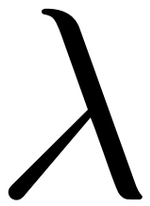
Machine code, formal verification and decompilation

Lecture 4

MPhil ACS & Part III course, Functional Programming: Implementation, Specification and Verification

Magnus Myreen
Michaelmas term, 2013

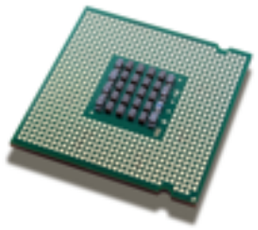
Target: verified machine code



High-level specification
(e.g. operational semantics from Lec 2-3)

Target: verified machine code

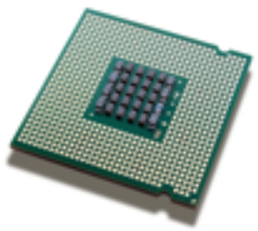
λ High-level specification
(e.g. operational semantics from Lec 2-3)



Concrete machine-code implementation
(e.g. x86, ARM, MIPS, Power)

Target: verified machine code

λ High-level specification
(e.g. operational semantics from Lec 2-3)



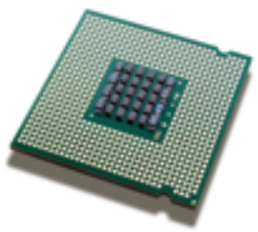
Concrete machine-code implementation
(e.g. x86, ARM, MIPS, Power)

Operational semantics of machine code

Target: verified machine code

λ

High-level specification
(e.g. operational semantics from Lec 2-3)



Concrete machine-code implementation
(e.g. x86, ARM, MIPS, Power)

Operational semantics of machine code

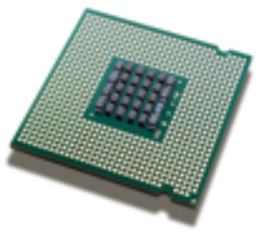
} connected by formal proof

Target: verified machine code

λ

High-level specification
(e.g. operational semantics from Lec 2-3)

memory management (GC)
parsing/printing
type inference
interpretation
compilation (dynamic)



Concrete machine-code implementation
(e.g. x86, ARM, MIPS, Power)

Operational semantics of machine code

} connected by formal proof

Floyd–Hoare logic

A programming logic

- a formal system for reasoning about program text
- Hoare: "An axiomatic basis for computer programming", 1969
- very influential still today (active research area)

Floyd–Hoare logic

A programming logic

- a formal system for reasoning about program text
- Hoare: "An axiomatic basis for computer programming", 1969
- very influential still today (active research area)

Hoare triple (the judgement of Floyd-Hoare logic):

$$\{ P \} C \{ Q \}$$

Floyd–Hoare logic

A programming logic

- a formal system for reasoning about program text
- Hoare: "An axiomatic basis for computer programming", 1969
- very influential still today (active research area)

Hoare triple (the judgement of Floyd-Hoare logic):

$$\{ P \} C \{ Q \}$$

precondition

Floyd–Hoare logic

A programming logic

- a formal system for reasoning about program text
- Hoare: "An axiomatic basis for computer programming", 1969
- very influential still today (active research area)

Hoare triple (the judgement of Floyd-Hoare logic):

$$\{ P \}\, C \,\{ Q \}$$

precondition

code

Floyd–Hoare logic

A programming logic

- a formal system for reasoning about program text
- Hoare: "An axiomatic basis for computer programming", 1969
- very influential still today (active research area)

Hoare triple (the judgement of Floyd-Hoare logic):

$$\{ P \}\, C \,\{ Q \}$$

precondition code postconditio

Floyd–Hoare logic

A programming logic

- a formal system for reasoning about program text
- Hoare: "An axiomatic basis for computer programming", 1969
- very influential still today (active research area)

Hoare triple (the judgement of Floyd-Hoare logic):

$$\{\, P \,\}\, C \,\{\, Q \,\}$$

precondition code postconditio

Example: $\{\, \texttt{x is even} \,\}\ \texttt{x := x+1}\ \{\, \texttt{x is odd} \,\}$

Hoare logic formally

A little language:

```
prog = Assign string (state → num)
     | Seq prog prog
     | If (state → bool) prog prog
     | While (state → bool) prog

where state = (string → num)
```

Hoare logic formally

A little language:

```
prog = Assign string (state → num)
     | Seq prog prog
     | If (state → bool) prog prog
     | While (state → bool) prog

where state = (string → num)
```

