Proving a compiler

Mechanized verification of program transformations and static analyses

Xavier Leroy

INRIA Paris-Rocquencourt

Oregon Programming Languages summer school 2011

Part I

Prologue: mechanized semantics, what for?

Formal semantics of programming languages

Provide a mathematically-precise answer to the question

What does this program do, exactly?

What does this program do, exactly?

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

(Raymond Cheong, 2001)

(It computes arbitrary-precision square roots.)

What about this one?

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

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

(Simon Tatham, author of PuTTY)

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

Why indulge in formal semantics?

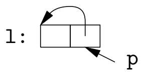
- An intellectually challenging issue.
- When English prose is not enough. (e.g. language standardization documents.)
- A prerequisite to formal program verification. (Program proof, model checking, static analysis, etc.)
- A prerequisite to building reliable "meta-programs" (Programs that operate over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

Is this program transformation correct?

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

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

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

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



What about this one?

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

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


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


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-2011/

- 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.
3. A menagerie of semantics: small-step, big-step, coinductive big-step, definitional interpreters, denotational semantics.
4. Finishing the proof of the IMP $\rightarrow$ 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

Compiling IMP to virtual machine code

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

Reminder: the IMP language

(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:

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

Commands (statements):

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

Reminder: IMP's semantics

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

- Evaluation function for arithmetic expressions

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

- Evaluation function for boolean expressions

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

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

$$c/st \Rightarrow st'$$

(`st` ranges over variable states: `ident` $\rightarrow$ `nat`.)

Execution models for a programming language

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

Compiling IMP to virtual machine code

The IMP virtual machine

Components of the machine:

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

The instruction set

i ::= `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`(δ)	unconditional jump forward
| `Ibranch_backward`(δ)	unconditional jump backward
| `Ibeq`(δ)	pop two values, jump if $=$
| `Ibne`(δ)	pop two values, jump if $\neq$
| `Ible`(δ)	pop two values, jump if $\leq$
| `Ibgt`(δ)	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).

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

Example

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

Semantics of the machine

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

```
Definition code := list instruction.
Definition stack := list nat.
Definition 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.

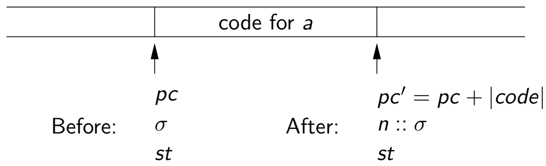
```
Definition mach_terminates (C: code) (s_init s_fin: state) :=
  exists pc,
  code_at C pc = Some Ihalt /\
  star (transition C) (0, nil, s_init) (pc, nil, s_fin).

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,

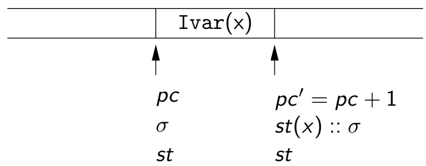


Compilation is just translation to "reverse Polish notation".

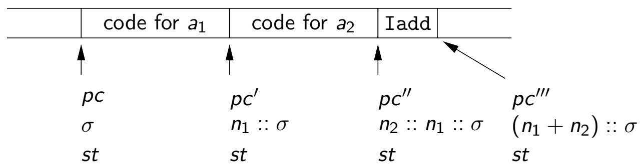
(See function `compile_aexpr` in `Compil.v`)

Compilation of arithmetic expressions

Base case: if $a = x$,

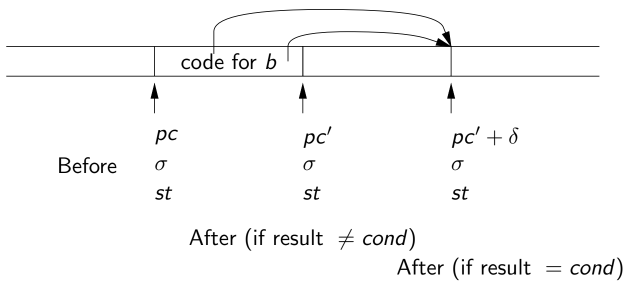


Recursive decomposition: if $a = a_1 + a_2$,



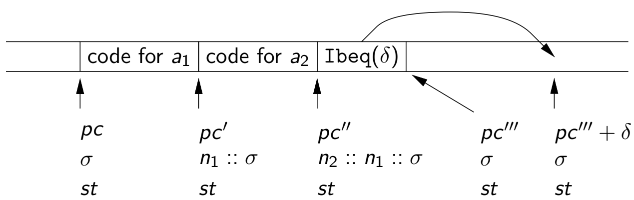
Compilation of boolean expressions

`compile_bexp` b $cond$ δ:
 skip δ instructions forward if b evaluates to boolean $cond$
 continue in sequence if b evaluates to boolean $\neg cond$



Compilation of boolean expressions

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



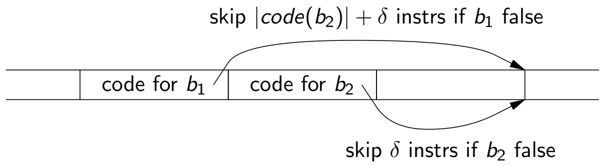
Short-circuiting "and" expressions

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

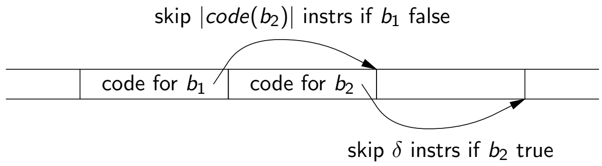
→ In this case, the code generated for b_1 `and` b_2 should skip over the code for b_2 and branch directly to the correct destination.

Short-circuiting "and" expressions

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

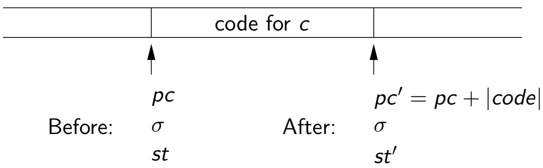


If $cond$ = `true` (branch if b_1 `and` b_2 is true):



Compilation of commands

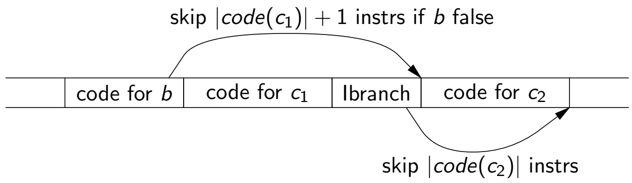
If the command c, started in initial state st, terminates in final state st',



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

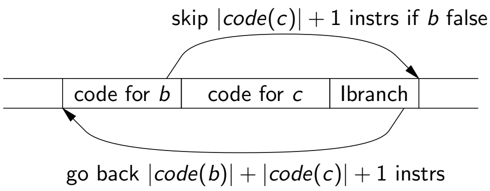
The mysterious offsets

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



The mysterious offsets

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



Compiling IMP to virtual machine code

Compiler verification

We now have two ways to run a program:

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

Will we get the same results either way?

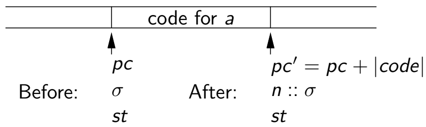
The compiler verification problem

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

First verifications

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

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



A formal claim along these lines:

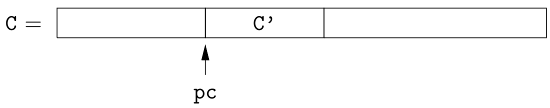
```
Lemma compile_aexp_correct:
  forall st a pc stk,
  star (transition (compile_aexp a))
       (0, stk, st)
       (length (compile_aexp a), aeval st a :: stk, st).
```


Verifying the compilation of expressions

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

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

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



Verifying the compilation of expressions

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

Proof: a simple induction on the structure of a.

The base cases are trivial:

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

An inductive case

Consider $a = a_1 + a_2$ and assume

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

We have the following sequence of transitions:

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

Historical note

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

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

John McCarthy
James Painter[1]

CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS[2]

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

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

Mathematical Aspects of Computer Science, 1967

3

Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch
Computer Science Department
Stanford University

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

Machine Intelligence (7), 1972.

APPENDIX 2: command sequence for McCarthy-Painter lemma

```
GOAL ∀e sp,Iswfse e::MT(compe e,sp)≡svof(sp)|((MSE(e,svof sp))&pdof(sp)),
     ∀e,Iswfse e::Iswft(compe e)≡TT,
     ∀e,Iswfse e::(count(compe e)=0)≡TT;
TRY 1 INDUCT 56;
 TRY 1 SIMPL;
 LABEL INDHYP;
 TRY 2 ABSTR;
  TRY 1 CASES wfsefun(f,e);
  LABEL TT;
   TRY 1 CASES type e=_N;
    TRY 1 SIMPL BY ,FMT1,,FMSE,,FCOMPE,,FISWFT1,,FCOUNT;
    TRY 2;SS-,TT;SIMPL,TT;QED;
    TRY 3 CASES type e=_E;
     TRY 1 SUBST ,FCOMPE;
     SS-,TT;SIMPL,TT;USE BOTH3 -;SS+,TT;
     INCL-,1;SS+-;INCL--,2;SS+-;INCL---,3;SS+-;
      TRY 1 CONJ;
       TRY 1 SIMPL;
        TRY 1 USE COUNT1;
         TRY 1;
         APPL ,INDHYP+2,arg1of e;
         LABEL CARG1;
         SIMPL-;QED;
         TRY 2 USE COUNT1;
          TRY 1;
```

(Even the proof scripts look familiar!)

