt

APLAS & CPP, 2012-12-13

The CompCert project

(X. Leroy, S. Blazy, et al)

Goal: develop and prove correct a realistic compiler

- from (a very large subset of) the C language
- to assembly code for popular processors (PowerPC, ARM, x86)
- producing reasonably efficient code ($\rightarrow$ some optimizations).

Verifying a compiler

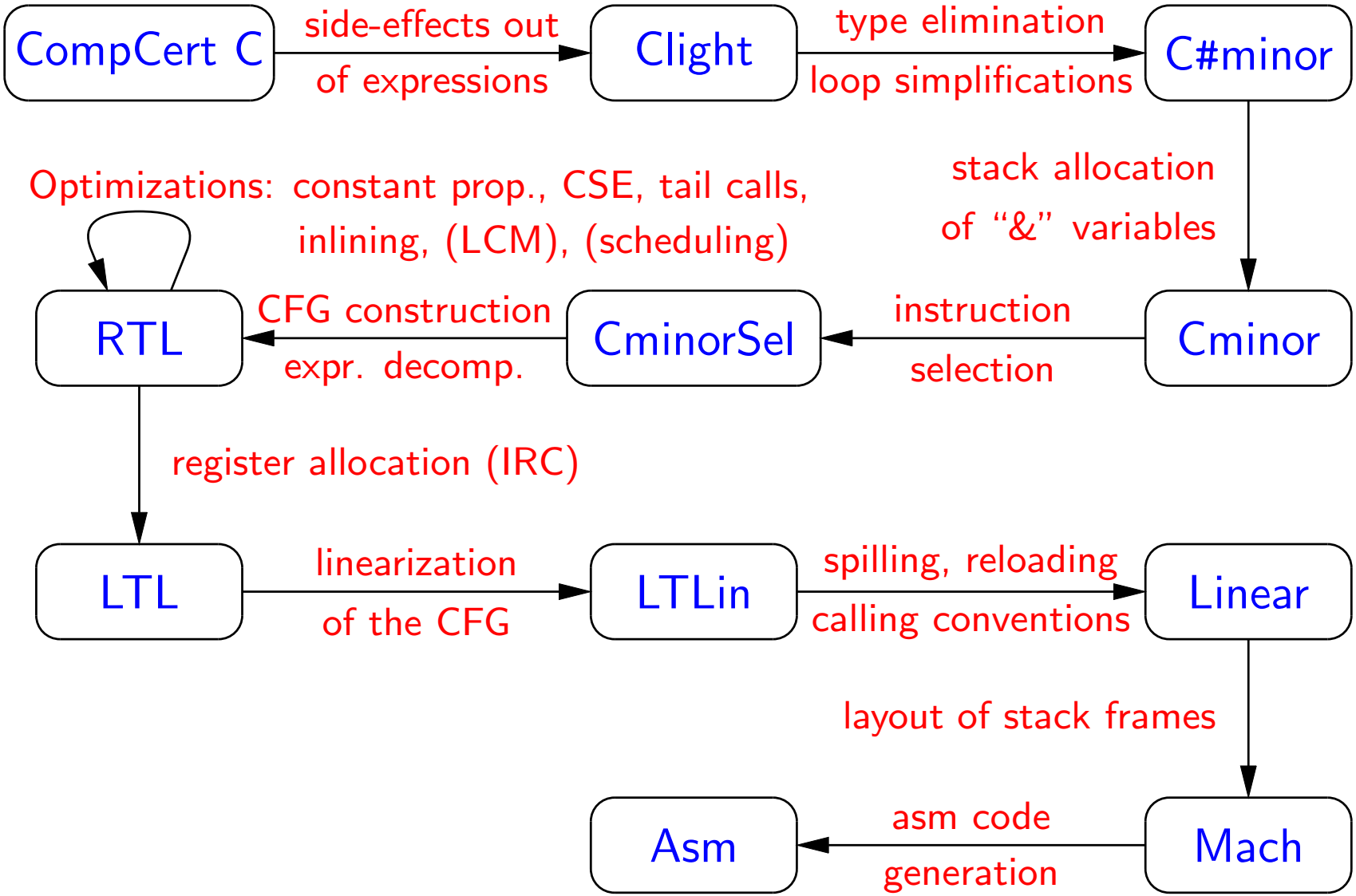
Using Coq, we prove the following semantic preservation property:

> *The observable behavior of the generated assembly code is one of the possible behaviors of the source program according to the C semantics, or improves on one of these possible behaviors.*

Behaviors =
termination / divergence / undefined ("going wrong")
+ trace of I/O operations performed.

"Improving" = replacing undefined behaviors by more defined behaviors.

The formally verified part of CompCert



A breakdown of the verification effort

Components	Size (lines)	Evolutions during the project
Compilation algorithms (the compilation passes)	8000	Minor extensions with new features Adding new compilation passes
Syntax of languages	2000	Minor extensions with new features
Semantics of languages	5000	Major changes of semantic styles At least one full rewrite per language
Proofs of semantic preservation	40000	Minor extensions for new features Major changes to accommodate changes in semantics

CompCert semantics as a function of time

Languages	V1	V2	V3	V4
(1) Clight ⋮ Cminor	Big-step	Big-step w/traces	Coinductive big-step	Small-step w/continuations
(2) RTL ⋮ Mach	Mixed-step	LTS	LTS	LTS
Asm	Trans. Syst.	LTS	LTS	LTS

(1) High-level languages with structured control.

(2) Intermediate languages with unstructured control (CFG).

This talk

A "post-mortem" analysis of the evolutions of CompCert,
trying to better understand:

- Why so much back-and-forth on the semantics of the source, intermediate and target CompCert languages?
- What is so difficult in the engineering of formal semantics and their mechanization?
- What are good styles of mechanized semantics?
  (as a function of what needs to be proved using them).

What is a formal semantics?

Associates a mathematically-precise meaning to a piece of syntax.

Example

Syntax: `x = x + 1`

Semantics:

ML:	$\lambda e.\ \texttt{false}$
Pascal-like:	$\lambda s.\ s[\texttt{x} \leftarrow s(\texttt{x}) + 1]$
C-like:	$\lambda e.\lambda s.\ s[e(\texttt{x}) \leftarrow (s(e(\texttt{x})) + 1) \bmod 2^{32}]$

Many different styles of semantics

Useful dividing lines:

- Operational vs. Denotational
- Syntactic vs. Higher Mathematics
- Compositional vs. Global

This talk: operational styles.

How to choose a style?

"It looks good!"

"It handles the language features I need."

"It lets me prove the properties I'm interested in."

"It fits my proof assistant."

First investigations: big-step vs. small-step on a toy language (IMP)

The IMP language

The prototypical imperative language with structured control.

Expressions:

$$a ::= cst \mid x \mid a_1 + a_2 \mid a_1 \leq a_2 \mid \ldots$$

Commands:

$$c ::= \texttt{skip} \mid x := a \mid c_1; c_2$$
$$\mid \texttt{if } a \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } a \texttt{ do } c$$

Reduction (small-step) semantics

Rewriting (command, state) configurations: $c/s \to c'/s'$

$$
\begin{array}{rcl}
(x := a)/s & \to & \texttt{skip}/s[x \leftarrow v] \qquad \text{if } s \vdash a \Rightarrow v \\
(\texttt{skip}; c)/s & \to & c/s \\
(\texttt{if } a \texttt{ then } c_1 \texttt{ else } c_2)/s & \to & \begin{cases} c_1/s & \text{if } s \vdash a \Rightarrow \texttt{true} \\ c_2/s & \text{if } s \vdash a \Rightarrow \texttt{false} \end{cases} \\
(\texttt{while } a \texttt{ do } c)/s & \to & \texttt{if } a \texttt{ then } (c; \texttt{while } a \texttt{ do } c) \\
 & & \qquad \texttt{else skip}
\end{array}
$$

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

Executions = sequences of reductions, e.g. $c/s \stackrel{*}{\to} \texttt{skip}/s'$.

