Proving the correctness of a compiler

Xavier Leroy

Collège de France and Inria

EUTypes Summer School 2019

The compilation process

In general: any translation from a computer language to another.

More specifically:

- automatic translation
- from a high-level language suitable for programming by humans
- to a low-level language executable by machines
- with a concern for efficiency ("optimizing" compilers).

Miscompilation

Traduttore, traditore ("Translator, traitor")

Bugs in the compiler can make it produce wrong executable code for a correct source program.

For ordinary software:

- negligible compared with bugs in the program itself;
- painful to track down.

For critical software:

- a risk that needs to be handled;
- can invalidate the guarantees obtained by formal verification of the source program.

The formal verification of compilers

To prove (with mathematical certainty) that a compiler is free of miscompilation and preserves the semantics of the source programs.

To transport the guarantees obtained by source-level verification all the way to the executable code.

Teaching compiler verification at EUTypes

An opportunity to study:

- An approach to program proof where the program and the proof are both written using a proof assistant (Coq).
- The semantics of two languages (source & target) and how to mechanize them.
- Some nontrivial algorithms and their correctness proofs.

Lecture material

`https://xavierleroy.org/courses/EUTypes-2019/`

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

Course outline

1. Compiling IMP to a simple virtual machine; first compiler proofs.
2. Notions of semantic preservation; more on semantics; finishing the proof of the IMP $\rightarrow$ VM compiler.
3. Verification of an optimizing program transformation (constant propagation) and the static analysis it uses.
4. More on static analyses: fixpoint iterations, liveness analysis, applications to dead code elimination.

Homework: exercises (some recommended, others optional).

I

The IMP language

Warm-up: Arithmetic expressions

A language of expressions comprising

- variables x, y, ...
- integer constants 0, 1, -5, ..., n
- $e_1 + e_2$ and $e_1 - e_2$
  where e_1, e_2 are themselves expressions.

Abstract syntax

We manipulate expressions not via their concrete syntax (`1 + x - 2`) but via their abstract syntax represented by an inductive type.

```
Definition ident := string.

Inductive aexp : Type :=
  | CONST (n: Z)                     (* a constant, or *)
  | VAR (x: ident)                   (* a variable, or *)
  | PLUS (a1: aexp) (a2: aexp)       (* a sum, or *)
  | MINUS (a1: aexp) (a2: aexp).     (* a difference *)
```

`CONST`, `VAR`, `PLUS`, `MINUS` are functions that construct terms of type `aexp`.

All terms of type `aexp` are finitely generated by these 4 functions
$\rightarrow$ enables case analysis and induction.

Semantics of arithmetic expressions

In denotational style: a function $[\![e]\!]\ s$ that gives the denotation of expression e (the integer it evaluates to) in store s (a mapping from variable names to integers).

In ordinary mathematics, the denotational semantics is presented as a set of equations:

$$\begin{aligned}
[\![x]\!]\ s &= s(x) \\
[\![n]\!]\ s &= n \\
[\![e_1 + e_2]\!]\ s &= [\![e_1]\!]\ s + [\![e_2]\!]\ s \\
[\![e_1 - e_2]\!]\ s &= [\![e_1]\!]\ s - [\![e_2]\!]\ s
\end{aligned}$$

In Coq: recursive function + pattern-matching. (See file `IMP.v`.)

The IMP language

A prototypical imperative language with structured control flow.

Composed of expressions (arithmetic, Boolean) and commands.

Arithmetic expressions:

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

Boolean expressions:

$$\begin{aligned} b ::= {} & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \le a_2 \\ \mid {} & \texttt{not}\ b \mid b_1\ \texttt{and}\ b_2 \end{aligned}$$

Commands (statements):

$$\begin{aligned} c ::= {} & \texttt{skip} && \text{(do nothing)} \\ \mid {} & x := a && \text{(assignment)} \\ \mid {} & c_1; c_2 && \text{(sequence)} \\ \mid {} & \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{fi} && \text{(conditional)} \\ \mid {} & \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done} && \text{(loop)} \end{aligned}$$

An example of an IMP program

Euclidean division by repeated subtraction.

```
// input: dividend in a, divisor in b

r := a;
q := 0;
while b <= r do
    r := r - b;
    q := q + 1
done

// output: quotient in q, remainder in r
```

Formalizing IMP

Abstract syntax: three inductive types: `aexp`, `bexp`, `com`.

Denotational semantics: not representable as a Coq function!

Classically, the denotation $[\![c]\!]\ s$ of a command is either $\perp$ (nontermination) or the final store s' (termination).

IMP being Turing-complete, this denotation is not computable and cannot be represented as a Coq function.

Operational semantics: in big-step style, as a relation $c/s \Rightarrow s'$ ("started in store s, command c terminates and the final store is s' ").

Big-step operational semantics

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

$$\frac{c_1/s \Rightarrow s_1 \qquad c_2/s_1 \Rightarrow s_2}{c_1; c_2/s \Rightarrow s_2} \qquad\qquad \frac{\begin{array}{c} c_1/s \Rightarrow s' \text{ if } [\![b]\!]\ s = \texttt{true} \\ c_2/s \Rightarrow s' \text{ if } [\![b]\!]\ s = \texttt{false} \end{array}}{\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2/s \Rightarrow s'}$$

$$\frac{[\![b]\!]\ s = \texttt{false}}{\texttt{while } b \texttt{ do } c \texttt{ done}/s \Rightarrow s}$$

$$\frac{[\![b]\!]\ s = \texttt{true} \qquad c/s \Rightarrow s_1 \qquad \texttt{while } b \texttt{ do } c \texttt{ done}/s_1 \Rightarrow s_2}{\texttt{while } b \texttt{ do } c \texttt{ done}/s \Rightarrow s_2}$$

In Coq: an inductive predicate `cexec s c s'`.

II

The IMP virtual machine

Virtual machines

Producing machine code for real processors (x86, ARM, ...) is rather difficult.

Many compilers (e.g. Java, C#) use a virtual machine as an intermediate step between source language and true machine code.

Like real machines, virtual machines execute sequences of simple instructions: no complex expressions, no control structures, ...

The instructions of the virtual machine are chosen to be close to the basic operations of the source language.

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 s : a mapping from variable names to integer values.
- The stack σ : a list of integer values (used to store intermediate results temporarily).

(Inspiration: old HP pocket calculators; the Java Virtual Machine.)

The instruction set

$$
\begin{array}{lll}
i ::= & \texttt{Iconst}(n) & \text{push } n \text{ on stack} \\
\mid & \texttt{Ivar}(x) & \text{push value of } x \\
\mid & \texttt{Isetvar}(x) & \text{pop value and assign it to } x \\
\mid & \texttt{Iadd} & \text{pop two values, push their sum} \\
\mid & \texttt{Iopp} & \text{pop one value, push its opposite} \\
\mid & \texttt{Ibranch}(\delta) & \text{unconditional jump} \\
\mid & \texttt{Ibeq}(\delta_1, \delta_0) & \text{pop two values, jump } \delta_1 \text{ if } = \text{, jump } \delta_0 \text{ if } \neq \\
\mid & \texttt{Ible}(\delta_1, \delta_0) & \text{pop two values, jump } \delta_1 \text{ if } \leq \text{, jump } \delta_0 \text{ if } > \\
\mid & \texttt{Ihalt} & \text{end of program}
\end{array}
$$

By default, each instruction increments pc by 1.

