CS4003: Formal Methods

Andrew Butterfield

1Foundations & Methods Group, Software Systems Laboratory
Andrew.Butterfield@cs.tcd.ie
Room G.39, O'Reilly Institute

Michaelmas Term, 2013

CS4003: Approach

<table>
<thead>
<tr>
<th>When</th>
<th>Where</th>
<th>What</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wed 10am</td>
<td>LB120</td>
<td>Lecture</td>
</tr>
<tr>
<td>Thursday 5pm</td>
<td>LB120</td>
<td>Lecture</td>
</tr>
<tr>
<td>Friday 2pm</td>
<td>LB107</td>
<td>mini-Solution;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tutorial/Examples;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mini-Exercise</td>
</tr>
</tbody>
</table>

CS4003: Assessment

- Exam: 80%
- Coursework: 20%
  - Project: 10%
  - Class (Mini-)Exercises: 10%
    - Mini-exercises handed out at end of Fri 2pm class.
    - Mini-solutions due in at start of following Friday 2pm class.
    - These deadlines are hard, as solutions will be given out at the start of the 2pm Friday class.

CS4003: Resources

- Blackboard (mymodule.tcd.ie)
- Main Reference Text:
  Unifying Theories of Programming,
  C.A.R. Hoare and Jifeng He,
  (out of print, will be uploaded to Blackboard)
- Secondary Texts:
  (out of print, will be uploaded to Blackboard)
What are “Formal” Systems?

- Specified collection of Symbols (lexicon)
- Specified ways of putting them together (well-formedness)
- Specific ways of manipulating symbol-sequences (inference rules)
- Typically the goal is to transform a starting sequence to a final one having desired properties.

Example: System $h\times\mathbb{I}$ (“H-Cross-I”)

- Symbols: $h$, $\times$, $\mathbb{I}$
- Well-Formed Sequences, $H$-Things and $I$-Things, where:
  - $H$-Thing: A $h$ followed by Crosses:
    - Crosses: Zero or more $\times$
  - $I$-Thing: A $\mathbb{I}$ followed by two $H$-Things
- Manipulations (let $f_1$ and $f_2$ stand for arbitrary Crosses).
  - $\langle\langle I$-absorb $\rangle\rangle$ $\mathbb{I}h_1h_2$ becomes $h_1$
  - $\langle\langle$ swap-Cross-H $\rangle\rangle$ $\mathbb{I}h_1h_2$ becomes $\mathbb{I}h_1h_2$ $f_1f_2$

- Goal: convert a starting I-Thing into a H-Thing

Example

$$\mathbb{I}h_1h_2h_3h_4h_5 = \langle\langle$ swap-Cross-H $\rangle\rangle \mathbb{I}h_1h_2h_3h_4h_5$$

Interpretation

<table>
<thead>
<tr>
<th>$h$</th>
<th>0</th>
<th>$\times$</th>
<th>+1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>zero</td>
<td>succ</td>
<td>plus</td>
</tr>
</tbody>
</table>

$$\begin{align*}
\mathbb{I}h_1h_2h_3h_4h_5 + 0 + 1 + 1 & = 2 + 3 \\
\mathbb{I}h_1h_2h_3h_4h_5 + 0 + 1 & = 1 + 4 \\
\mathbb{I}h_1h_2h_3h_4h_5 + 0 & = 0 + 5 \\
\mathbb{I}h_1h_2h_3h_4h_5 & = 5 \\
\mathbb{I}h_1h_2h_3h_4h_5 + (n + 1) + m & = n + (m + 1) \\
\mathbb{I}h_1 & = h_1 + 0 & m = m
\end{align*}$$
What’s the point?

- We can give very precise meanings to the symbols,
- The manipulations can have a very-well defined meaning,
- but, the symbols can be manipulated without our having to understand these meanings.
  - which is exactly how a computer does it!
- Formal Methods allow us to limit the scope for human error and to exploit the use of machines to help our analysis.

