Mechanized semantics, second lecture

Traduttore, traditore: formal verification of a compiler

Xavier Leroy

2019-12-12

Collège de France, chair of software sciences

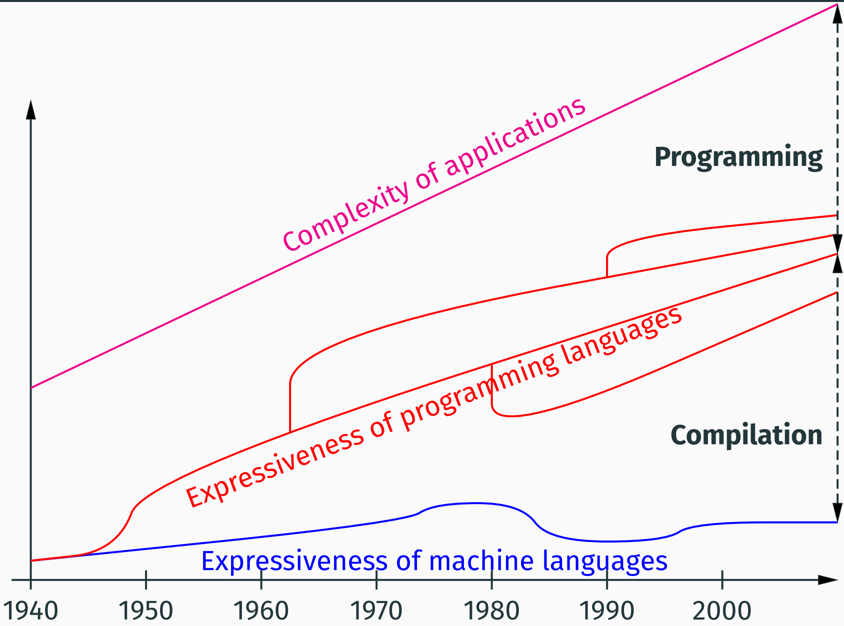
Compilation

Generally speaking: any automated translation from a computer language to another.

More specifically: an automated translation

- from a source language usable by programmers
- to a machine language executable by machines
- paying attention to efficiency: execution speed, code size, energy consumption.

A historical perspective on compilation



The first compilers

1953 The A-0, A-1, A-2 *autocoders* (G. Hopper, Rand Remington)
"I had a running compiler and nobody would touch it because, they carefully told me, computers could only do arithmetic; they could not do programs"

1957 The Fortran 1 *translator* (J. Backus et al, IBM)
First compiler featuring loop optimizations.

1960 First Algol 60 compiler (E. Dijkstra, U. Amsterdam)
Using a stack to implement recursion and call by name.

1962 First Lisp compiler (T. Hart et M. Levin, MIT)
First *bootstrapped* (self-hosting) compiler.

Trends in compilation

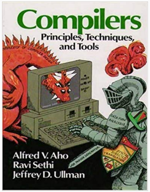
1970's Automatic generation of syntactic analyzers (e.g. `lex` for lexers, `yacc` for parsers).

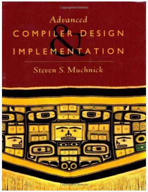
1980's The RISC approach: register allocation, instruction scheduling.

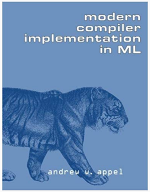
1990's The Static Single Assignment (SSA) intermediate representation.

2000's Dynamic, optimized compilation of scripting languages (JavaScript engines).

Compilation, today







A mature area of computer science.

Large corpus of algorithms for code generation and optimization.

Many compilers (free or closed-source) that implement subtle code transformations.

An example of optimizing compilation

$$\vec{a} \cdot \vec{b} = \sum_{i=0}^{i<n} a_i b_i$$

```
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 with a good optimizing compiler, then manually decompiled back to C.

