Language Based Security (LBT)

Secure compilation

Chiara Bodei, Gian-Luigi Ferrari

Lecture May, 13 2024

Outline

Compiler Correctness

Secure compilation

General overview*

Ken Thompson
What code can we trust?
…
Can we trust the compiler?
Solution: inspect the compiler

How can we trust a compiler is correct?

- Testing can help discover bugs, but it is not enough
- Disable any optimization: too restrictive
- So why not formally verify the compiler itself?
  - This is the aim of CompCert for instance, programmed and verified using the Coq proof assistant
- We need formal semantics for the source language and the target language

Compiler correctness

- Question: is this compilation correct?
- Is an easy or a difficult property to prove?
- By applying program proof to the compiler itself, we can obtain mathematically strong guarantees that the generated executable code is faithful to the semantics of the source program

Semantic preservation
For all source code S, if the compiler generates machine code C from source S without reporting any compilation error, then C behaves like S

Note that proving the correctness of a compiler is not a new idea

Compiler correctness

- Is this compilation correct?

```
for(i=0; i < 10; i++)
    printf("%d\n", i);
```

```
printf("42\n");
```

Compiler correctness

- Is this compilation correct? No

```
for(i=0; i < 10; i++)
    printf("%d\n", i);
```

```
printf("42\n");
```

Compiler correctness

- Is this compilation correct?

```
int n = some_pt->n;
if (some_pt == NULL)
    // Some code
use (n)
```

```
int n = some_pt->n;
use (n)
```

Compiler correctness

- Is this compilation correct? Not really

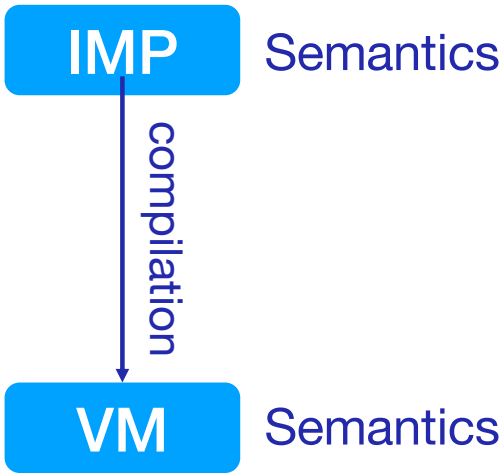
```
int n = some_pt->n;
if (some_pt == NULL)
    // Some code
use (n)
```

```
int n = some_pt->n;
use (n)
```

Compiling IMP to a simple virtual machine

We show the basics of verified compilation

- Compiling **IMP** to a simple virtual machine **VM**
- Semantic preservation notion
- Verification of an optimizing program transformation (dead store elimination)



IMP: arithmetic expressions

C-like imperative language with a notion of store

a:= n | x | a0 + a1 | a0 - a1 | a0 * a1

Abstract syntax

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

[Denotational] **Semantics**

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

IMP: boolean expressions

C-like imperative language with a notion of store

b:= true | false | a1 = a2 | a1 ≤ a2 | not b | b1 and b2

[Denotational] **Semantics**

...

Abstract syntax

```
Definition ident := string.
Inductive bexp : Type :=
|BTrue
| BFalse
| BEq (a1 a2 : aexp)
| BNeq (a1 a2 : aexp)
| BLe (a1 a2 : aexp)
| BGt (a1 a2 : aexp)
| BNot (b : bexp)
| BAnd (b1 b2 : bexp)
```

IMP: commands

C-like imperative language with a notion of store

c:= skip | x=a | c0; c1 | if b then c1 else c2| while b do c

Abstract syntax

```
Inductive com :=  | CSkip   |
CAss (x: string) (a: aexp)|
CSeq (c1 c2: com)         |
CIf (b: bexp) (c1 c2: com)|
CWhile (b: bexp) (c: com)
```

…

[Big-step operational] **Semantics**

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

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

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

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

The IMP virtual machine VM

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 (to store intermediate results temporarily)

Digression on Stack Machines

- Stack machine is one type of CPU organizations, e.g., the Java Virtual Machine is stack-based
- Stacks represent a natural and suitable tool to be used in processing well-structured code
- Machines with stacks are also required to compile and to do it efficiently
- Stacks can be used to support expression evaluation, subroutine return address storage, dynamically allocated local variable storage, and subroutine parameter passing