Exception: branch instructions increment it by $1 + \delta$.

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

Example

stack	ϵ	12	1 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	$\texttt{Ivar}(x)$;	$\texttt{Iconst}(1)$;	$\texttt{Iadd}$;	$\texttt{Isetvar}(x)$;	$\texttt{Ibranch}(-5)$

Semantics of the machine

Given in small-step operational style: a transition relation that represents the execution of one instruction.

```
Definition code := list instruction.
Definition stack := list Z.
Definition config : Type := (Z * stack * store)%type.

Inductive transition (C: code): config -> config -> 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 `Ihalt` instruction, empty stack.

```
Definition transitions (C: code): config -> config -> Prop :=
  star (transition C).

Definition machine_terminates
             (C: code) (s_init: store) (s_final: store) : Prop :=
  exists pc,
     transitions C (0, nil, s_init) (pc, nil, s_final)
  /\ instr_at C pc = Some Ihalt.
```

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

III

The compiler

Compilation of arithmetic expressions

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

	code for a	

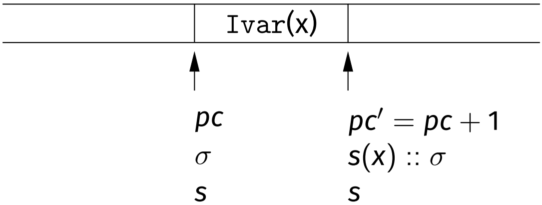
	Before:		After:
	pc		$pc' = pc + \|code\|$
Before:	σ	After:	$n :: \sigma$
	s		s

Compilation is just translation to “reverse Polish notation”.

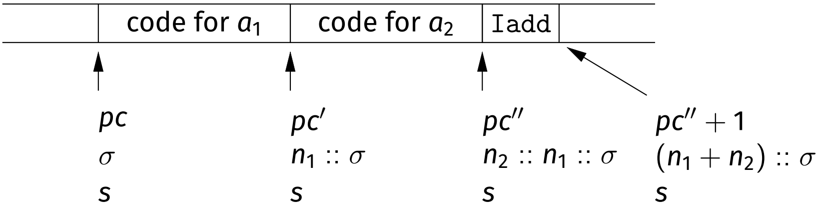
(See Coq function `compile_aexp`)

Compilation of arithmetic expressions

Base case: if $a = x$,

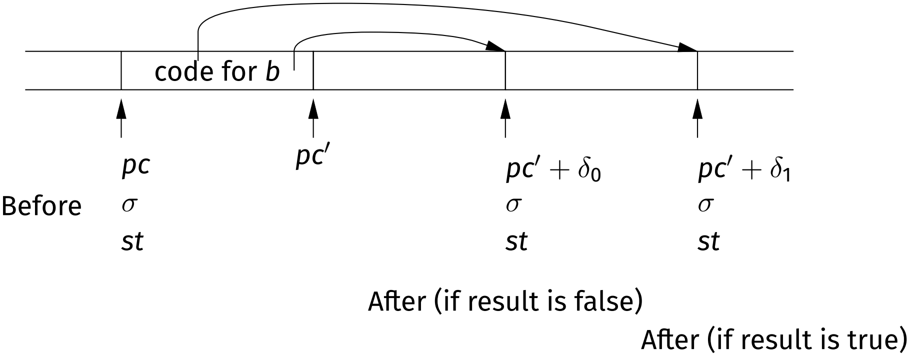


Recursive decomposition: if $a = a_1 + a_2$,



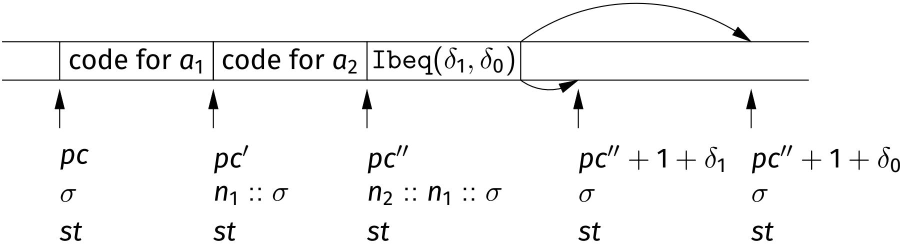
Compilation of boolean expressions

`compile_bexp` b $cond$ δ_1 δ_0 should
  skip δ_1 instructions forward if b evaluates to `true`
  skip δ_0 instructions forward if b evaluates to `false`.



Compilation of boolean expressions

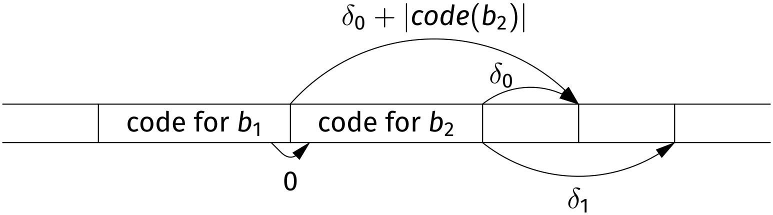
A base case: $b = (a_1 = a_2)$



Short-circuiting "`and`" expressions

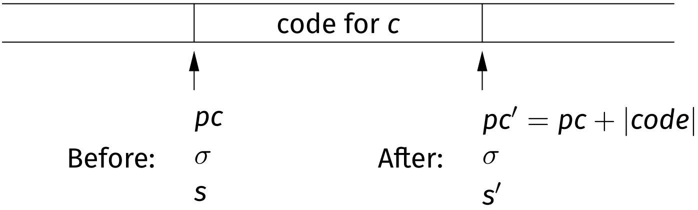
If b_1 evaluates to `false`, so does b_1 `and` b_2: no need to evaluate b_2!

$\rightarrow$ 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.



Compilation of commands

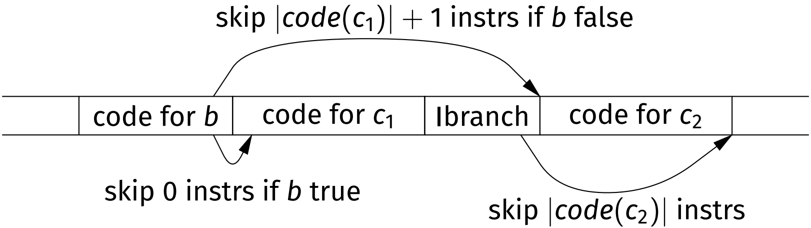
If the command c, started in initial store s, terminates in final store s',



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

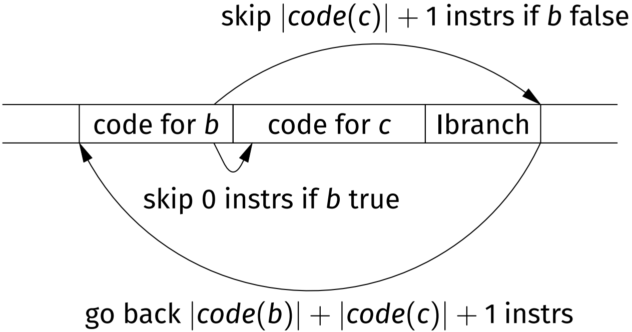
The mysterious offsets

Code for `IFTHENELSE` b c_1 c_2:



The mysterious offsets

Code for `WHILE` b c:



IV

First compiler correctness results

Compiler verification

We now have two ways to run a program:

- Interpret it using e.g. the `cexec_bounded` function (which follows the IMP semantics `cexec`)
- Compile it, then run the generated virtual machine code (following the VM semantics `transition`).

Will we get the same results either way?

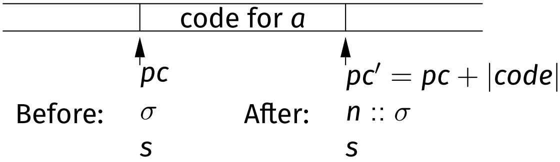
The compiler verification problem

Prove that the compiler preserves semantics: 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 s,



A formal claim along these lines:

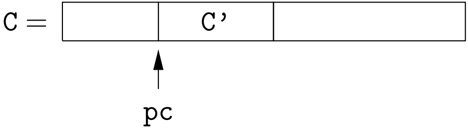
```
Lemma compile_aexp_correct:
  forall s a pc stk,
  transitions (compile_aexp a)
       (0, stk, s)
       (codelen (compile_aexp a), aeval s a :: stk, s).
```

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 `code_at C pc C'` capturing the following situation:



Verifying the compilation of expressions