```
double dotproduct(int n, double a[], double b[]) {
     dp = 0.0;
     if (n <= 0) goto L5;
     r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
     if (r2 > n || r2 <= 0) goto L19;
     prefetch(a[16]); prefetch(b[16]);
     if (4 >= r2) goto L14;
     prefetch(a[20]); prefetch(b[20]);
     f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
     r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
     f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
     f12 = a[4]; f16 = f18 * f16;
     f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
     f11 += f17; r1 += 4; f10 += f15;
     f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
     f1 += f16; dp += f19; b += 4;
     if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
     f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
     f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
     a += 4; b += 4; f14 = a[8]; f15 = b[8];
     f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
     f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
     a += 4; f28 = f29 * f28; b += 4;
     f10 += f14; f11 += f12; f1 += f26;
     dp += f28;  dp += f1; dp += f10; dp += f11;
     if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
     dp += f18;
     if (r1 < n) goto L19;
L5:  return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
```

```
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
     f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
     f12 = a[4]; f16 = f18 * f16;
     f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
     f11 += f17; r1 += 4; f10 += f15;
     f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
     f1 += f16; dp += f19; b += 4;
     if (r1 < r2) goto L17;
```

```
double dotproduct(int n, double a[], double b[]) {
     dp = 0.0;
     if (n <= 0) goto L5;
     r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
     if (r2 > n || r2 <= 0) goto L19;
     prefetch(a[16]); prefetch(b[16]);
     if (4 >= r2) goto L14;
     prefetch(a[20]); prefetch(b[20]);
     f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
     r1 = 8; if (8 >= r2) goto L16;









L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
     f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
     f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
     a += 4; b += 4; f14 = a[8]; f15 = b[8];
     f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
     f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
     a += 4; f28 = f29 * f28; b += 4;
     f10 += f14; f11 += f12; f1 += f26;
     dp += f28;  dp += f1; dp += f10; dp += f11;
     if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
     dp += f18;
     if (r1 < n) goto L19;
L5:  return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
```

The miscompilation risk

Miscompilation: production of wrong executable code from a correct source program.

> *We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables.*
>
> *E. Eide & J. Regehr, EMSOFT 2008*

> *To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input.*
>
> *X. Yang, Y. Chen, E. Eide & J. Regehr, PLDI 2011*

Formal verification of compilers

Compilers are complicated programs, but have a rather simple "end-to-end" specification:

> *The generated code must execute as prescribed by the semantics of the source program.*

This specification becomes mathematically precise as soon as we have formal semantics for the source language and the machine language.

Then, a formal verification of a compiler can be considered.

An old idea…

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

An old idea…

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.

Even proof scripts look familiar!

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;
```

In this lecture

In this lecture, we complete Milner and Weyrauch's agenda: the formal verification (in Coq) of a non-optimizing compiler for a simple imperative language (IMP).

We identify a number of approaches that extend all the way to the verification of compilers for "real-world" languages (CompCert, CakeML).

The next lecture will study code optimizations and their verification.

The IMP virtual machine

Virtual machines

Producing machine code for existing processors (x86, ARM, ...) is delicate.

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

Like a real machine, a virtual machine executes sequences of simple instructions: no compound expressions, no control structures.

The instructions of a virtual machine are not directly executable by hardware, but are chosen to match the base operations of the source language.

The IMP virtual machine

Four components:

- The code C: a list of instructions.
- The code pointer pc: an integer giving the position of the currently-executing instruction in C.
- The store s: associating a value to each variable.
- The stack σ: a list of integer values
  (used to save intermediate results)

(Inspired by old HP pocket calculators and by the Java Virtual Machine.)

The instruction set

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

All instructions increment *pc* by 1, except branch instructions, which increment *pc* by $1 + \delta$.

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

Example

pile	ε	12	1 12	13	ε
état	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 13$
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

Defined in operational style as a transition relation representing the execution of one instruction.

The instruction being executed is the one from code C at position pc.

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

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

(See Coq file `Compil.v`.)