Big-step semantics:

$$\frac{}{(s, \mathsf{Assign}\ v\ f) \Downarrow s[v \mapsto f(s)]} \qquad \frac{(s, p_1) \Downarrow s_1 \qquad (s_1, p_2) \Downarrow s_2}{(s, \mathsf{Seq}\ p_1\ p_2) \Downarrow s_2}$$

$$\frac{guard(s) \qquad (s, p_1) \Downarrow s_1}{(s, \mathsf{If}\ guard\ p_1\ p_2) \Downarrow s_1} \qquad \frac{\neg guard(s) \qquad (s, p_2) \Downarrow s_2}{(s, \mathsf{If}\ guard\ p_1\ p_2) \Downarrow s_2}$$

$$\frac{\neg guard(s)}{(s, \mathsf{While}\ guard\ body) \Downarrow s} \qquad \frac{guard(s) \quad (s, \mathsf{Seq}\ body\ (\mathsf{While}\ guard\ body)) \Downarrow t}{(s, \mathsf{While}\ guard\ body) \Downarrow t}$$

Hoare triple

Partial correctness:

$$\text{Hoare } P\ C\ Q \equiv \forall s\ t.\ P\ s \wedge (s, C) \Downarrow t \implies Q\ t$$

informally: "if P true and C terminates, then Q"

Hoare triple

Partial correctness:

$$\text{Hoare } P\ C\ Q \equiv \forall s\ t.\ P\ s \wedge (s, C) \Downarrow t \implies Q\ t$$

informally: "if P true and C terminates, then Q"

Total correctness:

$$\text{Total } P\ C\ Q \equiv \forall s.\ P\ s \implies \exists t.\ (s, C) \Downarrow t \wedge Q\ t$$

informally: "if P true, then C terminates and Q"

Hoare triple

Partial correctness:

$$\text{Hoare } P\ C\ Q \equiv \forall s\ t.\ P\ s \wedge (s, C) \Downarrow t \implies Q\ t$$

informally: "if P true and C terminates, then Q"

Total correctness:

$$\text{Total } P\ C\ Q \equiv \forall s.\ P\ s \implies \exists t.\ (s, C) \Downarrow t \wedge Q\ t$$

informally: "if P true, then C terminates and Q"

Both Hoare and Total usually written $\{\, P \,\}\, C \,\{\, Q \,\}$

Hoare logic proof rules

Assignment ‘axiom’:

$$\frac{}{\{\, Q[E/x] \,\}\, x := E \,\{\, Q \,\}}$$

Hoare logic proof rules

Assignment 'axiom':

$$\frac{}{\{\, Q[E/x] \,\}\, x := E \,\{\, Q \,\}}$$

In our formalisation this 'axiom' is a theorem in HOL

$$\forall Q\ x\ E.\ \text{Hoare}\ (\lambda s.\ Q\ (s[x \mapsto E(s)]))\ (\text{Assign}\ x\ E)\ Q$$

proved based on the definition of Hoare.

Hoare logic proof rules (cont.)

Some other proof rules:

$$\frac{\{\,P\,\}\,C_1\,\{\,R\,\} \qquad \{\,R\,\}\,C_2\,\{\,Q\,\}}{\{\,P\,\}\,\mathsf{Seq}\;C_1\;C_2\,\{\,Q\,\}}$$

Hoare logic proof rules (cont.)

Some other proof rules:

$$\frac{\{\,P\,\}\,C_1\,\{\,R\,\} \qquad \{\,R\,\}\,C_2\,\{\,Q\,\}}{\{\,P\,\}\,\mathsf{Seq}\;C_1\;C_2\,\{\,Q\,\}}$$

$$\frac{\{\,G \wedge P\,\}\,C_1\,\{\,Q\,\}}{\{\,G \wedge P\,\}\,\mathsf{If}\;G\;C_1\;C_2\,\{\,Q\,\}} \qquad \frac{\{\,\neg G \wedge P\,\}\,C_2\,\{\,Q\,\}}{\{\,\neg G \wedge P\,\}\,\mathsf{If}\;G\;C_1\;C_2\,\{\,Q\,\}}$$

Hoare logic proof rules (cont.)

Some other proof rules:

$$\frac{\{\,P\,\}\,C_1\,\{\,R\,\} \qquad \{\,R\,\}\,C_2\,\{\,Q\,\}}{\{\,P\,\}\,\mathsf{Seq}\;C_1\;C_2\,\{\,Q\,\}}$$

$$\frac{\{\,G \wedge P\,\}\,C_1\,\{\,Q\,\}}{\{\,G \wedge P\,\}\,\mathsf{If}\;G\;C_1\;C_2\,\{\,Q\,\}} \qquad \frac{\{\,\neg G \wedge P\,\}\,C_2\,\{\,Q\,\}}{\{\,\neg G \wedge P\,\}\,\mathsf{If}\;G\;C_1\;C_2\,\{\,Q\,\}}$$

$$\frac{\{\,G \wedge P\,\}\,C\,\{\,P\,\}}{\{\,P\,\}\,\mathsf{While}\;G\;C\,\{\,P \wedge \neg G\,\}}$$

Hoare logic proof rules (cont.)

