

# From INTERPRETATION TO COMPILATION

INTERPRETER FOR  $L$ : a program  $\mathcal{E}$  in a language  $L'$   
executing  $L$ -programs

$\mathcal{E}(P)$  = result of  $P$

COMPILER FOR  $L$  IN  $L'$ : a program  $\mathcal{C}$  translating  $L$ -programs to  
 $L'$ -programs

Usually

$L$ : source language

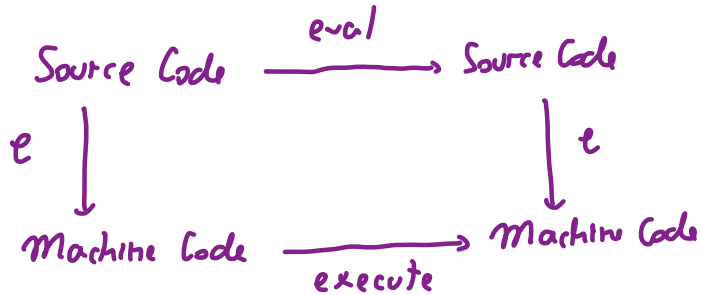
$L'$ : machine language (executable by machine)

Focus on efficiency, energy consumption, code optimisation

INTENSIONAL PROGRAM BEHAVIOURS

What is compiler correctness?

↳ The generated code must meet the semantics of the source program



$$\text{execute}(e(p)) \stackrel{?}{=} e(\text{eval}(p))$$

⇒ Need formal semantics of both  $\begin{cases} \text{source language} \\ \text{machine language} \end{cases}$

Why compiler correctness?

"We tested thirteen production-quality C-compilers and, for each, found situations in which the compiler generated incorrect code [...]"

Eide & Røgehaug Emsoft

"To improve the quality of C compilers, we created Csmith [...]  
Every compiler we tested was found to crash and also to silently generate wrong code [...]"

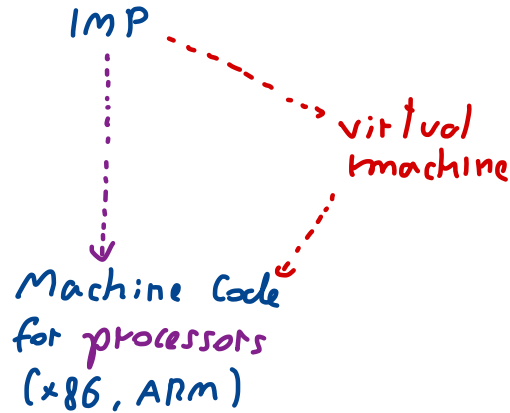
Yang, Chen, Eide & Røgehaug

PLDI 2011

Our goal : Formal verification of a (non-optimising)  
compiler for Imp

Use techniques that scale to real world languages

How to compile IMP?



- similar to a real machine
  - program = sequences of instructions

⇒ no compound exp  
no control structure

- close to the source lang.  
(instructions reflect base operations in source lang.)

# IMP virtual machine

4 components:

- Code  $C$  : list of instructions
- Code pointer  $pc$  : position of the instruction in  $C$  we are executing
- Store  $s$  : association variables-values
- Stack  $\sigma$  : list of integer values (to save intermediate results)

## INSTRUCTIONS

Inductive *instr* : Type :=

- | Iconst (n:  $\mathbb{Z}$ )
- | Ivar (x: ident)
- | Isetvar (x: ident)
- | Iadd
- | Iopp
- | Ibranch (d:  $\mathbb{Z}$ )
- | Ibeq (d1:  $\mathbb{Z}$ ) (d2:  $\mathbb{Z}$ )
- | Ible (d1:  $\mathbb{Z}$ ) (d2:  $\mathbb{Z}$ )
- | Ihalt

- "push integer n"
- "push value of x"
- "pop integer and assign it to x"
- "pop two integers; push their sum"
- "pop one integer; push its opposite"
- "skip 'forward' of d instructions"
- "pop two integers; skip d1 instructions if equal; skip d2 instructions if not"
- "stop"

NB. All instructions implement PC by 1; branching instructions of d+1



Ex.