Execution of a program

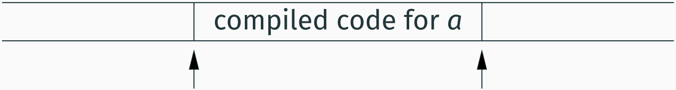
As a sequence of transitions:

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

The compiler

Compiling arithmetic expressions

Contract: if *a* evaluates to value *n* in store *s*,



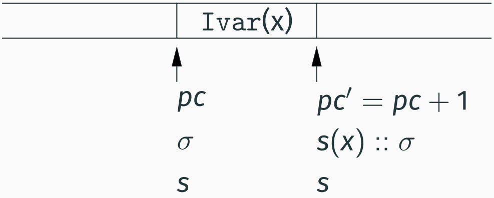
	pc		$pc' = pc + \lvert code \rvert$
Before:	σ	After:	$n :: \sigma$
	s		s

Compilation is translation to “reverse Polish notation”.

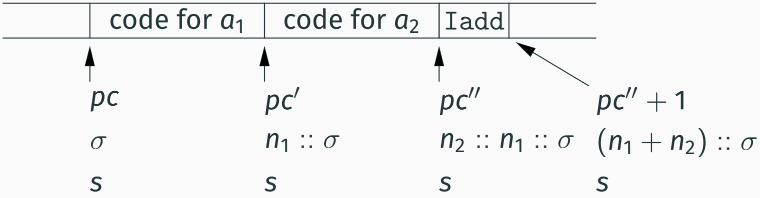
(Coq function: `compile_aexp`)

Compiling arithmetic expressions

A base case: if $a = x$,

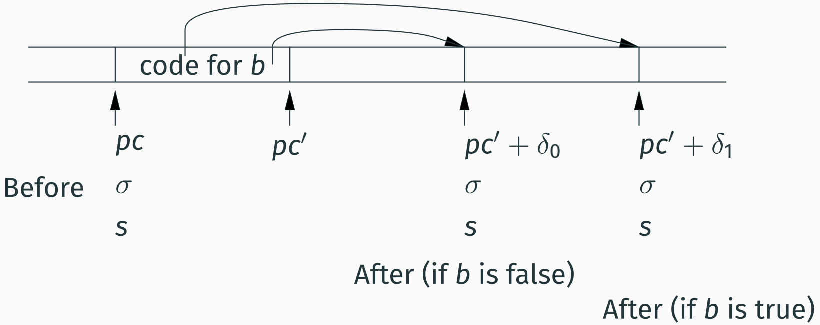


A recursive case: if $a = a_1 + a_2$,



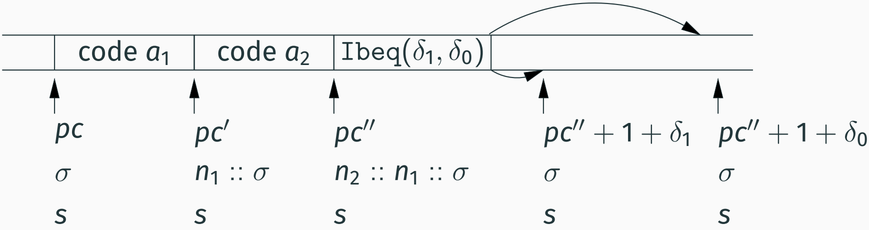
Compiling Boolean expressions

Contract: `compile_bexp` b δ_1 δ_0 should
  skip δ_1 instructions if b evaluates to true
  skip δ_0 instructions if b evaluates to false.



Compiling Boolean expressions

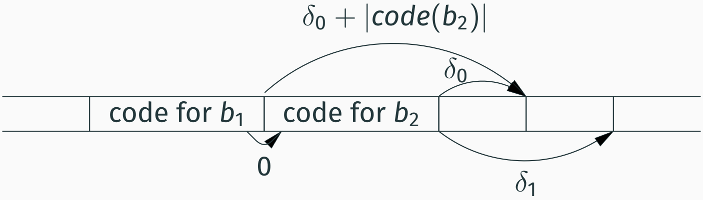
A base case: $b = (a_1 = a_2)$



Short-circuit evaluation of Boolean "and"

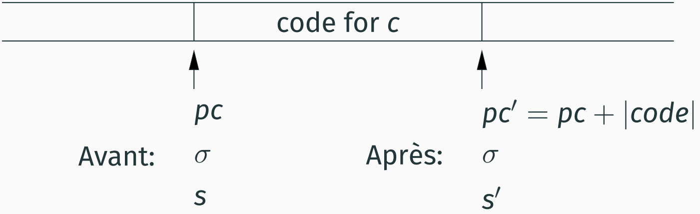
If b_1 evaluates to false, b_1 `and` b_2 evaluates to false as well: no need to evaluate b_2!