Some other proof rules:

$$\frac{\{\,P\,\}\,C_1\,\{\,R\,\} \qquad \{\,R\,\}\,C_2\,\{\,Q\,\}}{\{\,P\,\}\,\mathsf{Seq}\;C_1\;C_2\,\{\,Q\,\}}$$

$$\frac{\{\,G \wedge P\,\}\,C_1\,\{\,Q\,\}}{\{\,G \wedge P\,\}\,\mathsf{If}\;G\;C_1\;C_2\,\{\,Q\,\}} \qquad \frac{\{\,\neg G \wedge P\,\}\,C_2\,\{\,Q\,\}}{\{\,\neg G \wedge P\,\}\,\mathsf{If}\;G\;C_1\;C_2\,\{\,Q\,\}}$$

only for partial-correctness

$$\frac{\{\,G \wedge P\,\}\,C\,\{\,P\,\}}{\{\,P\,\}\,\mathsf{While}\;G\;C\,\{\,P \wedge \neg G\,\}}$$

Hoare logic proof rules (cont.)

Some other proof rules:

$$\frac{\{\,P\,\}\,C_1\,\{\,R\,\} \qquad \{\,R\,\}\,C_2\,\{\,Q\,\}}{\{\,P\,\}\ \mathsf{Seq}\ C_1\ C_2\,\{\,Q\,\}}$$

$$\frac{\{\,G \wedge P\,\}\,C_1\,\{\,Q\,\}}{\{\,G \wedge P\,\}\ \mathsf{If}\ G\ C_1\ C_2\,\{\,Q\,\}} \qquad \frac{\{\,\neg G \wedge P\,\}\,C_2\,\{\,Q\,\}}{}$$

P is called the invariant

only for partial-correctness

$$\frac{\{\,G \wedge P\,\}\,C\,\{\,P\,\}}{\{\,P\,\}\ \mathsf{While}\ G\ C\,\{\,P \wedge \neg G\,\}}$$

Hoare logic proof rules (cont.)

Some other proof rules:

$$\frac{\{\,P\,\}\,C_1\,\{\,R\,\} \qquad \{\,R\,\}\,C_2\,\{\,Q\,\}}{\{\,P\,\}\,\mathsf{Seq}\;C_1\;C_2\,\{\,Q\,\}}$$

$$\frac{\{\,G \wedge P\,\}\,C_1\,\{\,Q\,\}}{\{\,G \wedge P\,\}\,\mathsf{If}\;G\;C_1\;C_2\,\{\,Q\,\}} \qquad \frac{\{\,\neg G \wedge P\,\}\,C_2\,\{\,Q\,\}}{}$$

P is called the invariant

only for partial-correctness

$$\frac{\{\,G \wedge P\,\}\,C\,\{\,P\,\}}{\{\,P\,\}\,\mathsf{While}\;G\;C\,\{\,P \wedge \neg G\,\}}$$

(Consequence) $$\frac{P' \implies P \qquad \{\,P\,\}\,C\,\{\,Q\,\} \qquad Q \implies Q'}{\{\,P'\,\}\,C\,\{\,Q'\,\}}$$

Example proof: factorial

Code:

$\texttt{while (n} \neq \texttt{0) (r := r} \times \texttt{n; n := n - 1)}$

Invariant:

$Inv \equiv (\mathsf{fac}\ N = \mathsf{fac}\ \texttt{n} \times \texttt{r} \wedge 0 \leq \texttt{n})$

By Assignment and Seq rule:

$\{\, Inv[\texttt{n} - 1/\texttt{n}][\texttt{r} \times \texttt{n}/\texttt{r}] \,\}\ \texttt{r := r} \times \texttt{n; n := n - 1}\ \{\, Inv \,\}$

By Consequence rule:

$\{\, Inv \wedge \texttt{n} \neq 0 \,\}\ \texttt{r := r} \times \texttt{n; n := n - 1}\ \{\, Inv \,\}$

By While rule:

$\{\, Inv \,\}\ \texttt{while (n} \neq \texttt{0) (r := r} \times \texttt{n; n := n - 1)}\ \{\, Inv \wedge \texttt{n} = 0 \,\}$

By Consequence rule:

$\{\, N = \texttt{n} \wedge \texttt{r} = 1 \,\}\ \texttt{while (n} \neq \texttt{0) (r := r} \times \texttt{n; n := n - 1)}\ \{\, \texttt{r} = \mathsf{fac}\ N \,\}$

Machine code

Moving on to something more realistic.

For real machine code:

- state is more complex (consists of registers, memory, status bits, modes etc.)
- code lives in memory
- program counter (PC) points to next instruction
- no obvious termination (small-step semantics)

Next few slides formalise a simple machine language...

