Formal semantics of programming languages

Provide a mathematically-precise answer to the question

What does this program do, exactly?

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

(Raymond Cheong, 2001)

(It computes arbitrary-precision square roots.)

What about this one?

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

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

(Simon Tatham, author of PuTTY)

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

Why indulge in formal semantics?

- An intellectually challenging issue.
- When English prose is not enough.
  (e.g. language standardization documents.)
- A prerequisite to formal program verification.
  (Program proof, model checking, static analysis, etc.)
- A prerequisite to building reliable “meta-programs”
  (Programs that operate over programs: compilers, code generators, program verifiers, type-checkers, ….)
Is this program transformation correct?

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

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

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

What about this one?

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

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

Proof assistants

- Implementations of well-defined mathematical logics.
- Provide a specification language to write definitions and state theorems.
- Provide ways to build proofs in interaction with the user. (Not fully automated proving.)
- Check the proofs for soundness and completeness.

Some mature proof assistants:

- ACL2
- HOL
- PVS
- Agda
- Isabelle
- Twelf
- Coq
- Mizar

Using proof assistants to mechanize semantics

Formal semantics for realistic programming languages are large (but shallow) formal systems.

Computers are better than humans at checking large but shallow proofs.

- The proofs of the remaining 18 cases are similar and make extensive use of the hypothesis that [...]

- The proof was mechanically checked by the XXX proof assistant. This development is publically available for review at http://...

This lecture

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

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

http://gallium.inria.fr/~xleroy/courses/Eugene-2012/

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

Contents

1. Compiling IMP to a simple virtual machine; first compiler proofs.
2. Notions of semantic preservation.
4. Finishing the proof of the IMP → VM compiler.
5. An example of optimizing program transformation and its correctness proof: dead code elimination, with extension to register allocation.
6. A generic static analyzer (or: abstract interpretation for dummies).
7. Compiler verification "in the large": the CompCert C compiler.

Part II

Compiling IMP to virtual machine code

Reminder: the IMP language

(Already introduced in Benjamin Pierce’s “Software Foundations” course.)

A prototypical imperative language with structured control flow.

Arithmetic expressions:

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

Boolean expressions:

\[ b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \text{not} \ b \mid b_1 \text{ and } b_2 \]

Commands (statements):

\[ c ::= \text{SKIP} \mid x ::= a \mid c_1 ; c_2 \mid \text{IFB } b \text{ THEN } c_1 \text{ ELSE } c_2 \text{ FI} \mid \text{WHILE } b \text{ DO } c \text{ END} \]

(\(st\) ranges over variable states: \(\text{ident} \rightarrow \text{nat}\).)

Reminder: IMP’s semantics

As defined in file Imp.v of “Software Foundations”:

- Evaluation function for arithmetic expressions
  \[ \text{aeval} \ st \ a : \text{nat} \]
- Evaluation function for boolean expressions
  \[ \text{beval} \ st \ b : \text{bool} \]
- Evaluation predicate for commands (in big-step operational style)
  \[ c/st \Rightarrow st' \]

(\(st\) ranges over variable states: \(\text{ident} \rightarrow \text{nat}\).)
Execution models for a programming language

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

Compilation to native code:
before execution, the program is translated to a sequence of machine instructions. These instructions are those of a real microprocessor and are executed in hardware.

Compilation to virtual machine code:
before execution, the program is translated to a sequence of instructions. These instructions are those of a virtual machine. They do not correspond to that of an existing hardware processor, but are chosen close to the basic operations of the source language. Then, either the virtual machine instructions are interpreted (efficiently) or they are further translated to machine code (JIT).

Compiling IMP to virtual machine code

Reminder: the IMP 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 $st$: a mapping from variable names to integer values.
- The stack $\sigma$: a list of integer values (used to store intermediate results temporarily).

The instruction set

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

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

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

Example

<table>
<thead>
<tr>
<th>stack</th>
<th>stack</th>
<th>1</th>
<th>12</th>
<th>13</th>
<th>12</th>
<th>13</th>
<th>12</th>
<th>13</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \mapsto 12$</td>
<td>$x \mapsto 12$</td>
<td>$x \mapsto 12$</td>
<td>$x \mapsto 12$</td>
<td>$x \mapsto 13$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p.c.$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>code</td>
<td>Ivar($x$), Iconst(1), Iadd, Isetvar($x$), Ibranch_backward(5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Semantics of the machine

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

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

Inductive transition (C: code):

```
machine_state -> machine_state -> Prop := ...
```

(See file Compil.v.)
Executing machine programs

By iterating the transition relation:
- Initial states: pc = 0, initial store, empty stack.
- Final states: pc points to a halt instruction, empty stack.

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

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

Definition mach_goes_wrong (C: code) (s_init: state) :=
(* otherwise *)
(star is reflexive transitive closure. See file Sequences.v.)

Compiling IMP to virtual machine code

Compilation of arithmetic expressions

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

<table>
<thead>
<tr>
<th>code for a</th>
<th>pc</th>
<th>σ</th>
<th>st</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>pc' = pc +</td>
<td>code</td>
<td></td>
</tr>
<tr>
<td>Before: σ</td>
<td>After: n :: σ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st</td>
<td>st</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compilation is just translation to "reverse Polish notation".
(See function compile_aexpr in Compil.v)

Compilation of boolean expressions

compile_bexp b cond δ:
- skip δ instructions forward if b evaluates to boolean cond
- continue in sequence if b evaluates to boolean ¬cond

<table>
<thead>
<tr>
<th>code for b</th>
<th>pc</th>
<th>σ</th>
<th>st</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>pc' = pc + δ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Before: σ</td>
<td>After: n :: σ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st</td>
<td>st</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A base case: b = (a1 = a2) and cond = true:

<table>
<thead>
<tr>
<th>code for a1</th>
<th>code for a2</th>
<th>Ibeq(δ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>pc' = pc + δ</td>
<td></td>
</tr>
<tr>
<td>σ</td>
<td>n1 :: σ</td>
<td></td>
</tr>
<tr>
<td>st</td>
<td>st</td>
<td></td>
</tr>
</tbody>
</table>

After (if result ≠ cond)

After (if result = cond)
Short-circuiting “and” expressions

If $b_1$ evaluates to false, so does $b_1$ and $b_2$: no need to evaluate $b_2$!
→ 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 state $st$, terminates in final state $st'$,

$$
\begin{array}{c|c}
\text{Before:} & \text{After:} \\
\sigma & \sigma \\
st & st' \\
\end{array}
$$

$$pc = pc + |\text{code}|$$

(See function compile_com in Compil.v)

The mysterious offsets

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

$skip |\text{code}(c_1)| + 1$ instrs if $b$ false

$$
\begin{array}{c|c|c|c|c}
\text{code for } b & \text{code for } c_1 & \text{Ibranch} & \text{code for } c_2 \\
\end{array}
$$

$skip |\text{code}(c_2)|$ instrs

Compiling IMP to virtual machine code

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

We now have two ways to run a program:
- Interpret it using e.g. the ceval_step function defined in Imp.v.
- Compile it, then run the generated virtual machine code.

Will we get the same results either way?

The compiler verification problem

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

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 \( \text{compile}_aexp \ a \) appears as a fragment of a larger code \( C \).

To this end, we define the predicate \( \text{codeseq}_a exp \ C \ pc \ C' \) capturing the following situation:

\[
C = \begin{array}{c}
\text{pc} \\
\end{array}
\begin{array}{c}
C' \\
\end{array}
\]

An inductive case

Consider \( a = a_1 + a_2 \) and assume

\[
\text{codeseq}_a exp \ C \ pc \ (\text{code}(a_1) + +\text{code}(a_2) + +\text{Iadd} :: \text{nil})
\]

We have the following sequence of transitions:

\[
(pc, \sigma, \text{st}) \\
\downarrow \ast \ \text{ind. hyp. on } a_1 \\
(pc + |\text{code}(a_1)|, \text{aeval st } a_1 :: \sigma, \text{st}) \\
\downarrow \ast \ \text{ind. hyp. on } a_2 \\
(pc + |\text{code}(a_1)| + |\text{code}(a_2)|, \text{aeval st } a_2 :: \text{aeval st } a_1 :: \sigma, \text{st}) \\
\downarrow \ \text{Iadd transition} \\
(pc + |\text{code}(a_1)| + |\text{code}(a_2)| + 1, \text{aeval st } a_1 + \text{aeval st } a_2 :: \sigma, \text{st})
\]

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 \( \text{st} \),

\[
\begin{array}{c|c}
\text{code for } a \\
\hline
\text{pc} & \text{pc}' = \text{pc} + |\text{code}| \\
\text{Before: } & \text{After: } \\
\sigma & \sigma :: \text{st}
\end{array}
\]

A formal claim along these lines:

Lemma \( \text{compile}_aexp_correct \):
for all \( \text{st} \ a \ \text{pc} \ \text{stk} \),
\[
\text{star (transition (compile}_aexp \ a)} \rightarrow
\begin{array}{c}
\text{pc} \ \text{stk} \ \text{st} \\
\text{length (compile}_aexp \ a), \text{aeval st } a :: \text{stk}, \text{st}.
\end{array}
\]

Proof: a simple induction on the structure of \( a \).