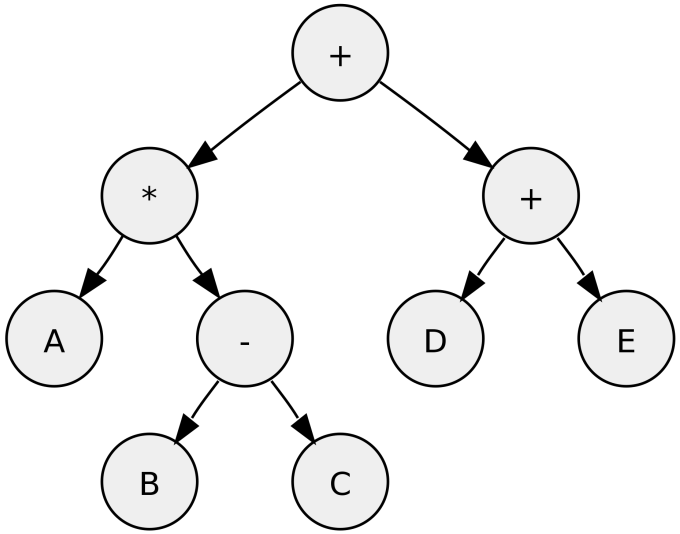
Digression on Stack Machines (cont.)

- The instruction set carries out most ALU actions that work only on the expression stack, not on data registers or main memory cells

- Arithmetic expression

A*(B-C)+(D+E)

Reverse ”polish” or post-fix notation
A B C - * D E + +



This can be very convenient for executing high-level languages because most arithmetic expressions can be easily translated into postfix notation

Digression on Stack Machines (cont.)

10 * (5-3) + (2+4) becomes

10 5 3 - * 2 4 +

	STACK
push	
10	10
push 5	5 10
push	
3	3 5 10
subtract	2 10
multiply	20
push 2	2 20
push 2	4 2 20
add	6 20
add	26

The IMP virtual machine VM

- Translation to a stack-based expression language with a program counter, a store and a stack
- **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

...

- The code for an arithmetic expression a executes in sequence and deposits the value of a at the top of the stack, e.g.:
- `1 + 2` is translated into `Iconst 1 :: Iconst 2 :: Iadd :: nil`

The IMP virtual machine VM (cont.)

A transition relation represents the execution of one instruction.

small-step semantics

```
Definition stack := list Z.
Definition store := ident → Z.
Definition config := (Z * stack * store).

Inductive transition (C: code): config -> Prop := |
trans_const: ∀ pc stack s n, instr_at C pc = Some(Iconst n)
→ transition C (pc, stack, s) (pc + 1, n :: stack, s)
```

...

Intially: pc = 0, initial store, empty stack

Finally: pc points to a Ihalt instruction, empty stack

Example of compilation

x = x+1 is translated into

```
Ivar(x):: Iconst 1 :: Iadd :: Isetvar(x):: nil
```

x: x+1 —compilation→

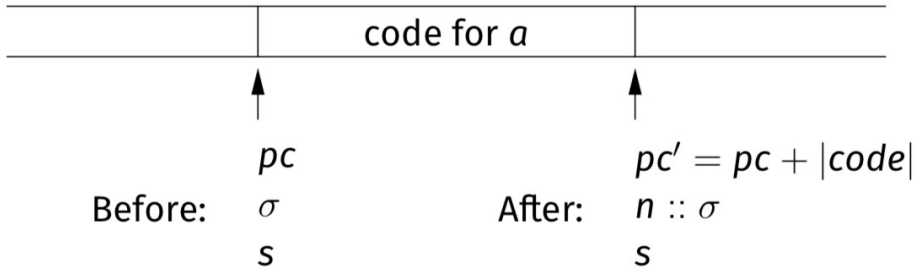
stack	ϵ	12	1 12	13	ϵ
store	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 13$
p.c.	0	1	2	3	4
code	Ivar(x);	Iconst(1);	Iadd;	Isetvar(x);	

At each instruction the program counter is incremented by one, unless the instruction prescribes a branch