Semantics of machine code

Let state be a partial finite map from `mc_name` to `mc_val`

```
reg = 4-bit word
word = 32-bit word
address = word

mc_name = Reg reg
        | Mem address
        | Flag
        | Pc

mc_val  = Word word
        | Bool bool
```

Real machine languages (x86, ARM, Power etc.) have roughly these components, but more registers, flags, modes etc.

Semantics (cont.)

Type of instructions:

```
mc_inst = Add reg reg reg
        | AddImm reg reg word
        | Jump offset
        | JumpIfEqual offset
        | Compare reg word
        | ...

offset = word
```

Assume we have a (partial) decode function that maps `word` into `mc_inst`.

Semantics (cont.)

Small-step semantics ($\xrightarrow{\texttt{eval}}$) defined using rules. Example:

```
s Pc = Word pc
s (Mem pc) = Word w
decode w = SubImm a b imm
s (Reg b) = Word x
```

```
s  --eval-->  s[Pc ↦ Word (pc + 4)][Reg a ↦ Word (x - imm)]
```

Naive machine-code Hoare logic

For the While-language, we defined a total-correctness Hoare-triple:

$$\text{Total } P\ C\ Q \equiv \forall s.\ P\ s \implies \exists t.\ (s, C) \Downarrow t \wedge Q\ t$$

informally: "if P true, then C terminates and Q"

Naive machine-code Hoare logic

For the While-language, we defined a total-correctness Hoare-triple:

$$\text{Total } P\ C\ Q \equiv \forall s.\ P\ s \implies \exists t.\ (s, C) \Downarrow t \wedge Q\ t$$

informally: "if P true, then C terminates and Q"

Similar total-correctness machine-code Hoare judgement:

$$\text{mcTotal } P\ Q \equiv \forall s.\ P\ s \implies \exists t.\ s \xrightarrow{\texttt{eval}}{}^{*} t \wedge Q\ t$$

informally: "if P true, then execution reaches Q"

Naive machine-code Hoare logic

For the While-language, we defined a total-correctness Hoare-triple:

$$\text{Total } P\ C\ Q \equiv \forall s.\ P\ s \implies \exists t.\ (s, C) \Downarrow t \wedge Q\ t$$

informally: "if P true, then C terminates and Q"

Similar total-correctness machine-code Hoare judgement:

$$\text{mcTotal } P\ Q \equiv \forall s.\ P\ s \implies \exists t.\ s \xrightarrow{\texttt{eval}}{}^{*} t \wedge Q\ t$$

informally: "if P true, then execution reaches Q"

makes sense only if $\xrightarrow{\texttt{eval}}$ is deterministic

Problems

`mcTotal` can be used:

- supports nice composition and sequence rules
- no need for special loop rule (composition suffices)
- ... however, assignment rules are dreadful

Problems

`mcTotal` can be used:

- supports nice composition and sequence rules
- no need for special loop rule (composition suffices)
- ... however, assignment rules are dreadful

Example:

- an mcHoare triple for `SubImm`

```
mcTotal
  (λs. ∃pc a b w x imm.
         Q (s[Pc ↦ Word (pc + 4)][Reg a ↦ Word (x - imm)]) ∧
         s Pc = Word pc ∧
         s (Mem pc) = Word w ∧
         decode w = SubImm a b imm ∧
         s (Reg b) = Word x)
  Q
```

Problems

`mcTotal` can be used:

- supports nice composition and sequence rules
- no need for special loop rule (composition suffices)
- ... however, assignment rules are dreadful

Example:

- an mcHoare triple for `SubImm`

```
mcTotal
  (λs. ∃pc a b w x imm.
         Q (s[Pc ↦ Word (pc + 4)][Reg a ↦ Word (x - imm)]) ∧
         s Pc = Word pc ∧
         s (Mem pc) = Word w ∧
         decode w = SubImm a b imm ∧
         s (Reg b) = Word x)
  Q
```

too many preconditions

Problems

`mcTotal` can be used:

- supports nice composition and sequence rules
- no need for special loop rule (composition suffices)
- ... however, assignment rules are dreadful

Example:

- an mcHoare triple for `SubImm`

```
mcTotal
  (λs. ∃pc a b w x imm.
         Q (s[Pc ↦ Word (pc + 4)][Reg a ↦ Word (x - imm)]) ∧
         s Pc = Word pc ∧
         s (Mem pc) = Word w ∧
         decode w = SubImm a b imm ∧
         s (Reg b) = Word x)
  Q
```

too many preconditions

affected by memory store instructions

Solution: separation logic

Separation logic

- a Hoare logic for 'local reasoning'
- developed recently by Reynolds, O'Hearn, Ishtiaq and Yang (based on some early ideas by Burstall)
- can express certain local concepts very succinctly, e.g. "and nothing else changed"

Separating conjunction

States are separated by the separating conjunction (*):

```
(p * q) state = ∃s1 s2. p s1 ∧ q s2 ∧
                        state = s1 ∪ s2 ∧
                        domain s1 ∩ domain s2 = {}
```