Natural (big-step) semantics

$s \vdash c \Rightarrow s'$: started in state s, command c terminates with state s'.

$$s \vdash \texttt{skip} \Rightarrow s$$

$$\frac{s \vdash a \Rightarrow v}{s \vdash (x := a) \Rightarrow s[x \leftarrow v]}$$

$$\frac{s \vdash c_1 \Rightarrow s' \qquad s' \vdash c_2 \Rightarrow s''}{s \vdash (c_1; c_2) \Rightarrow s''}$$

$$\frac{s \vdash a \Rightarrow \texttt{true} \qquad s \vdash c_1 \Rightarrow s'}{s \vdash (\texttt{if } a \texttt{ then } c_1 \texttt{ else } c_2) \Rightarrow s'}$$

$$\frac{s \vdash a \Rightarrow \texttt{false} \qquad s \vdash c_1 \Rightarrow s'}{s \vdash (\texttt{if } a \texttt{ then } c_1 \texttt{ else } c_2) \Rightarrow s'}$$

$$\frac{s \vdash a \Rightarrow \texttt{false}}{s \vdash (\texttt{while } a \texttt{ do } c) \Rightarrow s}$$

$$\frac{s \vdash a \Rightarrow \texttt{true} \qquad s \vdash c \Rightarrow s' \qquad s' \vdash (\texttt{while } a \texttt{ do } c) \Rightarrow s''}{s \vdash (\texttt{while } a \texttt{ do } c) \Rightarrow s''}$$

Comparing the two styles

Describe the same terminating executions:

$$s \vdash c \Rightarrow s' \quad \text{iff} \quad c/s \stackrel{*}{\rightarrow} \texttt{skip}/s'$$

Small-step semantics also capture

- divergence: infinite sequences of reductions
- "going wrong": $c/s \stackrel{*}{\rightarrow} c'/s' \not\rightarrow$ with $c' \neq \texttt{skip}$.

Big-step semantics make it easier to

- grow the language
- prove certain program equivalences

Proving program equivalences

Example: if c modifies no free variable of e, show

$$\texttt{if } e \texttt{ then } c; c_1 \texttt{ else } c; c_2 \quad \equiv \quad c; \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2$$

Big-step semantics: easy inversion on one evaluation derivation and construction of the other derivation.

Small-step semantics: need to invent a bisimulation $\sim$ containing the two commands above:

(Here, the bisimulation "only" has 4 cases, but it can get much worse.)

Growing the language

Let's add functions to IMP.

Commands: $c ::= \ldots \mid x := fn(a)$

Function definitions: $def ::= fn(x)\{c\}$

In big-step style: add another relation $\vdash fn(v) \Rightarrow v'$ for the evaluation of function calls.

$$\frac{s \vdash a \Rightarrow v \qquad \vdash fn(v) \Rightarrow v'}{s \vdash x := fn(a) \Rightarrow s[x \leftarrow v']}$$

$$\frac{fn(x)\{c\} \in Fundefs \qquad [x \leftarrow v] \vdash c \Rightarrow s}{\vdash fn(v) \Rightarrow s(fn)}$$

Functions in small-step style

In small-step, we need to extend the syntax of commands with a special form representing function calls being executed:

Commands: $c ::= \ldots \mid x := fn(a) \mid x := \texttt{call}(fn, c, s)$

Additional reduction rules:

$$(res := fn(a))/s \rightarrow (res := \texttt{call}(fn, c, [arg \leftarrow v])/s \quad \text{if } s \vdash a \Rightarrow v \text{ and } fn(arg)\{c\} \in Fundefs$$

$$(res := \texttt{call}(fn, \texttt{skip}, s')/s \rightarrow \texttt{skip}/s[res \leftarrow s'(fn)]$$

$$\frac{c_1/s_1 \rightarrow c_2/s_2}{(res := \texttt{call}(fn, c_1, s_1))/s \rightarrow (res := \texttt{call}(fn, c_2, s_2))/s}$$

Functions in small-step style

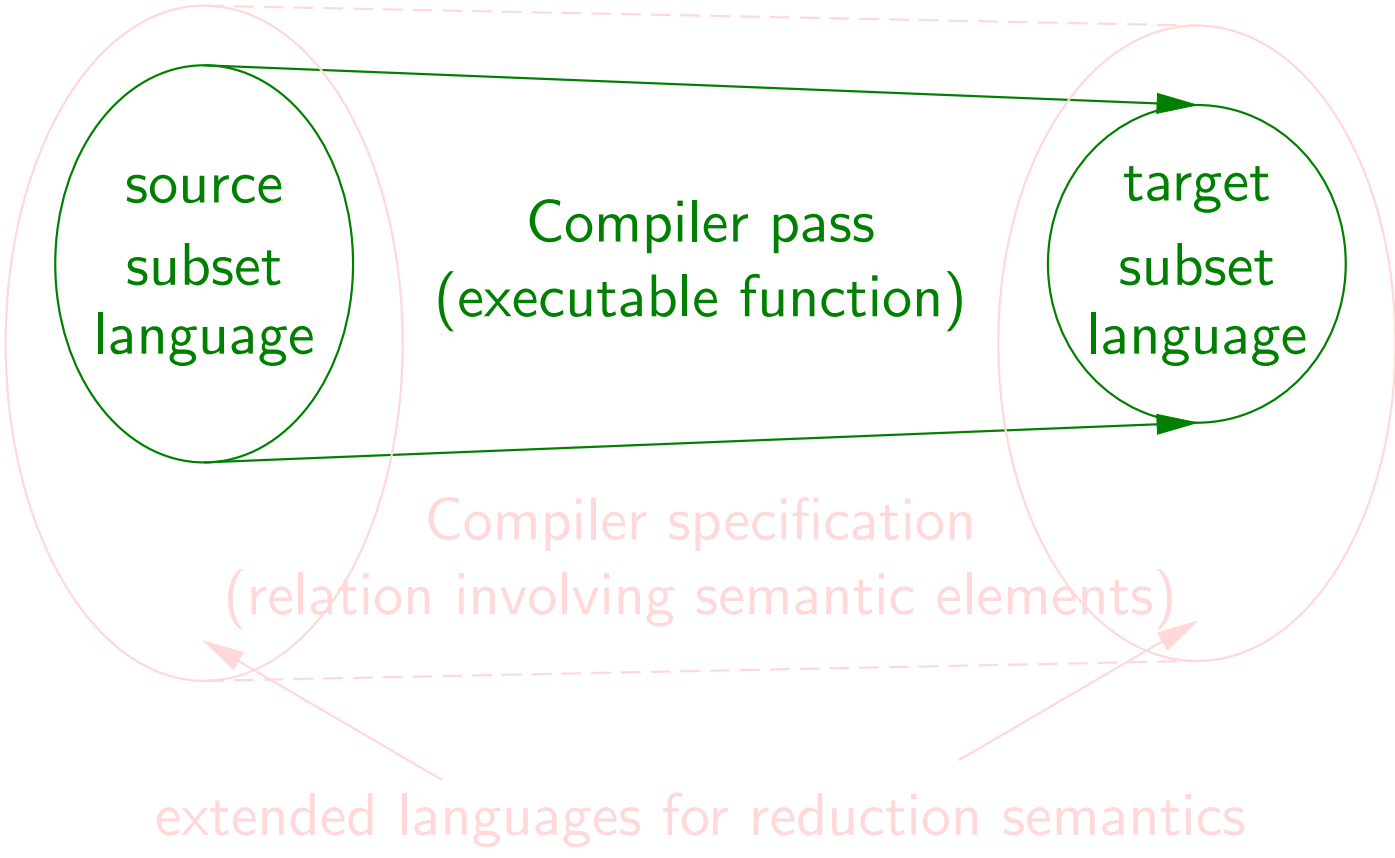
Commands: $c ::= \ldots \mid x := fn(a) \mid x := \texttt{call}(fn, c, s)$

Note that the syntax of the language must be extended with constructs that do not appear in the source language, but are here just for the purposes of the reduction rules.