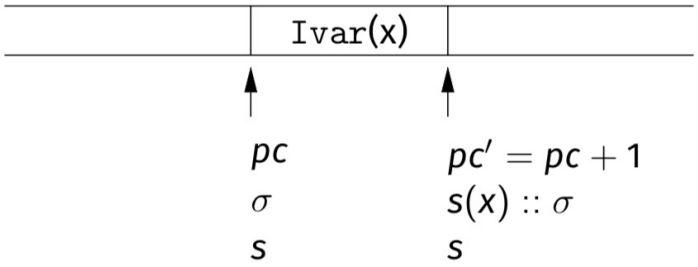
Compilation of arithmetic expressions

Compilation is just translation to “reverse Polish notation”

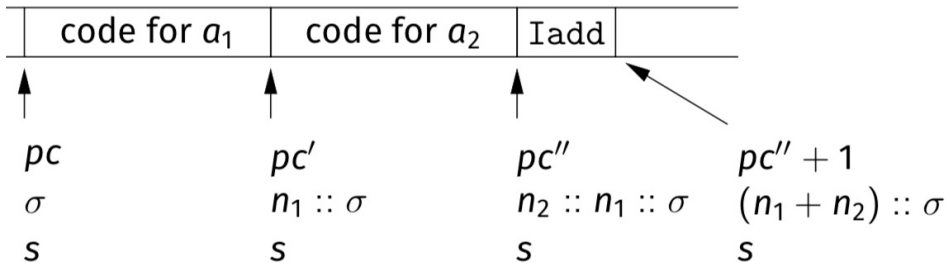
General contract: if a evaluates to n in store s,



Base case: if $a = x$,



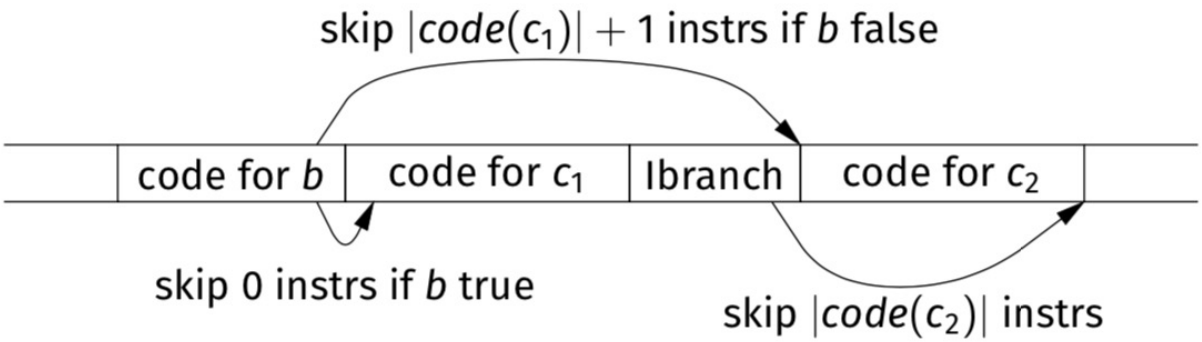
decomposition: if $a = a_1 + a_2$,



Similarly for Boolean expressions

Compilation of if then else

Code for IFTHENELSE b c_1 c_2:



Similarly for the other commands

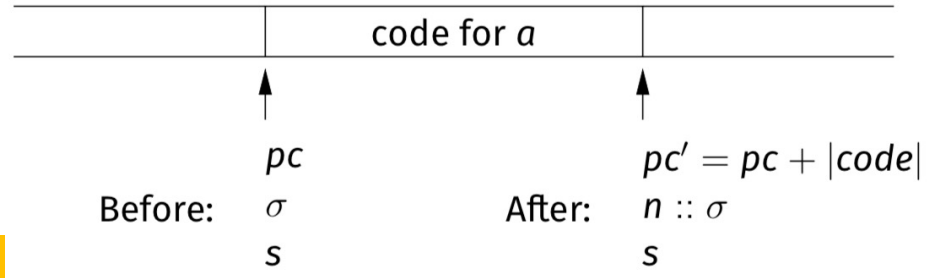
Verifying the compilation of expressions

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

Proof: a simple induction on the structure of *a*.

The generated code must behave as prescribed by the semantics of the source program:
if a evaluates to n in the store,
then in the translation
n ends up on the stack and vice versa

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



This is not enough to conclude that the compiler is correct!