Separating conjunction

States are separated by the separating conjunction (*):

$$(p * q)\ state = \exists s1\ s2.\ p\ s1 \wedge q\ s2 \wedge state = s1 \cup s2 \wedge domain\ s1 \cap domain\ s2 = \{\}$$

Let $(name \mapsto val) = \lambda s.\ (domain\ s = \{name\} \wedge s(name) = val)$

Separating conjunction

States are separated by the separating conjunction (*):

$$(p * q)\ state = \exists s1\ s2.\ p\ s1 \wedge q\ s2 \wedge state = s1 \cup s2 \wedge domain\ s1 \cap domain\ s2 = \{\}$$

Let $(name \mapsto val) = \lambda s.\ (domain\ s = \{name\} \wedge s(name) = val)$

Why local reasoning? Answer: * consumes state.

$$((a \mapsto x) * (b \mapsto y) * frame)\ state \implies a \neq b \wedge state(a) = x \wedge state(b) = y \wedge \ldots$$

Separating conjunction

States are separated by the separating conjunction (*):

$$(p * q)\ state = \exists s1\ s2.\ p\ s1 \wedge q\ s2 \wedge state = s1 \cup s2 \wedge domain\ s1 \cap domain\ s2 = \{\}$$

Let $(name \mapsto val) = \lambda s.\ (domain\ s = \{name\} \wedge s(name) = val)$

Why local reasoning? Answer: * consumes state.

$$((a \mapsto x) * (b \mapsto y) * frame)\ state \implies a \neq b \wedge state(a) = x \wedge state(b) = y \wedge \ldots$$

false if `frame` mentions `a` or `b`

Machine-code Hoare triple

We define an improved machine-code Hoare triple:

```
{ P } C { Q } = ∀frame. mcTotal (P * C * frame)
                                (Q * C * frame)
```

Machine-code Hoare triple

We define an improved machine-code Hoare triple:

```
{ P } C { Q } = ∀frame. mcTotal (P * C * frame)
                                (Q * C * frame)
```

Intention:

- can write pre- and postconditions using *-separated assertions
- assertion `C` holds code assertions (that stays invariant)
- the `frame` expresses "and nothing else changes"
- explanation on next few slides

Example: Sub instruction

A machine-code Hoare triple for a `SubImm` instruction:

```
{ PC pc * R 1 y * R 2 x }
INST(pc, SubImm 1 2 50)
{ PC (pc+4) * R 1 (x-50) * R 2 x }
```

where:

```
PC pc = Pc ↦ (Word pc)

R a x = (Reg a) ↦ (Word x)

M a x = (Mem a) ↦ (Word x)

INST(pc,inst) = ∃w. M pc w * pure (decode w = inst)

pure b = λs. domain s = {} ∧ b
```

Why local reasoning?

Frame rule:

$$\{\ P\ \}\ C\ \{\ Q\ \} \implies \forall \mathit{frame}.\ \{\ P * \mathit{frame}\ \}\ C\ \{\ Q * \mathit{frame}\ \}$$

Why local reasoning?

Frame rule:

$$\{\ P\ \}\ C\ \{\ Q\ \} \implies \forall \mathit{frame}.\ \{\ P * \mathit{frame}\ \}\ C\ \{\ Q * \mathit{frame}\ \}$$

Means e.g. that we can infer that reg. 3 was unaffected:

```
{ PC pc * R 1 y * R 2 x * R 3 z }
INST(pc, SubImm 1 2 50)
{ PC (pc+4) * R 1 (x-50) * R 2 x * R 3 z }
```

Composition

We also have frame rule for code segment:

$$\{ P \}\ C\ \{ Q \} \implies \forall frame.\ \{ P \}\ (C * frame)\ \{ Q \}$$

Needed before composition:

$$\{ P \}\ C\ \{ R \} \wedge \{ R \}\ C\ \{ Q \} \implies \{ P \}\ C\ \{ Q \}$$

Example of composition next slides...

Composition (cont.)

Want to derive behaviour of the SubImm-then-Jump:

```
{ PC pc * R 1 y * R 2 x }
INST(pc, SubImm 1 2 50)
{ PC (pc + 4) * R 1 (x-50) * R 2 x }

{ PC pc }
INST(pc, Jump 200)
{ PC (pc + 200) }
```

Composition (cont.)

Want to derive behaviour of the SubImm-then-Jump:

```
{ PC pc * R 1 y * R 2 x }
INST(pc, SubImm 1 2 50)
{ PC (pc + 4) * R 1 (x-50) * R 2 x }

{ PC pc }
INST(pc, Jump 200)
{ PC (pc + 200) }
```

Instantiation and application of frame rules:

```
{ PC pc * R 1 y * R 2 x }
INST(pc, SubImm 1 2 50) * INST(pc + 4, Jump 200)
{ PC (pc + 4) * R 1 (x-50) * R 2 x }

{ PC (pc + 4) * R 1 (x-50) * R 2 x }
INST(pc, SubImm 1 2 50) * INST(pc + 4, Jump 200)
{ PC (pc + 204) * R 1 (x-50) * R 2 x }
```

Composition (cont.)