Verifying the compilation of expressions

Similar approach for boolean expressions:

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

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

Verifying the compilation of commands

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

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

Summary so far

Piecing the lemmas together, and defining

$$\texttt{compile\_program c} = \texttt{compile\_command c} \; ++ \; \texttt{Ihalt} \; :: \; \texttt{nil}$$

we obtain a rather nice theorem:

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

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

What could have we missed?

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

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

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

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

Part III

Notions of semantic preservation

Comparing the behaviors of two programs

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

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

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

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

Observable behaviors

For an IMP-like language:

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

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

For a functional language like STLC:

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

where v is the value of the program.

Observable behaviors

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

`x := 1;  x := 2;`	$\approx$	`x := 2;`
(trace: ϵ)		(trace: ϵ)
`print(1); print(2);`	$\not\approx$	`print(2);`
(trace: out(1).out(2))		(trace: out(2))

Bisimulation (observational equivalence)

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

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.

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

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

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

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

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

Reducing non-determinism during optimization

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

Original program:

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

Program after common subexpression elimination:

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

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

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

Those for the optimized program are

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

Backward simulation (refinement)

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

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` (goes wrong if $\mathtt{y} = 0$) optimized to `x := 42` (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:

$$\mathtt{goeswrong} \notin \mathcal{B}(P_1) \implies \mathcal{B}(P_1) \supseteq \mathcal{B}(P_2)$$

Let *Spec* be the functional specification of a program: a set of correct behaviors, not containing `goeswrong`.

A program P satisfies *Spec* iff $\mathcal{B}(P) \subseteq \mathit{Spec}$.

Lemma

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

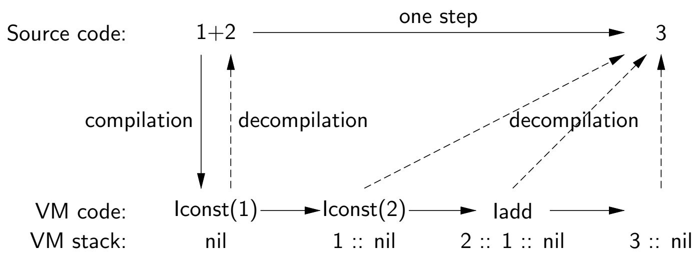
The pains of backward simulations

"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



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

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

Forward simulations

Forward simulation property:

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

Safe forward simulation property:

$$\texttt{goeswrong} \notin \mathcal{B}(P_1) \implies \mathcal{B}(P_1) \subseteq \mathcal{B}(P_2)$$

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!

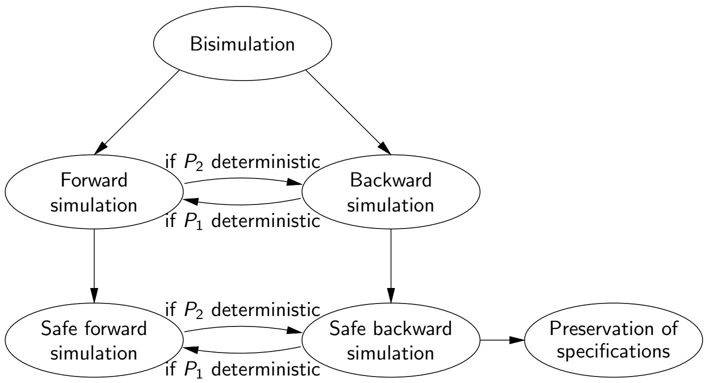
Lemma

If P_2 is deterministic (i.e. $\mathcal{B}(P_2)$ is a singleton), then

- *"forward simulation" implies "backward simulation"*
- *"forward simulation for correct programs" implies "backward simulation for correct programs"*

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

Relating preservation properties

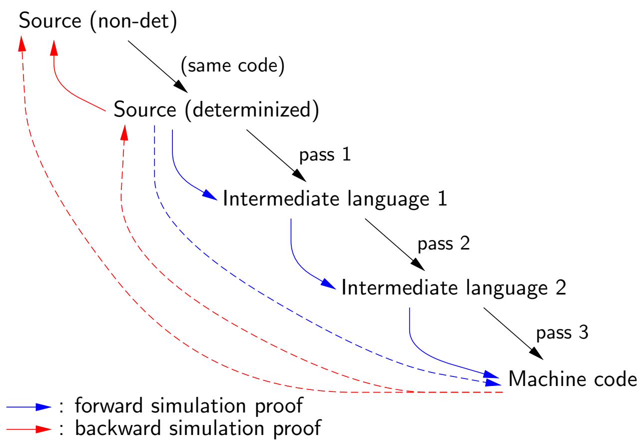


Our plan for verifying a compiler

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 langage 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).

Handling multiple compilation passes



Back to the IMP $\rightarrow$ VM compiler

We have already proved half of a safe forward simulation result:

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

It remains to show the other half:

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

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

Part IV

More on mechanized semantics

More on mechanized semantics

Big-step semantics

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

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

$$\frac{c_1/s \Rightarrow s_1 \qquad c_2/s_1 \Rightarrow s_2}{c_1; c_2/s \Rightarrow s_2} \qquad \frac{\begin{array}{c} c_1/s \Rightarrow s' \text{ if } \mathtt{beval}\ s\ b = \mathtt{true} \\ c_2/s \Rightarrow s' \text{ if } \mathtt{beval}\ s\ b = \mathtt{false}\end{array}}{\mathtt{IFB}\ b\ \mathtt{THEN}\ c_1\ \mathtt{ELSE}\ c_2\ \mathtt{FI}/s \Rightarrow s'}$$

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

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

Pros and cons of big-step semantics

Pros:

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

Cons:

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

Big-step semantics and divergence

For IMP, a negative characterization of divergence:

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

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

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

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

More on mechanized semantics

Small-step semantics

Also called "structured operational semantics".

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

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

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

Small-step semantics for IMP

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

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

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

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

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

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

Sequences of reductions

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

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

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

- Divergence: infinite sequence of reductions.

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

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

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

Equivalence small-step / big-step

A classic result:

$$c/s \Rightarrow s' \iff c/s \xrightarrow{*} \texttt{SKIP}/s'$$

(See Coq file `Semantics.v`.)

Pros and cons of small-step semantics

Pros:

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

Cons:

- Does not follow the structure of programs; lack of a powerful induction principle.
- This is not the way interpreters are written!
- Some extensions require unnatural extensions of the syntax of terms (e.g. with call contexts in the case of IMP + procedures).

More on mechanized semantics

Towards a big-step presentation of divergence

Big-step semantics can be viewed as adding structure to terminating sequences of reductions. Consider such a sequence for $c; c'$:

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

It contains a terminating reduction sequence for c:

$$(c, s) \to (c_1, s_1) \to \cdots \to (\texttt{SKIP}, s_2)$$

followed by another for c'.

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

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

Towards a big-step presentation of divergence

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

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

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

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

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

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

Big-step rules for divergence

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

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

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

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

Induction vs. coinduction in a nutshell

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

Inductive interpretation:

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

Coinductive interpretation:

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

(See Coq illustration in file `Coinduction.v`, and section 2 of *Coinductive big-step semantics* by H. Grall and X. Leroy.)

Example of divergence

Let's interpret coinductively the inference rules defining $c/s \Rightarrow \infty$.
(In Coq: use `CoInductive` instead of `Inductive`.)

We can easily show that classic examples of divergence are captured. Consider $c = \texttt{WHILE true DO SKIP END}$. We can build the following infinite derivation of $c/s \Rightarrow \infty$:

$$\dfrac{\begin{array}{c}\texttt{beval}\ s\ \texttt{true} = \texttt{true} \\ \texttt{SKIP}/s \Rightarrow s\end{array} \qquad \dfrac{\begin{array}{c}\texttt{beval}\ s\ \texttt{true} = \texttt{true} \\ \texttt{SKIP}/s \Rightarrow s\end{array} \qquad \dfrac{\begin{array}{c}\texttt{beval}\ s\ \texttt{true} = \texttt{true} \\ \texttt{SKIP}/s \Rightarrow s\end{array}}{c/s \Rightarrow \infty}}{c/s \Rightarrow \infty}}{c/s \Rightarrow \infty}$$

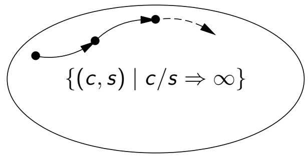
Big-step divergence vs. small-step divergence

Does the $c/s \Rightarrow \infty$ coinductive predicate capture the same notion of divergence as the existence of infinite reduction sequences?

From big-step divergence to small-step divergence

Lemma

If $c/s \Rightarrow \infty$, there exists c' and s' such that $c/s \rightarrow c'/s'$ and $c'/s' \Rightarrow \infty$.



Theorem

If $c/s \Rightarrow \infty$, then there exists an infinite sequence of reductions from c/s.

From small-step divergence to big-step divergence

Theorem

If c/s reduces infinitely, then $c/s \Rightarrow \infty$.

The proof uses inversion lemmas such as:

> *If $c_1; c_2$ reduces infinitely, then*
> *either c_1 reduces infinitely,*
> *or c_1 reduces finitely to* `SKIP` *and c_2 reduces infinitely.*

Note that these lemmas cannot be proved in Coq's constructive logic and require the excluded middle axiom ($\forall P,\ P \vee \neg P$) from classical logic.

Constructive logic in a nutshell

In Coq's constructive logic, a proof is a terminating functional program:

A proof of ...		is ...
$A \rightarrow B$	$\approx$	a total function from proofs of A to proofs of B.
$A \wedge B$	$\approx$	a pair of proofs, one for A and another for B.
$A \vee B$	$\approx$	a procedure that decides which of A and B holds and returns either a proof of A or a proof of B.
$\forall x : A.\ B(x)$	$\approx$	a total function from values $v : A$ to proofs of $B(v)$.
$\exists x : A.\ B(x)$	$\approx$	a pair of a value $v : A$ and a proof of $B(v)$.