The base cases are trivial:
- \( a = \star \): a single \text{Iconst} transition.
- \( a = \star \ x \): a single \text{Ivar}(x) transition.

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).
**Mathematical Aspects of Computer Science, 1967**

**Machine Intelligence** (7), 1972.

### 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 I.C., an implementation of a logic for computable functions due to Dana Scott, in which the abstract syntax and extended semantics of programming languages can be naturally expressed. The source language in our example is a simple ALGO/60-like language with constructs, conditionals, while and compound statements. The target language is an assembly language for a machine with pushdown stores. Algorithmic methods are used to give structure to the proof, which is presented only schematically. However, we present full the expression-complaining 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.

### Verifying the compilation of expressions

Similar approach for boolean expressions:

**Lemma compile_bexp_correct:**

\[
\forall C. st. b.\text{ cond of st. } pc.\text{ stk}.\quad \text{codeseq_at } C\text{ } pc\text{ } (\text{compile_bexp } b\text{ } \text{cond of st}) \rightarrow \text{star } (\text{transition } C)\text{ } (pc + \text{length of } \text{compile_bexp } b\text{ } \text{cond of st},\text{ stk},\text{ st}).
\]

Proof: induction on the structure of \( b \), plus copious case analysis.

(Even the proof scripts look familiar!)

### Verifying the compilation of commands

**Lemma compile_com_correct_terminating:**

\[
\forall C. st. c. st'.\quad (c \mid st || st') \rightarrow \forall \text{ stk } \text{ pc}.\quad \text{codeseq_at } C\text{ } pc\text{ } (\text{compile_com } c) \rightarrow \text{star } (\text{transition } C)\text{ } (pc,\text{ stk},\text{ st})
\]

\[
(pc + \text{length of } \text{compile_com } c,\text{ stk},\text{ st'}).\quad \text{An induction on the structure of } c \text{ fails because of the WHILE case. An induction on the derivation of } c \mid st \mid st' \text{ works perfectly.}
\]

### Summary so far

Piecing the lemmas together, and defining

\[
\text{compile_program } c = \text{compile_command } c + + \text{Ihalt} :: \text{nil}
\]

we obtain a rather nice theorem:

**Theorem compile_program_correct_terminating:**

\[
\forall c.\text{ st}.\quad (c \mid st || st') \rightarrow \text{mach_terminates } (\text{compile_program } c)\text{ st st'}.
\]

But is this enough to conclude that our compiler is correct?
What could have we missed?

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

What if the generated VM code could terminate on a state other than st'? or loop? or go wrong?
What if the program c started in st diverges instead of terminating?
What does the generated code do in this case?

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

Part III
Notions of semantic preservation

Comparing the behaviors of two programs

Consider two programs P1 and P2, possibly in different languages.
(For example, P1 is an IMP command and P2 is virtual machine code generated by compiling P1.)
The semantics of the two languages associate to P1, P2 sets B(P1), B(P2) of observable behaviors.

\[ \text{card}(B(P)) = 1 \text{ if } P \text{ is deterministic, and } \text{card}(B(P)) > 1 \text{ if it is not.} \]

Observable behaviors

For an IMP-like language:
\[ \text{observable behavior ::= terminates(st) | diverges | goeswrong} \]
(Alternative: in the terminates case, observe not the full final state st but only the values of specific variables.)

For a functional language like STLC:
\[ \text{observable behavior ::= terminates(v) | diverges | goeswrong} \]
where v is the value of the program.

Bisimulation (observational equivalence)

\[ B(P_1) = B(P_2) \]
The source and transformed programs are completely undistinguishable.

Often too strong in practice . . .
Reducing non-determinism during compilation

Languages such as C leave evaluation order partially unspecified.

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

The expression `f() + g()` can evaluate either to 1 if `f()` is evaluated first (returning 1), then `g()` (returning 0); to -1 if `g()` is evaluated first (returning -1), then `f()` (returning 0).

Every C compiler chooses one evaluation order at compile-time. The compiled code therefore has fewer behaviors than the source program (1 instead of 2).

---

Reducing non-determinism during optimization

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

Original program:

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

Program after common subexpression elimination:

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

Assuming `x = 0` initially, the final states for the original program are

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

Those for the optimized program are

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

---

Backward simulation (refinement)

\( B(P_1) \supseteq B(P_2) \)

All possible behaviors of \( P_2 \) are legal behaviors of \( P_1 \), but \( P_2 \) can have fewer behaviors (e.g. because some behaviors were eliminated during compilation).

---

Should “going wrong” behaviors be preserved?

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

\( x := 1 / y; x := 42 \) optimized to \( x := 42 \)

\( \text{(goes wrong if } y = 0) \quad \text{(always terminates normally)} \)

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 just “garbage in, garbage out”.

---

Safe backward simulation

Restrict ourselves to source programs that cannot go wrong:

\[ \text{goeswrong} \notin B(P_1) \implies B(P_1) \supseteq B(P_2) \]

Let Spec be the functional specification of a program: a set of correct behaviors, not containing \text{goeswrong}.

A program \( P \) satisfies Spec iff \( B(P) \subseteq \text{Spec} \).

**Lemma**

If “safe backward simulation” holds, and \( P_1 \) satisfies Spec, then \( P_2 \) satisfies Spec.

---

The pains of backward simulations

“Safe backward simulation” looks like “the” semantic preservation property we expect from a correct compiler.

It is however rather difficult to prove:

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

Source code: \[1 + 2 \rightarrow 3\]

VM code: \(I\text{const}(1) \rightarrow I\text{const}(2) \rightarrow \text{Iadd}\)

VM stack: \(\text{nil} \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow \text{nil} \rightarrow 3 \rightarrow \text{nil}\)

Intermediate VM code sequences like \(I\text{const}(2)\); \text{Iadd} or just \text{Iadd} do not correspond to the compilation of any source expression.

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

Forward simulations

Forward simulation property:

\[B(P_1) \subseteq B(P_2)\]

Safe forward simulation property:

\[\text{goeswrong} \notin B(P_1) \Rightarrow B(P_1) \subseteq B(P_2)\]

Significantly easier to prove than backward simulations, but not informative enough, apparently:

The compiled code \(P_2\) has all the good behaviors of \(P_1\), but could have additional bad behaviors . . .

Determinism to the rescue!

Lemma
If \(P_2\) is deterministic (i.e. \(B(P_2)\) is a singleton), then

- “forward simulation” implies “backward simulation”
- “forward simulation for correct programs” implies “backward simulation for correct programs”

Trivial result: follows from \(\emptyset \subset X \subseteq \{y\} = \Rightarrow X = \{y\}\).

Our plan for verifying a compiler

1. Prove “forward simulation for correct programs” between source and compiled codes.
2. Prove that the target language (machine code) is deterministic.
3. Conclude that all functional specifications are preserved by compilation.

Note: \((1) + (2)\) imply that the source language has deterministic semantics. If this isn’t naturally the case (e.g. for C), start by determinizing its semantics (e.g. fix an evaluation order a priori).

Relating preservation properties

Handling multiple compilation passes
We have already proved half of a safe forward simulation result:

Theorem compile_program_correct_terminating:
\[ \forall c \text{ st st'}, \quad \text{c} / \text{st} \parallel \text{st'} \rightarrow \text{mach_terminates} (\text{compile_program c} \text{ st st'}) \]

It remains to show the other half:

If command \( c \) diverges when started in state \( s \),
then the virtual machine, executing code \( \text{compile_program c} \) from initial state \( s \), makes infinitely many transitions.

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

--

**Big-step semantics**

A predicate \( \text{c/s ⇒ s'} \), meaning “started in state \( s \), command \( c \) terminates and the final state is \( s' \).”