Want to derive behaviour of the SubImm-then-Jump:

```
{ PC pc * R 1 y * R 2 x }
INST(pc, SubImm 1 2 50)
{ PC (pc + 4) * R 1 (x-50) * R 2 x }

{ PC pc }
INST(pc, Jump 200)
{ PC (pc + 200) }
```

In

read: this code is *sufficient* to get execution to the postcondition

```
{ PC pc * R 1 y * R 2 x }
INST(pc, SubImm 1 2 50) * INST(pc + 4, Jump 200)
{ PC (pc + 4) * R 1 (x-50) * R 2 x }

{ PC (pc + 4) * R 1 (x-50) * R 2 x }
INST(pc, SubImm 1 2 50) * INST(pc + 4, Jump 200)
{ PC (pc + 204) * R 1 (x-50) * R 2 x }
```

Composition (cont.)

Result of composition:

```
{ PC pc * R 1 y * R 2 x }
INST(pc, SubImm 1 2 50) * INST(pc + 4, Jump 200)
{ PC (pc + 204) * R 1 (x-50) * R 2 x }
```

Example: Store instruction

Separating conjunction keeps memory separate from code:

```
{ PC pc * R 1 addr * R 2 val * M addr x }
INST(pc, Store 1 2)
{ PC (pc+4) * R 1 addr * R 2 val * M addr val }
```

Example: Store instruction

Separating conjunction keeps memory separate from code:

```
{ PC pc * R 1 addr * R 2 val * M addr x }
INST(pc, Store 1 2)
{ PC (pc+4) * R 1 addr * R 2 val * M addr val }
```

Built-in assumption that `pc` and `addr` are separate. See definitions.

Example: Store instruction

Separating conjunction keeps memory separate from code:

```
{ PC pc * R 1 addr * R 2 val * M addr x }
INST(pc, Store 1 2)
{ PC (pc+4) * R 1 addr * R 2 val * M addr val }
```

Built-in assumption that `pc` and `addr` are separate. See definitions.

Support for self-modifying code:

```
{ P } (C1 * C2) { Q }   ⟺   { P * C2 } C1 { Q * C2 }
```

Example: Store instruction

Separating conjunction keeps memory separate from code:

```
{ PC pc * R 1 addr * R 2 val * M addr x }
INST(pc, Store 1 2)
{ PC (pc+4) * R 1 addr * R 2 val * M addr val }
```

Built-in assumption that `pc` and `addr` are separate. See definitions.

Support for self-modifying code:

$$\{ P \}\ (C1 * C2)\ \{ Q \} \iff \{ P * C2 \}\ C1\ \{ Q * C2 \}$$

Code can be moved into the pre and post and ‘become data’.

Reasoning still cumbersome...

Statements made succinct using separation logic.

However, reasoning (e.g. composition) is still very low-level...

Decompilation into logic

Decompilation automates low-level reasoning (from my PhD).

Decompilation into logic

Decompilation automates low-level reasoning (from my PhD).

Example: given some machine code (assembly syntax & comments):

```
FAC:
  cmp r0,0          compare r0 with zero
  jeq EXIT          jump to EXIT, if r0 was zero
  mul r1,r1,r0      perform: r1 := r1 × r0
  sub r0,r0,1       perform: r0 := r0 - 1
  jmp FAC           jump to FAC
EXIT:
```

Decompilation into logic

Decompilation automates low-level reasoning (from my PhD).

Example: given some machine code (assembly syntax & comments):

```
FAC:
  cmp r0,0          compare r0 with zero
  jeq EXIT          jump to EXIT, if r0 was zero
  mul r1,r1,r0      perform: r1 := r1 × r0
  sub r0,r0,1       perform: r0 := r0 - 1
  jmp FAC           jump to FAC
EXIT:
```

Decompiler extracts function describing the machine code:

```
fac (r0,r1) =
  if r0 = 0 then (r0,r1) else
    let r1 = r1 × r0 in
    let r0 = r0 - 1 in
      fac (r0,r1)
```

Why believe it?

Decompilation proves a certificate theorem:

Why believe it?

Decompilation proves a certificate theorem:

```
fac_pre (r0,r1) ⟹
{ PC pc * R 0 r0 * R 1 r1 * F _ }
  “ FAC: cmp r0,0
         jeq EXIT
         mul r1,r1,r0
         sub r0,r0,1
         jmp FAC
    EXIT:              ”
{ let (r0,r1) = fac (r0,r1) in
    PC (pc + 20) * R 0 r0 * R 1 r1 * F _ }
```

Why believe it?