Reasoning by cases about termination

A proposition such as

> *For all c and s,*
> *either c/s reduces infinitely,*
> *or there exists c', s' such that $c/s \xrightarrow{*} c'/s' \not\rightarrow$*

cannot be proved constructively.

A proof would be a total function that decides whether c/s terminates or diverges, solving the halting problem.

The obvious proof uses the principle of excluded middle ($\forall P,\ P \vee \neg P$), which is not constructive.

Excluded middle or the axiom of choice can however be added to Coq as axioms without breaking logical consistency.

More on mechanized semantics

Definitional interpreter for IMP

File `Imp.v` in "Software Foundations" defines a Coq function

```
ceval_step: state -> com -> nat -> option state
```

that executes (by interpretation) a given command in a given state.

The `nat` argument bounds the recursion depth and ensures that `ceval_step` always terminates.

- `ceval_step c st n = Some st'` denotes termination with final state `st'`.
- `ceval_step c st n = None` means that the interpretation "runs out of fuel".

Definitional interpreter for IMP

```
Fixpoint ceval_step (st : state) (c : com) (i : nat)
                    : option state :=
  match i with
  | O => None
  | S i' =>
    match c with
      | SKIP =>
          Some st
      | l ::= a1 =>
          Some (update st l (aeval st a1))
      | c1 ; c2 =>
          bind_option
            (ceval_step st c1 i')
            (fun st' => ceval_step st' c2 i')
      | IFB b THEN c1 ELSE c2 FI =>
          if (beval st b) then ceval_step st c1 i' else ceval_step st c2 i'
      | WHILE b1 DO c1 END =>
          if (beval st b1)
          then bind_option
                 (ceval_step st c1 i')
                 (fun st' => ceval_step st' c i')
          else Some st
    end
  end.
```


Equivalence with big-step semantics

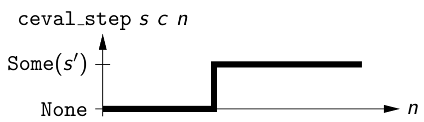
```
Theorem ceval_step__ceval:
  forall c st st',
  (exists i, ceval_step st c i = Some st') ->
  c / st ==> st'.

Theorem ceval__ceval_step:
  forall c st st',
  c / st ==> st' ->
  exists i, ceval_step st c i = Some st'.

Theorem cevalinf_ceval_step_bottom:
  forall n c st,
  c / st ==> ∞ -> ceval_step st c n = None.

Theorem ceval_step_bottom_cevalinf:
  forall c st m,
  (forall n, m <= n -> ceval_step st c n = None) ->
  c / st ==> ∞
```


More on mechanized semantics

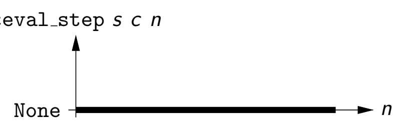
From definitional interpreter to denotational semantics

A simple form of denotational semantics can be obtained by "letting n go to infinity" in the definitional interpreter.

For a terminating command:



For a diverging command:



A denotational semantics

Lemma

For every c, there exists a function $[\![c]\!]$ from states to optional states such that $\forall s,\ \exists m,\ \forall n \geq m,\ \texttt{ceval\_step}\ c\ s\ n = [\![c]\!]\ s$.

The proof uses excluded middle and an axiom of description, but no domain theory.

$[\![c]\!]\ s = \texttt{Some}(s')$ denotes termination with final state s'.
$[\![c]\!]\ s = \texttt{None}$ denotes divergence. (`None` represents $\perp$.)

The equations of denotational semantics

The denotation function $[\![\cdot]\!]$ satisfies the equations of denotational semantics:

$$
\begin{aligned}
[\![\texttt{SKIP}]\!]\ s &= \texttt{Some}(s) \\
[\![x := e]\!]\ s &= \texttt{Some}(s[x \leftarrow [\![e]\!]\ s]) \\
[\![c_1; c_2]\!]\ s &= [\![c_1]\!]\ s \rhd (\lambda s'.\ [\![c_2]\!]\ s') \\
[\![\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI}]\!]\ s &= [\![c_1]\!]\ s \text{ if } \texttt{beval}\ s\ b = \texttt{true} \\
[\![\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI}]\!]\ s &= [\![c_2]\!]\ s \text{ if } \texttt{beval}\ s\ b = \texttt{false} \\
[\![\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}]\!]\ s &= \texttt{Some}(s) \text{ if } \texttt{beval}\ s\ b = \texttt{false} \\
[\![\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}]\!]\ s &= [\![c]\!]\ s \rhd (\lambda s'.\ [\![\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}]\!]\ s') \\
&\quad \text{if } \texttt{beval}\ s\ b = \texttt{true}
\end{aligned}
$$

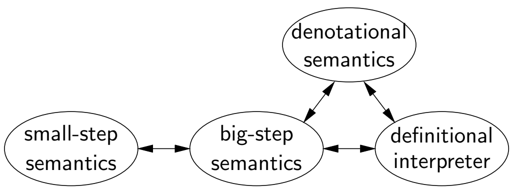
Moreover, $[\![\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}]\!]$ is the smallest function from states to results that satisfies the last two equations.

Relating denotational and big-step semantics

```
Lemma denot_ceval:
  forall c st st',
  c / st ==> st'  <->  denot st c = Some st'.

Lemma denot_cevalinf:
  forall c st,
  c / st ==> ∞  <->  denot st c = None.
```


In summary...

A toolbox of 4 mechanized semantics, all proved equivalent:



Each semantics has its strengths:

- Big-step: structured; powerful (co-) induction principles.
- Small-step: unified treatment of termination & divergence; all-terrain.
- Definitional interpreter: executable.
- Denotational: equational reasoning.

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

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

Compiling IMP to virtual machine code, continued

Using coinductive big-step semantics

The desired result:

```
Lemma compile_com_correct_diverging:
  forall c st C pc stk,
  c / st ==> ∞ -> codeseq_at C pc (compile_com c) ->
  infseq (transition C) (pc, stk, st).
```

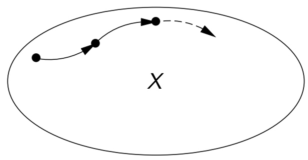
where the `infseq` operator is defined in `Sequences.v` as a coinductive predicate:

```
CoInductive infseq (A: Type) (R: A -> A -> Prop): A -> Prop :=
  | infseq_step: forall a b,
      R a b -> infseq R b -> infseq Ra.
```


The basic coinduction principle

Let X be a set of machine states.
(Encoded in Coq as a predicate `machstate` $\rightarrow$ `Prop`.)

Assume that $\forall x \in X, \exists y \in X, x \rightarrow y$.

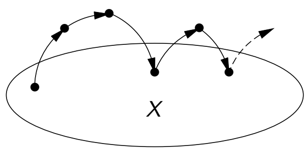


Then, for any $x \in X$, there exists an infinite sequence of machine transitions starting at x.

A more flexible coinduction principle

Let X be a set of machine states.
(Encoded in Coq as a predicate `machstate` $\rightarrow$ `Prop`.)

Assume that $\forall x \in X, \exists y \in X, x \xrightarrow{+} y$.



Then, for any $x \in X$, there exists an infinite sequence of machine transitions starting at x.

Using the coinduction principle

Let C be the compiled code for the whole program and take

$$X = \{(pc, stk, s) \mid \exists c,\ c/s \Rightarrow \infty \wedge \texttt{codeseq\_at}\ C\ pc\ c\}$$

We show that X is "plus-productive", i.e.

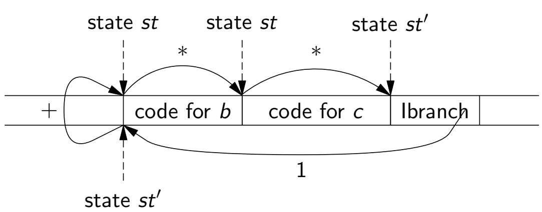
$$\forall x \in X, \exists y \in X, \texttt{plus}\ (\texttt{transition C})\ x\ y$$

The proof is by structural induction on the command c.

Base case: while loops

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

Assume pc points to the code for `WHILE` b `DO` c `END`.



Inductive cases

Just prepend a $\xrightarrow{*}$ sequence to the $\xrightarrow{+}$ sequence obtained by induction hypothesis.

(See Coq proof.)

Wrap-up

The "plus" coinduction principle now shows

```
Lemma compile_com_correct_diverging:
  forall c st C pc stk,
  c / st ==> ∞ -> codeseq_at C pc (compile_com c) ->
  infseq (transition C) (pc, stk, st).
```

from which the second half of forward simulation follows:

```
Theorem compile_program_correct_diverging:
  forall c st,
  c / st ==> ∞ ->
  mach_diverges (compile_program c) st.
```

Small regret: some duplication of proof effort between the terminating and diverging cases.