**Pros and cons of big-step semantics**

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

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

**Big-step semantics and divergence**

For IMP, a negative characterization of divergence:

\[ \text{c/s diverges} \iff \neg(\exists s', \text{c/s ⇒ s'}) \]

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

\[ \text{c/s diverges} \lor \text{c/s goes wrong} \iff \neg(\exists s', \text{c/s ⇒ s'}) \]

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

**Reminder: big-step semantics for terminating programs**

\[
\begin{align*}
\text{SKIP/s ⇒ s} & \quad x := a/s ⇒ s[x ← \text{aeval s a}] \\
\text{c}_1/s ⇒ s_1 & \quad \text{IFB} b \text{ THEN } c_1 \text{ ELSE } c_2 \text{ FI/s ⇒ s'} \quad \text{beval s b = true} \\
\text{c}_2/s ⇒ s_2 & \quad \text{beval s b = false} \\
\text{c}_1 \text{; } c_2/s ⇒ s_2 & \\
\text{beval s b = true } & \quad c/s ⇒ s_1 & \quad \text{WHILE } b \text{ DO } c \text{ END/s ⇒ s}_1 ⇒ s_2 \\
\text{beval s b = true } & \quad \text{WHILE } b \text{ DO } c \text{ END/s ⇒ s}_2 \\
\end{align*}
\]
More on mechanized semantics

Reminder: big-step semantics for terminating programs

Small-step semantics

Small-step semantics with continuations

Small-step semantics for IMP

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

$$x := a/s \rightarrow \text{SKIP} / s[x \leftarrow \text{aeval} s a]$$

$$c_1/s \rightarrow c_1'/s'$$

$$(c_1; c_2)/s \rightarrow (c_1'; c_2)/s'$$

(SKIP; c)/s \rightarrow c/s$$

beval s b = \text{true}

IFB b THEN c_1 ELSE c_2 FI/s \rightarrow c_1/s$$

beval s b = \text{false}

IFB b THEN c_1 ELSE c_2 FI/s \rightarrow c_2/s$$

WHILE b DO c END/s \rightarrow IFB b THEN c; WHILE b DO c END ELSE SKIP/s

Sequences of reductions

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

- Termination with final state $s'$: finite sequence of reductions to SKIP.
  $c/s \rightarrow \cdots \rightarrow \text{SKIP}/s'$$

- Divergence: infinite sequence of reductions.
  $c/s \rightarrow c_1/s_1 \rightarrow \cdots \rightarrow c_n/s_n \rightarrow \cdots$

- Going wrong: finite sequence of reductions to an irreducible command that is not SKIP.
  $$(c, s) \rightarrow \cdots \rightarrow (c', s') \not\rightarrow \text{ with } c \neq \text{SKIP}$$

Equivalence small-step / big-step

A classic result:

$$c/s \Rightarrow s' \iff c/s \Rightarrow_{\text{small-step}} \text{SKIP}/s'$$

(See Coq file Semantics.v.)

Pros and cons of small-step semantics

Pros:

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

Cons:

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

Small-step semantics

Also called “structured operational semantics”.

Like $\beta$-reduction in the $\lambda$-calculus: view computations as sequences of reductions

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

Each reduction $M \rightarrow M'$ represents an elementary computation. $M'$ represents the residual computations that remain to be done later.
Reasoning with or without structure

Reasoning, big-step style: by pre- and post-conditions
- Single program: if \( c/s \Rightarrow s' \) and \( P/s \), then \( Q/s' \).
- Program transformation: if \( c/s \Rightarrow s' \) and \( T/c \ c_1 \) and \( P/s \ s_1 \), there exists \( s_2 \) s.t. \( c_1/s_1 \Rightarrow s'_2 \) and \( Q/s' \ s'_2 \).
Proofs: by induction on a derivation of \( c/s \Rightarrow s' \).

Reasoning, small-step style: by invariants and simulations.
- Single program: if \( c/s \rightarrow c'/s' \) and \( I(c,s) \) then \( I(c',s') \).
- Program transformation: a relation \( I(c,s) \) is a (bi)-simulation for the transitions of the two programs.
Proofs: by case analysis on each transition.

Spontaneous generation of terms

\[(\text{IF} b \text{ THEN } c_1 \text{ ELSE } c_2 \text{ IF} \ ; c)/s \rightarrow (c_1; c)/s\]

Compiled code for initial command:

\[
\text{code for } b \quad \text{code for } c_1 \quad \text{Ibranch} \quad \text{code for } c_2 \quad \text{code for } c
\]

This code nowhere contains the compiled code for \( c_1; c \), which is:

\[
\text{code for } c_1 \quad \text{code for } c
\]

(Similar problem for \( \text{WHILE } b \text{ DO } c \text{ END } s \rightarrow \text{IF} b \text{ THEN } c \text{ END ELSE } \text{SKIP } s \).

More on mechanized semantics

Intermediate forms extending the syntax

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

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

Commands: \( c ::= \ldots | \text{BREAK} \ | \text{INLOOP } c_1 \ c_2 \)

Intuition: \( \text{INLOOP } c_1 \ c_2 \approx c_1; c_2 \) but with special treatment of \( \text{BREAK} \) arising out of \( c_1 \).

\[
\begin{align*}
\text{INLOOP } c_1 \ c_2 & \rightarrow \text{IF} b \text{ THEN } c_1 \text{ END ELSE } \text{SKIP } c_2 \\
\text{BREAK } c_1 \ c_2 & \rightarrow \text{BREAK } c_1 \ c_2
\end{align*}
\]

Small-step semantics with continuations

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

Idea: instead of rewriting whole commands:
\[ c/s \rightarrow c'/s' \]
rewrite pairs of (subcommand under focus, remainder of command):
\[ c/k/s \rightarrow c'/k'/s' \]
(Vaguely related to focusing in proof theory.)

Standard small-step semantics

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

\[
\begin{array}{ccc}
  c = C[\text{redex}] & \rightarrow & c' = C[\text{reduct}] \\
  \text{Context } C & & \text{Context } C \\
  \text{redex} & \rightarrow & \text{reduct} \\
  \text{head reduction} & & \text{reduct}
\end{array}
\]
Focusing the small-step semantics

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

\[ x ::= a \quad \Rightarrow \quad \text{SKIP} \]

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

\[ (c_1; c_2), \quad \Rightarrow \quad c_1, \quad \text{SKIP}, \quad \Rightarrow \quad c_2, \quad \text{Continuing a sequence} \]

Resuming a sequence

Transition rules

\[ x ::= a/k/s \rightarrow \text{SKIP}/k/s[x \leftarrow \text{beval } s \ a] \]

\[ (c_1; c_2)/k/s \rightarrow c_1/Kseq c_1 k/s \]

IFB b THEN c_1 ELSE c_2/k/s \rightarrow c_1/k/s \quad \text{if beval } s \ b = \text{true} \]

IFB b THEN c_1 ELSE c_2/k/s \rightarrow c_2/k/s \quad \text{if beval } s \ b = \text{false} \]

WHILE b DO c END/k/s \rightarrow c/Kseq (WHILE b DO c END) k/s \quad \text{if beval } s \ b = \text{true} \]

WHILE b DO c END/k/s \rightarrow \text{SKIP}/c/k/s \quad \text{if beval } s \ b = \text{false} \]

\[ \text{SKIP}/Kseq c k/s \rightarrow c/k/s \]

Note: no spontaneous generation of fresh commands.

Representing contexts "upside-down"

Inductive ctx := Inductive cont :=
| CThole: ctx | Kstop: cont |
| CTseq: com \rightarrow \text{ctx} \rightarrow \text{ctx.} | Kseq: com \rightarrow \text{cont} \rightarrow \text{cont.} |

Inductive cont :=
| CTseq: \text{com} \rightarrow \text{cont} \rightarrow \text{cont.} |

Inductive cont :=
| CTseq: \text{com} \rightarrow \text{cont} \rightarrow \text{cont.} |

CTseq (CTseq (CTseq CThole z) y) x \rightarrow Kseq z (Kseq y (Kseq x Kstop))

Upside-down context \( \approx \) continuation.

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

Enriching the language

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

Commands: \( c ::= \ldots | \text{BREAK} \)

Continuations: \( k ::= \text{Kstop} | \text{Kseq} c k | \text{Kwhile} b c k \)

New or modified rules:

\[ \text{WHILE } b \text{ DO } c \text{ END} /k/s \rightarrow c/\text{Kwhile } b c k/s \quad \text{if beval } s \ b = \text{true} \]

\[ \text{SKIP}/\text{Kwhile } b c k/s \rightarrow \text{WHILE } b \text{ DO } c \text{ END} /k/s \]

\[ \text{BREAK}/\text{Kseq} c k/s \rightarrow \text{BREAK}/k/s \]

\[ \text{BREAK}/\text{Kwhile } b c k/s \rightarrow \text{SKIP}/k/s \]

(Exercise: what about continue?)

Equivalence with the other semantics

\[ c/\text{Kstop} /s \rightarrow \text{SKIP}/\text{Kstop}/s' \iff c/s \Rightarrow s' \iff c/s \rightarrow \text{SKIP}/s' \]

\[ c/k/s \rightarrow \infty \iff c/s \rightarrow \infty \]

(See Coq file Semantics.v)

Part V

Compiling IMP to virtual machine code, continued
Finishing the proof of forward simulation

One half already proved: the terminating case.

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

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

Forward simulations, small-step style

Show that every transition in the execution of the source program
is simulated by some transitions in the compiled program
while preserving a relation between the states of the two programs.