Moreover, this `call` construct injects a semantic element, the store s, into the syntax.

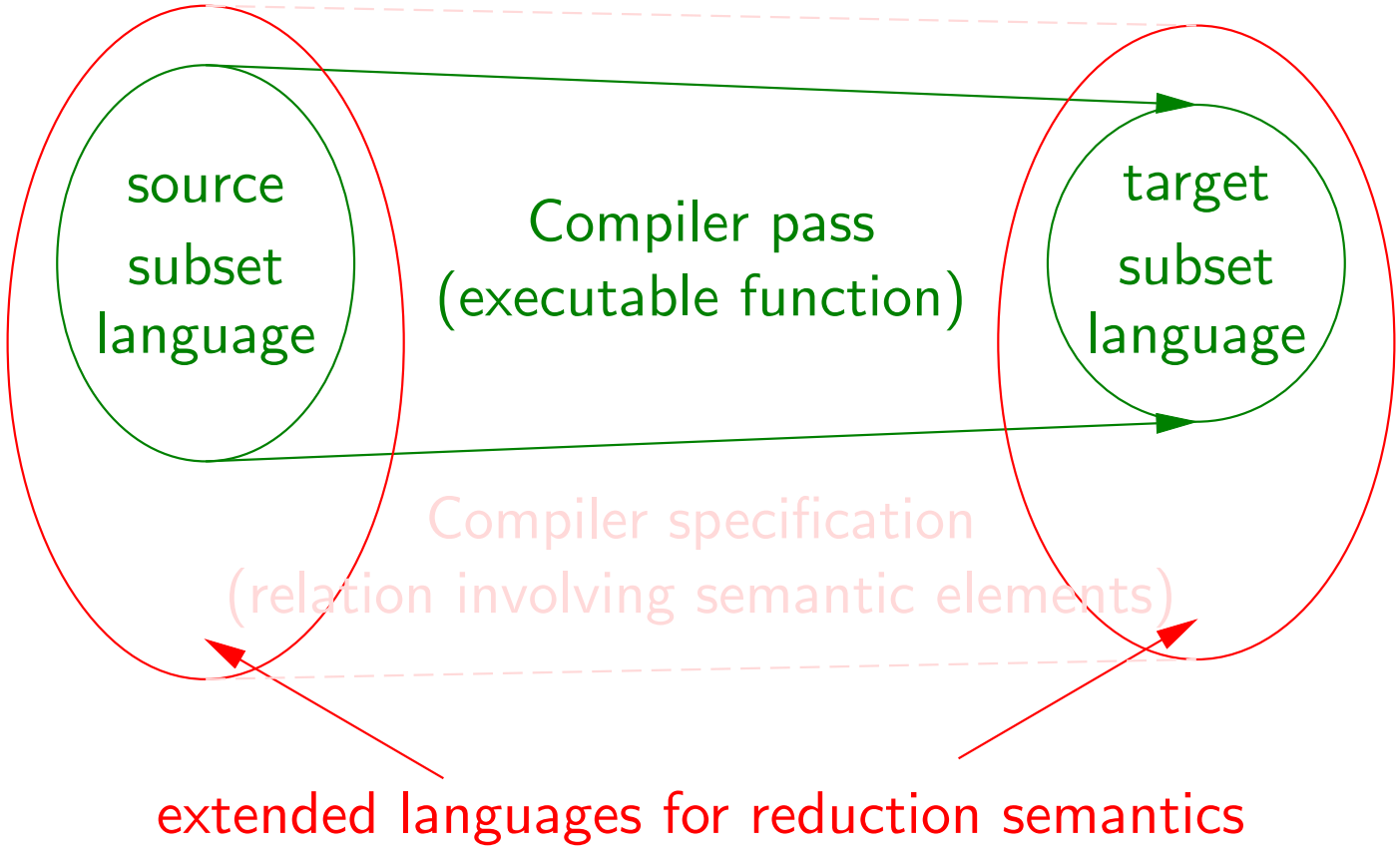
Impact on compiler verification

Two (pairs of) languages ⇒ two compilers: one for code generation, the other for proving semantic preservation.



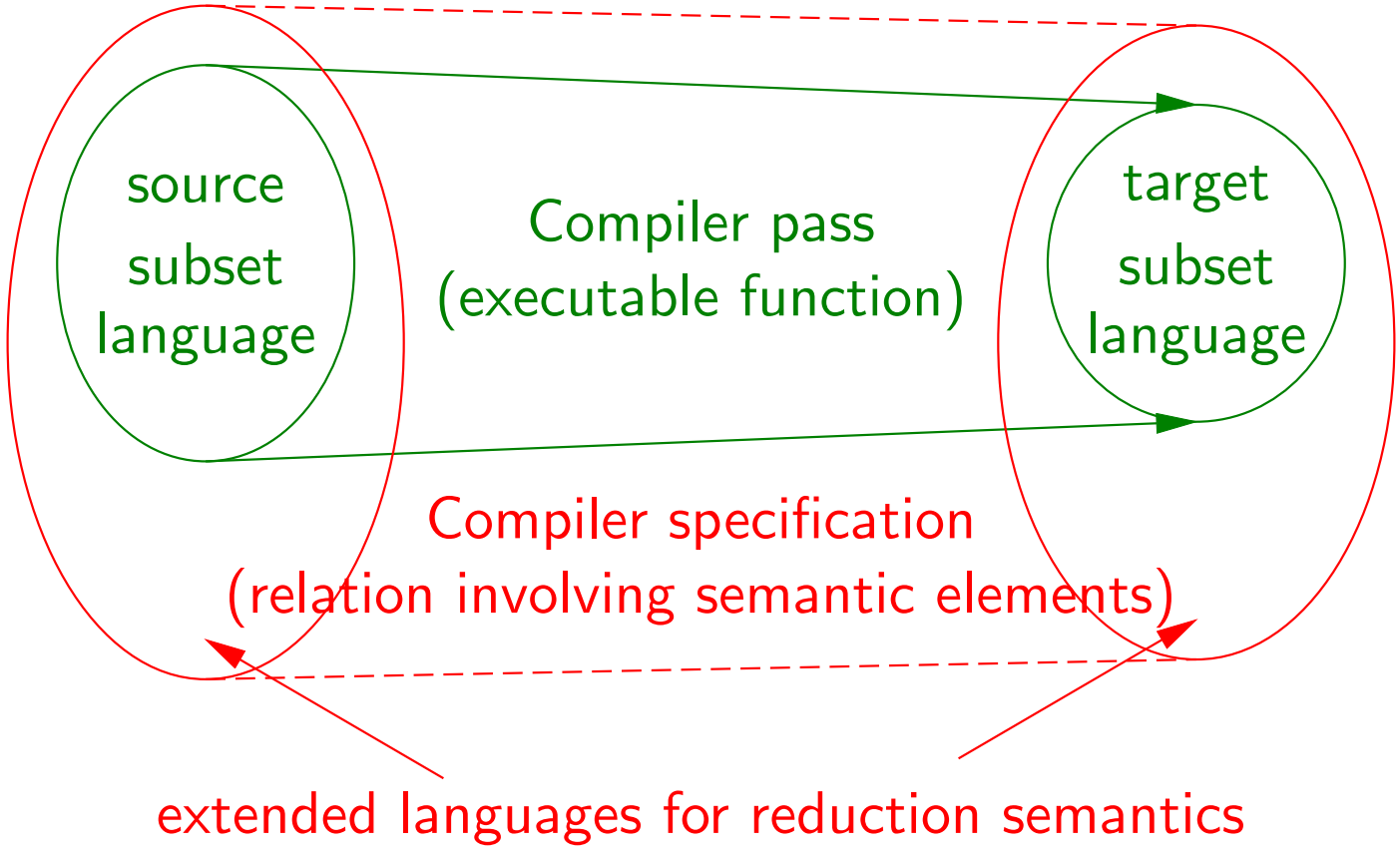
Impact on compiler verification

Two (pairs of) languages $\Rightarrow$ two compilers: one for code generation, the other for proving semantic preservation.