Compiling IMP to virtual machine code, continued

10 A proof using coinductive big-step semantics

11 A proof using small-step semantics

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.

$$\begin{tikzcd}
c_1/s_1 \arrow[r, dash, "\approx"] \arrow[d] & C,(pc_1,\sigma_1,s'_1) \arrow[d, dashed] \\
c_2/s_2 \arrow[r, dash, dashed, "\approx"] & C,(pc_2,\sigma_2,s'_2)
\end{tikzcd}$$

Lock-step simulation

Further show that initial states are related:

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

Further show that final states are quasi-related:

$$\texttt{SKIP}/s \approx (C, mst) \Longrightarrow (C, mst) \xrightarrow{*} (C,(pc,nil,s)) \land C(pc) = \texttt{Ihalt}$$

Lock-step simulation

Forward simulation follows easily:

$$\begin{tikzcd}
c_1/s_1 \arrow[r, dash, "\approx"] \arrow[d] & C,(pc_1,\sigma_1,s'_1) \arrow[d, dashed] \\
c_2/s_2 \arrow[r, dash, dashed, "\approx"] \arrow[d, dotted, no head] & C,(pc_2,\sigma_2,s'_2) \arrow[d, dotted, no head] \\
{} \arrow[r, dash, dashed, "\approx"] \arrow[d] & {} \arrow[d, dashed] \\
\texttt{SKIP}/s_n \arrow[r, dash, dashed, "\approx"] & C,(pc_n,\sigma_n,s'_n) \arrow[d, dashed, "*"] \\
& \text{halt with store} = s_n
\end{tikzcd}$$

(Likewise if c_1/s_1 reduces infinitely.)

“Plus” simulation diagrams

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

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

$$\begin{tikzcd}
c_1/s_1 \arrow[r, dash, "\approx"] \arrow[d] & C, (pc_1, \sigma_1, s'_1) \arrow[d, dashed, "+"] \\
c_2/s_2 \arrow[r, dash, dashed, "\approx"] & C, (pc_2, \sigma_2, s'_2)
\end{tikzcd}$$

Forward simulation still holds.

“Star” simulation diagrams (incorrect)

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

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

$$\begin{tikzcd}
c_1/s_1 \arrow[r, dash, "\approx"] \arrow[d] & C, (pc_1, \sigma_1, s'_1) \arrow[d, dashed, "*"] \\
c_2/s_2 \arrow[r, dash, dashed, "\approx"] & C, (pc_2, \sigma_2, s'_2)
\end{tikzcd}$$

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

The “infinite stuttering” problem

$$\begin{tikzcd}
c_1/s_1 \arrow[r, dash, "\approx"] \arrow[d] & C, (pc, \sigma, s') \\
c_2/s_2 \arrow[ur, dash, "\approx"] \arrow[d, dotted, dash] & \\
c_n/s_n \arrow[uur, dash, "\approx"] \arrow[d] & \\
c_{n+1}/s_{n+1} \arrow[uuur, dash, "\approx"] \arrow[d, dotted, dash] & \\
 &
\end{tikzcd}$$

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

An incorrect optimization that exhibits infinite stuttering

Add special cases to `compile_com` so that the following three trivially infinite loops get compiled to no instructions at all:

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

compile_com (IFB true THEN SKIP; WHILE true DO SKIP END
                      ELSE SKIP) = nil

compile_com (SKIP; WHILE true DO SKIP END) = nil
```


Infinite stuttering

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

$$\begin{tikzcd}
\texttt{WHILE true DO SKIP END}/s \arrow[r, dash, "\approx"] \arrow[d] & C, (pc, \sigma, s') \\
\texttt{IFB true THEN SKIP;} \ldots \ \texttt{ELSE SKIP}/s \arrow[ur, dash, "\approx"] \arrow[d] & \\
\texttt{SKIP;} \ldots \ / \ s \arrow[uur, dash, "\approx"] \arrow[d] & \\
\texttt{WHILE true DO SKIP END}/s \arrow[uuur, dash, "\approx"] \arrow[d, dotted, dash] & \\
 &
\end{tikzcd}$$

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

“Star” simulation diagrams (corrected)

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

$$\begin{tikzcd}
c_1/s_1 \arrow[r, dash, "\approx"] \arrow[d] & C, (pc_1, \sigma_1, s'_1) \arrow[d, dashed, "+"] & \text{OR} & c_1/s_1 \arrow[r, dash, "\approx"] \arrow[d] & C, (pc_1, \sigma_1, s'_1) \arrow[dl, dash, dashed, "\approx"] \\
c_2/s_2 \arrow[r, dash, dashed, "\approx"] & C, (pc_2, \sigma_2, s'_2) & & c_2/s_2 & \text{and } M(c_2) < M(c_1)
\end{tikzcd}$$

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 c (which changes during reductions) with the compiled code C (which is fixed throughout execution).

Stuttering woes

Stuttering reduction = no machine instruction executed. These include:

$$\begin{array}{rcl}
(\texttt{SKIP}; c)/s & \rightarrow & c/s \\
(c_1; c)/s & \rightarrow & (c_2; c)/s \text{ if } c_1/s \rightarrow c_2/s \text{ stutters} \\
(\texttt{IFB true THEN } c_1 \texttt{ ELSE } c_2)/s & \rightarrow & c_1/s \\
(\texttt{IFB false THEN } c_1 \texttt{ ELSE } c_2)/s & \rightarrow & c_2/s \\
\texttt{WHILE } b \texttt{ DO } c \texttt{ END}/s & \rightarrow & \texttt{IFB } b \\
& & \texttt{THEN c; WHILE } b \texttt{ DO } c \texttt{ END} \\
& & \texttt{ELSE SKIP}
\end{array}$$

Stuttering woes

Therefore, the measure M must satisfy (at least):

$$\begin{array}{rcl}
M(\texttt{SKIP}; c) & > & M(c) \\
M(c_1; c) & > & M(c_2; c) \text{ if } M(c_1) > M(c_2) \\
M(\texttt{IFB true THEN } c_1 \texttt{ ELSE } c_2) & > & M(c_1) \\
M(\texttt{WHILE } b \texttt{ DO } c \texttt{ END}) & > & M(\texttt{IFB } b \\
& & \quad \texttt{THEN c; WHILE } b \texttt{ DO } c \texttt{ END} \\
& & \quad \texttt{ELSE SKIP})
\end{array}$$

This is impossible:

$$\begin{array}{rcl}
M(\texttt{WHILE true DO SKIP END}) & > & M(\texttt{IFB true THEN } \ldots \texttt{FI}) \\
& > & M(\texttt{SKIP; WHILE true DO SKIP END}) \\
& > & M(\texttt{WHILE true DO SKIP END})
\end{array}$$

Stuttering woes

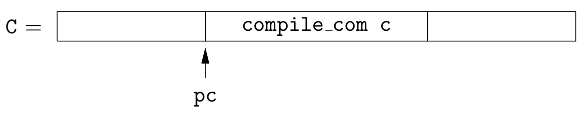
Only solution known to the teacher: change the compilation scheme for `WHILE` loops so that the machine always takes one transition at the beginning of each loop iteration.

$$\texttt{compile\_com}(\texttt{WHILE } b \texttt{ DO } c \texttt{ END}) = \texttt{Ibranch\_backward}(0); \ldots$$

This way, the `WHILE` reduction is no longer stuttering: it is simulated by the execution of the dummy `Ibranch_backward(0)` instruction.

Relating commands with compiled code

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

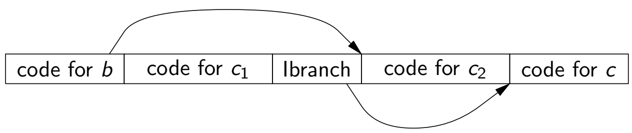


In a small-step proof: no longer works because reductions create commands that did not occur in the original program.

Spontaneous generation of commands

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

Compiled code for initial command:



This code nowhere contains the compiled code for $c_1; c$, which is:

code for c_1	code for c

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

Relating commands with compiled code

Solution: define a (nondeterministic) relation

$$\texttt{spec\_compile\_com}\ C\ c\ pc_1\ pc_2$$

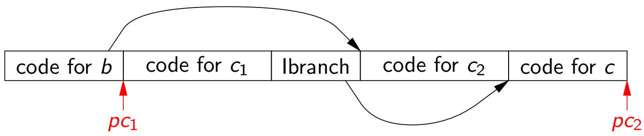
that says, roughly:

> *There exists a path from pc_1 to pc_2 in compiled code C that spells out machine instructions that execute command c.*

This relation tolerates the insertion of unconditional branches in the middle of the path.

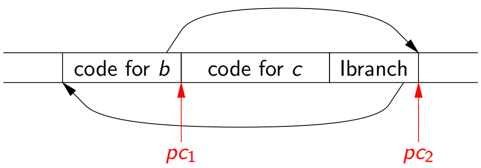
Relating commands with compiled code

According to this relation, the code below "contains" the instructions for $c_1; c$ between pc_1 and pc_2.



Relating commands with compiled code

Likewise, the code below "contains" the instructions for $c;$ `WHILE` b `DO` c `END` between pc_1 and pc_2.



Wrap up

We can finally prove a "star" simulation diagram:

```
forall C c1 s1 c2 s2 pc1 pc3,
c1 / s1 --> c2 / s2 ->
spec_compile_com C c1 pc1 pc3 ->
exists pc2,
(plus (transition C) (pc1, nil, s1) (pc2, nil, s2)
 \/ com_size c1 < com_size c2
    /\ star (transition C) (pc1, nil, s1) (pc2, nil s2))
/\ spec_compile_com C c2 pc2 pc3.
```

where the measure `com_size` is simply the number of constructors in a command.

From this diagram, forward simulation follows easily.

Conclusions

Compiler proofs based on big-step semantics:

+ Statements of lemmas are easy to find.
+ The structure of the proof follows the structure of the compiled code.
– Separate proofs for termination & divergence

Compiler proofs based on small-step semantics:

+ Termination & divergence handled at the same time.
+ Proof is minimal in terms of number of cases.
– Need to invent invariant between states & measure.
– Sometimes the compilation scheme needs tweaking for the proof to go through.

Part VI

Optimizations based on liveness analysis

Compiler optimizations

Automatically transform the programmer-supplied code into equivalent code that

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

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

Compiler optimization and static analysis

Some optimizations are unconditionally valid, e.g.:

$$\begin{array}{rcl} \mathtt{x * 2} & \to & \mathtt{x + x} \\ \mathtt{x * 4} & \to & \mathtt{x << 2} \end{array}$$

Most others apply only if some conditions are met:

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

→ need a static analysis prior to the actual code transformation.

Static analysis

Determine some properties of all concrete executions of a program.

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

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

Requirements:

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

Running example: dead code elimination via liveness analysis

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

Example

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

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

Builds on a static analysis called liveness analysis.

Optimizations based on liveness analysis

12 Liveness analysis

13 Dead code elimination

14 Advanced topic: computing exact fixpoints

15 Advanced topic: register allocation

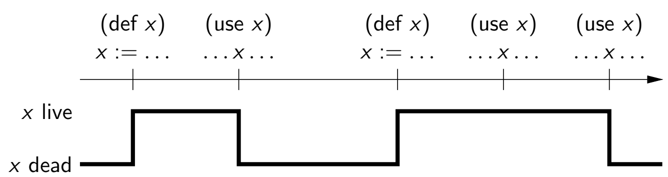
Notions of liveness

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

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

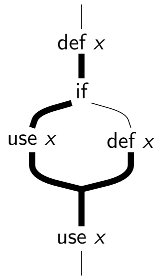
A variable is live if it is not dead.

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



Notions of liveness

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



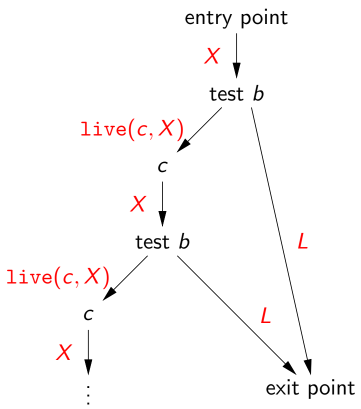
Conservatively over-approximate liveness, assuming all `if` conditionals can be true or false, and all `while` loops are taken 0 or several times.

Liveness equations

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

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

Liveness for loops



We must have:

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

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

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

To analyze `while` loops, we need to compute a 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 $F : A \to A$.

Theorem (Knaster-Tarski)

The sequence

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

converges to the smallest fixpoint of F, provided that

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

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

Problems with Knaster-Tarski

1. Formalizing and exploiting the ascending chain property
   $\to$ well-founded orderings and Noetherian induction.
2. In our case (liveness analysis), the ordering $\subset$ has infinite ascending chains: $\emptyset \subset \{x_1\} \subset \{x_1, x_2\} \subset \cdots$
   Need to restrict ourselves to subsets of a given, finite universe of variables ($=$ all variables free in the program).
   $\to$ dependent types.

We will revisit this approach later. For now, time for plan B...

The engineer's approach to post-fixpoints

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

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

A compromise between analysis time and analysis precision.

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

Optimizations based on liveness analysis

Dead code elimination

The program transformation eliminates assignments to dead variables:

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

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

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

The semantic meaning of liveness

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

Hmmm. . .

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

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

Liveness as an information flow property

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

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

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

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

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

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

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

Agreement and its properties

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

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

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

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

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

Agreement and its properties

Agreement is preserved by parallel assignment to a variable:

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

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

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


Forward simulation for dead code elimination

For terminating source programs:

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

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

$$\begin{tikzcd} st \arrow[r, dash, "\texttt{agree}\ (\texttt{live}\ c\ L)"] \arrow[d, "\texttt{ceval}\ c"'] & st_1 \arrow[d, dashed, "\texttt{ceval}\ (\texttt{dce}\ c\ L)"] \\ st' \arrow[r, dash, dashed, "\texttt{agree}\ L"] & st_1' \end{tikzcd}$$

Forward simulation for dead code elimination

The result extends simply to diverging source programs:

```
Theorem dce_correct_diverging:
  forall st c L st1,
  c / st ==> ∞ ->
  agree (live c L) st st1 ->
  dce c L / st1 ==> ∞.
```

$$\begin{tikzcd} s \arrow[r, dash, "\texttt{agree}\ (\texttt{live}\ c\ a)"] \arrow[d, "\texttt{cevalinf}\ c"'] & s_1 \arrow[d, dashed, "\texttt{cevalinf}\ (\texttt{dce}\ c\ a)"] \\ \infty & \infty \end{tikzcd}$$

(Exercises: re-do the proof using small-step or denotational semantics.)

Optimizations based on liveness analysis

Knaster-Tarski's fixpoint theorem

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.
(See Coq file `Fixpoint.v`).

The ascending chain condition in Coq

Captured by well-founded orderings.

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

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

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

Since `Acc` is an inductive predicate, `Acc` x holds iff all chains x_n `R` x_{n-1} `R` $\cdots$ `R` x_1 `R` x are finite.

Therefore, `well_founded` holds iff the ascending chain condition is true.

Moreover, induction on a derivation of `Acc` x $\Longleftrightarrow$ Noetherian induction.

Examples of well-founded orderings

```
fun (x y: nat) => x < y

fun (x y: nat) => y < x <= N

fun (x y: A)   => measure x < measure y
                  where measure: A -> nat
```

Lexicographic product of two well-founded orderings.

Ordering subsets

For liveness analysis of loops, we need to compute a fixpoint of the operator

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

over sets X of variables, ordered by inclusion.

This ordering has infinite ascending chains!

$$\emptyset \subset \{x_1\} \subset \{x_1, x_2\} \subset \cdots$$

We need to exploit two facts:

- that there are finitely many variables $x_1, \ldots, x_n$ mentioned in a given program;
- that liveness analysis manipulates only subsets of $\{x_1, \ldots, x_n\}$.

Dependent types to the rescue

Let `U : VS.t` be a finite set of variables. Define the type

```
Definition vset : Type := { X : VS.t | VS.Subset X U }
```

Elements of `vset` are pairs of an `X: VS.t` and a proof that `VS.Subset X U` holds.

Subset types, a.k.a. Sigma-types

Defined in the Coq standard library:

```
Inductive sig (A:Type) (P:A -> Prop) : Type :=
    exist : forall x:A, P x -> sig A P.

Notation "{ x  |  P }" := (sig (fun x => P)).

Definition proj1_sig (A: Type) (P: A -> Prop) (x: sig A P) : A :=
  match x with exist a b => a end.

Definition proj2_sig (A: Type) (P: A -> Prop) (x: sig A P)
         : P (proj1_sig A P x) :=
  match x with exist a b => b end.
```


Application to liveness analysis

In file `Fixpoints.v`:

- Redefine usual set operations and free variable computations over the type `vset` (of subsets of `U`).
- Show that the ordering $\subset$ over `vset` is well-founded. (The cardinal of the complement $\mathrm{card}(U \setminus X)$ strictly decreases.)
- This enables us to take smallest fixpoints of monotone operators over `vset` . . .
- . . . making it possible to compute the live variables of a `while` loop exactly.

Optimizations based on liveness analysis

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 (often stack-allocated)
  (available in unbounded quantity, but slower access)

Try to maximize the use of hardware registers.

(A crucial step for the generation of efficient machine code.)

Approaches to register allocation

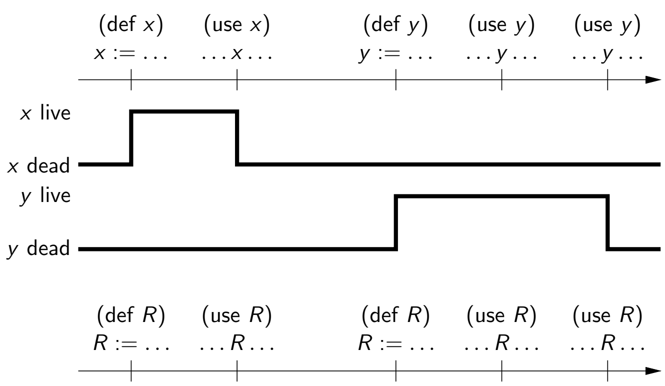
Naive approach (injective allocation):

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

Optimized approach (non-injective allocation):

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

Example of register sharing



Register allocation for IMP

Properly done:

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

Simplified as follows in this lecture:

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

The program transformation

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

The program transformation consists of:

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

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

- Coalescing:

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

Correctness conditions on the register assignment

Clearly, not all register assignments f preserve semantics.

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

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

Computes 4 instead of 3 ...

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

Agreement, revisited

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

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

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


Agreement, revisited

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

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

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

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


Agreement, revisited

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

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

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

A special case for moves

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

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

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

The interference graph

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

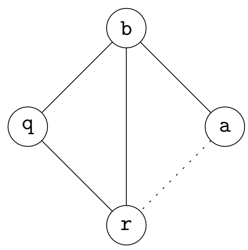
- Nodes = program variables.
- Undirected edge between x and y = x and y cannot be assigned the same location.

Chaitin's algorithm to construct this graph:

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

Example of an interference graph

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



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

Register allocation as a graph coloring problem

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

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

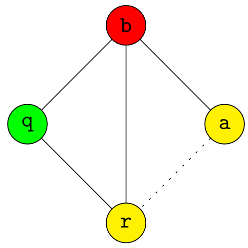
under the constraint that the two ends of an interference edge have different colors;

with the objective to

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

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

Example of coloring




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


What needs to be proved in Coq?

Full compiler proof:
formalize and prove correct a good graph coloring heuristic.

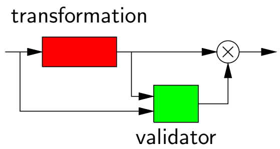
George and Appel's *Iterated Register Coalescing* $\approx$ 6 000 lines of Coq.

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

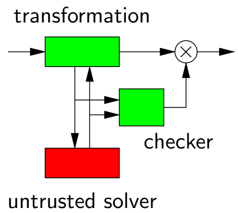
The verified transformation–verified validation spectrum

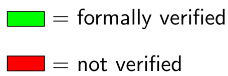
Verified transformation

Verified translation validation



External solver with verified validation





Validating candidate allocations in Coq

It is easy to write a Coq boolean-valued function

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

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

(See file `Regalloc.v`.)

Semantic preservation

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

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

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

Part VII

A generic static analyzer

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
$x = n_1 \pmod{n_2}$	congruences
$\mathtt{valid}(p[n_1 \ldots n_2])$	pointer validity
$p \ \mathtt{pointsTo}\ x$ or $p \neq q$	(non-) aliasing of pointers

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

Examples of properties that can be statically inferred

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

(a_i, c are rational constants determined by the analysis.)

"Non-functional" properties:

- Memory consumption.
- Worst-case execution time (WCET).

Using static analysis for optimization

Applying algebraic laws when their conditions are met:

`x / 4` $\rightarrow$ `x >> 2` if analysis says $\mathtt{x} \geq 0$

`x + 1` $\rightarrow$ `1` if analysis says $\mathtt{x} = 0$

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 $\mathtt{i} \neq \mathtt{j}$

`*p = a; x = *q;` $\rightarrow$ `x = *q; *p = a;`
if analysis says $\mathtt{p} \neq \mathtt{q}$

Automatic parallelization:

$$loop_1; loop_2 \rightarrow loop_1 \parallel loop_2 \quad \text{if } polyh(loop_1) \cap polyh(loop_2) = \emptyset$$

Using static analysis for verification

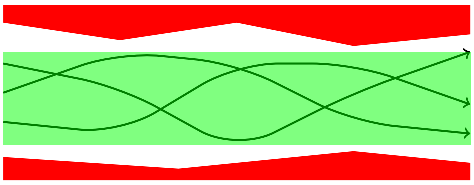
(Also known as "static debugging")

Use the results of static analysis to prove the absence of run-time errors:

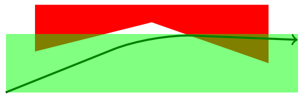
$$b \in [n_1, n_2] \wedge 0 \notin [n_1, n_2] \implies a/b \text{ cannot fail}$$

$$\texttt{valid}(p[n_1 \ldots n_2]) \wedge i \in [n_1, n_2] \implies *(\texttt{p + i}) \text{ cannot fail}$$

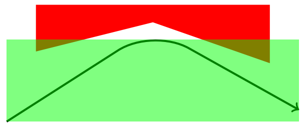
Signal an alarm otherwise.



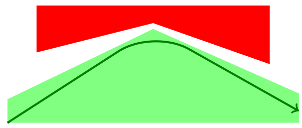
True alarms, false alarms



True alarm
(dangerous behavior)



False alarm
(imprecise analysis)



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

Some properties verifiable by static analysis

Absence of run-time errors:

- Arrays and pointers:
  - No out-of-bound accesses.
  - No dereferencing of null pointers.
  - No accesses after a `free`.
  - Alignment constraints of the processor.
- Integers:
  - No division by zero.
  - No overflows in (signed) arithmetic.
- Floating-point numbers:
  - No arithmetic overflows (infinite results).
  - No undefined operations (not-a-number results).
  - No catastrophic cancellations.

Variation intervals for program outputs.

Floating-point subtleties and their analysis

Taking rounding into account:

```
float x, y, u, v;          // x ∈ [1.00025, 2]
                           // y ∈ [0.5, 1]
u = 1 / (x - y);           // OK
v = 1 / (x*x - y*y);       // ALARM: undefined result
```

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

Second division:

$$\begin{array}{rcll} (x{*}x) & \in & [1, 4] & \text{(float rounding!)} \\ (y{*}y) & \in & [0.25, 1] & \\ (x{*}x - y{*}y) & \in & [0, 3.75] & \end{array}$$

and division by zero is possible, resuting in $+\infty$

A 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: collecting semantics

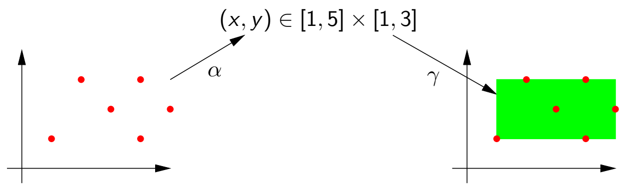
Define a semantics that collects all possible concrete states at every program point.

```
// initial value of x is N

y := 1;
                (x, y) ∈ { (N,1) }
WHILE x > 0 DO
                (x, y) ∈ { (N,1); (N−1, 2); ...; (1,2^(N−1)) }
  y := y * 2;
                (x, y) ∈ { (N,2); (N−1, 4); ...; (1,2^N) }
  x := x - 1
                (x, y) ∈ { (N−1, 2); ...; (0,2^N) }
END
                (x, y) ∈ { (0,2^N) }
```


Orthodox presentation: Galois connection

Define a lattice $\mathcal{A}, \leq$ of abstract states and two functions:

- Abstraction function α : sets of concrete states $\rightarrow$ abstract state
- Concretization function γ : abstract state $\rightarrow$ sets of concrete states



α and γ monotonic; $X \subseteq \gamma(\alpha(X))$; and $x^\sharp \leq \alpha(\gamma(x^\sharp))$.

Orthodox presentation: calculating abstract operators

For each operation of the language, compute its abstract counterpart (operating on elements of $\mathcal{A}$ instead of concrete values and states).

Example: for the + operator in expressions,

$$a_1 +^\sharp a_2 = \alpha\{n_1 + n_2 \mid n_1 \in \gamma(a_1), n_2 \in \gamma(a_2)\}$$

(...calculations omitted ...)

$$[l_1, u_1] +^\sharp [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$$

$+^\sharp$ 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 *concrete-thing* $\rightarrow$ *abstract-thing* $\rightarrow$ `Prop`.

Forget about the abstraction function α
(generally not computable; sometimes not uniquely defined.)

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

Forget about optimality; focus on soundness only.

A 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` and `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) : V.t :=
  match a with
  | 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)
  end.
```

(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) : 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 s c1) (abstr_interp s c2)
  | WHILE b DO c END =>
      fixpoint (fun x => S.join s (abstr_interp x c)) s
  end.
```


Abstract interpretation of commands

```
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 s c1) (abstr_interp s c2)
  | WHILE b DO c END =>
      fixpoint (fun x => S.join s (abstr_interp x c)) s
  end.
```

For the time being, we do not try to guess the value of a boolean test
→ consider the THEN branch and the ELSE branch as both taken
→ take an upper bound of their final states.

Abstract interpretation of commands

```
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 s c1) (abstr_interp s c2)
  | WHILE b DO c END =>
      fixpoint (fun x => S.join s (abstr_interp x c)) s
  end.
```

Let s' be the abstract state "before" the loop body c.

- entering c on the first iteration $\Rightarrow s \leq s'$.
- re-entering c after $\Rightarrow$ `abstr_interp` $s'\ c \leq s'$.

We therefore compute a post-fixpoint s' with $s \sqcup$ `abstr_interp` $s'\ c \leq s'$

Soundness results

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

```
Lemma abstr_eval_sound:
  forall st s, S.smatch st s ->
  forall a, V.vmatch (aeval st a) (abstr_eval s a).

Theorem abstr_interp_sound:
  forall c st st' s,
  S.smatch st s ->
  c / st ==> st' ->
  S.smatch st' (abstr_interp s c).
```

(Easy structural inductions on a and c.)

An example of state abstraction

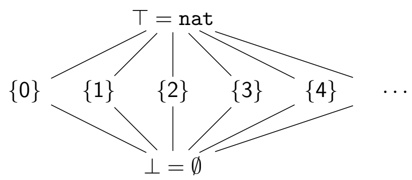
Parameterized by a value abstraction V.

Abstract states = $\bot$ | finite maps *ident* $\to$ V.t. (Default value: V.top.)

Appropriate for all non-relational analyses.

An example of value abstraction: constants

Abstract domain = the flat lattice of integers:



Obvious interpretation of operations:

$\bot +^\sharp x = x +^\sharp \bot = \bot \quad \top +^\sharp x = x +^\sharp \top = \top \quad \{n_1\} +^\sharp \{n_2\} = \{n_1 + n_2\}$

A generic static analyzer

First improvement: static analysis of boolean expressions

Our analyzer makes no attempt at analyzing boolean expressions
→ 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;
IF x = 0 THEN y := 1 ELSE y := 2 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# = ⊤ initially
IF x = 0 THEN
                      learn that x# = {0}
    y := x + 1
                      hence y# = {1}
ELSE
    y := 1
                      y# = {1} as well
FI
                      hence y# = {1}, not ⊤
```


Static analysis of boolean expressions

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

```
                      x# = ⊤ initially
WHILE not (x = 42) DO
    x := x + 1
DONE
                      learn that x# = 42# = {42}
```

More realistic example using intervals instead of constants:

```
                      x# = ⊤ = [0, ∞] initially
WHILE x <= 1000 DO
    x := x + 1
DONE
                      learn that x# = [1001, ∞]
```


Inverse analysis of expressions

`learn_from_test` s b res :
return abstract state $s' \leq s$ reflecting the fact that b (a boolean expression) evaluates to res (one of `true` or `false`).

`learn_from_eval` s a res :
return abstract state $s' \leq s$ reflecting the fact that a (an arithmetic expression) evaluates to a value matching res (an abstract value).