Lock-step simulation

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

c1/k1/s1 ⇠ C,(pc1,σ1,s′1)
c2/k2/s2 ⇠ C,(pc2,σ2,s′2)

Further show that initial states are related:
c/Kstop/s ≈ (C,(0,nil,s)) with C = compile_program(c)

Further show that final states are quasi-related:
SKIP/Kstop/s ≈ (C,mst) ⇒ (C,(pc,nil,s))\land C(pc) = Ihalt

“Plus” simulation diagrams

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

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

c1/k1/s1 ⇠ C,(pc1,σ1,s′1)
c2/k2/s2 ⇠ C,(pc2,σ2,s′2)
...
halt with store = sn

(Likewise if c1/k1/s1 reduces infinitely.)

Forward simulation still holds.
“Star” simulation diagrams (incorrect)

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

(Example: source reduction (SKIP; c)/s → c/s makes zero transitions in the machine code.)

\[
\begin{align*}
  &c_1/k_1/s_1 \xrightarrow{} C.(p_1c_1, \sigma_1, s'_1) \\
  &c_2/k_2/s_2 \xrightarrow{} C.(p_2c_2, \sigma_2, s'_2)
\end{align*}
\]

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

An incorrect optimization that exhibits infinite stuttering

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

\[
\text{compile_com (WHILE true DO SKIP END)} = \text{nil}
\]

The “infinite stuttering” problem

Adding special cases to the \(\approx\) relation, we can prove the following naive “star” simulation diagram:

\[
\begin{align*}
  &c_1/k_1/s_1 \xrightarrow{} C.(p_1c, \sigma, s') \\
  &c_2/k_2/s_2 \\
  &c_n/k_n/s_n \xrightarrow{} C.(p_n c, \sigma, s')
\end{align*}
\]

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

Infinite stuttering

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

“Star” simulation diagrams (corrected)

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

\[
\begin{align*}
  &c_1/k_1/s_1 \xrightarrow{} C.(p_1c_1, \sigma_1, s'_1) \\
  &c_2/k_2/s_2 \xrightarrow{} C.(p_2c_2, \sigma_2, s'_2) \\
  &c_n/k_n/s_n \xrightarrow{} C.(p_n c, \sigma, s')
\end{align*}
\]

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

Application to the IMP \(\rightarrow\) VM compiler

Let’s try to prove a “star” simulation diagram for our compiler.

Two difficulties:

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

Stuttering reduction = no machine instruction executed. These include:

\[
(c_1; c_2)/k/s \rightarrow c_1/Kseq c_2 k/s \\
\text{SKIP}/kseq c/k/s \rightarrow c/k/s \\
(\text{IFB true THEN} c_1 \text{ ELSE} c_2)/k/s \rightarrow c_1/k/s \\
(\text{WHILE true DO c END})/k/s \rightarrow c/Kwhile true c/k/s
\]

No measure \( M \) on the command \( c \) can rule out stuttering. For \( M \) to decrease in the second case above, we should have

\[ M(\text{SKIP}) > M(c) \quad \text{for all command} \quad c \]

\[ \rightarrow \text{We must measure } (c, k) \text{ pairs.} \]

Relating commands and continuations with compiled code

In the big-step proof: \( \text{codeseq}_\text{at} \ C \ pc \ (\text{compile}_\text{com} \ c) \).

\[
C = \begin{array}{c}
\text{compile}_\text{com} \ c \\
\text{pc}
\end{array}
\]

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

\[
\text{C} = \begin{array}{c}
\text{machine instructions that "execute" } k \\
\text{pc}
\end{array}
\]

Relating continuations with compiled code

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

\[
\text{lbranch} \\
\text{pc \ s.t. compile}_\text{cont} \ C \ k \ pc'
\]

Useful to handle continuations arising out of \( \text{IFB b THEN} c_1 \text{ ELSE} c_2 \):

\[
\text{code for } b \quad \text{code for } c_1 \quad \text{lbranch} \quad \text{code for } c_2 \\
\text{pc s.t. compile}_\text{cont} \ C \ k \ pc'
\]

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 \( pc + |\text{code}(c)| \).

Anti-stuttering measure

After some trial and error, an appropriate measure is:

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

(In other words, every constructor of \( \text{com} \) counts for 1, and every constructor of \( \text{cont} \) counts for 0.)

\[
M((c_1; c_2), k) = M(c_1, Kseq c_2 k) + 1 \\
M(\text{SKIP}, Kseq c k) = M(c, k) + 1 \\
M(\text{IFB} b \text{ THEN} c_1 \text{ ELSE} c_2, F_1, k) \geq M(c_1, k) + 1 \\
M(\text{WHILE} b \text{ DO} c \text{ END}, k) = M(c, Kwhile b c k) + 1
\]
The simulation diagram

\[ C \vdash c_1/k_1/s_1 \approx (pc_1, \epsilon, s'_1) \]
\[ C \vdash c_2/k_2/s_2 \approx (pc_2, \epsilon, s'_2) \]

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

Wrapping up

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

- An alternate proof of compiler correctness for terminating programs:
  
  if \( c/K_{\text{stop}}/s \rightarrow \text{SKIP}/K_{\text{stop}}/s' \)
  
  then \( \text{mach terminates} (\text{compile program } c) \ s \ s' \)

- A proof of compiler correctness for diverging programs:
  
  if \( c/K_{\text{stop}}/s \) reduces infinitely,
  
  then \( \text{mach diverges} (\text{compile program } c) \ s \)

Mission complete!

Compiler optimizations

Automatically transform the programmer-supplied code into equivalent code that

- Runs faster
  - Remoes redundant or useless computations.
  - Use cheaper computations (e.g. \( x \times 2 \rightarrow x + x \))
  - 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.

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:

\[ x = n \quad x \in [n,m] \quad x = \text{expr} \quad a \times b + y \leq n \]

Requirements:

- The inputs to the program are unknown.
- The analysis must terminate.
- The analysis must run in reasonable time and space.
Running example: dead code elimination via liveness analysis

Remove assignments \( x := e \), turning them into \( \text{skip} \), whenever the variable \( x \) is never used later in the program execution.

Example
Consider: \( x := 1; \ y := y + 1; \ x := 2 \)
The assignment \( x := 1 \) can always be eliminated since \( x \) is not used before being redefined by \( x := 2 \).

Builds on a static analysis called liveness analysis.

Optimizations based on liveness analysis

- Liveness analysis
- Dead code elimination
- Advanced topic: register allocation

Notions of liveness

A variable is **dead** at a program point if its value is not used later in any execution of the program:
- either the variable is not mentioned again before going out of scope
- or it is always redefined before further use.

A variable is **live** if it is not dead.

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

\[
\begin{align*}
\text{(def } x \text{)} & \quad (\text{use } x) \\
 x := \ldots & \quad \ldots x \ldots & (\text{def } x) & \quad (\text{use } x) & \quad (\text{use } x)
\end{align*}
\]

\( x \) live \( x \) dead

Liveness equations

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

\[
\begin{align*}
\text{live(SKIP, } L \text{)} & = \ L \\
\text{live}(x := a, \ L) & = \ (L \setminus \{x\}) \cup \text{FV}(a) \quad \text{if } x \in L; \quad L \quad \text{if } x \notin L. \\
\text{live}([c_1; c_2], \ L) & = \ \text{live}(c_1, \text{live}(c_2, L)) \\
\text{live}((\text{IF} b \text{ THEN } c_1 \text{ ELSE } c_2), \ L) & = \ \text{FV}(b) \cup \text{live}(c_1, L) \cup \text{live}(c_2, L) \\
\text{live}((\text{WHILE } b \text{ DO } c \text{ END}), \ L) & = \ X \quad \text{such that} \\
\ & \quad X \supseteq L \cup \text{FV}(b) \cup \text{live}(c, X)
\end{align*}
\]

Liveness for loops

We must have:
- \( \text{FV}(b) \subseteq X \) (evaluation of \( b \))
- \( L \subseteq X \) (if \( b \) is false)
- \( \text{live}(c, X) \subseteq X \) (if \( b \) is true and \( c \) is executed)
Fixpoints, a.k.a “the recurring problem”

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

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

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

The mathematician’s approach to fixpoints

Let \( A, \leq \) be a partially ordered type. Consider \( F : A \rightarrow A \).

Theorem (Knaster-Tarski)

The sequence \( \bot, F(\bot), F(F(\bot)), \ldots, F^n(\bot), \ldots \)

converges to the smallest fixpoint of \( F \), provided that

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

This provides an effective way to compute fixpoints.

(Coq implementation: see file Fixpoint.v)

Problems with Knaster-Tarski

1. Formalizing and exploiting the ascending chain property
   → well-founded orderings and Noetherian induction.

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

Time for plan B…

The engineer’s approach to post-fixpoints

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

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

A compromise between analysis time and analysis precision.

(Coq implementation: see file Deadcode.v)

Optimizations based on liveness analysis

1. Liveness analysis

2. Dead code elimination

3. Advanced topic: register allocation

Dead code elimination

The program transformation eliminates assignments to dead variables:

\[ x := a \quad \text{becomes} \quad \text{SKIP} \quad \text{if} \ x \text{ is not live “after” the assignment} \]

Presented as a function \( \text{dce} : \text{com} \rightarrow \text{VS.t} \rightarrow \text{com} \)

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!