Impact on compiler verification

Two (pairs of) languages $\Rightarrow$ two compilers: one for code generation, the other for proving semantic preservation.



Episode 1:
The early days of CompCert

CompCert V1

2005–2006

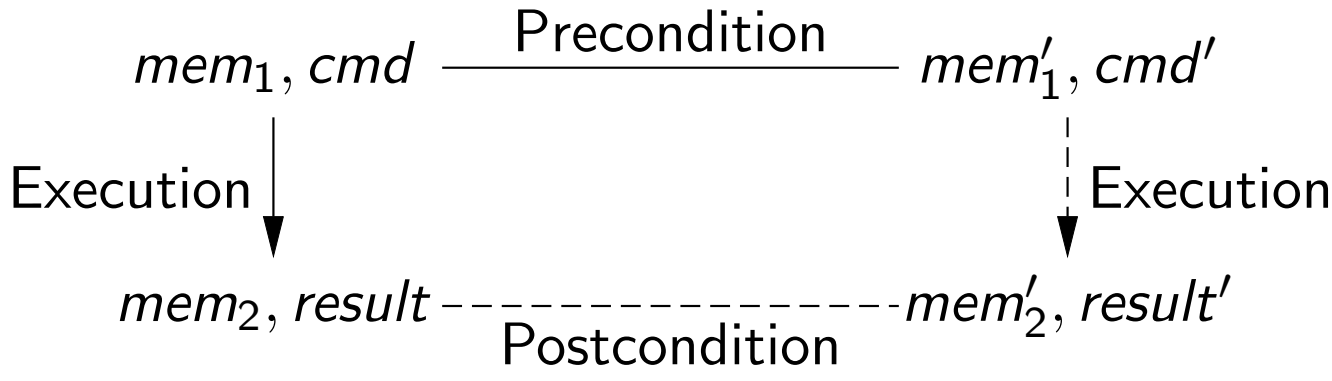
	Languages	Semantics, V1
	Clight	
(1)	⋮	Big-step semantics
	Cminor	
	RTL	
(2)	⋮	"Mixed-step" semantics (*)
	Mach	
	Asm	Transition system

(1) High-level languages, structured control.
(2) Intermediate languages, unstructured control (CFG).
(*) Small-step transitions for all instructions, except function calls treated in big-step style.

Assessment

Semantics: a bit verbose, but easy to write and understand.
("Natural semantics" lived up to its name.)

Proofs of compiler passes: by forward simulations, big-step style



The compositional nature of big-step semantics is a good match for the compositional nature of compilation functions.

Big-step gives powerful induction principles
(but: mutual inductions painful in Coq).

Extension 1: tail calls

Initially: "mixed-steps" semantics:

- Big-step function calls $fn, args, mem \Rightarrow res, mem'$
- Intra-function transitions $localstate, mem \rightarrow localstate', mem'$.

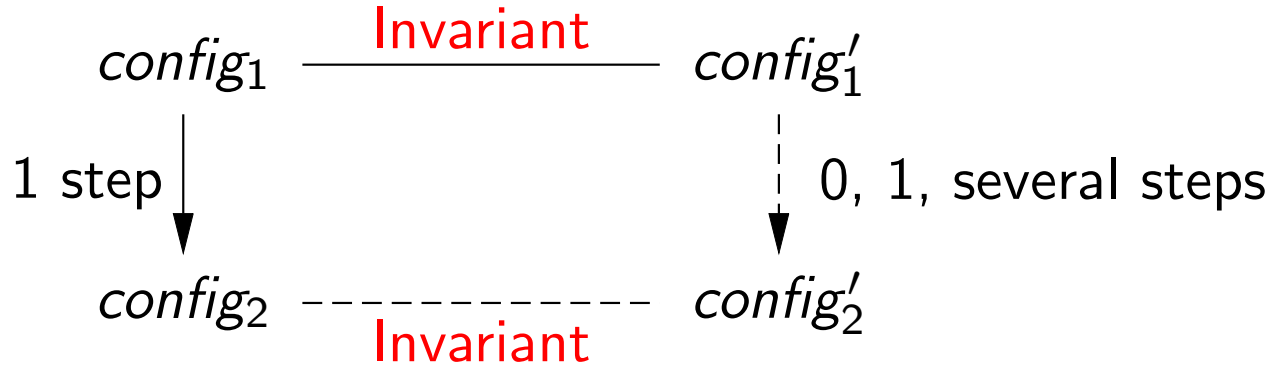
The call stack is implicit in the evaluation derivation.

To support tail function calls in the back-end:
make the call stack explicit in semantics of intermediate languages.
$\rightarrow$ small-step transition systems.

$$localstate, callstack, mem \rightarrow localstate', callstack', mem'$$

Extension 1: tail calls

Impact on proofs: become standard simulation diagrams



The invariants are more subtle than precondition $\wedge$ postcondition.

Extension 2: traces

Original compiler correctness theorem quite weak:

> *If the source Clight program terminates, the generated Asm program terminates and has no other behavior.*

Strengthening this result to show preservation of I/O operations performed by the program:

> *If the source Clight program terminates and performs observable effects t, the generated Asm program terminates and performs the same effects t, and has no other behavior.*

Observable effects = calls to external functions (system calls) + (later) reading and writing `volatile` global variables.

Extension 2: traces

Instrumenting the semantics to collect traces of observables:

transition systems:	$c_1 \to c_2$	becomes	$c_1 \xrightarrow{t} c_2$ (LTS)
big-step predicates:	$cmd \Rightarrow res$	becomes	$cmd \overset{t}{\Rightarrow} res$

An invasive but fairly systematic change. E.g. for IMP:

$$\frac{s \vdash c_1 \overset{t}{\Rightarrow} s' \qquad s' \vdash c_2 \overset{t'}{\Rightarrow} s''}{s \vdash (c_1; c_2) \overset{t.t'}{\Rightarrow} s''}$$

CompCert V2

2006–2007

	Languages	V1	V2
(1)	Clight ⋮ Cminor	Big-step	Big-step + traces
(2)	RTL ⋮ Mach	Mixed-step	Labeled transition systems
	Asm	Transition system	Labeled transition system

(1) High-level languages, structured control.
(2) Intermediate languages, unstructured control (CFG).