Formal Logic

We present a Formal System, called **Predicate Calculus**

**Symbols**
Those used for expressions and propositional logic, as well as $\exists$, $\forall$, $\cdot$.

**Well-Formedness**
Predicates: Well structured expressions whose type is Boolean.

**Manipulation Rules**
Classified as **Axioms** and **Inference Rules**.

**Goal**
To **prove** a given Predicate is “True”.

Expressions

We build basic expressions out of constants ($k \in \text{Const}$), variables ($v \in \text{Var}$), tuples, functions and operators:

$$e \in \text{Expr} ::= k \mid v \mid (e_1, \ldots, e_n) \mid \{e_1, \ldots, e_n\} \mid (e)$$

- Read $e \in \text{Expr} ::= \ldots$ as “$e$, an Expr, is $\ldots$”
- Read $k$ as “a constant $k$”
- Read $|$ as “or”
- We do not give a complete definition here of expressions, and will extend this as the course progresses.

Expressions (Aggregates)

Aggregates are distinguished by different brackets:

- Tuples: $\langle e_1, \ldots, e_n \rangle$
- Sets: $\{e_1, \ldots, e_n\}$
- Sequences: $\langle e_1, \ldots, e_n \rangle$

$$e \in \text{Expr} ::= (e_1, \ldots, e_n) \mid \{\ldots\} \mid (\ldots)$$
Expressions (Function application)

- We define function application without brackets, so \( f \ x \) denotes function \( f \) applied to argument \( x \) (if \( f \) is a prefix function — most are !)
- We can write function application with brackets if we prefer so \( f(x) \) and \( f \ x \) are equivalent.
- In some of the literature, function application is shown explicitly with a dot, so \( f . x \) is the same as \( f(x) \) (or \( f(x) \)).
- In the case where \( f \) is a postfix function, then applying \( f \) to \( x \) is written as \( xf \) (e.g. raising to a power — \( x^2 \) is the squaring function \(^2 \) post-applied to \( x \)).
- The notation used is often a matter of style, and depends on context.

Expressions (Infix operators)

\[
e \in \text{Expr} ::= e_1 \oplus e_2 \quad \text{infix operator application}
\]

- Here \( \oplus \) denotes any infix binary operator, such as \(+, -, *, /, =, <, >, \leq, \geq, \cup, \cap, \ldots\)
- Parentheses and precedence behave in the same manner as found in most programming languages, so \( x + y \times z \) is the same as \( x + (y \times z) \), but different from \((x + y) \times z\).

Well-Typedness

- We require expressions to be well typed.
  - e.g. we want to outlaw nonsense like \( 3 \div (A \cup B) \) or \( \langle 1, 0, 1, 0 \rangle - 5 = \text{True} \)
- we write the assertion that expression \( e \) has type \( T \) as \( e : T \)
- We can consider (for now) a type \( T \) as being the set of values that are allowed for something of that type.

Types

We make use of a rich variety of given types and type constructors:

\[
S, T \in \text{Type} ::= 1 \quad \text{unit type} \\
| \mathbb{B} \quad \text{boolean values} \\
| A \quad \text{(ASCII) characters} \\
| N, Z, Q, R, C \quad \text{numbers} \\
| P T \quad \text{set of } T \\
| T_1 \times \cdots \times T_n \quad \text{cross-product} \\
| T^* \quad \text{sequence of } T \\
| S \rightarrow T \quad \text{function from } S \text{ to } T
\]

- There are rules for checking (and/or inferring) expression types, which we ignore for now.
Expressions (Type annotations - 1)

- \( e \in \text{Expr} ::= \ldots \mid (e : T) \) type annotation

- As a standalone expression, it has type Boolean, and is true if \( e \) has type \( T \)
  - Note, if \( e \) is a variable (\( v \)) not in scope, then \( v : T \) is always true, and so can be considered a type declaration.
  - Note that an incorrect annotation is not a type error, but is simply a statement, so \( ((A \cup B) : \mathbb{N}) \) is a well-typed boolean-valued expression that just happens to be false (provided \( A \cup B \) is well-typed).