Correctness, formally

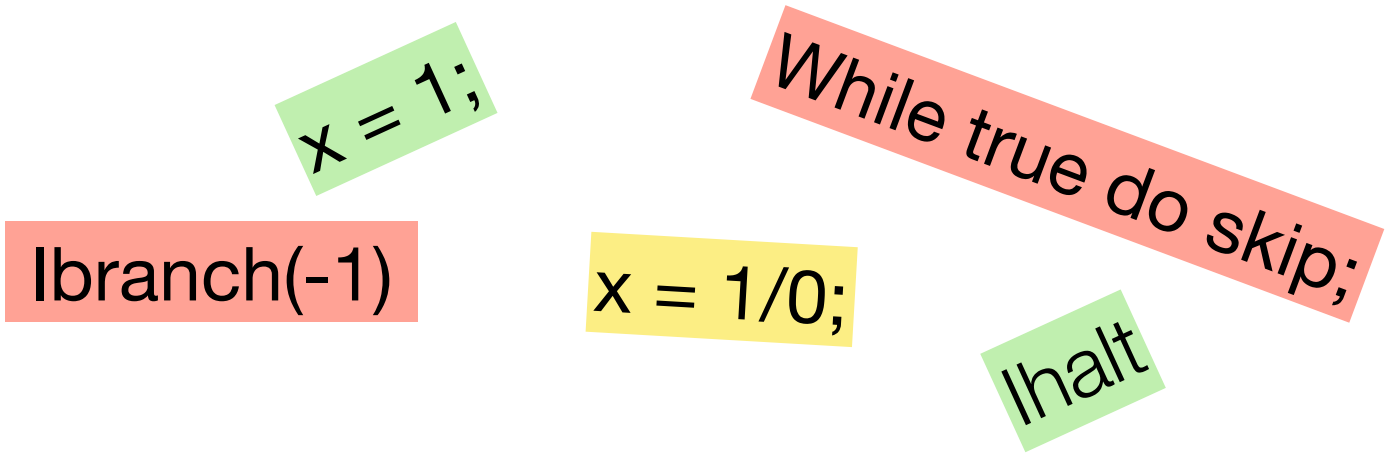
Formally, a compiler is a function $[\![\cdot]\!]_T^S$ that translates programs s written in a source language S into programs t written in a target language T

What does it mean that a compiler is correct? It means that the compiled code behaves as the source code

Correctness

A program may:

- Terminate and return a result value
- Diverge (= indefinitely run, e.g. infinite loop)
- Go wrong (= crash, e.g. division by zero, or invalid memory access)



The previous approach only works with normal terminating programs

Semantic Preservation

What should be preserved?

Observable behaviors

What should be preserved?

Answer: observable behaviors $\mathcal{B}(s)$

- Hence, observable behaviors are normal termination or abnormal termination and divergence

- Inputs and outputs can be observed as well

Observable behaviors

We define the behaviour of a source code in terms of the set of observable actions, e.g., I/O actions, memory operations, …

For example, in terms of traces, composed by strings of observable actions

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

```
x := 1;  x := 2;        ≈        x := 2;
```

(trace: ϵ) (trace: ϵ)

```
print(1); print(2);     ≉        print(2);
```

(trace: out(1).out(2)) (trace: out(2))

Semantic Preservation (cont.)

A correct compilation is semantics preserving if the generated code behaves as prescribed by the semantics of the source

Preservation

A correct compilation is semantics preserving if the generated code behaves as prescribed by the semantics of the source

How to characterize preservation?
Answer: we need to compare the behaviors of two programs, by using simulations and program equivalences

Two programs are **equivalent** when they behave the same, even if they are different (same semantics and possibly different syntax)

Digression on equivalences

Suppose to have two tea/coffee machines:

- **M_1** allows you to:
  - Put a coin
  - Choose to press the tea or the coffee button
  - After providing you with the required beverage, the machine is again waiting for requests
- **M_2** allows you to:
  - Put a coin
  - When a tea or coffee button is pressed, non-deterministically delivers tea or coffee
  - …
- Do **M_1** and **M_2** behave the same?

Digression on equivalences

What does it mean that two machines “are the same” for an observer?

Intuitively, if you do something with one machine, you must be able to do the same with the other, and on the two states which the machines evolve to, the same is again true

Bisimulation (observation equivalence)

How to characterize preservation?

Answer: simulations and program equivalences

Definition (Bisimulation)

The source program S and the compiled program C have exactly the same observable behaviors (they are equivalent).