Episode 2:

non-termination,

non-termination,

non-termination,

non-termination,

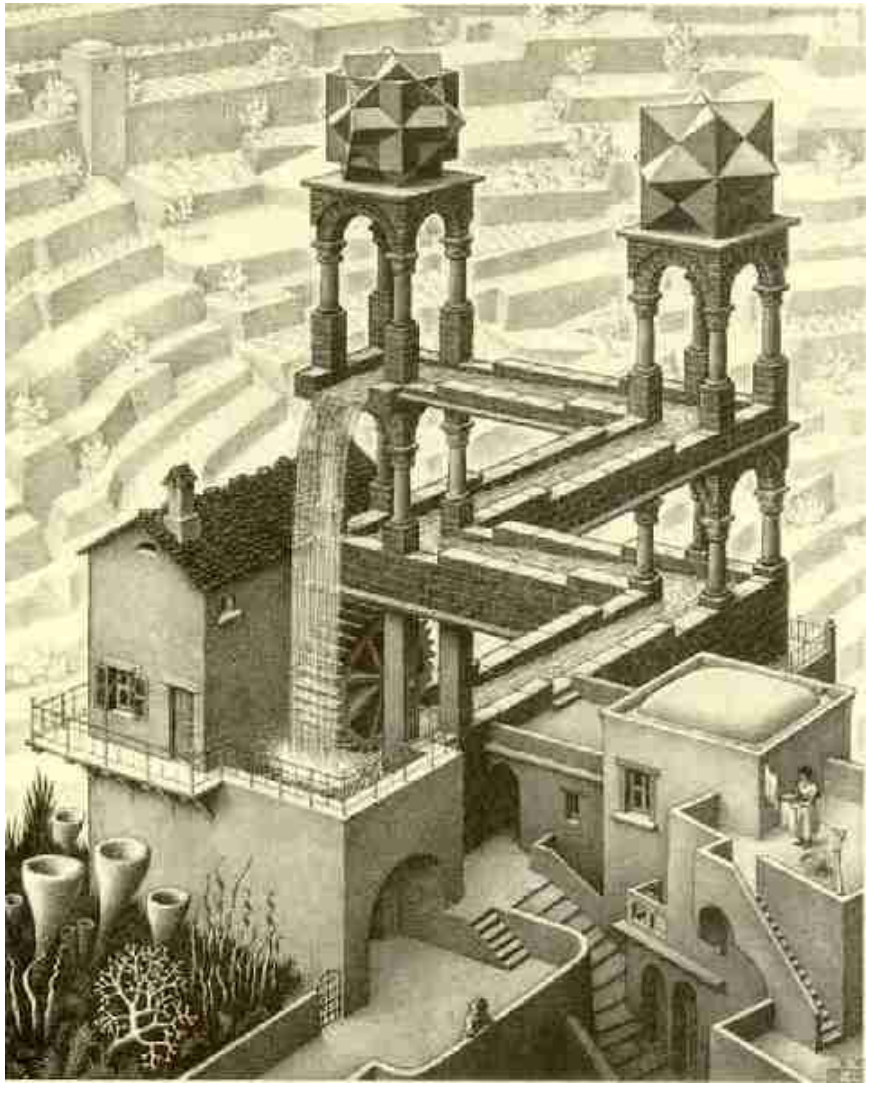
non-termination,

non-termination,

non-termination,

non-termination,

. . .



Non-termination

The original compiler correctness theorem applies only to terminating source programs:

> *If the source Clight program terminates and performs observable effects t, the generated Asm program terminates and performs the same effects, and has no other behavior.*

If the source runs forever (without going wrong), we expect the generated code to do the same, but we have no proof.

Reality check

Mr. Aircraft Manufacturer:

Sir! Our flight control software does not terminate by itself! It keeps running flawlessly until the pilot parks the plane and turns power off.

Semantics for divergence

In small-step style: obvious!

- Termination: $S \to S_1 \to \cdots \to S_n \not\to$
- Divergence: $S \to \cdots \to S_n \to \cdots$ (infinite sequence)

In big-step style: problematic!

Milner & Tofte: a negative characterization

$$a \text{ diverges} \iff a \not\Rightarrow \texttt{wrong} \ \wedge \ \forall v, a \not\Rightarrow v$$

More desirable: a positive characterization of divergence, so that we do not have to give rules for "going wrong" behaviors.

Towards a big-step view of divergence

Big-step semantics $\approx$ adding structure to terminating sequences of reductions. Consider such a sequence for $c; c'$:

$$(c; c')/s \to (c_1; c')/s_1 \to \cdots \to (\texttt{skip}; c')/s_2 \to c'/s_2 \to \cdots \to \texttt{skip}/s_3$$

It contains a terminating reduction sequence for c:
$(c, s) \stackrel{*}{\to} (\texttt{skip}, s_2)$ followed by another for c'.

The big-step semantics reflects this structure in its rule for sequences:

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

Towards a big-step view of divergence

Let's play the same game for infinite sequences of reductions!

Consider an infinite reduction sequence for $c; c'$. It must be of one of the following two forms:

$$(c; c')/s \xrightarrow{*} (c_i; c')/s_i \to \cdots$$

$$(c; c')/s \xrightarrow{*} (\texttt{skip}; c')/s_i \to c'/s_i \xrightarrow{*} c'_j/s_j \to \cdots$$

I.e. either c diverges, or it terminates normally and c' diverges.

Idea: write inference rules that follow this structure and define a predicate $s \vdash c \Rightarrow \infty$, meaning "in initial state s, the command c diverges".

Big-step rules for divergence

$$\frac{s \vdash c_1 \Rightarrow \infty}{s \vdash c_1; c_2 \Rightarrow \infty} \qquad \frac{s \vdash c_1 \Rightarrow s_1 \quad s_1 \vdash c_2 \Rightarrow \infty}{s \vdash c_1; c_2 \Rightarrow \infty}$$

$$\frac{s \vdash c_1 \Rightarrow \infty \text{ if } s \vdash a \Rightarrow \texttt{true} \quad s \vdash c_2 \Rightarrow \infty \text{ if } s \vdash a \Rightarrow \texttt{false}}{s \vdash \texttt{if } a \texttt{ then } c_1 \texttt{ else } c_2 \Rightarrow \infty} \qquad \frac{s \vdash a \Rightarrow \texttt{true} \quad s \vdash c \Rightarrow \infty}{s \vdash \texttt{while } a \texttt{ do } c \Rightarrow \infty}$$

$$\frac{s \vdash a \Rightarrow \texttt{true} \quad s \vdash c \Rightarrow s_1 \quad s_1 \vdash \texttt{while } a \texttt{ do } c \Rightarrow \infty}{s \vdash \texttt{while } a \texttt{ do } c \Rightarrow \infty}$$

Problem: there are no axioms! So, isn't it the case that these rules define a predicate $s \vdash c \Rightarrow \infty$ that is always false?

Induction vs. coinduction in a nutshell

A set of axioms and inference rules can be interpreted in two ways:

Inductive interpretation:

- In set theory: the least defined predicate that satisfies the axioms and rules (smallest fixpoint).
- In proof theory: conclusions of finite derivation trees.

Coinductive interpretation:

- In set theory: the most defined predicate that satisfies the axioms and rules (greatest fixpoint).
- In proof theory: conclusions of finite or infinite derivation trees.

Example of divergence

Let's interpret coinductively the inference rules defining $s \vdash c \Rightarrow \infty$.

We can easily show that classic examples of divergence are captured. Consider $c = \texttt{while true do skip}$. We can build the following infinite derivation of $s \vdash c \Rightarrow \infty$:

$$\dfrac{\dfrac{s \vdash \texttt{true} \Rightarrow \texttt{true} \qquad s \vdash \texttt{skip} \Rightarrow s}{s \vdash \texttt{while true do skip} \Rightarrow \infty} \qquad \dfrac{\dfrac{s \vdash \texttt{true} \Rightarrow \texttt{true} \qquad s \vdash \texttt{skip} \Rightarrow s}{s \vdash \texttt{while true do skip} \Rightarrow \infty} \qquad \vdots}{s \vdash \texttt{while true do skip} \Rightarrow \infty}}{s \vdash \texttt{while true do skip} \Rightarrow \infty}$$

Coinductive big-step semantics

X. Leroy and H. Grall, *Inf. & Comp.* 207(2), 2009

A paper that studies the coinductive big-step approach on a simple language (CBV λ-calculus with constants):

- Equivalence with the existence of infinite reduction sequences (and other established characterizations of divergence).
- Use for a type soundness proof.
- Use for a compiler correctness proof (compilation to a SECD-like abstract machine)
- Extension with traces.

It all seems to work! Time to scale this approach to CompCert...