Decompilation proves a certificate theorem:

```
fac_pre (r0,r1) ⟹
{ PC pc * R 0 r0 * R 1 r1 * F _ }
  “ FAC: cmp r0,0
         jeq EXIT
         mul r1,r1,r0
         sub r0,r0,1
         jmp FAC
    EXIT:              ”
{ let (r0,r1) = fac (r0,r1) in
    PC (pc + 20) * R 0 r0 * R 1 r1 * F _ }
```

where `fac_pre` collects termination and side-conditions.

```
fac_pre (r0,r1) =
  if r0 = 0 then true else
    let r1 = r1 × r0 in
    let r0 = r0 - 1 in
      fac_pre (r0,r1)
```

Why believe it?

Decompilation proves a certificate theorem:

```
fac_pre (r0,r1) ⟹
{ PC pc * R 0 r0 * R 1 r1 * F _ }
  “ FAC: cmp r0,0
         jeq EXIT
         mul r1,r1,r0
         sub r0,r0,1
         jmp FAC
    EXIT:              ”
{ let (r0,r1) = fac (r0,r1) in
    PC (pc + 20) * R 0 r0 * R 1 r1 * F _ }
```

Benefit:
Anything proved about `fac` and `fac_pre` is also true of the machine code via this theorem.

where `fac_pre` collects termination and side-conditions.

```
fac_pre (r0,r1) =
  if r0 = 0 then true else
    let r1 = r1 × r0 in
    let r0 = r0 - 1 in
      fac_pre (r0,r1)
```

Verification using decomp.

Suffices to prove properties of `fac` and `fac_pre`.

```
fac (n,1) = (0,n!)
fac_pre (n,1) = true
```

Verification using decomp.

Suffices to prove properties of `fac` and `fac_pre`.

```
fac (n,1) = (0,n!)
fac_pre (n,1) = true
```

Then:

```
{ PC pc * R 0 n * R 1 1 * F _ }
  “ FAC: cmp r0,0
         jeq EXIT
         mul r1,r1,r0
         sub r0,r0,1
         jmp FAC
    EXIT:              ”
{ PC (pc + 20) * R 0 0 * R 1 (n!) * F _ }
```

Verification using decomp.

Suffices to prove properties of `fac` and `fac_pre`.

```
fac (n,1) = (0,n!)
fac_pre (n,1) = true
```

Then:

```
{ PC pc * R 0 n * R 1 1 * F _ }
  “ FAC: cmp r0,0
         jeq EXIT
         mul r1,r1,r0
         sub r0,r0,1
         jmp FAC
    EXIT:              ”
{ PC (pc + 20) * R 0 0 * R 1 (n!) * F _ }
```

Flag component modified, but uninteresting value hidden

How decompilation works

Implementation:

1. compose Hoare triples along each path through code
2. apply loop rule, if applicable
3. read off function and precondition

How decompilation works

Implementation:

1. compose Hoare triples along each path through code
2. apply loop rule, if applicable
3. read off function and precondition

Example: composition of triples gives:

```
{ PC pc * R 0 r0 * R 1 r1 * F _ }
  " FAC: cmp r0,0 ... "
{ if r0 = 0 then
    PC (pc + 20) * R 0 r0 * R 1 r1 * F _
  else
    let r1 = r1 × r0 in
    let r0 = r0 - 1 in
      PC pc * R 0 r0 * R 1 r1 * F _ }
```

Loop rule

Loop rule introduces tail rec. function:

```
(∀x. p x ⇒ { P x } code { if g x then P (f x) else Q (d x) }) ⇒
(∀x. side g f p x ⇒ { P x } code { Q (tailrec g f d x) })
```

where `tailrec` is a function template:

```
tailrec g f d x = if g x then tailrec g f d (f x) else d x
```

Loop rule

Loop rule introduces tail rec. function:

```
(∀x. p x ⇒ { P x } code { if g x then P (f x) else Q (d x) }) ⇒
(∀x. side g f p x ⇒ { P x } code { Q (tailrec g f d x) })
```

where `tailrec` is a function template:

```
tailrec g f d x = if g x then tailrec g f d (f x) else d x
```

can be defined in HOL *without* a termination proof (trick)

Loop rule

Loop rule introduces tail rec. function:

```
(∀x. p x ⇒ { P x } code { if g x then P (f x) else Q (d x) }) ⇒
(∀x. side g f p x ⇒ { P x } code { Q (tailrec g f d x) })
```

every tail rec. function fits this template

w

```
tailrec g f d x = if g x then tailrec g f d (f x) else d x
```

can be defined in HOL *without* a termination proof (trick)

Loop rule

Loop rule introduces tail rec. function:

```
(∀x. p x ⇒ { P x } code { if g x then P (f x) else Q (d x) }) ⇒
(∀x. side g f p x ⇒ { P x } code { Q (tailrec g f d x) })
```

where `tailrec` is a function template:

```
tailrec g f d x = if g x then tailrec g f d (f x) else d x
```