Every possible behavior of S is a possible behavior of C

Every possible behavior of C is a possible behavior of S

$$\mathcal{B}([[s]]) = \mathcal{B}(s)$$

Very strong notion of semantic preservation: often too strong in practice

Forward Simulation

Definition (Forward Simulation)
Every possible behavior of the source program S is a possible behavior of the compiled program C

$$\mathcal{B}([[s]]) \subseteq \mathcal{B}(s)$$

Easier to prove

Our compilation: if the source code terminates (with the same final store)/diverges, then the compiled code terminates/diverges

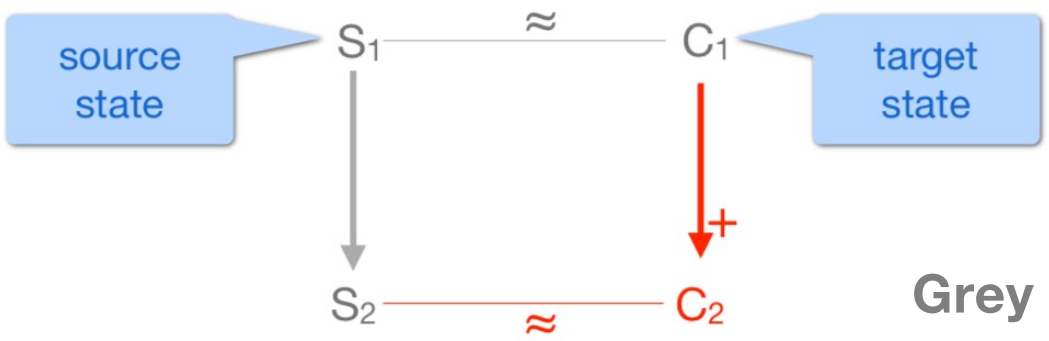
But, what if the compiled code C has more behaviors (maybe udesirable) than the source S, e.g., what if C can terminate or go wrong?

Simulation diagrams

Behaviors are defined in terms of sequences of transitions.

Forward simulation can be proved as follows:

- show that every transition in S is simulated by some transitions in C
- while preserving an invariant ≈ between the states of S and C

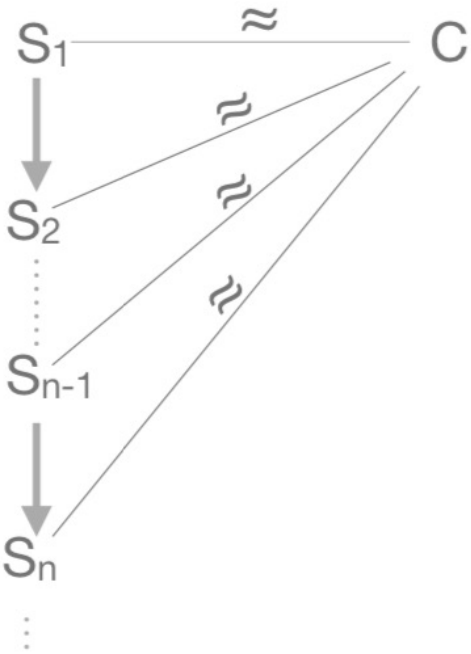


Grey = hypotheses
Red= conclusions

For instance, for the translation of x = x+1 it works

Infinite stuttering

- The source program diverges but the compiled code can terminate normally or by going wrong
- This denotes an incorrect optimization of a diverging program, e.g. compiling (while true skip) into skip
- To solve this problem, we need continuation-based small-step semantics of IMP



Reducing non-determinism during compilation

C does not guarantee the order of evaluation of the operands of the + operator

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

int y = f() + g();
```

`f()+ g()` can be evaluated in two ways

- It returns 1, if `f()` is called first
- It returns -1, if `g()` is called first

This is a form of non-determinism

The compiler chooses one behaviour*: hence the compiled code has fewer behaviours. Forward simulation and bisimulation fail

*This is a form of refinement

Reducing non-determinism in concurrent setting

Shared Counter

```
class C {
  private int counter;
  void inc() { synchronized (this) {
   counter++; }
   }
  int get() {synchronized(this){
              return counter;}
   }
  void set(int val) { synchronized (this) {
   counter= val }
   }
:
}
...
C c = new C(0);
.....
```

There is a common sub-expression that can be eliminated

THREAD 1

```
:
aval = c.get() +1:
bval = c.get()+1;
:
```

THREAD 2

```
:
c.set(1);
:
```

If c is initially 0, (aval,bval) can range in {(1,1), (1,2), (2,2)}

Reducing non-determinism in concurrent setting

Shared Counter