CompCert V3

2007–2009

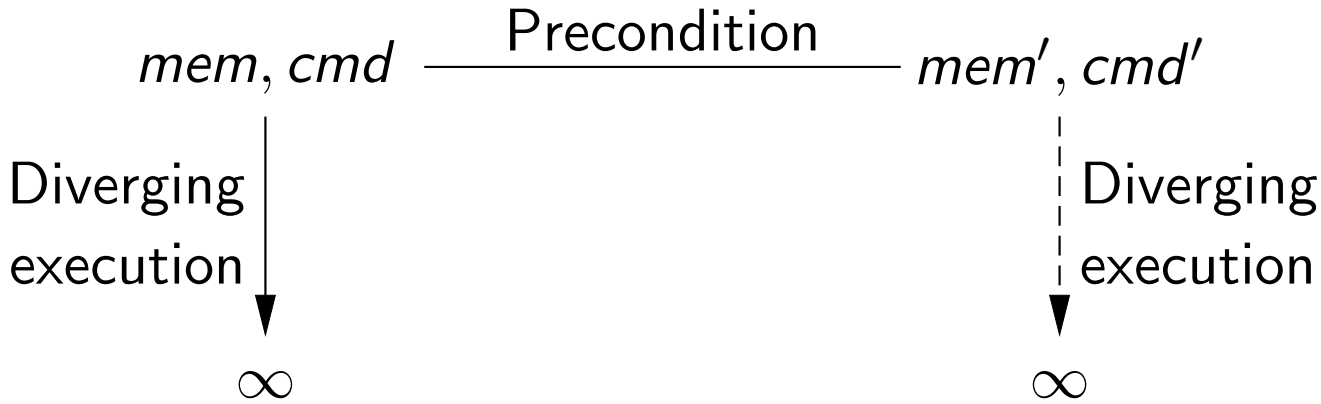
Languages	V2	V3
Clight ⋮ Cminor	Big-step (w/ traces)	Coinductive big-step (w/ traces)
RTL ⋮ Mach	LTS	LTS
Asm	LTS	LTS

Assessment

With some elbow grease, could extend the proof of semantic preservation to diverging source programs:

> *If the source Clight program diverges, producing trace T of observables, the generated Asm program diverges with the same trace T.*

Proof: using simulation diagrams of the form



Assessment

The costs are relatively high:

- Separate semantic definitions for termination & divergence
  $\rightarrow$ size of semantics $\times 1.7$
- Separate proofs of simulation for termination & divergence
  $\rightarrow$ size of proofs $\times 1.20$
  (The crucial invariants and lemmas are shared, though).
- Coq's support for coinductive proofs is temperamental.
  (The very syntactic "guard condition".)

Episode 3:
trouble ahead!

Time to reconsider...

Next on the list of features to support in CompCert C:

- General `goto` (unstructured control);
- Partially-unspecified evaluation order for expressions (e.g. in `f() + g() + h()`, the 3 functions can be called in any of the 6 possible orders `fgh`, `fhg`, `gfh`, `ghf`, `hfg`, `hgf`).

This rings the death knell for big-step semantics...

Big-step semantics for `goto`

A proposal by Hendrik Tews, *Verifying Duff's device: A simple compositional denotational semantics for Goto and computed jumps*, 2004, unpublished. The execution relation becomes:

$$s_{init}, cmd, income \Rightarrow outcome, s_{final}$$

$$\begin{array}{rcl} income & = & \text{how the command is entered} \\ & & \text{(normally or while searching for label } \ell\text{)} \\ outcome & = & \text{how the command terminates} \\ & & \text{(normally or by goto } \ell\text{)} \end{array}$$

Incredibly clever, but causes an explosion in the number of rules, esp. in conjuction with coinductive divergence.

Big-step for unspecified evaluation orders

Scheme-style: OK!
(each operator has one evaluation order, which is not specified).

$$\frac{s \vdash a_1 \Rightarrow n_1, s' \qquad s' \vdash a_2 \Rightarrow n_2, s''}{s \vdash a_1 + a_2 \Rightarrow n_1 + n2, s''}$$

$$\frac{s \vdash a_2 \Rightarrow n_2, s' \qquad s' \vdash a_1 \Rightarrow n_1, s''}{s \vdash a_1 + a_2 \Rightarrow n_1 + n2, s''}$$

C-style: not enough!
Must be able to reduce arbitrarily deep subexpressions in almost arbitrary order, like in the pure λ-calculus.

(Michael Norrish, *C formalized in HOL*, PhD, 1998.)

CompCert V4

2009–2012

Languages	V3	V4
Clight ⋮ Cminor	Coinductive big-step (w/ traces)	Small-step: – reductions for expressions – LTS for statements
RTL ⋮ Mach	LTS	LTS
Asm	LTS	LTS

Biting the bullet: no more big-step semantics!

- For C expressions: reductions under context (Wright-Felleisen)
- For statements and function calls: a transition system based on focused statements and continuation terms.

Small-step semantics with continuations

A. Appel and S. Blazy, 2007

A variant of standard small-step semantics that avoids extending the syntax of command with forms useful only during reductions.

Idea: instead of rewriting whole commands:

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

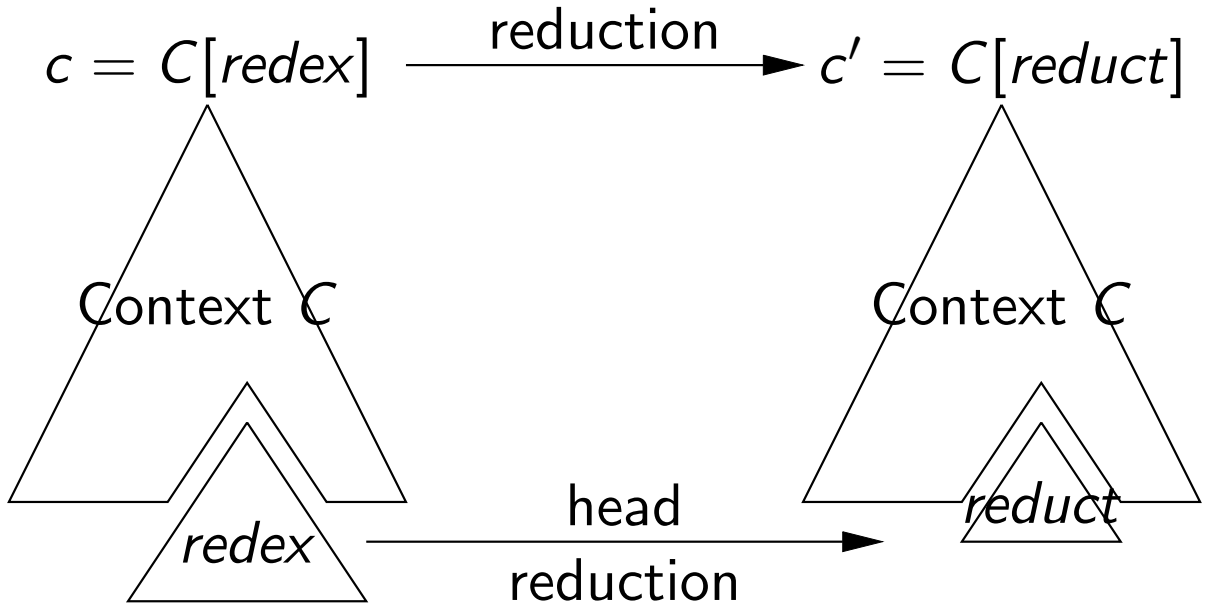
rewrite pairs of (subcommand under focus, remainder of command):

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

(Related to focusing in proof theory.)