```
Lemma compile_aexp_correct:
  forall C s a pc stk,
  code_at C pc (compile_aexp a) ->
  transitions C
       (pc, stk, s)
       (pc + codelen (compile_aexp a), aeval st a :: stk, s).
```

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

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

We have the following sequence of transitions:

$$
\begin{array}{c}
(pc, \sigma, s) \\
\downarrow * \ \text{ind. hyp. on } a_1 \\
(pc + |code(a_1)|, \texttt{aeval}\ s\ a_1 :: \sigma, s) \\
\downarrow * \ \text{ind. hyp. on } a_2 \\
(pc + |code(a_1)| + |code(a_2)|, \texttt{aeval}\ s\ a_2 :: \texttt{aeval}\ s\ a_1 :: \sigma, s) \\
\downarrow \quad \texttt{Iadd}\ \text{transition} \\
(pc + |code(a_1)| + |code(a_2)| + 1, (\texttt{aeval}\ s\ a_1 + \texttt{aeval}\ s\ a_2) :: \sigma, s)
\end{array}
$$

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)=0)≡TT;

TRY 1 INDUCT 56;
 TRY 1 SIMPL;
 LABEL INDHYP;
 TRY 2 ABSTR;
  TRY 1 CASES wfsefun(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 type 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 s b d1 d0 pc stk,
  code_at C pc (compile_bexp b d1 d0) ->
  transitions C
       (pc, stk, s)
       (pc + codelen (compile_bexp b d1 d0)
           + (if beval s b then d1 else d0), stk, s).
```

Proof: induction on the structure of `b`.

Verifying the compilation of commands

```
Lemma compile_com_correct_terminating:
  forall s c s',
  cexec s c s' ->
  forall C pc stk,
  code_at C pc (compile_com c) ->
  transitions C
      (pc, stk, s)
      (pc + codelen (compile_com c), stk, s').
```

An induction on the structure of `c` fails because of the `WHILE` case.
An induction on the derivation of `cexec s c s'` 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 s c s',
  cexec s c s' ->
  machine_terminates (compile_program c) s s'.
```

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

What could have we missed?