Therefore, if b_1 is false, the compiled code for b_1 `and` b_2 can skip the code for b_2 and jump directly to the expected target.



Compiling commands

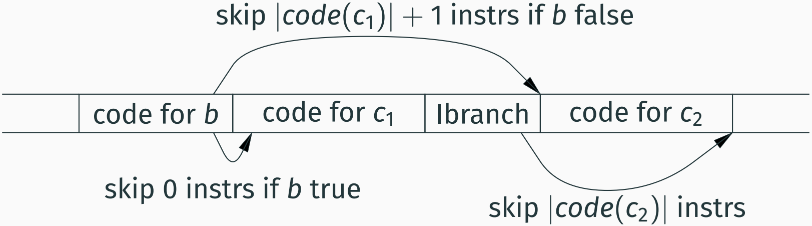
Contract: if command c, started in initial store s, terminates in final store s',



(Coq function: `compile_com`)

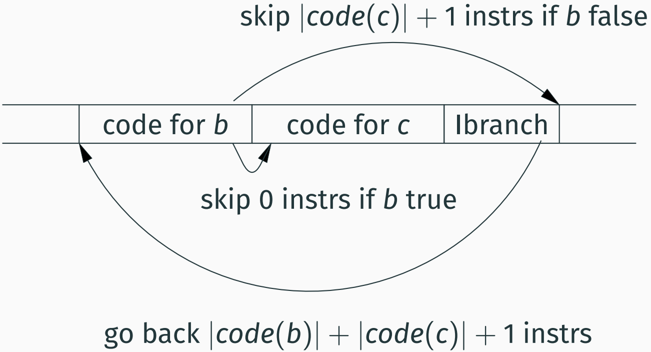
The mysterious branch offsets

Compiled code for `IFTHENELSE` b c_1 c_2:



The mysterious branch offsets

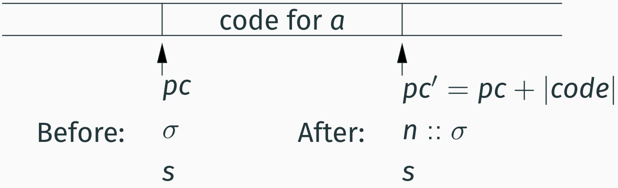
Compiled code for `WHILE` b c:



First compiler correctness results

First verifications

The "contract" for arithmetic expressions: if a evaluates to n in store s,



A plausible formal claim for this "contract":

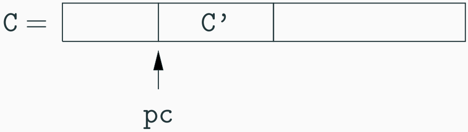
```
Lemma compile_aexp_correct:
  forall s a pc σ,
  transitions (compile_aexp a)
       (0, σ, s)
       (codelen (compile_aexp a), aeval a s :: σ, s).
```

Verifying the compilation of expressions

This claim cannot be proved by induction on the structure of a. It must be generalized so that

- the initial `pc` is not necessarily 0;
- the code `compile_aexp` a occurs within a larger piece of code C, at position `pc`.

To this end, we define the predicate `code_at C pc C'` that holds in the following case:



Verifying the compilation of expressions

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

Proof: an induction on the structure of a.
(It's the proof by McCarthy and Painter, 1967!)

Base cases are trivial:

- $a = n$: execution of one `Iconst` transition
- $a = x$: execution of one `Ivar`(x) transition.

