Content

→ Intro & motivation, getting started

→ Foundations & Principles
  • Lambda Calculus, natural deduction \[1,2\]
  • Higher Order Logic \[3^a\]
  • Term rewriting \[4\]

→ Proof & Specification Techniques
  • Inductively defined sets, rule induction \[5\]
  • Datatypes, recursion, induction \[6, 7\]
  • Hoare logic, proofs about programs, invariants \[8^b, 9\]
  • (mid-semester break)
  • C verification \[10\]
  • CakeML, Isar \[11^c\]
  • Concurrency \[12\]

\(^a\text{a1 due}; ^b\text{a2 due}; ^c\text{a3 due}\)
Program verification so far

If the following true?

\[
\begin{align*}
\{ & x = 0 \} \\
& y := x; \\
& x := x + 1; \\
& \{ & x = 1 \land y = 0 \}
\end{align*}
\]

YES!
Is it still true?

\[
\begin{align*}
\{x = 0\} \\
y := x; \quad \parallel \quad x := 4 \\
x := x + 1; \\
\{x = 1 \land y = 0\}
\end{align*}
\]

NO!
Program verification so far

So far we have assumed **sequential execution**

\[
\begin{align*}
\{x = 0\} & \quad x \mapsto 0 \quad y \mapsto - \\
y := x; & \quad x \mapsto 0 \quad y \mapsto 0 \\
x := x + 1; & \quad x \mapsto 1 \quad y \mapsto 0 \\
\{x = 1 \land y = 0\} & 
\end{align*}
\]

i.e. a single thread of execution accessing the memory state

This is not always the case!
Types of concurrency

**Multithreading**

<table>
<thead>
<tr>
<th>ProgA</th>
<th>ProgB</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>→ ...</td>
</tr>
<tr>
<td>→ ...</td>
<td>...</td>
</tr>
</tbody>
</table>

CPU

Memory

**Multicore**

<table>
<thead>
<tr>
<th>ProgA</th>
<th>ProgB</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>→ ...</td>
</tr>
<tr>
<td>→ ...</td>
<td>...</td>
</tr>
</tbody>
</table>

CPU

Memory

**Distributed**

<table>
<thead>
<tr>
<th>ProgA</th>
<th>ProgB</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>→ ...</td>
</tr>
<tr>
<td>→ ...</td>
<td>...</td>
</tr>
</tbody>
</table>

CPU

Memory

All need communication and synchronisation mechanisms

- Shared memory
- Interleaved execution

- Shared memory
- Parallel execution

- Message passing

**Here:** we’ll look at shared-memory concurrency

(and we’ll ignore further complications such as caches, weak memory,...)
Goal

We want to be able to reason about parallel composition of programs:

\[
\begin{align*}
\{ \text{precondition} \} \\
\text{Prog}_A \quad \| \quad \text{Prog}_B \\
\text{...} \\
\rightarrow \text{...} \\
\end{align*}
\]  

\{ \text{postcondition} \}

2 kinds of properties:

Safety:  
“something bad does not happen”  
(no bad state can be reached)  
e.g. \( \{ x = 0 \} \)

Liveness:  
“something good must happen”  
(specific states must be reached)  
e.g. the program terminates

With concurrency: much harder!  
With concurrency: new problems!
Goal

We want to be able to reason about parallel composition of programs:

\[
\{\text{precondition}\} \\
\text{Prog}_A \quad || \\ 
\text{Prog}_B
\]

\[
\{\text{postcondition}\}
\]

Here:
- We focus on safety properties: postcondition holds if reached
- We will define parallel composition (||) as non-deterministic interleaving
- We go back to our minimal IMP language (forget about C and monads)

\text{datatype com} \equiv \text{SKIP}
Program verification so far

If the following true?

\{x = 0\}
y := x;
x := x + 1;
\{x = 1 \land y = 0\}

YES!
Is it still true?

\[
\begin{align*}
\{x = 0\} \\
y &:= x; \quad || \quad x := 4 \\
x &:= x + 1; \\
\{x = 1 \land y = 0\}
\end{align*}
\]

NO!

What is going wrong?
What do we need to change?

