FROM INTERPRETATION TO COMPILATION

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

$$\mathcal{I}(P) = \text{result of } P$$

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

Usually

$\mathcal{L}$: source language

$\mathcal{L}'$: machine language (executable by machine)

Focus on $\underbrace{\text{efficiency, energy consumption, } \boxed{\text{code optimisation}}}_{\text{INTENSIONAL PROGRAM BEHAVIOURS}}$

What is compiler correctness?

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

$$\begin{tikzcd}
\text{Source Code} \arrow[r, "\text{eval}"] \arrow[d, "e"'] & \text{Source Code} \arrow[d, "e"] \\
\text{Machine Code} \arrow[r, "\text{execute}"'] & \text{Machine Code}
\end{tikzcd}$$

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

$\Rightarrow$ 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 & Regehr EMSOFT 2008

"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 & Regehr
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 ?

$$\begin{tikzcd}[column sep=large] \text{IMP} \arrow[d, dotted] \arrow[r, dashed] & \text{virtual machine} \arrow[dl, dotted] \\ \begin{array}{c}\text{Machine Code} \\ \text{for processors} \\ \text{(x86, ARM)}\end{array} & \end{tikzcd}$$

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

    $\Longrightarrow$ 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 σ : list of integer values (to save intermediate results)

Instructions

```
Inductive instr : Type :=
  | Iconst (n : Z)                    "push integer n"
  | Ivar (x : ident)                  "push value of x"
  | Isetvar (x : ident)               "pop integer and assign it to x"
  | Iadd                              "pop two integers; push their sum"
  | Iopp                              "pop one integer; push its opposite"
  | Ibranch (d : Z)                   "skip "forward" of d instructions"
  | Ibeq (d1 : Z) (d2 : Z)            "pop two integers; skip d1 instructions
                                       if equal; skip d0 instructions if not"
  | Ible (d1 : Z) (d2 . Z)            ...
  | Ihalt                             "stop"
```

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

Ex.

$\langle$ code, pc, store, stack $\rangle$ } configurations : states of the machine

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

Ex.

$\langle$ code, pc, store, stack $\rangle$ } configurations : states of the machine

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

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

Ex.

$\langle code, pc, store, stack \rangle$ } configurations : states of the machine

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

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

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

Ex.

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

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

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

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

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

Ex.

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

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

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

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

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

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

Ex.

$\langle code, pc, store, stack \rangle$ } configurations : states of the machine

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

$\longrightarrow$ $\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 1, $x \mapsto 12$, [12] $\rangle$

$\longrightarrow$ $\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 2, $x \mapsto 12$, [1, 12] $\rangle$

$\longrightarrow$ $\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 3, $x \mapsto 12$, [13] $\rangle$

$\longrightarrow$ $\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 4, $x \mapsto 13$, [] $\rangle$

$\longrightarrow$ $\langle$ Ivar(x); Iconst(1); Iadd; Isetvar(x); Ibranch(−5), 0, $x \mapsto 13$, [] $\rangle$

We specify the vm using operational semantics

The pc-th element of C is $\mathrm{IConst}(n)$

$$\frac{C[pc] = \mathrm{IConst}(n)}{\langle C, pc, s, \sigma \rangle \to \langle C, pc+1, s, n :: \sigma \rangle}$$

$$\frac{C[pc] = \mathrm{IVar}(x)}{\langle C, pc, s, \sigma \rangle \to \langle C, pc+1, s, s(x) :: \sigma \rangle}$$

$$\frac{C[pc] = \mathrm{ISetvar}(x)}{\langle C, pc, s, n :: \sigma \rangle \to \langle C, pc+1, s[x \mapsto n], \sigma \rangle}$$

$$\frac{C[pc] = \mathrm{Iadd}}{\langle C, pc, s, n :: m :: \sigma \rangle \to \langle C, pc+1, s, (n+m) :: \sigma \rangle}$$

$$\vdots$$

$$\frac{C[pc] = \mathrm{Ibranch}(d)}{\langle C, pc, s, \sigma \rangle \to \langle C, pc+1+d, s, \sigma \rangle}$$

$$\vdots$$

How to encode op. sem in Coq?

Definition code := list instruction
Definition store := ident → option Z
Definition stack := list Z
Definition config := Z * store * stack   (⇝ no code!)

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

| rule_iconst : ∀ pc s σ,
if c[pc] = Some (Ivar (n))