```
Theorem compile_program_correct_terminating:
  forall s c s',
  cexec s c s' ->
  machine_terminates (compile_program c) s s'.
```

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

What if the program `c` started in `s` 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).

V

Notions of semantic preservation

Semantic preservation

We've claimed that compilers should "preserve semantics" or "produce code that executes in accordance with the semantics of the source program".

What does this mean, exactly?

What should be preserved? Answer: observable behaviors

How to characterize preservation? Answer: simulations

Observable behaviors

For classroom languages, observable behaviors are, typically:

- Normal termination, with final value or final state.
- Divergence, a.k.a. nontermination.
- Abnormal termination, a.k.a. “going wrong”, “crashing”, …

For more realistic languages, we also observe

- Inputs and outputs,
  for example as a trace of I/O actions performed.

Examples of behaviors

	Normal termination	Divergence	Going wrong
IMP	`x := 1` (result: store $[x \mapsto 1]$)	`while true do skip done`	impossible
VM	`Ihalt` (result: initial store)	`Ibranch(-1)`	`Iadd`
λ-calculus with constants	$(\lambda x.x)\ 0$ (result: 0)	$(\lambda x.x\ x)(\lambda x.x\ x)$	$0\ 1$
C	`return 0;`	`for(;;) { }`	`*NULL = 42;`

Notions of preservation: Bisimulation

Definition (Bisimulation)

The source program S and the compiled program C have exactly the same behaviors.

- Every possible behavior of S is a possible behavior of C.
- Every possible behavior of C is a possible behavior of S.

Example (for the IMP to VM compiler)

`compile_com`(c) terminates if and only if c terminates
(with the same final store)
`compile_com`(c) diverges if and only if c diverges.
`compile_com`(c) never goes wrong.

Forward simulation

Definition (Forward simulation)

Every possible behavior of the source program S is a possible behavior of the compiled program C.

Example (for the IMP to VM compiler)

If c terminates, `compile_com`(c) terminates with the same final store.
(theorem `compile_com_correct_terminating`)
If c diverges, `compile_com`(c) diverges.

This looks insufficient: what if the compiled code C has more behaviors than the source S? For example, if C can terminate or go wrong?

Forward simulation + determinism = bisimulation

A language is deterministic if every program has only one observable behavior.

Lemma

If the target language is deterministic, forward simulation implies backward simulation and therefore bisimulation.

Proof.

Let C be a compiled program and S its source.
Let b be a behavior of C and b' a behavior of S.
By forward simulation, b' is a behavior of C.
By determinism of C, $b' = b$.
Hence every behavior b of C is a behavior of S. □

Reducing non-determinism during compilation

If the source language has internal nondeterminism, forward simulation may not hold. For example, the C language leaves 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). Forward simulation and bisimulation fail.

Backward simulation, a.k.a. refinement

Definition (Backward simulation)

Every possible behavior of the compiled program C is a possible behavior of the source program S. However, C may have fewer behaviors than S.

Backward simulation suffices to show the preservation of properties established by source-level verification:

If all behaviors of S satisfy a specification *Spec*,
then all behaviors of C satisfy *Spec* as well.

Should “going wrong” behaviors be preserved?

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

$\mathtt{x} := 1 / \mathtt{y};\ \mathtt{x} := 42$	optimized to	$\mathtt{x} := 42$
(goes wrong if $\mathtt{y} = 0$)		(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”. (This is what the C standards say.)

Simulations for safe programs

Safe forward simulation: any behavior of the source program S other than "going wrong" is a possible behavior of the compiled code C.

Safe backward simulation: for any behavior b of the compiled code C, the source program S can either have behavior b or go wrong.

Small-step semantics based on transition systems

For many languages we have semantics presented in small-step operational style, as a transition relation $a \rightarrow a'$

- machine languages (real or virtual, e.g. our VM)
- lambda-calculi
- process calculi (with labeled transitions $a \xrightarrow{\tau} a'$).

Transition systems

Behaviors are defined in terms of sequences of transitions:

- Termination: finite sequence of transitions to a final state.

$$a \to a_1 \to \cdots \to a_n \in \mathit{Final}$$

- Divergence: infinite sequence of transitions.

$$a \to a_1 \to \cdots \to a_n \to \cdots$$

- Going wrong: finite sequence of transitions to a state that cannot make a transition and is not final

$$a \to a_1 \to \cdots \to a_n \not\to \ \text{with } a_n \notin \mathit{Final}$$

Simulation diagrams

Forward simulation from a source S to a compiled code C can be proved as follows:

Show that every transition in the execution of S

- is simulated by some transitions in C
- while preserving a relation between the states of S and C.

(Backward simulation is similar, but simulates transitions of C by transitions of S.)

Lock-step simulation

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

$$\begin{tikzcd} s_1 \arrow[r, dash, "\approx"] \arrow[d] & c_1 \arrow[d, red] \\ s_2 \arrow[r, dash, red, "\approx"] & \color{red}{c_2} \end{tikzcd}$$

(Black = hypotheses; red = conclusions.)

Lock-step simulation

Further show that initial configurations are related:

$$s_{init} \approx c_{init}$$

Further show that final configurations are related:

$$s \approx c \;\wedge\; s \in \mathit{Final} \implies c \in \mathit{Final}$$

Lock-step simulation

Forward simulation follows easily:

$$
\begin{tikzcd}
s_{init} \arrow[r, dash, "\approx"] \arrow[d] & c_{init} \arrow[d] \\
s_1 \arrow[r, dash, "\approx"] \arrow[d, dotted, no head] & c_1 \arrow[d, dotted, no head] \\
{} \arrow[r, dash, "\approx"] \arrow[d] & {} \arrow[d] \\
\mathit{Final} \ni s_n \arrow[r, dash, "\approx"] & c_n \in \mathit{Final}
\end{tikzcd}
$$

Likewise if s_{init} makes an infinity of transitions.

“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 `ASSIGN` x a consists of several instructions.)

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[d] & c_1 \arrow[d, "+"] \\
s_2 \arrow[r, dash, "\approx"] & c_2
\end{tikzcd}$$

Forward simulation still holds.

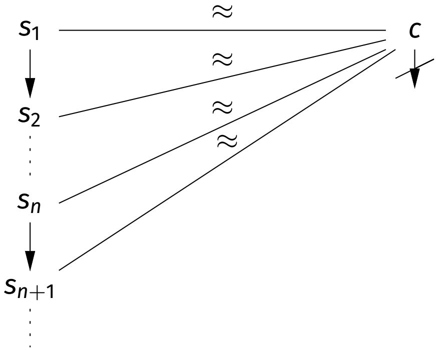
“Star” simulation diagrams (incorrect)

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

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[d] & c_1 \arrow[d, "*"] \\
s_2 \arrow[r, dash, "\approx"] & c_2
\end{tikzcd}$$

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.

This denotes an incorrect optimization of diverging programs, e.g. adding a special case `compile_com (WHILE TRUE SKIP) = nil`.

“Star” simulation diagrams (corrected)

Find a measure $M(s)$: `nat` over source terms that decreases strictly when a stuttering step is taken. Then show:

$$\begin{tikzcd} s_1 \arrow[r, dash, "\approx"] \arrow[d] & c_1 \arrow[d, "+"] \\ s_2 \arrow[r, dash, "\approx"] & c_2 \end{tikzcd} \quad \textbf{or} \quad \begin{tikzcd} s_1 \arrow[r, dash, "\approx"] \arrow[d] & c_1 \arrow[dl, dash, "\approx"'] \\ s_2 & \text{and } M(s_2) < M(s_1) \end{tikzcd}$$

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

Forward simulation, diverging case: OK.
(If s diverges, it must perform infinitely many non-stuttering steps, so the compiled code executes infinitely many transitions.)

(Note: can use any well-founded ordering between source terms s.)

The next steps

Equip IMP with a small-step semantics.

Prove a forward simulation diagram (of the "star" kind) between IMP transitions and VM transitions.

Conclude that all IMP programs, terminating or not, are correctly compiled.

VI

Small-step semantics for IMP

A reduction semantics for IMP

Broadly similar to β-reduction in the λ-calculus:

- $M \xrightarrow{\beta} M'$ represents an elementary computation.