```
class C {
  private int counter;
  void inc() { synchronized (this) {
   counter++; }
   }
  int get() {synchronized(this){
              return counter;}
   }
  void set(int val) { synchronized (this) {
   counter= val }
   }
:
}
...
C c = new C(0);
.....
```

THREAD 1

```
:
aval = c.get() +1:
bval = aval;
:
```

THREAD 2

```
:
c.set(1);
:
```

If c is initially 0, (aval,bval) can range in {(1,1), ~~(1,2)~~, (2,2)}

Backward Simulation (a.k.a. refinement)

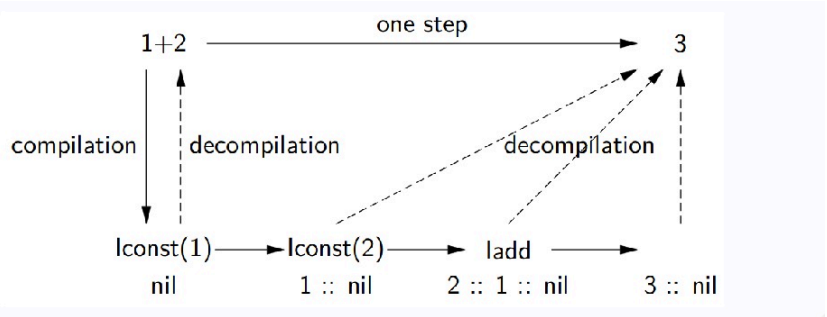
Definition (Backward Simulation)
Every possible behavior of the compiled program C is a possible behavior of the source program S. C may have fewer behaviors than S

$$\mathcal{B}([[s]]) \supseteq \mathcal{B}(s)$$

- Our compilation: if the compiled code terminates/diverges, then the source code terminates/diverges
- Backward simulation can be proved similarly to forward simulation, but simulates transitions of C by transitions of S.
- Backward simulation suffices to show the preservation of properties established by source-level verification

Backward Simulation (a.k.a. refinement)

- Difficult to prove: all steps that the compiled code can take can be traced back to steps the source program can take
- Problematic: one single step may be translated into many steps (e.g., 1+2 is one step in IMP, but several instructions in the VM)



Solution: introduce a **decompilation**, defined on all intermediate states, and that is left-inverse of compilation (comp ∘ decomp = id)

Fwd simulation + determinism = bisimulation

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

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

Our compilation: if the source code terminates/diverges, then the compiled code terminates/diverges

Once solved the stuttering issue, all IMP programs, terminating or not, are correctly compiled

Should “going wrong” behaviors be preserved?

As we know, compilers usually “optimize away” going-wrong behaviors, maybe under the assumption that there are none, as in C

There are weaker simulation properties, by restricting ourselves to the cases in which the computations do not go wrong (“invoke undefined behavior”)

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

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

A program is safe when it either terminates or diverges

Optimizations

Optimizations automatically transform the programmer-supplied code into equivalent code that should be more efficient

For instance, DSE is a form of compilation from initial programs to programs where dead store have been removed

Back to dead store elimination

DSE is a form of **compilation** from initial programs to programs where dead store have been removed