Liveness as an information flow property

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

\[
\begin{align*}
\frac{c/s_1}{s_2} & \quad \frac{c/s_1'}{s_2'}
\end{align*}
\]

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

\[
\forall x \in \text{live}(c, L), \quad s_1(x) = s_2'(x)
\]

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

\[
\forall x \in L, \quad s_2(x) = s_2'(x)
\]

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

Agreement and its properties

Definition agree $(L: VS.t) (s_1 s_2: state) : \text{Prop} :=$

\[
\forall x, VS.\text{In} x L \rightarrow s_1 x = s_2 x.
\]

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

Lemma agree_mon:

\[
\forall L s_1 s_2, \text{agree} L s_1 s_2 \rightarrow \text{agree} L (\text{update} s_1 x v) s_2.
\]

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

Lemma aeval_agree:

\[
\forall b, VS.\text{Subset}(\text{fv}_{bexp} b) L \rightarrow \text{aeval} s_1 a = \text{aeval} s_2 a.
\]

Lemma beval_agree:

\[
\forall L s_1 s_2, \text{agree} L s_1 s_2 \rightarrow \\
\forall b, VS.\text{Subset}(\text{fv}_{bexp} b) L \rightarrow \text{beval} s_1 b = \text{beval} s_2 b.
\]

Forward simulation for dead code elimination

For terminating source programs:

Theorem dce_correct_terminating:

\[
\forall c s_1 s_2, \text{dce} c L / s_1 || s_2 \rightarrow \\
\exists s_1', \text{dce} c L / s_1' \land \\text{agree} L s_1' s_2\]

(Proof: an induction on the derivation of $c / st \Rightarrow st'$.)

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

\[
\begin{align*}
c_1/s_1 & \rightarrow \text{agree} (\text{live} c_1 L) s_1 s_1' \
& \text{dce} c_1 L/s_1' \rightarrow \\
c_2/s_2 & \rightarrow \text{agree} (\text{live} c_2 L) s_2 s_2' \
& \text{dce} c_2 L/s_2' \rightarrow \end{align*}
\]

Liveness as an information flow property

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

\[
\begin{align*}
\frac{c/s_1}{s_2} & \quad \frac{c/s_1'}{s_2'}
\end{align*}
\]

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

\[
\forall x \in \text{live}(c, L), \quad s_1(x) = s_2'(x)
\]

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

\[
\forall x \in L, \quad s_2(x) = s_2'(x)
\]

The proof of semantic preservation for dead-code elimination follows this pattern, relating executions of $c$ and $\text{dce} c L$ instead.
Optimizations based on liveness analysis

- Liveness analysis
- Dead code elimination
- Advanced topic: register allocation

The register allocation problem

Place the variables used by the program (in unbounded number) into:
- either hardware registers (very fast access, but available in small quantity)
- or memory locations (generally allocated on the stack) (available in unbounded quantity, but slower access)

Try to maximize the use of hardware registers.
A crucial step for the generation of efficient machine code.

Approaches to register allocation

Naive approach (injective allocation):
- Assign the \( N \) most used variables to the \( N \) available registers.
- Assign the remaining variables to memory locations.

Optimized approach (non-injective allocation):
- Notice that two variables can share a register as long as they are not simultaneously live.

Example of register sharing

(def x)
x := ...
(use x)
... x ...
(def y)
y := ...
(use y)
... y ...
(use y)
... y ....
(def R)
R := ...
(use R)
... R ...
(use R)
... R ...

Register allocation for IMP

Properly done:
- Break complex expressions by introducing temporaries.
  (E.g. \( x = (a + b) \cdot y \) becomes \( \text{tmp} = a + b; x = \text{tmp} \cdot y \).)
- Translate IMP to a variant IMP’ that uses registers \( \cup \) memory locations instead of variables.

Simplified as follows in this lecture:
- Do not break expressions.
- Translate from IMP to IMP, by renaming identifiers.
  (Convention: low-numbered identifiers \( \approx \) hardware registers.)

The program transformation

Assume given a “register assignment” \( f : \text{id} \rightarrow \text{id} \).

The program transformation consists of:
- Renaming variables: all occurrences of \( x \) become \( f \ x \).
- Dead code elimination:
  \( x := a \rightarrow \text{SKIP} \) if \( x \) is dead “after”
- Coalescing:
  \( x := y \rightarrow \text{SKIP} \) if \( f \ x = f \ y \)
Correctness conditions on the register assignment

Clearly, not all register assignments \( f \) preserve semantics.

Example: assume \( f(x) = f(y) = f(z) = R \)

\[
\begin{align*}
x &::= 1; & R &::= 1; \\
y &::= 2; & R &::= 2; \\
z &::= x + y; & R &::= R + R;
\end{align*}
\]

Computes 4 instead of 3 . . .

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

Agreement, revisited

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

\( \text{Lemma agree_mon:} \)

\[
\begin{align*}
\forall L, L' \subseteq \text{VS} \quad \text{agree}(L') \Rightarrow \text{agree}(L) \land \text{agree}(L \cup L' - L)
\end{align*}
\]

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

\( \text{Lemma agree_update_dead:} \)

\[
\begin{align*}
\forall L, L' \subseteq \text{VS} \quad \text{agree}(L') \Rightarrow \forall x \in L \setminus L' \exists y \in L : f(y) = f(x) \land \text{agree}(L - \{x\} + \{y\})
\end{align*}
\]

Counter-example: assume \( f(x) = f(y) = R \).

\[
\begin{align*}
\text{agree}(x = 0, y = 0) \land (R = 0) \land \text{agree}(x = 1, y = 0) \land (R = 1) \not\Rightarrow \text{agree}(x = 1, y = 0)
\end{align*}
\]

A special case for moves

Consider a variable-to-variable copy \( x ::= y \).

In this case, the value \( v \) assigned to \( x \) is not arbitrary, but known to be \( s_1 y \). We can, therefore, weaken the non-interference criterion:

\( \text{Lemma agree_update_move:} \)

\[
\begin{align*}
\forall L, L' \subseteq \text{VS} \quad \text{agree}(L', s_1 y + s_2 y - s_1 x, l = v) \Rightarrow \forall x \in L \setminus L' \exists y \in L : f(y) = f(x) \land \text{agree}(L - \{x\} + \{y\})
\end{align*}
\]

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

The interference graph

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

- **Nodes**: program variables.
- **Undirected edge**: between \( x \) and \( y \) if \( x \) and \( y \) cannot be assigned the same location.

Chaitin’s algorithm to construct this graph:

- For each move \( x ::= y \), add edges between \( x \) and every variable \( z \) live “after” except \( x \) and \( y \).
- For each other assignment \( x ::= a \), add edges between \( x \) and every variable \( z \) live “after” except \( x \).
Example of an interference graph

\[ \begin{align*}
    r &:= a; \\
    q &:= 0; \\
    \text{WHILE } b \leq r \text{ DO} \\
    &\quad \begin{align*}
    &r := r - b; \\
    &q := q + 1
    \end{align*} \\
    \text{END}
\end{align*} \]

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

Register allocation as a graph coloring problem


Color the interference graph, assigning a register or memory location to every node;
under the constraint that the two ends of an interference edge have different colors;
with the objective to
- minimize the number (or total weight) of nodes that are colored by a memory location
- maximize the number of preference edges whose ends have the same color.

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

Example of coloring

\[ \begin{align*}
    \text{yellow} &:= \text{yellow}; \\
    \text{green} &:= 0; \\
    \text{WHILE } \text{red} \leq \text{yellow} \text{ DO} \\
    &\quad \begin{align*}
    &\text{yellow} := \text{yellow} - \text{red}; \\
    &\text{green} := \text{green} + 1
    \end{align*} \\
    \text{END}
\end{align*} \]

What needs to be proved in Coq?

Full compiler proof:
formalize and prove correct a good graph coloring heuristic.
George and Appel’s Iterated Register Coalescing ≈ 6 000 lines of Coq.

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

The verified transformation–verified validation spectrum

<table>
<thead>
<tr>
<th>Verified transformation</th>
<th>Verified translation validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>transformation</td>
<td>transformation validation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>External solver with verified validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>transformation</td>
</tr>
<tr>
<td>untrusted solver</td>
</tr>
<tr>
<td>checker</td>
</tr>
</tbody>
</table>

Validating candidate allocations in Coq

It is easy to write a Coq boolean-valued function
\[ \text{correct_allocation}: (\text{id} \to \text{id}) \to \text{com} \to \text{VS.t} \to \text{bool} \]
that returns true only if the expected non-interference properties are satisfied.

(See file Regalloc.v.)
Semantic preservation

The proofs of forward simulation that we did for dead code elimination then extend easily, under the assumption that $\text{correct}\_\text{allocation}$ returns true:

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

Part VII

A generic static analyzer

Static analysis in a nutshell

Statically infer properties of a program that are true of all executions.