Examples:

$$\begin{aligned}
\texttt{learn_from_test}\ (x \mapsto \top)\ (x = 0)\ \texttt{true} &= (x \mapsto \{0\}) \\
\texttt{learn_from_test}\ (x \mapsto \{1\})\ (x = 0)\ \texttt{true} &= \bot \\
\texttt{learn_from_eval}\ (x \mapsto \top)\ (x + 1)\ \{10\} &= (x \mapsto \{9\})
\end{aligned}$$

Inverse analysis of expressions

The abstract domain for values is enriched with inverse abstract operators add_inv, etc and inverse abstract tests eq_inv, etc.

Examples with intervals:

```
le_inv [0,10] [2,5] = ([0,5], [2,5])

add_inv [0,1] [0,1] [0,0] = ([0,0], [0,0])
```


Inverse analysis of expressions

In orthodox presentation:

$$\begin{aligned}
\texttt{le_inv}\ x^\sharp\ y^\sharp &= (\alpha\{x \mid x \in \gamma(x^\sharp), y \in \gamma(y^\sharp), x \le y\}, \\
&\qquad \alpha\{y \mid x \in \gamma(x^\sharp), y \in \gamma(y^\sharp), x \le y\}) \\
\texttt{add_inv}\ x^\sharp\ y^\sharp\ z^\sharp &= (\alpha\{x \mid x \in \gamma(x^\sharp), y \in \gamma(y^\sharp), x + y \in \gamma(z^\sharp)\}, \\
&\qquad \alpha\{y \mid x \in \gamma(x^\sharp), y \in \gamma(y^\sharp), x + y \in \gamma(z^\sharp)\}
\end{aligned}$$

In Coq: see file Analyzer2.v.

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.
```


Second improvement: accelerating convergence

Consider the computation of (post-) fixpoints when analyzing loops.

Remember the two approaches previously discussed:

1. The mathematician's approach based on the Knaster-Tarski theorem. (Only if the abstract domain is well-founded, e.g. the domain of constants.)
2. The engineer's approach: force convergence to $\top$ after a bounded number of iterations.

1- is often not applicable or too slow.
2- produces excessively coarse results.

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;
WHILE unpredictable-condition DO x := x + 1 END
```

($x^\sharp$ is successively $[0,0]$ then $[0,1]$ then $[0,2]$ then ...)

Slow convergence

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

```
x := 0;
WHILE x <= 1000 DO x := x + 1 END
```

(Starting with $x^\sharp = [0,0]$, it takes 1000 iterations to reach $x^\sharp = [0,1000]$, 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.

```
x := 0;
y := 0;
WHILE x <= 1000 DO x := x + 1 END
```

In the final abstract state, not only $x^\sharp = \top$, but also $y^\sharp = \top$.

Widening

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

$$X_0 = \bot \qquad X_{i+1} = \begin{cases} X_i & \text{if } F(X_i) \leq X_i \\ X_i \mathbin{\nabla} F(X_i) & \text{otherwise} \end{cases}$$

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

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

$$[l_1, u_1] \mathbin{\nabla} [l_2, u_2] = [(\texttt{if } l_2 < l_1 \texttt{ then } 0 \texttt{ else } l_1, \; \texttt{if } u_2 > u_1 \texttt{ then } \infty \texttt{ else } u_1)]$$

Example of widening

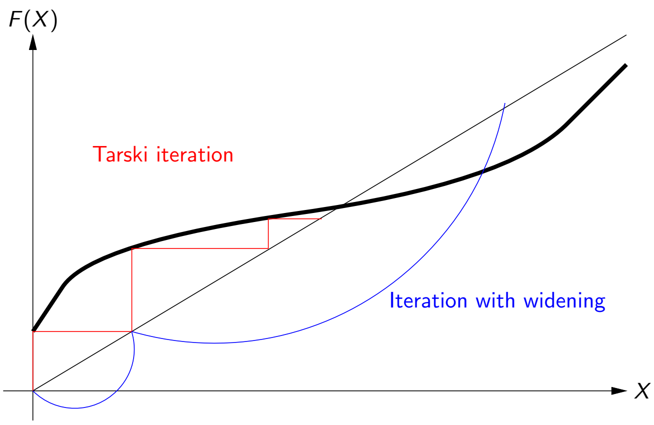
```
x := 0;
WHILE x <= 1000 DO x := x + 1 END
```

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

$X_0 = \bot$
$X_1 = X_0 \mathbin{\nabla} F(X_0) = \bot \mathbin{\nabla} [0, 0] = [0, 0]$
$X_2 = X_1 \mathbin{\nabla} F(X_1) = [0, 0] \mathbin{\nabla} [0, 1] = [0, \infty]$
X_2 is a post-fixpoint: $F(X_2) = [0, 1001] \subseteq [0, \infty]$.

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

Widening in action



Narrowing

The quality of a post-fixpoint can be improved by iterating F some more, combining it with narrowing.

A narrowing operator $\Delta : \mathcal{A} \to \mathcal{A} \to \mathcal{A}$ computes a middle point between its two arguments in such a way that the following fixpoint iteration always converges (and converges quickly):

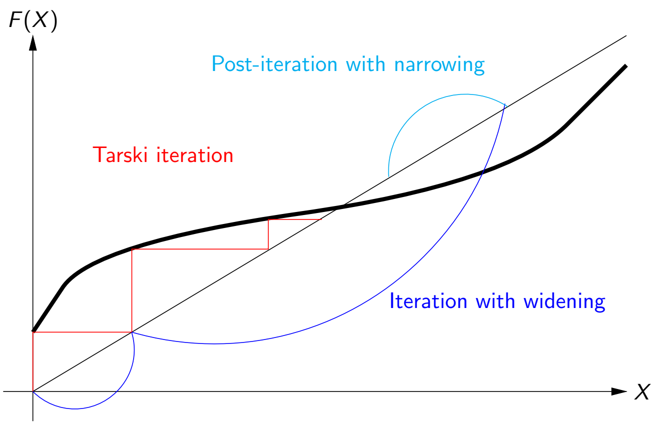
$$Y_0 = \text{a post-fixpoint} \qquad Y_{i+1} = Y_i \mathbin{\Delta} F(Y_i)$$

The limit Y of this sequence is a post-fixpoint: $F(Y) \leq Y$, as well as any of the Y_i.

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

$$[l_1, u_1] \mathbin{\Delta} [l_2, u_2] = [l_1, \texttt{if } u_1 = \infty \texttt{ then } u_2 \texttt{ else } u_1)]$$

Widening and narrowing in action



Example of narrowing

```
x := 0;
WHILE x <= 1000 DO x := x + 1 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 narrowing is $[0,\infty]$.

$Y_0 = [0,\infty]$
$Y_1 = Y_0 \,\Delta\, F(Y_0) = [0,\infty] \,\Delta\, [0,1001] = [0,1001]$
$Y_2 = Y_1 \,\Delta\, F(Y_1) = [0,1000] \,\Delta\, [0,1001] = [0,1001]$

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

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

Specification of widening and narrowing operators

For reference:

- $y \le x \,\nabla\, y$ for all x, y.
- For all increasing sequences $x_0 \le x_1 \le \ldots$, the sequence $y_0 = x_0,\ y_{i+1} = y_i \,\nabla\, x_i$ is not strictly increasing.
- $y \le x \,\Delta\, y \le x$ for all $y \le x$.
- For all decreasing sequences $x_0 \ge x_1 \ge \ldots$, the sequence $y_0 = x_0,\ y_{i+1} = y_i \,\Delta\, x_i$ is not strictly decreasing.

Coq implementation of accelerated convergence

Because we have not proved the monotonicity of `abstr_interp` nor the nice properties of widening and narrowing, we still bound arbitrarily the number of iterations.