- M' is the residual: it represents all the other computations that remain to be done

Since IMP is an imperative language, we reduce not commands but pairs c/s of a command c and the current store s.

The reduction relation is, therefore: $c/s \to c'/s'$.

A reduction semantics for IMP

$$x := a \ / \ s \to \texttt{skip} \ / \ s[x \leftarrow [\![a]\!] \ s]$$

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

$$\frac{[\![b]\!] \ s = \texttt{true}}{(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2) \ / \ s \to c_1/s} \qquad \frac{[\![b]\!] \ s = \texttt{false}}{(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2) \ / \ s \to c_2/s}$$

$$\frac{[\![b]\!] \ s = \texttt{false}}{(\texttt{while } b \texttt{ do } c \texttt{ done}) \ / \ s \to \texttt{skip}/s}$$

$$\frac{[\![b]\!] \ s = \texttt{true}}{(\texttt{while } b \texttt{ do } c \texttt{ done}) \ / \ s \to (c; \texttt{while } b \texttt{ do } c \texttt{ done}) \ / \ s}$$

Equivalence with the big-step semantics

A classic result:

$$c/s \Rightarrow s' \qquad \text{if and only if} \qquad c/s \xrightarrow{*} \texttt{skip}/s'$$

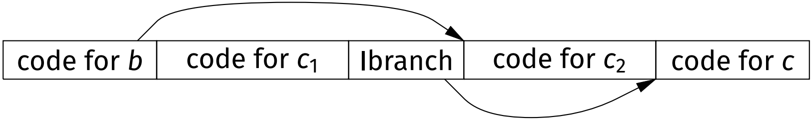
(See Coq file `IMP.v`.)

Spontaneous generation of commands

IMP reductions, like β-reduction in the λ-calculus, can create commands that are "fresh", that is, not sub-terms of the original program:

$$((\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2); c)/s \to (c_1; c)/s$$

This is problematic for compiler verification because the compiled code does not change during execution! The compiled code for the initial command $(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2); c$



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

code for c_1	code for c

A transition semantics with continuations

A variant of reduction semantics that avoids the spontaneous generation of commands.

Idea: instead of rewriting whole commands:

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

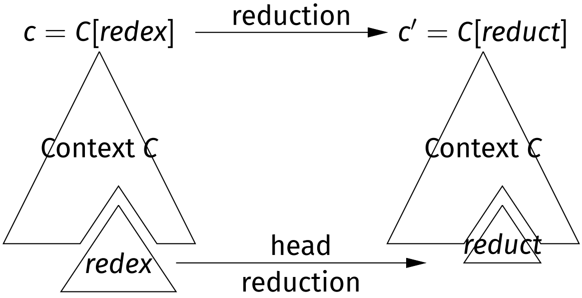
rewrite pairs of (subcommand under focus, remainder of command):

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

(Very related to continuation-based abstract machines such as the CEK.)
(Also related to focusing in proof theory.)

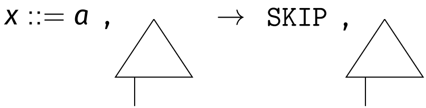
Standard reduction semantics

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

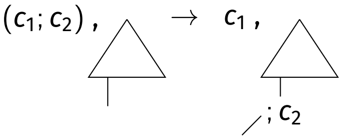


Focusing the reduction semantics

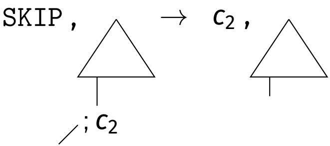
Rewrite pairs (subcommand, context in which it occurs).



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

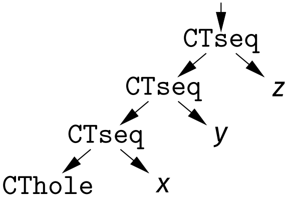


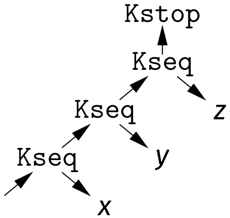
Resuming a sequence

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 ≈ continuation.
(“Eventually, do x, then do y, then do z, then stop.”)

Transition rules

$$
\begin{array}{rcll}
x := a/k/s & \to & \mathtt{skip}/k/s[x \leftarrow [\![a]\!]\, s] & \\
(c_1; c_2)/k/s & \to & c_1/\mathtt{Kseq}\ c_2\ k/s & \\
\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2/k/s & \to & c_1/k/s & \text{if } [\![b]\!]\, s = \mathtt{true} \\
\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2/k/s & \to & c_2/k/s & \text{if } [\![b]\!]\, s = \mathtt{false} \\
\mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{end}/k/s & \to & c/\mathtt{Kwhile}\ b\ c\ k/s & \\
 & & \text{if } [\![b]\!]\, s = \mathtt{true} & \\
\mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{end}/k/s & \to & \mathtt{skip}/c/k & \text{if } [\![b]\!]\, s = \mathtt{false} \\
\mathtt{skip}/\mathtt{Kseq}\ c\ k/s & \to & c/k/s & \\
\mathtt{skip}/\mathtt{Kwhile}\ b\ c\ k/s & \to & \mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{done}/k/s &
\end{array}
$$

Note: no spontaneous generation of fresh commands.

VII

Full proof of compiler correctness

A proof by simulation diagram

Let's build a forward simulation diagram between source transitions (in the continuation-based semantics of IMP) and machine transitions.

This will show behavior preservation both for terminating IMP programs (we already proved this) and for diverging IMP programs (new!).

Since the machine has deterministic semantics, we will get full bisimulation between the source and compiled code.

Two difficulties:

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

Anti-stuttering measure

Stuttering reduction $=$ no machine instruction executed. These include:

$$
\begin{array}{rcl}
(c_1; c_2)/k/s & \rightarrow & c_1/\mathtt{Kseq}\ c_2\ k/s \\
\mathtt{SKIP}/\mathtt{Kseq}\ c\ k/s & \rightarrow & c/k/s \\
(\mathtt{IFTHENELSE}\ \mathtt{TRUE}\ c_1\ c_2)/k/s & \rightarrow & c_1/k/s \\
(\mathtt{WHILE}\ \mathtt{TRUE}\ c)/k/s & \rightarrow & c/\mathtt{Kwhile}\ \mathtt{TRUE}\ c\ k/s
\end{array}
$$

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

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

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

Anti-stuttering measure

After some trial and error, an appropriate measure is:

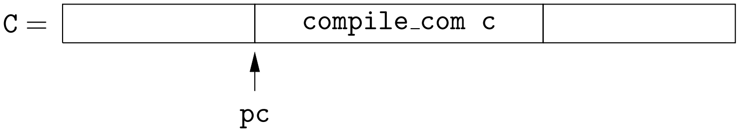
$$M(c, k) = size(c) + \sum_{c' \text{ appears in } k} size(c')$$

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

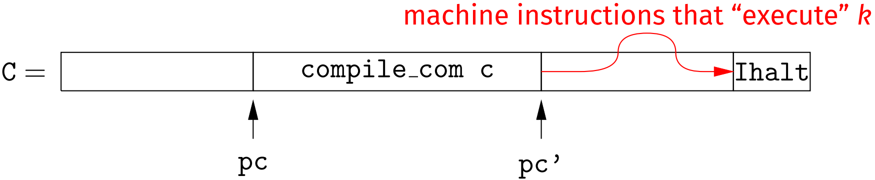
$$\begin{array}{rcl}
M((c_1; c_2), k) & = & M(c_1, \texttt{Kseq}\ c_2\ k) + 1 \\
M(\texttt{SKIP}, \texttt{Kseq}\ c\ k) & = & M(c, k) + 1 \\
M(\texttt{IFTHENELSE}\ b\ c_1\ c_2, k) & \geq & M(c_1, k) + 1 \\
M(\texttt{WHILE}\ b\ c, k) & = & M(c, \texttt{Kwhile}\ b\ c\ k) + 1
\end{array}$$

Relating continuations with compiled code

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



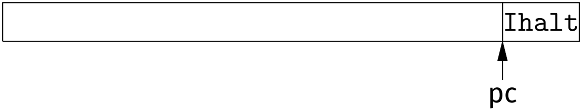
In a proof based on the small-step continuation semantics: we must also relate continuations *k* with the compiled code:



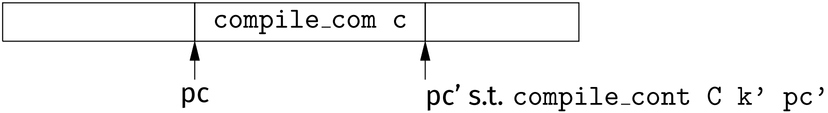
Relating continuations with compiled code

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`:

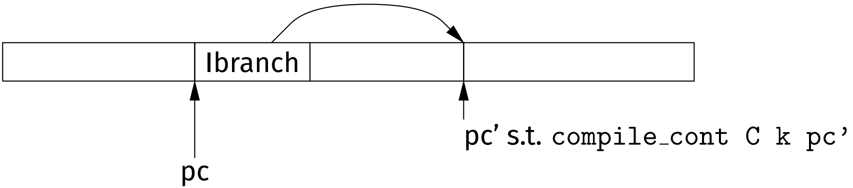


Sequence case $k = $ `Kseq` c k':

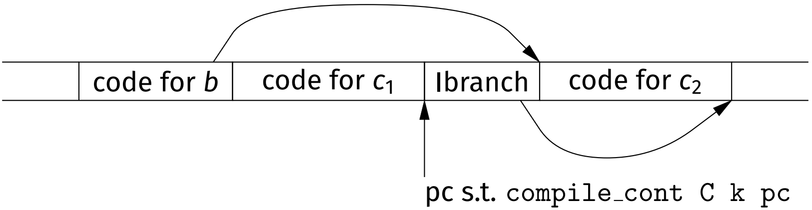


Relating continuations with compiled code

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



Useful to handle continuations arising out of IFB b THEN c_1ELSE 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

$$
\begin{tikzcd}
c_1/k_1/s_1 \arrow[r, dash, "C \vdash c_1/k_1/s_1 \approx (pc_1, \epsilon, s_1)"] \arrow[d] & (pc_1, \epsilon, s_1') \arrow[d, "+ \lor * \land M(c_2, k_2) < M(c_1, k_1)"] \\
c_2/k_2/s_2 \arrow[r, dash, "C \vdash c_2/k_2/s_2 \approx (pc_2, \epsilon, s_2)"'] & (pc_2, \epsilon, s_2')
\end{tikzcd}
$$

Proof: by copious 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 $c/\texttt{Kstop}/s \xrightarrow{*} \texttt{SKIP}/\texttt{Kstop}/s'$
  then $\texttt{machine\_terminates}\ (\texttt{compile\_program}\ c)\ s\ s'$

- A proof of compiler correctness for diverging programs:
  if $c/\texttt{Kstop}/s$ reduces infinitely,
  then $\texttt{machine\_diverges}\ (\texttt{compile\_program}\ c)\ s$

Mission accomplished!

VIII

An optimization: constant propagation

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.:

$$\begin{aligned} \mathtt{x} * 2 &\;\to\; \mathtt{x} + \mathtt{x} \\ \mathtt{x} * 4 &\;\to\; \mathtt{x} << 2 \end{aligned}$$

Most others apply only if some conditions are met:

$$\begin{array}{rcll} \mathtt{x} \,/\, 4 & \to & \mathtt{x} >> 2 & \text{only if } \mathtt{x} \geq 0 \\ \mathtt{x} + 1 & \to & 1 & \text{only if } \mathtt{x} = 0 \\ \mathtt{if}\ \mathtt{x} < \mathtt{y}\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 & \to & c_1 & \text{only if } \mathtt{x} < \mathtt{y} \\ \mathtt{x} := \mathtt{y} + 1 & \to & \mathtt{skip} & \text{only if } \mathtt{x} \text{ unused later} \end{array}$$

$\to$ 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:

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

Requirements:

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

Running example: constant propagation

Perform at compile-time all arithmetic operations involving known quantities, e.g. constants, or variables whose values are known at compile-time.

Examples: (`x` is unknown)