*At this program point, $0 < x \leq y$ and pointer $p$ is not NULL.*

Emphasis on infer: no programmer intervention required. (E.g., no need to annotate the source with loop invariants.)

Emphasis on statically:
- Inputs to the program are unknown.
- Analysis must always terminate.
- Analysis must run in reasonable time and space.

Examples of properties that can be statically inferred

**Properties of the value of a single variable:** (value analysis)

- $x = n$ constant propagation
- $x > 0$ or $x = 0$ or $x < 0$ signs
- $x \in [n_1, n_2]$ intervals
- $x = n_1 \mod n_2$ congruences
- valid($p[n_1 \ldots n_2]$) pointer validity
- $p \text{ pointsTo} x$ or $p \neq q$ (non-) aliasing of pointers

($n, n_1, n_2$ are constants determined by the analysis.)

**Properties of several variables:** (relational analysis)

- $\sum a_i x_i \leq c$ polyhedras
- $\pm x_1 \pm \cdots \pm x_n \leq c$ octagons
- expr$_1 = expr$_2 Herbrand equivalences, a.k.a. value numbering

("Non-functional" properties:
- Memory consumption.
- Worst-case execution time (WCET).

Examples of properties that can be statically inferred
Using static analysis for optimization

Applying algebraic laws when their conditions are met:
- \( x / 4 \rightarrow x >> 2 \) if analysis says \( x \geq 0 \)
- \( x + 1 \rightarrow 1 \) if analysis says \( x = 0 \)

Optimizing array and pointer accesses:
- \( a[i]=1; a[j]=2; x=a[i] \rightarrow a[i]=1; a[j]=2; x=1 \) if analysis says \( i \neq j \)
- \( *p = a; x = *q; \rightarrow x = *q; *p = a \) if analysis says \( p \neq q \)

Automatic parallelization:
- \( \text{loop 1} \parallel \text{loop 2} \) if \( \text{polyh(\text{loop 1})} \cap \text{polyh(\text{loop 2})} = \emptyset \)

Using static analysis for verification

(Also known as “static debugging”)

Use the results of static analysis to prove the absence of run-time errors:
- \( b \in [n_1, n_2] \land 0 \notin [n_1, n_2] \implies a/b \text{ cannot fail} \)
- \( \text{valid}(p(n_1 \ldots n_2)) \land i \in [n_1, n_2] \implies *(p + 1) \text{ cannot fail} \)

Sign an alarm otherwise.

True alarms, false alarms

- True alarm (dangerous behavior)
- False alarm (imprecise analysis)

More precise analysis (polyhedra instead of intervals): false alarm goes away.

Floating-point subtleties and their analysis

Taking rounding into account:
- \( \text{float x, y, u, v; } // x \in [1.00025, 2] \)
- \( u = 1 / (x - y); // y \in [0.5, 1] \)
- \( v = 1 / (x*x - y*y); // \text{ALARM: undefined result} \)

First division: \( (x - y) \in [0.00025, 1.5] \) and division cannot result in infinity or not-a-number.

Second division:
- \( (x*x) \in [1.4] \) (float rounding!)
- \( (y*y) \in [0.25, 1] \)
- \( (x*x - y*y) \in [0.375] \)

and division by zero is possible, resulting in +\( \infty \)

A generic static analyzer

- Introduction to static analysis
- Static analysis as an abstract interpretation
- An abstract interpreter in Coq
- Improving the generic static analyzer
Abstract interpretation for dummies

“Execute” the program using a non-standard semantics that:

- Computes over an abstract domain of the desired properties (e.g. \( x \in [n_1, n_2] \) 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.

Orthodox presentation: calculating abstract operators

Define a semantics that computes over an abstract domain of the desired properties (operating on elements of \( \mathbb{A} \) instead of concrete values and states).

Example: for the + operator in expressions,
\[
    a_1 +^\gamma a_2 = \gamma\{ n_1 + n_2 \mid n_2 \in \gamma(a_1), n_2 \in \gamma(a_2) \}
\]

\( +^\gamma \) is sound and optimally precise by construction.

Pedestrian Coq presentation

Focus on the concretization relation \( x \in \gamma(y) \) viewed as a 2-place predicate \( \text{concrete-thing} \to \text{abstract-thing} \to \text{Prop} \).

Forget about the abstraction function \( \alpha \) (generally not computable; often not uniquely defined.)

Forget about calculating the abstract operators: just guess their definitions and prove their soundness.

Forget about optimality; focus on soundness only.

Orthodox presentation: collecting semantics

Define a semantics that collects all possible concrete states at every program point:

```plaintext
// initial value of x is N
y := 1;
WHILE x > 0 DO
  y := y * 2;
  x := x - 1
END
```

Always terminates.

(Handles boolean conditions, even if they cannot be resolved statically.

...calculations omitted ...)
\[
    \left[ l_1, u_1 \right] +^\gamma \left[ l_2, u_2 \right] = \left[ l_1 + l_2, u_1 + u_2 \right]
\]

Orthodox presentation: Galois connection

Define a lattice \( \mathbb{A}, \leq \) of abstract states and two functions:

- Abstraction function \( \alpha : \text{sets of concrete states} \to \text{abstract state} \)
- Concretization function \( \gamma : \text{abstract state} \to \text{sets of concrete states} \)

\( \alpha \) and \( \gamma \) monotonic: \( X \subseteq \gamma(\alpha(X)) \); and \( x^\dagger \leq \alpha(\gamma(x^\dagger)) \).

A generic static analyzer

- Introduction to static analysis
- Static analysis as an abstract interpretation
- An abstract interpreter in Coq
- Improving the generic static analyzer
Abstract domains in Coq

Specified as module interfaces:
- **VALUE_ABSTRACTION**: to abstract integer values.
- **STATE_ABSTRACTION**: to abstract states.

(See Coq file `Analyzer1.v`.)

Each interface declares:
- A type `t` of abstract “things”
- A predicate `vmatch/smatch` relating concrete and abstract things.
- Abstract operations on type `t` (arithmetic operations for values; `get/set` operations for stores).
- Soundness properties of these operations.

Abstract interpretation of arithmetic expressions

Let `V` be a value abstraction and `S` a corresponding state abstraction.

Fixpoint `abstr_eval` `(s: S.t) (a: aexp)` :
- `ANum n` => `V.of_const n`
- `AId x` => `S.get s x`
- `APlus a1 a2` => `V.add (abstr_eval s a1) (abstr_eval s a2)`
- `AMinus a1 a2` => `V.sub (abstr_eval s a1) (abstr_eval s a2)`
- `AMult a1 a2` => `V.mul (abstr_eval s a1) (abstr_eval s a2)`

(What else could we possibly write?)

Abstract interpretation of commands

Computes the abstract state “after” executing command `c` in initial abstract state `s`.

Fixpoint `abstr_interp` `(s: S.t) (c: com)` :
- `SKIP` => `s`
- `(x ::= a)` => `S.set s x (abstr_eval s a)`
- `(c1; c2)` => `abstr_interp (abstr_interp s c1) c2`
- `IFB b THEN c1 ELSE c2 FI` => `S.join (abstr_interp s c1) (abstr_interp s c2)`
- `WHILE b DO c END` =>
  - fixpoint `(fun x => S.join s (abstr_interp x c))` s

Let `s'` be the abstract state “before” the loop body `c`.

- entering `c` on the first iteration => `s ≤ s'`.
- re-entering `c` at next iteration => `abstr_interp s' c ≤ s'`.

Therefore compute a post-fixpoint `s'` such that `s ⊔ abstr_interp s' c ≤ s'`.

Soundness results

Show that all concrete executions produce results that belong to the abstract things inferred by abstract interpretation.

**Lemma abstr_eval_sound**: for all `s`, `S.match (aeval s a)`.

**Theorem abstr_interp_sound**: for all `c` at `st` `st'`, `S.match st s -> c / st || st` at `st'` `S.match st' (abstr_interp s c)`.

(Easy structural inductions on `a` and `c`.)
An example of state abstraction

Parameterized by a value abstraction $V$.

Abstract states = $\bot | \text{finite maps } \text{ident} \rightarrow V.t$. (Default value: $V$.top.)

Appropriate for all non-relational analyses.

A generic static analyzer

An example of value abstraction: constants

Abstract domain = the flat lattice of integers:

\[
\begin{align*}
\top &= \text{nat} \\
\bot &= \emptyset \\
\{0\} \\
\{1\} \\
\{2\} \\
\{3\} \\
\{4\} & \vdots
\end{align*}
\]

Obvious interpretation of operations:

\[
\begin{align*}
\bot +^\sharp x &= x \\
\bot +^\sharp \bot &= \bot \\
\top +^\sharp x &= x +^\sharp \top = \top \\
\{n_1\} +^\sharp \{n_2\} &= \{n_1 + n_2\}
\end{align*}
\]

First improvement: static analysis of boolean expressions

Our analyzer makes no attempt at analyzing boolean expressions \rightarrow both arms of an IF are always assumed taken.