→ to make sure we don’t prove wrong statements!
→ to allow us to prove true statements about concurrent programs
Program verification so far

How would we have proved this?

\[
\{ x = 0 \} \implies \{ x + 1 = 1 \land x = 0 \} \\
y := x; \{ x + 1 = 1 \land y = 0 \} \\
x := x + 1; \\
\{ x = 1 \land y = 0 \}
\]

Using Hoare logic rules!

\[
\begin{align*}
\vdash \{ P \} \ c_1 \ \{ R \} & \quad \vdash \{ R \} \ c_2 \ \{ Q \} \\
\vdash \{ P \} \ c_1; c_2 \ \{ Q \} & \\
\vdash \{ P[x \mapsto e] \} \quad x := e \quad \{ P \}
\end{align*}
\]

Why does this make it true? What does it mean that it’s true?
It means:

*If the program “y := x; x := x + 1” is executed from a state satisfying \{ x = 0 \} then, if it terminates, the resulting state satisfied \{ x = 1 \land y = 0 \}*
Program verification so far

How would we have proved this? Using Hoare logic rules!

\{ x = 0 \} \implies \{ x + 1 = 1 \land x = 0 \}

\begin{align*}
y &:= x; \quad \{ x + 1 = 1 \land y = 0 \} \\
x &:= x + 1; \\
\{ x = 1 \land y = 0 \}
\end{align*}

\[ \vdash \{ P \} \ c_1 \ \{ R \} \vdash \{ R \} \ c_2 \ \{ Q \} \]

\[ \vdash \{ P \} \ c_1; c_2 \ \{ Q \} \]

\[ \vdash \{ P[x \mapsto e] \} \quad x := e \quad \{ P \} \]

Why does this make it true? What does it mean that it’s true?

It means:

\[ \langle y := x; \ x := x + 1, \sigma \rangle \rightarrow \sigma' \land x \ \sigma = 0 \quad \rightarrow \quad x \ \sigma' = 1 \land y \ \sigma' = 0 \]

Where:

\[ \langle c_1, \sigma \rangle \rightarrow \sigma' \quad \langle c_2, \sigma' \rangle \rightarrow \sigma'' \]

\[ e \ \sigma = \nu \]

\[ \langle x := e, \sigma \rangle \rightarrow \sigma[x \mapsto \nu] \]
Program verification with concurrency

\[
\begin{align*}
\{ x = 0 \} & \quad y := x; \quad \{ x + 1 = 1 \land y = 0 \} \\
\| & \quad x := 4 \\
\{ x = 1 \land y = 0 \} & \quad x := x + 1;
\end{align*}
\]

→ Execution is interleaved
→ Intermediate assertions can be interferred with

→ Need a new reasoning framework!
→ New syntax, new semantics,
   new proof rules (proved sound w.r.t semantics), new VCG

→ (1969: Hoare Logic (Tony Hoare))
→ 1976: Owicki-Gries (Susan Owicki and David Gries)
→ 1981: Rely-Guarantee (Cliff Jones)
Owicki-Gries framework

Intuition:

- Syntax: our IMP language + Parallel operator + Await operator
- Semantics:
  - $P || Q$: pick one program and execute its current instruction
  - `AWAIT b DO c OD`: if guard is true execute c atomically
- Proof rules:
  - you prove *local correctness* (as before)
  - your prove *interference-freedom* (assertions not interfered with)

$$\begin{align*}
\{ \text{is\_even } x \} \\
 x := x + 1; \ {\{ \text{is\_even } x + 1 \} } & \quad \parallel \quad x := x + 2 \\
 x := x + 1; \\
\{ \text{is\_even } x \}
\end{align*}$$

$\rightarrow$ Needs a fully annotated program!