An inductive case

Consider $a = a_1 + a_2$ and assume

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

We build a transition sequence:

$$(pc, \sigma, s)$$

$$\downarrow *\ \text{ind. hyp. for } a_1$$

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

$$\downarrow *\ \text{ind. hyp. for } a_2$$

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

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

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

Verifying the compilation of expressions

Same approach for Boolean expressions: the "contract", once formalized and generalized, is as follows:

```
Lemma compile_bexp_correct:
  forall C s b d1 d0 pc σ,
  code_at C pc (compile_bexp b d1 d0) ->
  transitions C
       (pc, σ, s)
       (pc + codelen (compile_bexp b d1 d0)
           + (if beval b s then d1 else d0), σ, s).
```

The proof is by 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 σ,
  code_at C pc (compile_com c) ->
  transitions C
      (pc, σ, s)
      (pc + codelen (compile_com c), σ, s').
```

An induction on the structure of `c` fails in the `WHILE` case.

An induction on the derivation of the predicate `cexec s c s'` works beautifully.

Summary so far

Combining the previous results, and taking

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

we obtain a nice theorem:

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

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 machine code stops on a store different from `s'`? or loops forever? or gets stuck on an error?

Impossible! because the machine is deterministic: every machine code program has at most one behavior (termination on a given store, divergence, or going wrong).

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 source program `c`, started in store `s`, diverges instead of terminating? What does the compiled machine code do in this case?

Example

Let's "optimize" `while true do` c into `skip`.
This feels wrong; yet, the theorem is still valid!

We need a more precise verification to show preservation of non-termination.

Simulation diagrams

Transition semantics

Defined by a relation $a \to a'$ representing one transition / one reduction step.

Also called "small-step operational semantics".

Examples:

- the reduction semantics for IMP
- the transition semantics for the virtual machine
- the lambda-calculus $M \to_\beta M'$
- process calculi $P \xrightarrow{\alpha} P'$

Transition semantics

Transition semantics define the possible behaviors of a programs in terms of transition sequences:

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

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

- Divergence: infinite sequence of transitions.

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

- Going wrong: finite sequence of transitions to a configuration that is stuck and is not final.

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

Simulation diagrams

Assume that the source program S and the compiled code C have transition semantics.

Show that every transition in the execution of S

- is "simulated" by transitions in the execution of C
- while preserving a relation between S configurations and C configurations.

Lock-step simulation diagrams

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

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

(In black: hypotheses; in red: conclusions.)

Also show that initial configurations are related:

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

Also show that final configurations are related:

$$s \approx c \ \wedge \ s \in Fin \implies c \in Fin$$

Lock-step simulation diagrams

It follows that if S terminates, C terminates as well:

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

Likewise, if we have infinitely many transitions from s_{init}, we have infinitely many transitions from c_{init}. Hence, if S diverges, C diverges as well.

“Plus” simulation diagrams

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

(Example: the compiled code for $x := a$ comprises several machine instructions.)

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

Again, termination and divergence are preserved.

"Star" simulation diagram (wrong!)

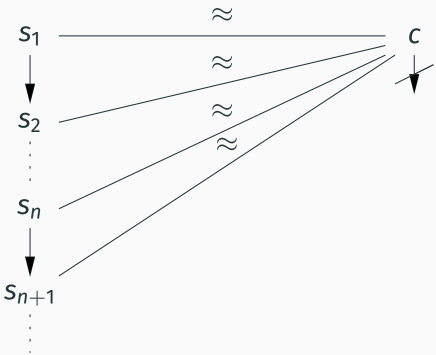
In some cases, every transition in the source program is simulated by zero, one or several transitions in the compiled code.

Example: the reduction $(\texttt{SKIP}; c)/s \to c/s$ corresponds with zero machine. This is called "stuttering".

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

Terminating executions are preserved.
Diverging executions are not always preserved!

The infinite stuttering problem



Here, the source program diverges, but the compiled code can terminate, normally or on an error.

This denotes an incorrect optimization of diverging programs, such as “optimizing” `while true do skip` into `skip`.

"Star" simulation diagrams (corrected)

Find a measure $M(s)$: `nat` for source configurations that decreases strictly on stuttering steps.

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

If s terminates, c terminates too (like before).

If s diverges, it must perform infinitely many non-stuttering transitions, hence c performs infinitely many transitions.

(Remark: we can use any well-founded ordering on source configurations.)

Towards a simulation diagram for IMP compilation

Let’s try to prove a simulation diagram between an IMP command and its compiled machine code.

Two difficulties with IMP’s reduction semantics:

- how to connect the IMP command and the machine code?
- how to build a measure to avoid infinite stuttering?

“Spontaneous generation” of commands

In natural semantics, all commands that appear in the derivation of $c/s \Downarrow s'$ are sub-terms of c.

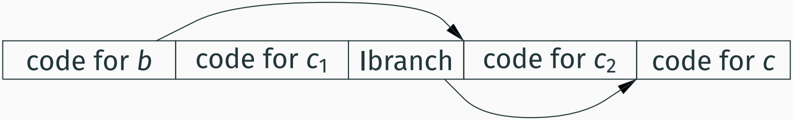
Not so in reduction semantics! Commands appear during reduction that are not sub-terms of the initial program c:

$$
\begin{aligned}
(\texttt{while}\ b\ \texttt{do}\ c)/s &\to (c; \texttt{while}\ b\ \texttt{do}\ c)/s && \text{if } [\![s]\!]\ b = \texttt{true} \\
((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2); c)/s &\to (c_1; c)/s && \text{if } [\![b]\!]\ s = \texttt{true}
\end{aligned}
$$

Matching commands with compiled code

The compiled code for the initial program does not change during execution. It may not contain the code for commands "spontaneously generated" during reductions.

Compiled code for $(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2); c$:



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

The anti-stuttering measure

The "stuttering" reduction steps, those that correspond to zero transitions of the machine, include:

$$(\texttt{skip}; c)/s \to c/s$$
$$(\texttt{if true then } c_1 \texttt{ else } c_2)/s \to c_1/s$$
$$(\texttt{while true do } c)/s \to (c; \texttt{while true do } c)/s$$