$\langle \text{code}, \text{pc}, \text{store}, \text{stack} \rangle$  } configurations : state of the machine

$\langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 0, x \mapsto 12, [] \rangle$



Ex.

$\langle \text{code}, \text{pc}, \text{store}, \text{stack} \rangle$  } configurations : state of the machine

$\langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 0, x \mapsto 12, [] \rangle$



$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 1, x \mapsto 12, [12] \rangle$



Ex.

$\langle \text{code}, \text{pc}, \text{store}, \text{stack} \rangle$  } configurations : state of the machine

$\langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 0, x \mapsto 12, [] \rangle$



$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 1, x \mapsto 12, [12] \rangle$



$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 2, x \mapsto 12, [1, 12] \rangle$



Ex.

$\langle \text{code}, \text{pc}, \text{store}, \text{stack} \rangle$  } configurations : state of the machine

$\langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 0, x \mapsto 12, [] \rangle$

↑

$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 1, x \mapsto 12, [12] \rangle$

↑

$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 2, x \mapsto 12, [1, 12] \rangle$

↑

$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 3, x \mapsto 12, [13] \rangle$

↑

Ex.

$\langle \text{code}, \text{pc}, \text{store}, \text{stack} \rangle$  } configurations : state of the machine

$\langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 0, x \mapsto 12, [] \rangle$

$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 1, x \mapsto 12, [12] \rangle$

$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 2, x \mapsto 12, [1, 12] \rangle$

$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 3, x \mapsto 12, [13] \rangle$

$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 4, x \mapsto 13, [] \rangle$

Ex.

$\langle \text{code}, \text{pc}, \text{store}, \text{stack} \rangle$  } configurations : state of the machine

$\langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 0, x \mapsto 12, [] \rangle$

$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 1, x \mapsto 12, [12] \rangle$

$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 2, x \mapsto 12, [1, 12] \rangle$

$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 3, x \mapsto 12, [13] \rangle$

$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 4, x \mapsto 13, [] \rangle$

$\rightarrow \langle \text{Ivar}(x); \text{Iconst}(1); \text{Iadd}; \text{Isetvar}(x); \text{Ibranch}(-5), 0, x \mapsto 13, [] \rangle$

We specify the vm using *operational semantics*

→ The  $pc$ -th element of  $C$  is  $I(\text{const}(m))$

$$C[pc] = I(\text{const}(m))$$

---

$$\langle C, pc, s, \sigma \rangle \rightarrow \langle C, pc+1, s, m :: \sigma \rangle$$

$$C[pc] = I\text{Var}(x)$$

---

$$\langle C, pc, s, \sigma \rangle \rightarrow \langle C, pc+1, s, s(x) :: \sigma \rangle$$

$$C[pc] = I\text{SetVar}(x)$$

---

$$\langle C, pc, s, m :: \sigma \rangle \rightarrow \langle C, pc+1, s[x \mapsto m], \sigma \rangle$$

$$C[pc] = I_{add}$$


---

$$\langle C, pc, s, m :: m :: \sigma \rangle \rightarrow \langle C, pc+1, s, (m+m) :: \sigma \rangle$$

$$\vdots$$

$$C[pc] = I_{branch}(d)$$


---

$$\langle C, pc, s, \sigma \rangle \rightarrow \langle C, pc+1+d, s, \sigma \rangle$$

$$\vdots$$



How to encode op. sem in Coq?

Definition **code** := list instruction

Definition **store** := ident  $\rightarrow$  option  $\mathbb{Z}$

Definition **stack** := list  $\mathbb{Z}$

Definition **config** :=  $\mathbb{Z} * \text{store} * \text{stack}$  ( $\leadsto$  no code!)

Inductive **Transition** ( $C : \text{code}$ ):  $\underbrace{\text{config} \rightarrow \text{config} \rightarrow \text{Prop}}_{\text{A relation on configuration}} :=$

| rule\_1const :  $\forall pc \ s \ \sigma,$   
                  if  $C[pc] = \text{Some } (Ivar \ n)$