```
a = 1 + 2;                         a = 3;
b = a - 4;           ---->         b = -1;
c = (x + 1) + 2;                   c = x + 3;
d = (x - 1) + a;                   d = x + 2;
```

Acieved by a combination of

- local, algebraic simplifications of expressions;
- global, static analysis to keep track of the values of variables.

Algebraic simplifications

Many algebraic identities can be used to make expressions simpler. The problem is to find a good strategy for applying them.

Example: using associativity and commutativity to bring constants together.

$$
\begin{aligned}
simp((a + N) + M) &= simp(a + (N + M)) \\
simp((N + a) + M) &= simp(a + (N + M)) \\
simp(M + (a + N)) &= simp(a + (N + M)) \\
simp(M + (N + a)) &= simp(a + (N + M)) \\
simp(a + b) &= simp(a) + simp(b)
\end{aligned}
$$

There are many patterns for the same simplification.
Recursive calls to *simp* are not structurally decreasing.

Smart constructors

An effective strategy based on bottom-up rewriting and smart constructors: functions that

- look like constructors of the AST

  ```
  mk_PLUS: aexp -> aexp -> aexp
  ```

- are proved to have the same semantics as a constructor

  ```
  aeval s (mk_PLUS a1 a2) = aeval s a1 + aeval s a2
  ```

- normalize the shape of generated expressions, e.g. `mk_PLUS` will never return `PLUS (CONST n) a`, returningn `PLUS a (CONST n)` instead
- perform simplifications "on the fly", e.g.

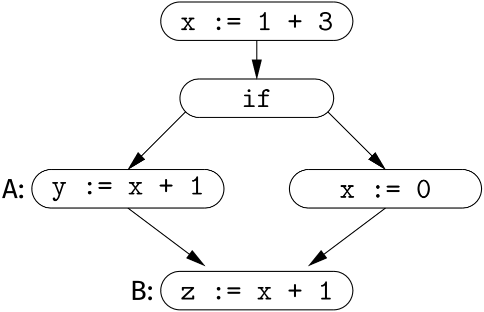
  ```
  mk_PLUS (PLUS a (CONST n)) (CONST m) = PLUS a (CONST (n+m))
  ```

(See Coq file `Constprop.v`.)

Static analysis: the dataflow view

(the traditional presentation in compiler textbooks)

Connect definitions and uses of variables in the control-flow graph so as to exploit, at use sites, properties established at definition sites (or conversely).



At use point A, only one definition of `x` reaches: `x = 4`.
At use point B, two incompatible definitions reach: `x = 4` and `x = 0`.

Static analysis: the abstract interpretation view

Execute ("interpret") the program using a non-standard semantics that:

- Computes over an abstract domain of the desired properties (e.g. "$x = N$" for constant propagation; "$x \in [n_1, n_2]$" for interval analysis) instead of concrete "things" like values and states.
- Handles boolean conditions, even if they cannot be resolved statically.
  (`then` and `else` branches of `if` are considered both taken.)
  (`while` loops execute arbitrarily many times.)
- Always terminates.

Abstract domains for constant propagation

Abstract integers (type `option Z`):

  `Some` *n* if statically known, `None` if unknown

Abstract Booleans (type `option bool`):

  `Some` *b* if statically known, `None` if unknown

Abstract stores (type `Store`):

  morally a function `ident -> option Z`

  for algorithmic reasons, a finite partial map from `ident` to `Z`

    (variables not represented are mapped to `None`)

The abstract evaluation functions

Evaluating arithmetic and Boolean expressions using abstract integers and abstract Booleans:

```
Aeval: Store -> aexp -> option Z
Beval: Store -> bexp -> option bool
```

Executing a command in the abstract. Input: the abstract store "before" execution. Output: the abstract store "after".

```
Cexec: Store -> com -> Store
```

(See Coq `Constprop.v`.)

Analyzing conditionals

```
Fixpoint Cexec (S: Store) (c: com) : Store :=
  match c with
  ...
  | IFTHENELSE b c1 c2 =>
      match Beval S b with
      | Some true => Cexec S c1
      | Some false => Cexec S c2
      | None => Join (Cexec S c1) (Cexec S c2)
      end
```

If the condition `b` is statically known, we known which branch `c1` or `c2` will always be executed, and analyze only this branch.

Otherwise, either branch can be taken at run-time, so we analyze both and take the join of the resulting abstract stores.

`Join s1 s2` maps *x* to a known value *n* only if `s1` and `s2` map *x* to *n*.

Analyzing loops

```
Fixpoint Cexec (S: Store) (c: com) : Store :=
  match c with
  ...
  | WHILE b c =>
      fixpoint (fun x => Join S (Cexec x c)) S
```

Let `X` be the abstract store at the beginning of the loop body `c`.

- On the first iteration, we enter `c` with abstract store `S`. Hence, $\mathtt{S} \sqsubseteq \mathtt{X}$
- On later iterations, we enter `c` with abstract store `Cexec X c` coming from the previous iteration. Hence, $\mathtt{Cexec\ X\ c} \sqsubseteq \mathtt{X}$.

The usual way to solve for `X` is to compute a post-fixpoint of the function

$$F \stackrel{\text{def}}{=} \lambda X.\, \mathtt{S} \sqcup \mathtt{Cexec}\ X\ \mathtt{c}$$

i.e. an X such as $F(X) \sqsubseteq X$.

The mathematician's approach to fixpoints

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

Theorem (Knaster-Tarski)

The sequence

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

converges to the smallest fixpoint of F, provided that

- *F is increasing:* $x \leq y \Rightarrow F(x) \leq F(y)$.
- $\perp$ *is a smallest element.*
- *There are no infinite, strictly ascending chains* $x_0 < x_1 < \ldots < x_n < \ldots$

This provides an effective way to compute the smallest post-fixpoint, but is difficult to implement in Coq. We'll attempt this in the next lecture. In the meantime...

The engineer's approach to fixpoints

$$F = \lambda X.\ \mathtt{S} \sqcup \mathtt{Cexec}\ X\ \mathtt{c}$$

- Compute $F(S), F(F(S)), \ldots, F^N(S)$ up to some fixed N.
- Stop as soon as a pre-fixpoint is found ($F^{i+1}(S) \sqsubseteq F^i(S)$).
- Otherwise, return a safe over-approximation: $\top$
  (the abstract store that maps all variables to "unknown").

A compromise between analysis time and analysis precision.

(Coq implementation: see function `fixpoint` in `Constprop.v`.)

The code transformation

The results of the analysis are used to optimize expressions by

- replacing a variable `VAR` x by `CONST` n if x is mapped to n in the abstract store
- further simplify the expression by applying the smart constructors.

$$\texttt{x + (1 + y)} \longrightarrow \texttt{3 + (1 + y)} \longrightarrow \texttt{y + 4}$$

Within commands, all expressions are optimized, then conditionals and loops can be simplified if their conditions are statically known:

$$\begin{aligned}
\texttt{IFTHENELSE TRUE c1 c2} &\longrightarrow \texttt{c1} \\
\texttt{IFTHENELSE FALSE c1 c2} &\longrightarrow \texttt{c2} \\
\texttt{WHILE FALSE c} &\longrightarrow \texttt{SKIP}
\end{aligned}$$

(Coq development: functions `cp_aexp`, `cp_bexp`, `cp_com`.)

Correctness proof

The soundness of the static analysis is expressed in terms of "matching" between concrete stores `s` arising during execution and abstract stores `S` inferred by the analysis:

```
Definition matches (s: store) (S: Store) : Prop :=
  forall x n, IdentMap.find x S = Some n -> s x = n.
```

In abstract interpretation terms, this is the γ concretization function: `matches s S` means that $\mathtt{s} \in \gamma(\mathtt{S})$.

Correctness proof

The two main results: if `cexec s1 c s2` and `matches s1 S`, then

- Soundness of the analysis: `matches s2 (Cexec S c)`
  (the final concrete store matches the prediction of the analysis)

- Semantic preservation for the code transformation:
  `cexec s1 (cp_com S c) s2`
  (the optimized code terminates on the same final store).

IX

More about fixpoints

Back to the mathematician's approach

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:* $x \le y \Rightarrow F(x) \le F(y)$.
- $\bot$ *is a smallest element.*
- *There are no infinite, strictly ascending chains* $x_0 < x_1 < \ldots < x_n < \ldots$

Can we formalize and prove this result in Coq?

In a way that is computationally effective and provides a "fixpoint calculator" that we can use in a static analysis?

The ascending chain condition

There are no infinite, strictly ascending chains
$x_0 < x_1 < \ldots < x_n < \ldots$

Too many negatives! Let's reformulate more positively:

All strictly ascending chains are finite:
$x_0 < x_1 < \ldots < x_n \not<$

Getting closer…

An element x is accessible if all strictly ascending chains starting with x are finite: $x < x_1 < \ldots < x_n \not<$.

An order $<$ is well-founded if all x are accessible.

Well-founded orders in type theory

Key insight: the “is accessible” predicate is inductive by nature!

- x is accessible iff all $y > x$ are accessible.
- This rule must be applied a finite number of times only.

```
Section Well_founded.

 Variable A : Type.
 Variable R : A -> A -> Prop.

 Inductive Acc (x: A) : Prop :=
     Acc_intro : (forall y:A, R y x -> Acc y) -> Acc x.

 Definition well_founded := forall a:A, Acc a.
```

Structural induction on a derivation of `Acc`(x) is Noetherian induction!
(“To prove $P(x)$ you can assume $P(y)$ for all y such that `R` y x”)

From Knaster-Tarski to effective fixpoint computation

Noetherian induction can prove the existence of a fixpoint:

```
exists x : A, eq x (F x)
```

Replacing `Prop` with `Type`, the proof shows that an `x` that is a fixpoint can be effectively computed:

```
{ x : A | eq x (F x) }
```

Alternate approach: use `Program Fixpoint` to write explicitly the fixpoint iteration algorithm, dropping into proof mode to fill in the necessary proof terms.

(See file `Fixpoints.v`)

Using the new fixpoint for constant analysis

The type of abstract states (finite maps) has the ascending chain property. So, we should be able to drop the new fixpoint function in the analysis of commands:

```
Fixpoint Cexec (S: Store) (c: com) : Store :=
  match c with
  | SKIP => S
  | ASSIGN x a => update' x (Aeval S a) S
  | SEQ c1 c2 => Cexec (Cexec S c1) c2
  | IFTHENELSE b c1 c2 => [...]
  | WHILE b c1 =>
      fixpoint (fun x => Join S (Cexec x c1)) S
  end.
```

Problem: our new fixpoint applies to increasing functions only. But we haven't proved yet that `Cexec` is increasing!

Using the new fixpoint for constant analysis

The solution is to define the static analysis function and simultaneously prove that it is increasing!

```
Program Fixpoint Cexec (c: com) :
                     { F: Store -> Store | increasing' F } :=
  match c with
  | SKIP => fun S => S
  | ASSIGN x a => fun S => update' x (Aeval S a) S
  | SEQ c1 c2 => compose (Cexec c2) (Cexec c1)
  | IFTHENELSE b c1 c2 => fun S => [...]
  | WHILE b c1 =>
      fun S => fixpoint_join S (fun S => Cexec c1 S) _
  end.
```

Many proof obligations related to monotonicity are generated, but it works in the end.

X

Liveness analysis and dead code elimination

Dead code elimination

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.

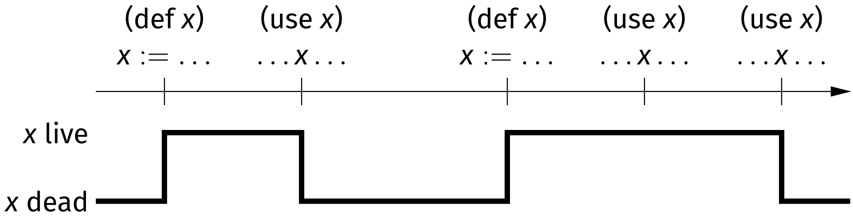
Notions of liveness

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.

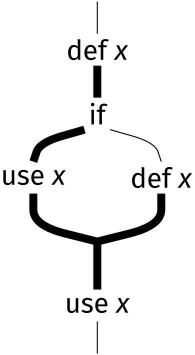
A variable is live if it is not dead.

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.

Note: this is a "backward" analysis that does not fit the abstract interpretation framework.

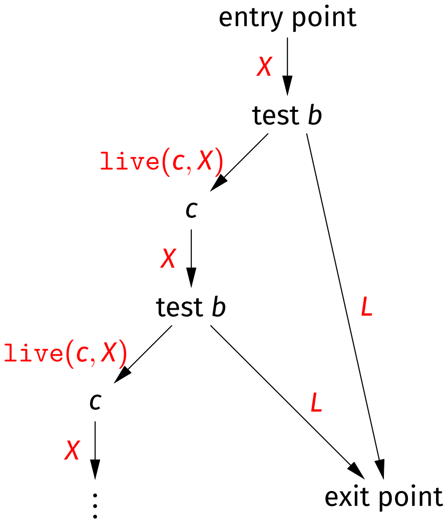
Liveness equations

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.

$$
\begin{array}{rcl}
\texttt{live}(\texttt{SKIP}, L) & = & L \\
\texttt{live}(x := a,\ L) & = & \begin{cases} (L \setminus \{x\}) \cup FV(a) & \text{if } x \in L; \\ L & \text{if } x \notin L. \end{cases} \\
\texttt{live}((c_1; c_2),\ L) & = & \texttt{live}(c_1, \texttt{live}(c_2, L)) \\
\texttt{live}((\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2),\ L) & = & FV(b) \cup \texttt{live}(c_1, L) \cup \texttt{live}(c_2, L) \\
\texttt{live}((\texttt{while } b \texttt{ do } c \texttt{ done}),\ L) & = & X \text{ such that} \\
& & X \supseteq L \cup FV(b) \cup \texttt{live}(c, X)
\end{array}
$$

The `while` case is solved by taking a fixpoint. See file `Deadcode.v`.

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)

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 $\texttt{dce} : \texttt{com} \rightarrow \texttt{IdentSet.t} \rightarrow \texttt{com}$
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…

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!

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 two initial states:

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

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

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

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

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

The proof of semantic preservation for dead-code elimination follows this pattern, relating executions of c and `dce` c L instead.

Agreement and its properties