Therefore, we must find a measure M such that

$$M(\texttt{skip}; c) > M(c)$$
$$M(\texttt{while true do } c) > M(c; \texttt{while true do } c)$$

This is impossible! Consider $M(\texttt{while true do skip})\ldots$

Tagging sequences

We can work around the issue by marking $;^\dagger$ the sequences generated by loop reductions:

$$
\begin{aligned}
(\texttt{while}\ b\ \texttt{do}\ c)/s &\to \texttt{skip}/s && \text{if } [\![b]\!]\ s = \texttt{false}\\
(\texttt{while}\ b\ \texttt{do}\ c)/s &\to (c;^\dagger \texttt{while}\ b\ \texttt{do}\ c)/s && \text{if } [\![s]\!]\ b = \texttt{true}\\
(c_1;^\dagger c_2)/s &\to (c_1';^\dagger c_2)/s' && \text{if } c_1/s \to c_1'/s'\\
(\texttt{skip};^\dagger c_2)/s &\to c_2/s
\end{aligned}
$$

The reduction $(\texttt{skip};^\dagger c_2)/s \to c_2/s$ is not stuttering, since it corresponds to the `Ibranch` instruction that restarts the loop. Therefore, we can take $M(c_1;^\dagger c_2) = M(c_1)$ and satisfy the constraints over M.

Time for a different approach

Red alert: we are about to change the syntax of the IMP language just to “push through” a compiler proof…

Saner approach: without changing the syntax of IMP, let’s find another semantics:

- of the small-step operational style, to support reasoning by simulation diagrams;
- that does not run into problems with “spontaneous generation” of commands, nor with stuttering control.

A semantics using continuations

A small-step semantics with continuations

Instead of reducing whole programs c

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

let us reduce commands under focus c and their continuations k:

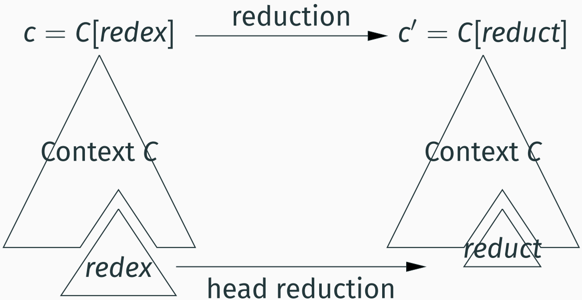
$$c/k/s \to c'/k'/s'$$

(Idea taken from A. W. Appel and S. Blazy, *Separation Logic for Small-Step Cminor*, 2007.)

(Close to "focusing" in proof theory.)

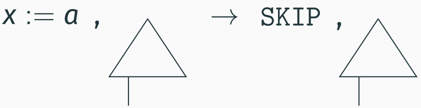
The usual reduction semantics

Rewrites the whole program even though only one sub-command changes (the *redex*).

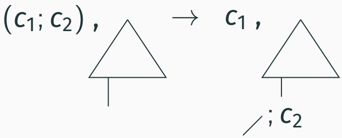


Focusing the reduction semantics

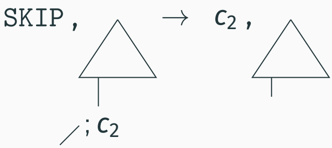
Rewrite pairs (sub-command, context where it appears).



The sub-command is not always the redex! We add explicit focusing and resumption rules to move terms between sub-command and context.



Focusing on the left of a sequence

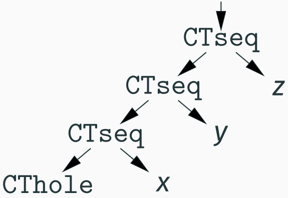


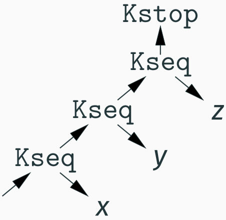
Resuming a sequence

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 it's over.")

Transition rules

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

Note: no spontaneous generation of commands!

Full correctness of the compiler

A simulation diagram

At last we can construct a simulation diagram of the transitions of the IMP continuation semantics by transitions of the machine.

This will prove semantic preservation for terminating executions (already proved) and diverging executions (new!).

Since the machine is deterministic, it follows a bisimulation between the source program and its compiled code.

Two delicate points:

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

The anti-stuttering measure

The main stuttering reduction steps are:

$$\begin{aligned}
(c_1; c_2)/k/s &\rightarrow c_1/\mathtt{Kseq}\ c_2\ k/s \\
\mathtt{skip}/\mathtt{Kseq}\ c\ k/s &\rightarrow c/k/s \\
(\mathtt{if\ true\ then}\ c_1\ \mathtt{else}\ c_2)/k/s &\rightarrow c_1/k/s \\