Can do better when the static information available allows to statically resolve the IF. Example:

\[
x := 0;
\text{IF } x = 0 \text{ THEN } y := 1 \text{ ELSE } y := 2 \text{ FI}
\]

Constant analysis in its present form returns \(y^\sharp = \top\) (joining the two branches where \(y^\sharp = \{1\}\) and \(y^\sharp = \{2\}\).)

Since \(x^\sharp = \{0\}\) before the IF, the ELSE branch cannot be taken, hence we should have \(y^\sharp = \{1\}\) at the end.

Static analysis of boolean expressions

Even when the boolean expression cannot be resolved statically, the analysis can learn much from which branch of an IF is taken.

\[
x^\sharp = \top \text{ initially}
\]

\[
\begin{align*}
\text{IF } x = 0 \text{ THEN } & \text{learn that } x^\sharp = \{0\} \\
\text{y := x + 1} & \text{hence } y^\sharp = \{1\} \\
\text{ELSE } & y := 1 \\
\text{FI} & y^\sharp = \{1\} \text{ as well}
\end{align*}
\]

\[
\text{hence } y^\sharp = \{1\}, \text{ not } \top
\]

We can also learn from the fact that a WHILE loop terminates:

\[
x^\sharp = \top \text{ initially}
\]

\[
\text{WHILE not (x = 42) DO} \\
x := x + 1 \\
\text{DONE} \text{ learn that } x^\sharp = 42 \rightarrow \{42\}
\]

More realistic example using intervals instead of constants:

\[
x^\sharp = [0. \infty] \text{ initially}
\]

\[
\text{WHILE } x \leq 1000 \text{ DO} \\
x := x + 1 \\
\text{DONE} \text{ learn that } x^\sharp = [1000. \infty]
\]
Using inverse analysis

Fixpoint abstr_interp (s: S.t) (c: com) : S.t :=
match c with
| SKIP => s
| x ::= a => S.set s x (abstr_eval s a)
| (c1; c2) => abstr_interp (abstr_interp s c1) c2
| IFB b THEN c1 ELSE c2 FI =>
  S.join (abstr_interp (learn_from_test s b true) c1)
  (abstr_interp (learn_from_test s b false) c2)
| WHILE b DO c END =>
  let s' :=
  fixpoint
  (fun x => S.join s
    (abstr_interp (learn_from_test x b true) c))
  s in
  learn_from_test s' b false
end.

Non-well-founded domains

Many interesting abstract domains are not well-founded.
Example: intervals.

\[ [0,0] \subset [0,1] \subset [0,2] \subset \cdots \subset [0,n] \subset \cdots \]

This causes problems for analyzing non-counted loops such as

\[ x := 0; \]
\[ \text{WHILE unpredictable-condition DO } x := x + 1 \text{ END} \]

\[ (x^2 \text{ is successively } [0,0] \text{ then } [0,1] \text{ then } [0,2] \text{ then } \ldots) \]
Slow convergence

In other cases, the fixpoint computation via Tarski’s method does terminate, but takes too much time.

\[
\begin{align*}
x &:= 0; \\
\text{WHILE } x <= 1000 \text{ DO } x := x + 1 \text{ END}
\end{align*}
\]

(Starting with \(x^0 = [0,0]\), it takes 1000 iterations to reach \(x^0 = [0,1000]\), which is a fixpoint.)

Imprecise convergence

The engineer’s algorithm (return \(\top\) after a fixed number of unsuccessful iterations) does converge quickly, but loses too much information.

\[
\begin{align*}
x &:= 0; \\
y &:= 0; \\
\text{WHILE } x <= 1000 \text{ DO } x := x + 1 \text{ END}
\end{align*}
\]

In the final abstract state, not only \(x^0 = \top\), but also \(y^0 = \top\).

Widening

A widening operator \(\nabla : A \rightarrow A \rightarrow A\) computes an upper bound of its second argument in such a way that the following fixpoint iteration always converges (and converges quickly):

\[
\begin{align*}
X_0 &= \bot \\
X_{i+1} &= \begin{cases} 
X_i & \text{if } F(X_i) \leq X_i \\
X_i \nabla F(X_i) & \text{otherwise}
\end{cases}
\end{align*}
\]

The limit \(X\) of this sequence is a post-fixpoint: \(F(X) \leq X\).

For intervals of natural numbers, the classic widening operator is:

\[
[[l_1, u_1]] \nabla [[l_2, u_2]] = 
\begin{cases}
[0, \max(l_2, l_1)] & \text{if } l_2 < l_1 \\
[0, \min(u_2, u_1)] & \text{if } u_2 > u_1
\end{cases}
\]

Example of widening

\[
\begin{align*}
x &:= 0; \\
\text{WHILE } x <= 1000 \text{ DO } x := x + 1 \text{ END}
\end{align*}
\]

The transfer function for \(x\)’s abstraction is

\[
F(X) = [0,0] \cup (X \cap [0,1000]) + 1.
\]

\[
\begin{align*}
X_0 &= \bot \\
X_1 &= X_0 \nabla F(X_0) = \bot \nabla [0,0] = [0,0] \\
X_2 &= X_1 \nabla F(X_1) = [0,0] \nabla [0,1] = [0,\infty] \\
X_2 &\text{ is a post-fixpoint: } F(X_2) = [0,1001] \subseteq [0,\infty].
\end{align*}
\]

Final abstract state is \(x^0 = [0,\infty] \cap [1001,\infty] = [1001,\infty]\).

Refining the fixpoint

The quality of a post-fixpoint can be improved by iterating \(F\) some more:

\[
Y_0 = \text{a post-fixpoint } \quad Y_{i+1} = F(Y_i)
\]

If \(F\) is monotone, each of the \(Y_i\) is a post-fixpoint: \(F(Y_i) \leq Y_i\).

Often, \(Y_i < Y_0\), so we obtain a more precise post-fixpoint.

We can stop iteration when a \(Y_i\) is a fixpoint, or at any convenient time.
Widening plus refinement in action

Example of refinement

\[ x := 0; \]
\[
\text{WHILE } x \leq 1000 \text{ DO } x := x + 1 \text{ END}
\]

The transfer function for \( x \)'s abstraction is
\[ F(X) = [0, 0] \cup (X \cap [0, 1000]) + 1. \]

The post-fixpoint found by iteration with widening is \([0, \infty]\).

\[ Y_0 = [0, \infty] \]
\[ Y_1 = F(Y_0) = [0,1001] \]
\[ Y_2 = F(Y_1) = [0,1001] \]

Final post-fixpoint is \( Y_1 \) (actually, a fixpoint).

Final abstract state is \( x^\# = [0,1001] \cap [1001, \infty] = [1001,1001]. \)

Specification of widening operators

For reference:
- \( y \leq x \triangledown y \) for all \( x, y \).
- For all increasing sequences \( x_0 \leq x_1 \leq \ldots \),
  the sequence \( y_0 = x_0, y_{i+1} = y_i \triangledown x_i \)
  is not strictly increasing.

Coq implementation of accelerated convergence

Because we have not proved the monotonicity of \texttt{abstr_interp} nor the nice properties of widening, we still bound arbitrarily the number of iterations.

\begin{verbatim}
Fixpoint iter_up (n: nat) (s: S.t) : S.t :=
  match n with
  | 0 => S.top
  | S n1 =>
      let s' := F s in
      if S.ble s' s then s else iter_up n1 (S.widen s s')
  end.

Fixpoint iter_down (n: nat) (s: S.t) : S.t :=
  match n with
  | 0 => s
  | S n1 =>
      let s' := F s in
      if S.ble (F s') s' then iter_down n1 s' else s
  end.

Definition fixpoint (start: S.t) : S.t :=
  iter_down num_iter_down (iter_up num_iter_up start).
\end{verbatim}

Static analysis tools in the real world

General-purpose tools:
- Coverity
- MathWorks Polyspace verifier.
- Frama-C value analyzer (open source!)
- Microsoft’s Code Contract

Tools specialized to an application area:
- Microsoft Static Driver Verifier (Windows system code)
- Astrée (control-command code at Airbus)
- Fluctuat (symbolic analysis of floating-point errors)

Tools for non-functional properties:
- aiT WCET (worst-case execution time)
- aiT StackAnalyzer (stack consumption)
Part VIII

Compiler verification in the large

- Compiler issues in critical software
- The CompCert project
- Status and ongoing challenges
- Closing

The classroom setting

- IMP
  - Compiler
  - Static analysis
  - Hoare logic

V.M.