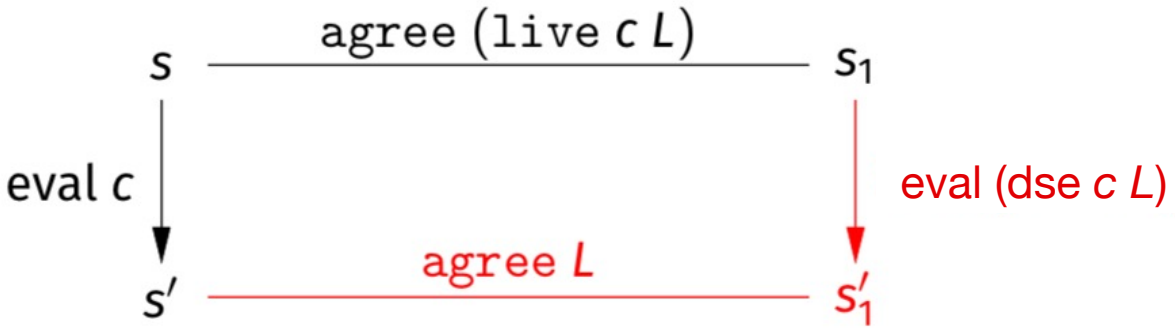
Liveness can be indeed seen as an information flow property

Semantic preservation for dead store elimination (DSE) relates executions of c and c in absence of dead stores:

Assume initial states agree on live variables “before” c:
Then, the two executions terminate on final states that agree on live variables “after” c

Fwd simulation for dead store elimination

Forward simulation: the execution of c after removal of dead stores simulates the execution of c while preserving the agree relation between the stores



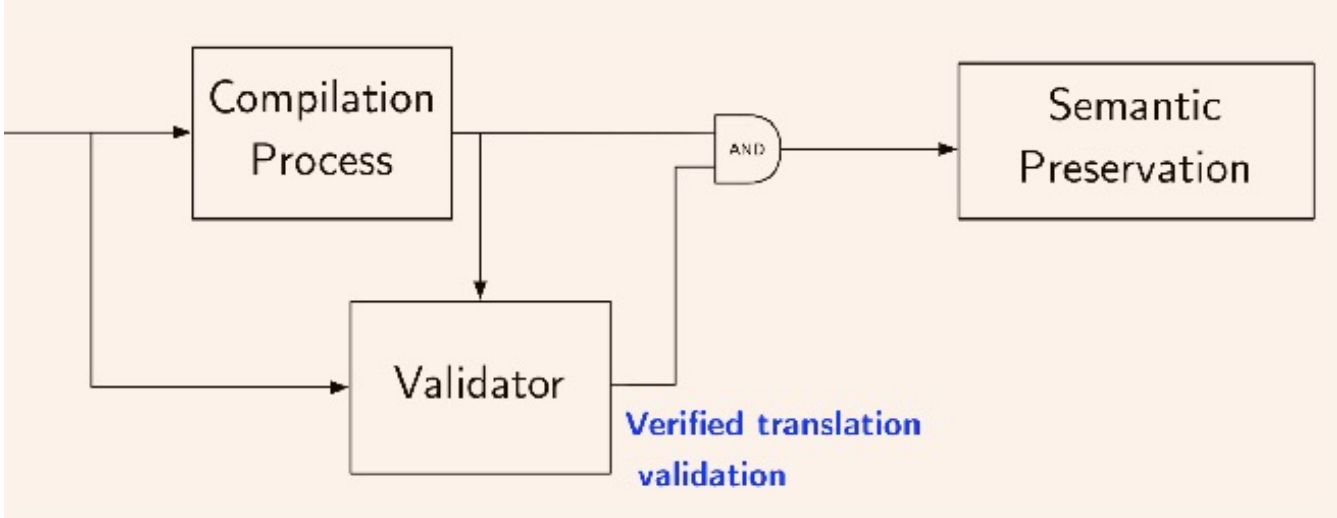
Real verified compilers

CompCert: fully written in Coq, it compiles C in many real-world architectures. It presents mechanized proofs of correctness

CakeML: compiles a subset of Standard ML. It uses HOL4 for mechanized proofs

Alternative approaches: Translation validation

Considering just a single run of the compiler at the time on the single program



It takes the source and compiled programs as input and checks whether the source program semantics is preserved in the compiled one

Beyond whole programs

- Many real-world programs are partial, i.e. they are not written as a whole by programmers
- Partial programs are made "full" by linking with a context
- Contexts model external definitions from standard libraries, code written by third parties, external components, ...

Issue: All the above cannot deal with partial programs

In case of partial programs resort to prove compiler correctness without the whole-program restriction

Separate compilation correctness

- Separately compile the source context and the source program and link them together

- A separate compiler independently translates a programs components in a way that preserves correctness of the program as a whole
- Compile the partial source program
- Compile the source context with the same compiler
- Link them together
- Correctness of the result is guaranteed

Compositional compilation correctness

- Aim: to guarantee correct compilation of components

- Compile the source program, choose a target context that correctly implements the source one and then link them together
  - Compile the partial program
  - Choose a target context that correctly implements the source one
  - Link them together
- Correctness of the result is guaranteed!
- This variant is stronger, more useful and more difficult to obtain
- Note that in Separate Compilation linking is only defined with linking contexts produced by the same compiler

Bibliography

- Xavier Leroy. *Proving the correctness of a compiler*. EUTypes Summer School 2019
- Sandrine Blazy. *Verified compilation An introduction to CompCert*. OPLSS, Eugene, 2023-07-05

End