(\mathtt{while\ true\ do}\ c)/k/s &\rightarrow c/\mathtt{Kwhile\ true}\ c\ k/s
\end{aligned}$$

Like before, measuring c leads us nowhere. We must measure (c, k) pairs.

The anti-stuttering measure

After trial and error, the following measure works:

$$M(c, k) = \|c\| + M(k)$$

where

$$M(\texttt{Kskip}) = 0 \quad M(\texttt{Kseq}\, c\, k) = \|c\| + M(k) \quad M(\texttt{Kwhile}\, b\, c\, k) = M(k)$$

It satisfies

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

Matching commands, continuations, and compiled code

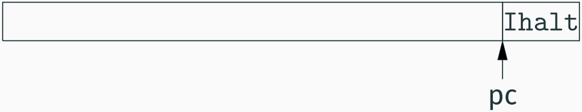
For a command *c*: `code_at C pc (compile_com c)`.

For a (command *c*, continuation *k*) pair:

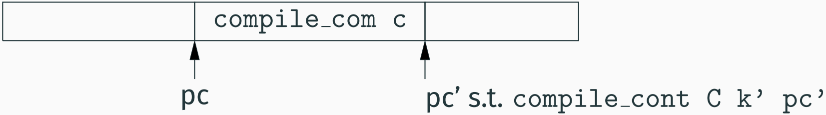
Matching continuations and compiled code

A predicate `compile_cont` $C\ k\ pc$, read "there exists a path in code C starting at pc, ending on a `Ihalt` instruction, and executing the pending computations described by k".

Base case $k =$ `Kstop`:

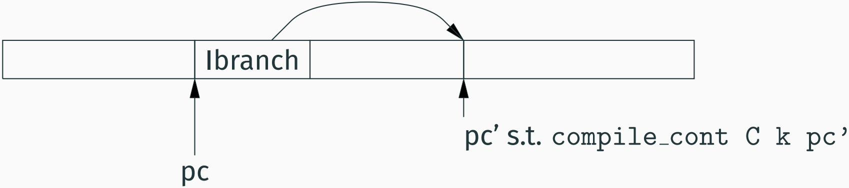


Sequence case $k =$ `Kseq` $c\ k'$:

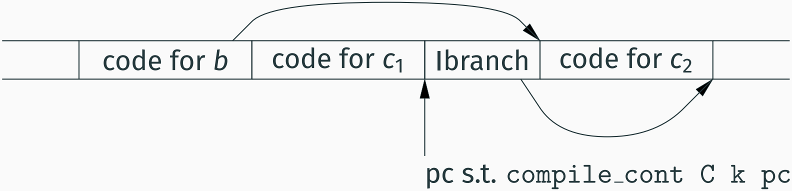


Matching continuations and compiled code

A “non-structural” case lets us insert branches when convenient:



Makes it possible to handle continuation produced by
`if` b `then` c_1 `else` c_2:



The simulation relation

A source program configuration (c, k, s) matches a machine configuration $C, (pc, \sigma, s')$ iff:

- the stores are the same: $s' = s$
- the stack is empty: $\sigma = \varepsilon$
- C contains compiled code for c starting at position pc
- C contains compiled code for k starting at position $pc + |code(c)|$.

The simulation diagram

$$
\begin{tikzcd}
c_1/k_1/s_1 \arrow[r, dash, "C \vdash c_1/k_1/s_1 \approx (pc_1, \varepsilon, s_1)"] \arrow[d] & (pc_1, \varepsilon, s'_1) \arrow[d, "+ \;\vee\; * \wedge M(c_2, k_2) < M(c_1, k_1)"] \\
c_2/k_2/s_2 \arrow[r, dash, "C \vdash c_2/k_2/s_2 \approx (pc_2, \varepsilon, s_2)"'] & (pc_2, \varepsilon, s'_2)
\end{tikzcd}
$$

Proof: large case analysis on the left-hand transition.

To conclude

From this diagram, it follows:

- Another proof of compiler correctness for terminating programs:
  if $c/\mathtt{Kstop}/s \xrightarrow{*} \mathtt{SKIP}/\mathtt{Kstop}/s'$
  then $\mathtt{machine\_terminates}\ (\mathtt{compile\_program}\ c)\ s\ s'$
- A proof of compiler correctness for diverging programs:
  if $c/\mathtt{Kstop}/s$ reduces infinitely,
  then $\mathtt{machine\_diverges}\ (\mathtt{compile\_program}\ c)\ s$

Mission accomplished!

Summary

Summary

Using a non-optimizing compiler for the toy IMP language, we have shown several approaches that scale to more ambitious compiler verification projects such as CompCert:

- Code generation by recursion over the abstract syntax tree.
- Natural semantics for initial explorations.
- Simulation diagrams between two transition semantics for the final proof.
- Continuation semantics for languages with structured control.

References

References

An excellent compiler textbook:

- A. W. Appel, *Modern Compiler Implementation in Java / ML / C*, Cambridge University Press, 1998.

Another verification of an IMP compiler:

- T. Nipkow, G. Klein, *Concrete Semantics*, Springer, 2014, chapitre 8.

Two formal verification of realistic compilers:

- X. Leroy, *Formal verification of a realistic compiler*, Comm. ACM 52(7), 2009.
- R. Kumar, M. O. Myreen, M. Norrish, S. Owens, *CakeML: A Verified Implementation of ML*, POPL 2014.