The reality of critical embedded software

- Simulink
- Scade
- Code gen.
- Code prov
- Program prover
- Static analyzers
- Assembly
- Compilers
- Executable
- Test
- Hand-written

Example: fly-by-wire software

Requirements for qualification

- Compilers and code generation tools: Can introduce bugs in programs!
  - Either: the code generator is qualified at the same level of assurance as the application.
    (Implies: much testing, rigorous development process, no recursion, no dynamic allocation, . . . )
  - Or: the generated code needs to be qualified as if hand-written.
    (Implies: testing, code review and analysis on the generated code . . . )

Verification tools used for bug-finding:

Cannot introduce bugs, just fail to notice their presence.
→ can be qualified at lower levels of assurance.

Verification tools used to establish the absence of certain bugs:

Status currently unclear.
The compiler dilemma

If the compiler is untrusted (= not qualified at the highest levels of assurance):

- We still need to review & analyze the generated assembly code, which implies turning off optimizations, and is costly, and doesn’t scale.
- We cannot fully trust the results obtained by formal verification of the source program.
- Many benefits of programming in a high-level language are lost.

Yet: the traditional techniques to qualify high-assurance software do not apply to compilers.

Could formal verification of the compiler help?

The CompCert project


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

- Source language: a subset of C.
- Target language: PowerPC, ARM and x86-32 assembly.
- Generates reasonably compact and fast code ⇒ some optimizations.

This is “software-proof codesign” (as opposed to proving an existing compiler).

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

The formally verified part of the compiler

The subset of C supported

Supported:

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

Not supported:

- The long long and long double types.
- Unstructured switch, longjmp/setjmp.
- Variable-arity functions.

Supported via de-sugaring (not proved!):

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

The whole CompCert compiler
Verified in Coq

Theorem transf_c_program_is_refinement:
forall p tp,
transf_c_program p = OK tp ->
(forell beh, exec_C_program p beh -> not_wrong beh) ->
(forell beh, exec_asm_program tp beh -> exec_C_program p beh).

A composition of
- 15 proofs of the "safe forward simulation" kind
- 1 proof of the "safe backward simulation" kind.

Observable behaviors

Inductive program_behavior: Type :=
| Terminates: trace -> int -> program_behavior
| Diverges: trace -> program_behavior
| Reacts: traceinf -> program_behavior
| Goes_wrong: trace -> program_behavior.

trace = list of input-output events.
traceinf = infinite list (stream) of i-o events.

I/O events are generated for:
- Calls to external functions (system calls)
- Memory accesses to global volatile variables (hardware devices).

Styles of semantics used (as a function of time)

<table>
<thead>
<tr>
<th>Clight . . . Cminor</th>
<th>RTL . . . Mach</th>
<th>Asm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st gen.</td>
<td>big-step</td>
<td>small-step</td>
</tr>
<tr>
<td>2nd gen.</td>
<td>big-step</td>
<td>small-step</td>
</tr>
<tr>
<td>(+ divergence)</td>
<td>(coinductive)</td>
<td>(w/ call stacks)</td>
</tr>
<tr>
<td>3rd gen.</td>
<td>small-step</td>
<td>small-step</td>
</tr>
<tr>
<td>(+ goto &amp; tailcalls)</td>
<td>(w/ continuations)</td>
<td>(w/ call stacks)</td>
</tr>
</tbody>
</table>

The Coq proof

4 person-years of work.
Size of proof: 50000 lines of Coq.
Size of program proved: 8000 lines.
Low proof automation (could be improved).

Programmed in Coq

The verified parts of the compiler are directly programmed in Coq’s specification language, in pure functional style.
- Monads are used to handle errors and state.
- Purely functional data structures.

Coq’s extraction mechanism produces executable Caml code from these Coq definitions, which is then linked with hand-written Caml parts.

Claim: pure functional programming is the shortest path between an executable program and its proof.

Performance of generated code
(On a PowerPC G5 processor)
Compiler verification in the large

Preliminary conclusions

At this stage of the CompCert experiment, the initial goal – proving correct a realistic compiler – appears feasible.

Moreover, proof assistants such as Coq are adequate (but barely) for this task.

What next?

Enhancements to CompCert

In the middle:

- More static analyses: nonaliasing, intervals, ...
- More optimizations? Possibly using verified translation validation?

(Cf. Moore’s Piton & Gypsy projects circa 1995)

Connections with hardware verification

Hardware verification:

- A whole field by itself.
- At the circuit level: a strong tradition of formal synthesis and verification, esp. using model checking.
- At the architectural level (machine language semantics, memory model, ...): almost no publically available formal specifications, let alone verifications.

A very nice work in this area: formalizing the ARM architecture and validating it against the ARM6 micro-architecture. (Anthony Fox et al, U. Cambridge.)

The ARM6 micro-architecture
The ARM6 instruction pipeline

Difficulty for verification: several instructions are "in flight" at any given time.

Redeeming feature: synchrony. The machine state is determined as a function of time and the initial state.

Connections with verification tools

Code generators, static analyzers, model checkers, program provers, . . .

- deserve formal verification if we are to fully trust their results
- . . . and must be verified against the same semantics as the compiler.

The Verasco project (just started):
- an abstract interpreter for the CompCert languages
- will include advanced relational domains and combinations thereof
- formally verified in Coq.

Towards shared-memory concurrency

Programs containing data races are generally compiled in a non-semantic-preserving manner.

Issue #1: apparently atomic operations are decomposed into sequences of instructions, exhibiting more behaviors.

\[
\begin{align*}
x &= \ast p + \ast p; & \text{||} & \ast p = 1; \\
t_1 &= \text{load}(p) & \text{||} & \text{store}(p, 1) \\
t_2 &= \text{load}(p) \\
x &= \text{add}(t_1, t_2)
\end{align*}
\]

In Clight (top): final \( x \in \{0, 2\} \).
In RTL (bottom): final \( x \in \{0, 1, 2\} \).

Other source languages

New problem: run-time system verification (allocator, GC, etc).

Towards shared-memory concurrency

Issue #2: weakly-consistent memory models, as implemented in hardware, introduce more behaviors than just interleavings of loads and stores.

\[
\begin{align*}
\text{store}(q, 1); & \text{||} & \text{store}(p, 1); \\
x &= \text{load}(p) & \text{||} & y = \text{load}(q)
\end{align*}
\]

Interleaving semantics: \( (x, y) \in \{(0, 1), (1, 0), (1, 1)\} \).
Hardware semantics: \( x = 0 \) and \( y = 0 \) is also possible!
Plan A

Expose all behaviors in the semantics of all languages (source, intermediate, machine):
- “Very small step” semantics (expression evaluation is not atomic).
- Weakly-consistent model of memory.

Turn off optimizations that are wrong in this setting.
(common subexpression elimination; uses of nonaliasing properties).

Prove backward simulation results for every pass.

→ The CompCertTSO project at Cambridge
http://www.cl.cam.ac.uk/~pes20/CompCertTSO/

Plan B

Restrict ourselves to data-race free source programs . . .

. . . as characterized by concurrent separation logic.

Separation logic (quick reminder)

Like Hoare triples \( (P \rightarrow Q) \),
but assertions \( P, Q \) control the memory footprint of commands \( c \).

Application: the frame rule

\[
\frac{P \rightarrow Q}{P \star R \rightarrow Q \star R}
\]

Concurrent separation logic (intutions)

Two concurrently-running threads do not interfere if their memory footprints are disjoint:

\[
\frac{P_1 \rightarrow Q_1 \quad P_2 \rightarrow Q_2}{P_1 \star P_2 \rightarrow (c_1 \parallel c_2) \star Q_1 \star Q_2}
\]

But how can two threads communicate through shared memory?

Concurrent separation logic (intutions)

Locks \( L \) are associated with resource invariants \( R \).

\( R \)'s footprint describes the set of shared data protected by lock \( L \).

Locking ⇒ acquire rights to access this shared data.
Unlocking ⇒ forego rights to access this shared data.

\[
\frac{P \rightarrow \text{lock} L \star P \star R(L) \star \text{unlock} L \rightarrow P}{P \star R(L) \star \text{unlock} L \rightarrow P}
\]

Quasi-sequential semantics


For parallel programs provable in concurrent separation logic, we can restrict ourselves to “quasi-sequential” executions:
- In between two lock / unlock operations, each thread executes sequentially; other threads are stopped.
- Interleaving at lock / unlock operations only.
- Interleaving is determined in advance by an “oracle”.

Claim: for programs provable in CSL, quasi-sequential semantics and concrete semantics (arbitrary interveilings + weakly-consistent memory) predict the same sets of behaviors.
Verifying a compiler for data-race free programs

"Just" have to show that quasi-sequential executions are preserved by compilation:

- Easy?? extensions of the sequential case.
- Can still use forward simulation arguments.
- Most classic sequential optimizations remain valid.
- The only "no-no": moving memory accesses across lock and unlock operations.

Work in progress, stay tuned . . .

To finish . . .

The formal verification of compilers and related programming tools

... could be worthwhile,
... appears to be feasible,
... and is definitely exciting!