Expressions (Type annotations 2)

- Sometimes we use type annotations embedded in expressions as an abuse of notation e.g.,
  \[ (x : \mathbb{Q})^2 = 2 \quad \text{instead of} \quad x^2 = 2 \]

- Strictly speaking, this is ill-typed (why?), but the intention was to say
  \[ (x : \mathbb{Q}) \land x^2 = 2 \]

- Usually we shall avoid this abuse of notation.
Expression Meaning

- What is our intended meaning for an expression?
  (e.g. \( x^2 + y \))
  - it depends on the value of \( x \) and \( y \)
  - so, let \( x = 3 \) and \( y = 5 \) (say)
  - OK, so then \( x^2 + y \) has value 14.
- The “meaning” of an expression is the relationship it creates between the values of its variables and its own overall value:
  "meaning : values \((x, y)\) \(\rightarrow\) value"

- Alternatively, we can view the meaning as a function from variables to the expression’s value.

Environments

- We call the association of values with variables an Environment (or Interpretation) and is a table associating a value with every variable (whose value is defined).
- We assume environments are always well-formed, in that the associated value is always of the correct type.

```
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>N*</td>
<td>(1, 1, 2, 3, 5, 8, 13)</td>
</tr>
<tr>
<td>ix</td>
<td>N</td>
<td>4</td>
</tr>
<tr>
<td>offset</td>
<td>N</td>
<td>2</td>
</tr>
</tbody>
</table>
```

Evaluating Expressions against Environments

Given an environment (as per previous slide), it is possible to determine the value for an expression in a systematic way.

\[
\text{ix} + \text{offset} \leq \text{length list} = \text{"lookup above table" = 4 + 2} \leq \text{length \((1, 1, 2, 3, 5, 8, 13)\) = "defn. of + and length. (?)" = 6} \leq 7 \Rightarrow \text{"defn. of \(\leq\) (?)" = True}
\]

(?) Where do we find the definitions +, length and \(\leq\)?
Modelling Environments

We usually model Environments mathematically as a finite partial map \((\rightarrow)\) from variables to values:

\[
\rho \in \text{Env} = \text{Var} \rightarrow \text{Value}
\]

Here the ‘type’ \(\text{Value}\) should considered as the union of all possible types.

- A table entry mapping variable \(v\) to value \(k\) is written as \(v \rightarrow k\).
- A table is a set of table entry mappings (order is irrelevant):

\[
\{\text{list} \rightarrow \langle 1, 1, 2, 3, 5, 8, 13 \rangle, \ ix \rightarrow 4, \ offset \rightarrow 2\}
\]

Expressions as Functions

- Given an expression, and then an environment, we can get the value of that expression in that environment.
- We can therefore view the meaning of an expression as a (partial \((?))\) function from environments to values.
- Let \(\llbracket e \rrbracket\) denote the “meaning of \(e\)”, and \(\llbracket e \rrbracket_\rho\) the meaning given environment \(\rho\), then we can say

\[
\llbracket e \rrbracket : \text{Env} \rightarrow \text{Value} \\
\llbracket e \rrbracket_\rho = \ldots
\]

The meaning function, given an expression, returns a partial function from environment to values.

Example

- What is the meaning of expression

\[
ix + offset \leq length \text{ list}
\]

w.r.t. environment

\[
\rho = \{\text{list} \rightarrow \langle 1, 1, 2, 3, 5, 8, 13 \rangle, \ ix \rightarrow 4, \ offset \rightarrow 2\}
\]

- \[
\llbracket ix + offset \leq length \text{ list} \rrbracket_\rho = \text{True}
\]

- We shall see how to define \(\llbracket \rrbracket\) later.

Predicates

- A \textit{Predicate} is an “expression” whose type is boolean \((\mathbb{B})\).

\[
P \in \text{Pred} ::= \ldots \\
\llbracket P \rrbracket : \text{Env} \rightarrow \mathbb{B}
\]

Remember that “type” \(\text{Value}\) contains values of all types, including \(\mathbb{B}\), so \(\mathbb{B} \subseteq \text{Value}\).