Loop rule

Loop rule introduces tail rec. function:

```
(∀x. p x ⇒ { P x } code { if g x then P (f x) else Q (d x) }) ⇒
(∀x. side g f p x ⇒ { P x } code { Q (tailrec g f d x) })
```

where `tailrec` is a function template:

```
tailrec g f d x = if g x then tailrec g f d (f x) else d x
```

and `side` is ensures that `tailrec` terminates, definition:

$$\texttt{side}\ g\ f\ p\ x = (\forall k.\ (\forall m.\ m < k \Rightarrow g\ (f^m\ x)) \Rightarrow p\ (f^k\ x)) \wedge (\exists n.\ \neg g\ (f^n\ x))$$

Loop rule

Loop rule introduces tail rec. function:

```
(∀x. p x ⇒ { P x } code { if g x then P (f x) else Q (d x) }) ⇒
(∀x. side g f p x ⇒ { P x } code { Q (tailrec g f d x) })
```

where `tailrec` is a function template:

```
tailrec g f d x = if g x then tailrec g f d (f x) else d x
```

and `side` is ensures that `tailrec` terminates, definition:

$$\texttt{side}\ g\ f\ p\ x = (\forall k.\ (\forall m.\ m < k \Rightarrow g\ (f^m\ x)) \Rightarrow p\ (f^k\ x)) \wedge (\exists n.\ \neg g\ (f^n\ x))$$

derived equation:

```
side g f p x = (p x ∧ (g x ⇒ side g f p (f x)))
```

Instantiation for fac example

State assertions:

```
P = λ(r0,r1). PC pc * R 0 r0 * R 1 r1 * F _

Q = λ(r0,r1). PC (pc + 20) * R 0 r0 * R 1 r1 * F _
```

Function components:

```
f = d = λ(r0,r1). if r0 = 0 then (r0,r1) else
                      let r1 = r1 × r0 in
                      let r0 = r0 - 1 in
                        (r0,r1)

g = λ(r0,r1). if r0 = 0 then false else true

p = λ(r0,r1). true
```

Proof sketch for loop rule

Assume:

```
(∀x. p x ⇒ { P x } code { if g x then P (f x) else Q (d x) })
```

Want to show:

```
(∀x. side g f p x ⇒ { P x } code { Q (tailrec g f d x) })
```

Goal simplifies to:

```
(∀x. ... ∧ (∃n. ¬g (f^n x)) ⇒ { P x } code { Q (tailrec g f d x) })
```

which simplifies to:

```
(∀n x. ... ∧ ¬g (f^n x) ⇒ { P x } code { Q (tailrec g f d x) })
```

which we can prove by induction on `n`:

case `n = 0`: `(∀x. ... ∧ ¬g x ⇒ { P x } code { Q (tailrec g f d x) })`

which is the same as:

```
(∀x. ... ∧ ¬g x ⇒ { P x } code { Q (d x) })
```

which follows from the assumption.

Proof sketch (cont.)

case n = k+1:

can assume:

$(\forall x.\ \ldots \wedge \neg g\ (f^k\ x) \Rightarrow \{\ P\ x\ \}\ \texttt{code}\ \{\ Q\ (\texttt{tailrec}\ g\ f\ d\ x)\ \})$

need to show:

$(\forall x.\ \ldots \wedge \neg g\ (f^{k+1}\ x) \Rightarrow \{\ P\ x\ \}\ \texttt{code}\ \{\ Q\ (\texttt{tailrec}\ g\ f\ d\ x)\ \})$

for any x, if $\neg g\ x$ then same as base case, otherwise:

$\ldots \wedge \neg g\ (f^k\ (f\ x)) \Rightarrow \{\ P\ x\ \}\ \texttt{code}\ \{\ Q\ (\texttt{tailrec}\ g\ f\ d\ (f\ x))\ \})$

which is true by composition of the following two facts

fact 1: $p\ x \Rightarrow \{\ P\ x\ \}\ \texttt{code}\ \{\ P\ (f\ x)\ \}$

which is an instantiation of the assump. from last slide

fact 2: $\ldots \wedge \neg g\ (f^k\ (f\ x)) \Rightarrow \{\ P\ (f\ x)\ \}\ \texttt{code}\ \{\ Q\ (\texttt{tailrec}\ g\ f\ d\ (f\ x))\ \})$

which is an instantiation of the assump. from this slide
(the inductive hypothesis)

Summary

Hoare logic

- from Hoare's seminal paper "An axiomatic basis for computer programming", 1969
- a logic with judgements about program text

Hoare logic for machine code

- based on separation logic (due Reynolds et al.)
- supports the frame rule
- and self-modifying code

Decompilation into logic

- automation that aids in machine code verification
- extracts function from code
- automatically proves certificate theorem