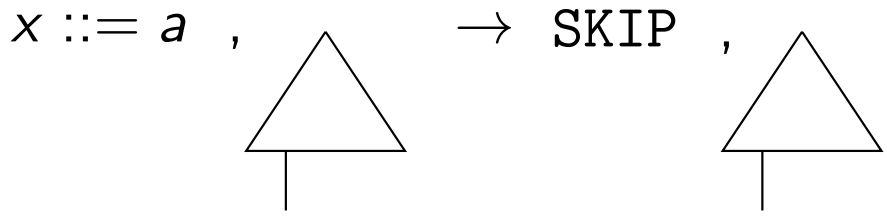
Standard small-step semantics

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

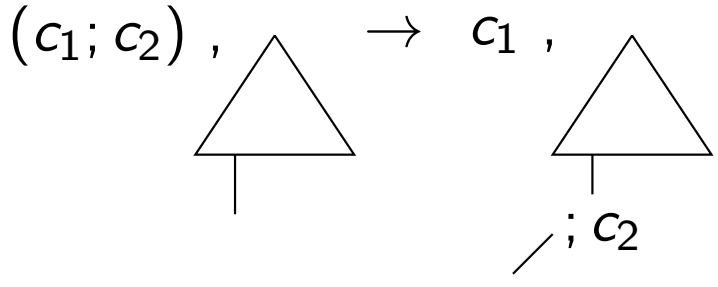


Focusing the small-step semantics

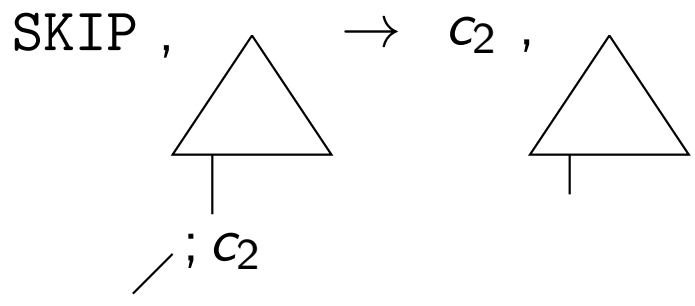
Rewrite pairs (subcommand, context in which it occurs).



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



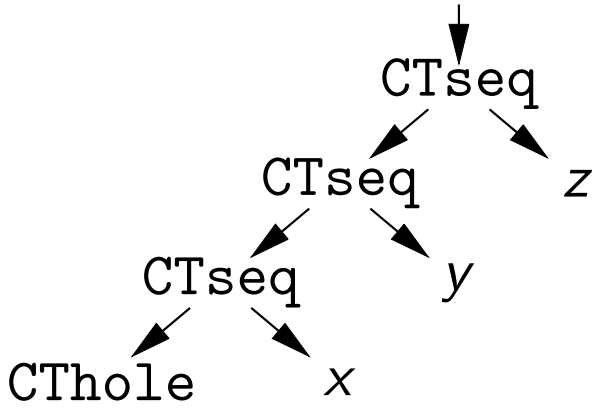
Resuming a sequence

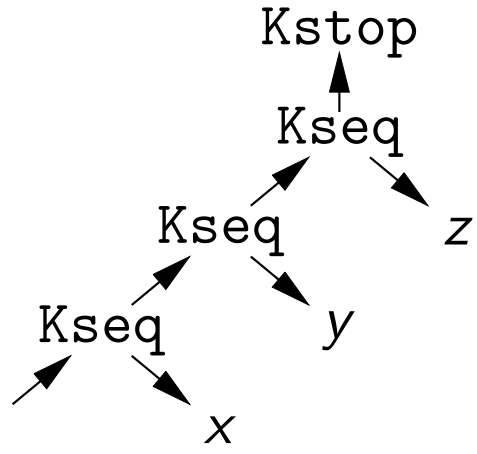
Representing contexts "upside-down"