```
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' := S.narrow 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).
```


In summary...

The abstract interpretation approach leads to highly modular static analyzers:

- The language-specific parts of the analyzer are written once and for all.
- It can then be combined with various abstract domains, which are largely independent of the programming language analyzed.
- Domains can be further combined together (e.g. by reduced product).

The technical difficulty is concentrated in the definition and implementation of domains, esp. the widening and narrowing operators.

Relational analyses are much more difficult (but much more precise!) than the non-relational analyses presented here.

Static analysis tools in the real world

General-purpose tools:

- Coverity
- MathWorks Polyspace verifier.
- Frama-C value analyzer (open source!)

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:

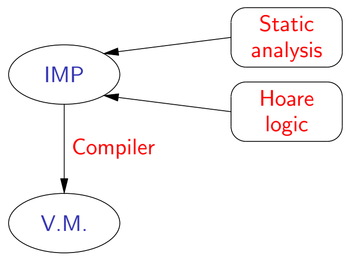
- aiT WCET (worst-case execution time)
- aiT StackAnalyzer (stack consumption)

Part VIII

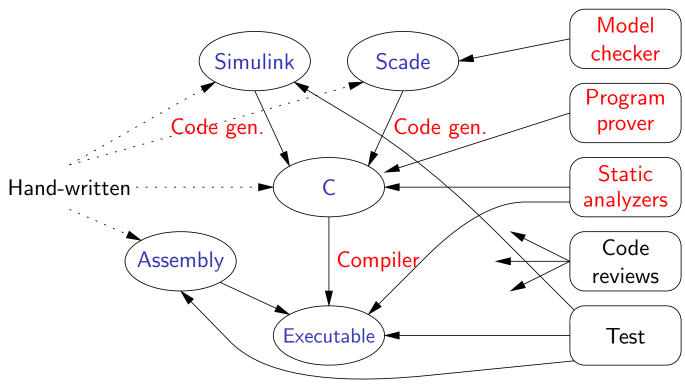
Compiler verification in the large

Compiler verification in the large

The classroom setting



The reality of critical embedded software



Requirements for qualification

(E.g. DO178-B in avionics.)

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?

Compiler verification in the large

The CompCert project

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

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

- Source language: a subset of C.
- Target language: PowerPC and ARM 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 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.

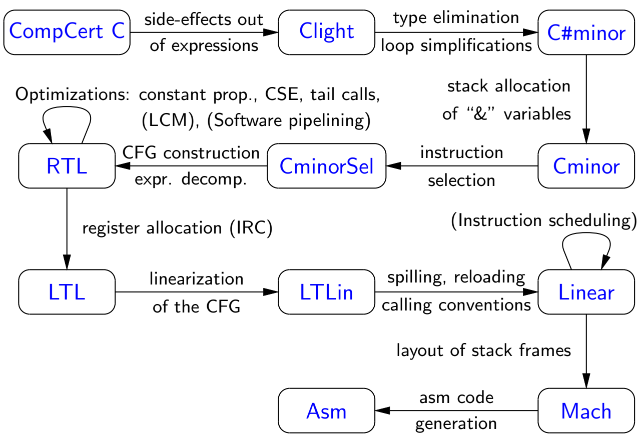
Not supported:

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

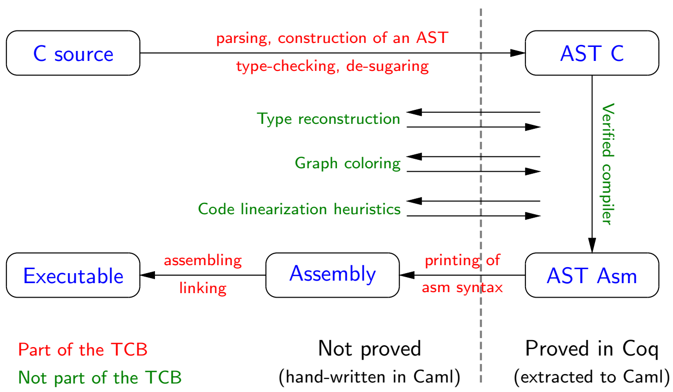
Supported via de-sugaring (not proved!):

- Block-scoped variables.
- Assignment & pass-by-value of `struct` and `union`
- Bit-fields.

The formally verified part of the compiler



The whole CompCert compiler



Verified in Coq

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

A composition of

- 13 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)

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

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).

13%	8%	17%	55%	7%
Code	Sem.	Statements	Proof scripts	Misc

Programmed in Coq

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

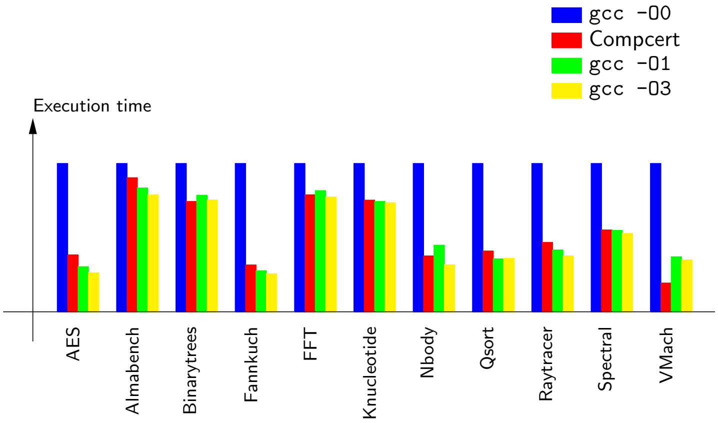
- Monads are used to handle errors and state.
- Purely functional data structures.

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

Upstream:

- Formalize some of the emulated features (bitfields, etc).
- Is there anything to prove about the C parser? preprocessor??

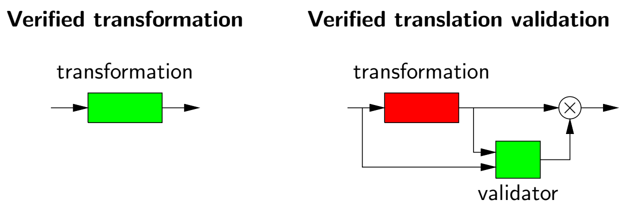
Downstream:

- Currently, we stop at assembly language with a C-like memory model.
- Refine the memory model to a flat array of bytes.
  (Issues with bounding the total stack size used by the program.)
- Refine to real machine language?
  (Cf. Moore's Piton & Gypsy projects circa 1995)

Enhancements to CompCert

In the middle:

- More static analyses, esp. for nonaliasing.
- More optimizations? Possibly using verified translation validation?



(See e.g. J.B. Tristan's verified translation validators for instruction scheduling, lazy code motion, and software pipelining.)

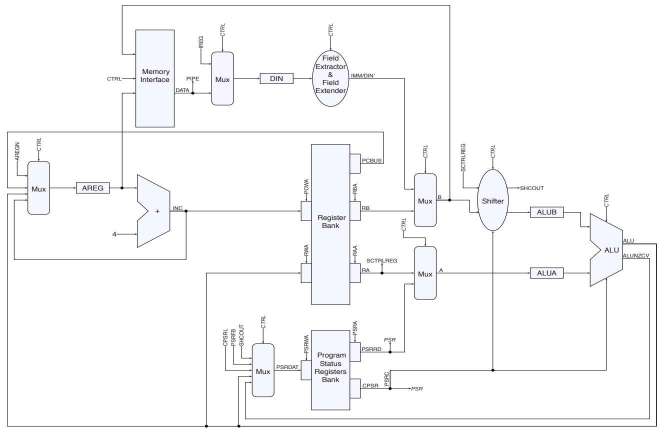
Connections with hardware verification

Hardware verification:

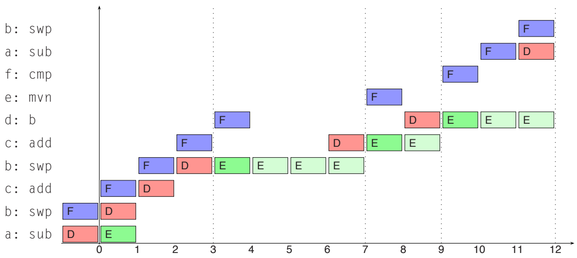
- A whole field by itself.
- At the circuit level: a strong tradition of formal synthesis and verification, esp. using model checking.
- At the architectural level (machine language semantics, memory model, . . .): almost no publically available formal specifications, let alone verifications.

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



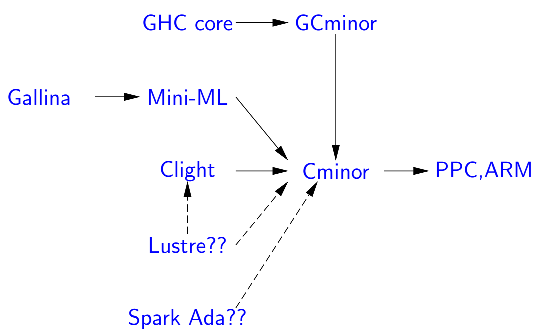
The ARM6 instruction pipeline



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

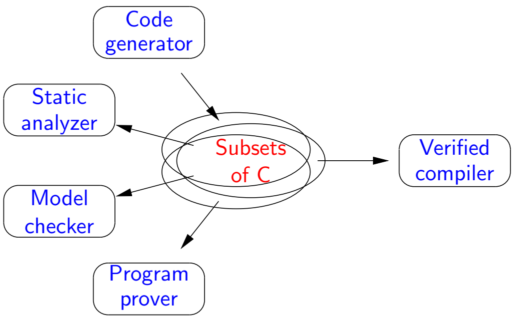
Redeeming feature: synchrony. The machine state is determined as a function of time and the initial state.

Other source languages



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

Connections with verification tools



Connections with verification tools

Consider other C-related tools involved in the production and verification of critical software: code generators, static analyzers, model checkers, program provers, . . .

- (Long term) Formally verify these tools as well?
  (E.g. verification condition generators, abstract interpreters, abstract domains, etc)
- (Medium term) Validate the operational semantics used in CompCert against the other semantics used in these tools?
  (E.g. axiomatic semantics, collecting semantics, etc)
- (More modestly) Agree on a common subset of the C language?

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.

```
x = *p + *p;        ||      *p = 1;


t1 = load(p)        ||      store(p, 1)
t2 = load(p)
x = add(t1,t2)
```

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

Towards shared-memory concurrency

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

```
store(q, 1)      ||      store(p, 1)
x = load(p)      ||      y = load(q)
```

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).
- Model of the hardware 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 $\{P\}\ c\ \{Q\}$,
but assertions P, Q control the memory footprint of commands c.

Application: the frame rule

$$\frac{\{P\}\ c\ \{Q\}}{\{P \star R\}\ c\ \{Q \star R\}}$$

Concurrent separation logic (intutions)

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

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \qquad \{P_2\}\ c_2\ \{Q_2\}}{\{P_1 \star P_2\}\ (c_1 \parallel c_2)\ \{Q_1 \star Q_2\}}$$

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 $\Rightarrow$ acquire rights to access this shared data.
Unlocking $\Rightarrow$ forego rights to access this shared data.

$$\begin{array}{rcl} \{P\} & \texttt{lock}\ L & \{P \star R(L)\} \\ \{P \star R(L)\} & \texttt{unlock}\ L & \{P\} \end{array}$$

Quasi-sequential semantics

(Hobor, Appel, Zappa Nardelli, *Oracle Semantics for Concurrent Separation Logic*, ESOP 2008).

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: quasi-sequential semantics and concrete semantics (arbitrary interleavings + weakly-consistent memory) predict the same sets of behaviors for programs provable in CSL.

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 ...

Compiler verification in the large

To finish . . .

The formal verification of compilers and related programming tools

. . . could be worthwhile,

. . . appears to be feasible,

. . . and is definitely exciting!