$\rightarrow$ Needs a “small-step semantics” $\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$
Owicki-Gries framework

Formally:

- **Syntax**: our IMP language + Parallel operator + Await operator
- **Semantics**:

\[
\begin{align*}
\langle c_1, \sigma \rangle &\rightarrow \langle c'_1, \sigma' \rangle \\
\langle c_1 \parallel c_2, \sigma \rangle &\rightarrow \langle c'_1 \parallel c_2, \sigma' \rangle \\
\langle c_2, \sigma \rangle &\rightarrow \langle c'_2, \sigma' \rangle \\
\langle c_1 \parallel c_2, \sigma \rangle &\rightarrow \langle c_1 \parallel c'_2, \sigma' \rangle
\end{align*}
\]

- **Hoare rules**:

\[
\begin{align*}
\{ P_1 \} \quad c_1 \quad \{ Q_1 \} \quad \{ P_2 \} \quad c_2 \quad \{ Q_2 \} \quad \text{interfree } c_1 \quad c_2 \\
\{ P_1 \land P_2 \} \quad c_1 \parallel c_2 \quad \{ Q_1 \land Q_2 \}
\end{align*}
\]

Where

\[
\text{interfree } c_1 \quad c_2 \equiv \\
\forall p \in (\text{assertions } c_1) \quad \forall (a, c) \in (\text{atomics } c_2) \quad \{ p \land a \} c \{ p \}
\]
Owicki-Gries framework

→ Quadratic explosion of proof obligations! (verification conditions)
→ Not compositional
→ Not complete: sometimes need auxiliary/ghost variables

\[
\{ x = 0 \} \\
x := x + 1; \parallel x := x + 1 \\
\{ x = 2 \}
\]

\[
\{ x = 0 \land a_1 = 0 \land a_2 = 0 \} \\
\{ a_2 = 0 \land x = 0 \lor a_2 = 1 \land x = 1 \} \quad \parallel \quad \{ a_1 = 0 \land x = 0 \lor a_1 = 1 \land x \} \\
< x := x + 1; a_1 := 1 > \\
\{ a_2 = 0 \land x = 1 \lor a_2 = 1 \land x = 2 \} \quad \parallel \quad \{ a_1 = 1 \land x = 1 \lor a_1 = 1 \land x \} \\
\{ x = 2 \} \\
\]
Demo
Rely-Guarantee?

Intuition:

- Syntax, semantics: as before (but no need for assertions)
- Proof rules:
  - each program is specified in isolation, assuming a behavior of the “environment” (other programs in parallel)
  - each program has: precondition, postcondition, rely and guarantee
  - rely and guarantee are relations between 2 states
  - rely expresses the maximum behavior of the environment (the interference that the program can tolerate)
  - guarantee expresses a maximum behavior promised to the environment

\[
\begin{align*}
&c 
\begin{cases}
P, R, G, Q
\end{cases}

\begin{cases}
P', R', G', Q'
\end{cases}

c' \\
\sigma_0 \xrightarrow{c} \sigma_1 \xrightarrow{c} \sigma_2 \xrightarrow{c'} \sigma_3 \xrightarrow{c} \sigma_4 \xrightarrow{c'} \sigma_5 \xrightarrow{c'} \sigma_6 \xrightarrow{c} \sigma_7
\end{align*}
\]
Rely-Guarantee?

Formally:

- Syntax, semantics: as before (but no need for assertions)
- Proof rules (examples):

\[ P \subseteq \{ s. f s \in Q \} \quad \{ (s, t). P s \land (t = f s \lor t = s) \} \subseteq G \quad \text{stable } P \quad R \quad \text{stable } Q \]

\[ \text{Basic } f \{ P, R, G, Q \} \]

\[ c_1\{ P_1, R_1, G_1, Q_1 \} \quad c_2\{ P_2, R_2, G_2, Q_2 \} \quad G_1 \subseteq R_2 \quad G_2 \subseteq R_1 \]

\[ c_1 || c_2\{ P_1 \cap P_2, R_1 \cap R_2, G_1 \cup G_2, Q_1 \cap Q_2 \} \]

Where \( \text{stable } P \quad R \quad = \quad \forall \quad \sigma \quad \sigma'. \quad (P\sigma \land R(\sigma, \sigma')) \rightarrow P\sigma' \)

\( \text{(doing an environment step before or after } P \text{ should not make } P \text{ invalid)} \)

Intuition: the guarantee of one program is the rely of the other program
Demo
We have seen today ...

- Need for new reasoning framework for parallel/concurrent programs
- Owicki-Gries
- Rely-Guarantee