G. Huet, *The Zipper*, 1997

```
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 $\approx$ a continuation.

("Eventually, do x, then do y, then do z, then stop.")

Transition rules for IMP

$$
\begin{array}{rcll}
x := a/k/s & \to & \texttt{skip}/k/s[x \leftarrow v] & \text{if } s \vdash a \Rightarrow v \\
(c_1; c_2)/k/s & \to & c_1/\texttt{Kseq}\ c_2\ k/s & \\
\texttt{if}\ a\ \texttt{then}\ c_1\ \texttt{else}\ c_2/k/s & \to & c_1/k/s & \text{if } s \vdash a \Rightarrow \texttt{true} \\
\texttt{if}\ a\ \texttt{then}\ c_1\ \texttt{else}\ c_2/k/s & \to & c_2/k/s & \text{if } s \vdash a \Rightarrow \texttt{false} \\
\texttt{while}\ a\ \texttt{do}\ c/k/s & \to & c/\texttt{Kseq}\ (\texttt{while}\ a\ \texttt{do}\ c)\ k/s & \\
 & & \text{if } s \vdash a \Rightarrow \texttt{true} & \\
\texttt{while}\ a\ \texttt{do}\ c/k/s & \to & \texttt{skip}/c/k & \text{if } s \vdash a \Rightarrow \texttt{false} \\
\texttt{skip}/\texttt{Kseq}\ c\ k/s & \to & c/k/s &
\end{array}
$$

Adding functions

To add functions, we need a new form of continuations representing pending function calls ($\approx$ the call stack), but no ad-hoc extension to the syntax of commands.

Commands: $c ::= \ldots \mid x := fn(a)$

Continuations: $k ::= \texttt{Kstop} \mid \texttt{Kseq}\ c\ k \mid \texttt{Kcall}\ s\ res\ fn\ k$

New rules:

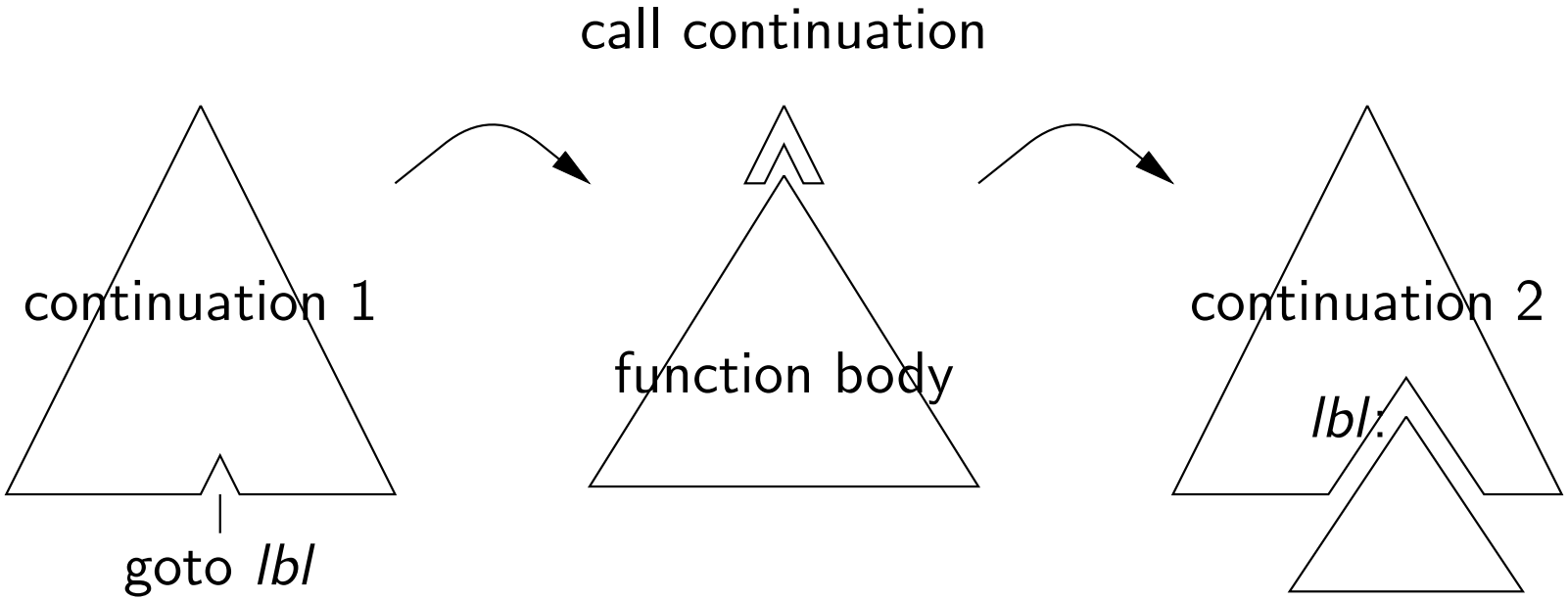
$$res := fn(a)/k/s \quad \rightarrow \quad c/\texttt{Kcall}\ s\ res\ fn\ k/[arg \leftarrow v]$$
$$\text{if } s \vdash a \Rightarrow v \text{ and } fn(arg)\{c\} \in Fundefs$$

$$\texttt{skip}/\texttt{Kcall}\ s'\ res\ fn\ k/s \quad \rightarrow \quad \texttt{skip}/k/s'[res \leftarrow s(fn)]$$

Handling goto

by zipper surgery

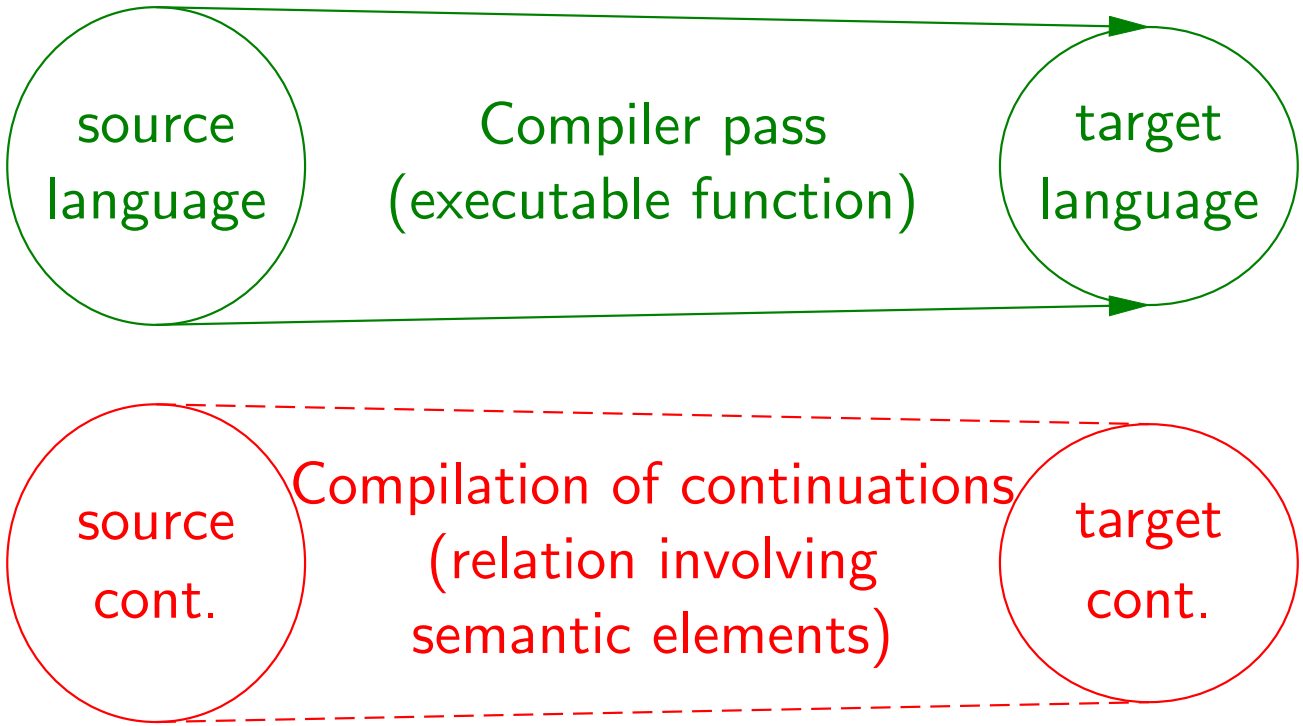
A search function that finds a subcommand labeled *lbl* while manufacturing the corresponding continuation:



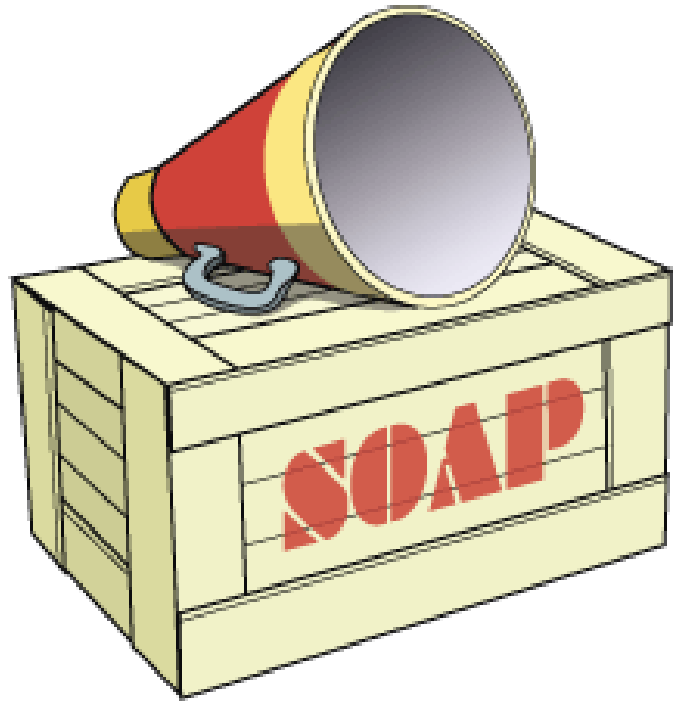
Implements the transition goto $lbl/k_1/s \to lbl : c/k_2/s$.

Impact on compiler verification

To prove semantic preservation, reason over the normal compilation function for source terms, complemented with a compilation relation (nonexecutable) for continuations.



Epilogue:
Some lessons learned



Looking back...

CompCert's path through the landscape of operational semantics has been tortuous indeed!

Languages	V1	V2	V3	V4
Clight ⋮ Cminor	Big-step	Big-step w/traces	Coinductive big-step	Small-step w/continuations
RTL ⋮ Mach	Mixed-step	LTS	LTS	LTS
Asm	Trans. Syst.	LTS	LTS	LTS

Some lessons learned

Semantics engineering is like software engineering:

> *Plan to throw one away; you will, anyhow. (F. Brooks)*

Exploration on toy languages (IMP, STLC) is essential, but don't expect the results to scale to big languages.

The detour through big-step was costly, but helped find the correct proof invariants and get the project off the ground.

In the end, Labeled Transition Systems (in one form or other) win, despite leading to semantics that look more like abstract machines than like high-level specifications.

The sensitivity is disturbingly high: add one language feature, redo the whole semantics.

Are we there yet?

No! Some features yet to be accommodated in CompCert:

Shared-memory concurrency:

- Verified Software Toolchain (A. Appel et al, Princeton): coarse interleaving, for race-free programs
- CompCertTSO (P. Sewell et al, Cambridge): fine-grained interleaving, data races, relaxed TSO memory.

Separate compilation and linking:

- based on operational semantics with interleaving (at Yale)
- or using some form of logical relations to explicate contracts between compilation units (N. Benton, C.K. Hur, A. Amal)

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; ongoing efforts on Javascript).

A great opportunity to challenge the state of the art and invent new approaches and mechanization techniques!