```
Definition agree (L: IdentSet.t) (s1 s2: state) : Prop :=
  forall x, IdentSet.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 -> IdentSet.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, IdentSet.Subset (fv_aexp a) L -> aeval s1 a = aeval s2
Lemma beval_agree:
  forall L s1 s2, agree L s1 s2 ->
  forall b, IdentSet.Subset (fv_bexp b) L -> beval s1 b = beval s2
```

Agreement and its properties

Agreement is preserved by parallel assignment to a variable:

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

Forward simulation for dead code elimination

```
Theorem dce_correct_terminating:
  forall s c s', cexec s c s' ->
  forall L s1, agree (live c L) s s1 ->
  exists s1', cexec s1 (dce c L) s1' /\ agree L s' s1'.
```

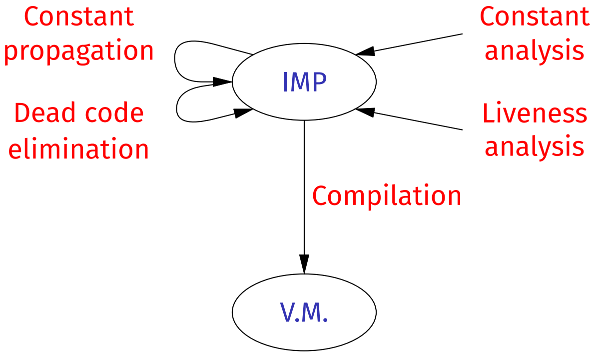
(Proof: an induction on the derivation of `cexec s c s'`.)

$$\begin{tikzcd}
s \arrow[r, dash, "\texttt{agree}\,(\texttt{live}\ c\ L)"] \arrow[d, "\mathrm{eval}\ c"'] & s_1 \arrow[d, "\mathrm{eval}\ (\texttt{dce}\ c\ L)"] \\
s' \arrow[r, dash, "\texttt{agree}\ L"] & s'_1
\end{tikzcd}$$

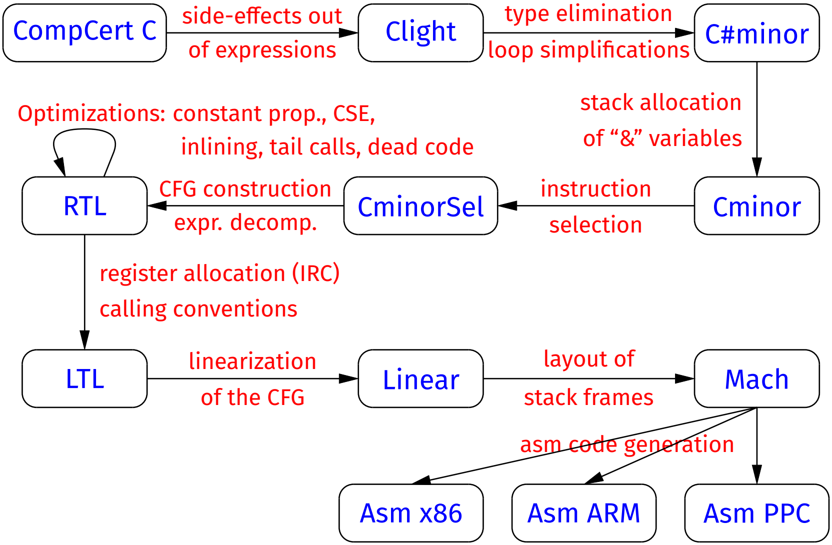
XI

In closing

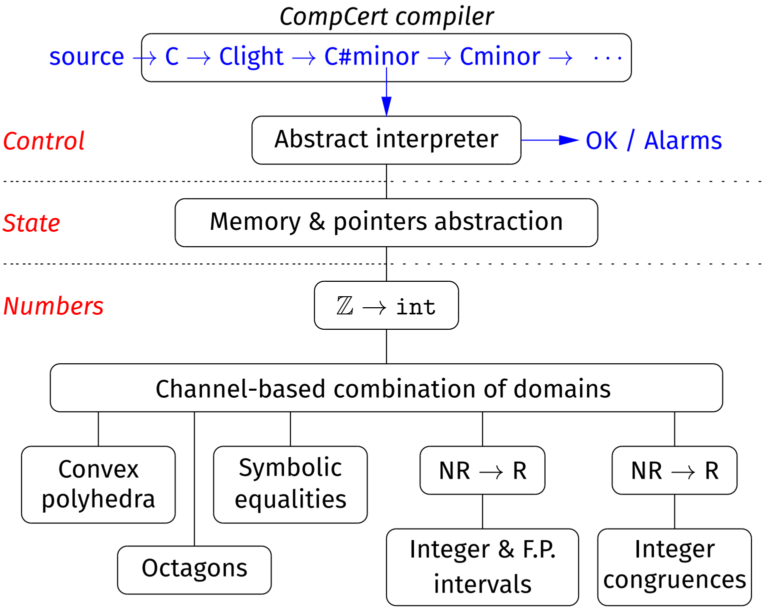
From this lecture…



… to the CompCert verified C compiler …



… and the Verasco verified static analyzer …



…some key ideas scale very well !

- Operational semantics based on transition systems (using continuations to handle structured control).
- Forward simulation diagrams.
- Big-step semantics to help with discovery.
- A “naive abstract interpretation” view of static analyses. (Concretization relations, but no full Galois connections.)
- Bounded fixpoint iterations.
- Programming the analyses and transformations as Coq functions (followed by extraction to executable OCaml code).

Other key ideas not seen in this lecture

For verified compilers: (e.g. CompCert)

- Labeled transition semantics to deal with I/O.
- Other representations of control: control-flow graphs, assembly-style code with labels and jumps.
- Complex, low-level memory model.
- Optimizing memory accesses despite pointers and aliasing.

For verified static analyzers: (e.g. Verasco)

- Modular, compositional construction of abstract domains.
- Relational analyses.
- Fixpoint iteration with widening and narrowing.

Other applications of mechanized semantics

Embedding powerful program logics in a proof assistant, e.g.

- Iris @ MPI SWS and Aarhus
- VST @ Princeton
- the seL4 verification infrastructure @ NICTA / Data61

Verifying properties of testing frameworks, e.g.

- Quickchick @ UPenn (randomized property testing)

In closing

Interactive or automatic theorem provers are taking programming language research to new heights, and producing programming tools that we can really trust.

Go forth and mechanize!