- Unlike expressions, where evaluation w.r.t. an environment may be undefined, we insist that predicates always evaluate to either \text{True} or \text{False}.
Atomic Predicates

- An Atomic Predicate is an expression whose overall type is Boolean, and whose constituent parts have non-Boolean types
  \[ A, B, C \in \text{AtmPred} \quad ::= \quad e : \mathbb{B} \]
- They can be viewed as a function from an environment to the values \text{true} or \text{false}.
  \[ [A] : \text{Env} \rightarrow \mathbb{B} \]
- Examples:
  \[
  x + 3 \leq y \\
  f(x) = g(x) - h(x) \\
  \text{reverse}(\langle x \rangle \sqcup xs) = (\text{reverse xs}) \sqcup \langle x \rangle
  \]

True & False

- Two special predicates \text{true} and \text{false} always return \text{True} and \text{False} respectively, regardless of the environment.
- \text{True} and \text{true} are not the same:
  - \text{True} : \mathbb{B} is a boolean value
  - \text{true} : \text{Env} \rightarrow \mathbb{B} is a predicate, a function from environments to \mathbb{B}.
- Similarly for \text{False} and \text{false}.
- In practice we can often ignore the distinction, using them interchangeably.

Identities

- (Atomic) Predicate Expressions of the form:
  \[ e_1 = e_2 \]
  where \text{e}_1 and \text{e}_2 have the same type, are known as \text{Identities}.
- Identities play a major role in what is to come.

Predicate Logic Syntax

Given atomic predicates we can build a richer language using Propositional Connectives (\neg, \land, \lor, \Rightarrow, \equiv):

\[
P \in \text{Pred} \\
::= \quad \text{true} \mid \text{false} \mid A \quad \text{atomic predicates} \\
     \mid \neg P \quad \text{logical negation} \\
     \mid P_1 \land P_2 \quad \text{conjunction (logic-and)} \\
     \mid P_1 \lor P_2 \quad \text{disjunction (logic-or)} \\
     \mid P_1 \Rightarrow P_2 \quad \text{implication} \\
     \mid P_1 \equiv P_2 \quad \text{equivalence}
\]

This gives us the same power as propositional calculus, a.k.a. “digital logic”.
Predicate Logic Syntax

Considerably more power is obtained by adding Quantifiers \( (\forall, \exists, \exists_1) \) to the language:

\[ P \in \text{Pred} \]

\[
\begin{align*}
| \forall x : T \bullet P & \text{ universal quantification (for-all)} \\
| \exists x : T \bullet P & \text{ existential quantification (there-exists)} \\
| \exists_1 x : T \bullet P & \text{ unique existence quantification} \\
| [P] & \text{ universal closure}
\end{align*}
\]

The type annotations \( (: T) \) are optional and are often omitted when clear from context (or irrelevant!)

Predicate Meanings (Propositions)

The predicate meaning function can be described as follows:

\[ [\text{true}]_\rho = \text{True} \]
\[ [\text{false}]_\rho = \text{False} \]
\[ [e]_\rho = [e]_\rho \text{ pred. meaning = expr. meaning} \]
\[ [\neg P]_\rho = \text{logical inverse of } [P]_\rho \]
\[ [P_1 \land P_2]_\rho = \text{True iff both } [P_1]_\rho \text{ and } [P_2]_\rho \text{ are} \]
\[ [P_1 \lor P_2]_\rho = \text{True iff either } [P_1]_\rho \text{ or } [P_2]_\rho \text{ are} \]

We define implication and equivalence by translation

\[ P_1 \Rightarrow P_2 = \neg P_1 \lor P_2 \]
\[ P_1 \equiv P_2 = (P_1 \Rightarrow P_2) \land (P_2 \Rightarrow P_1) \]

Manipulating Environments (Maps)

We define the domain of an environment \( (\text{dom } \rho) \) as the set of all variables mentioned in \( \rho \).

We can use one environment \( (\rho') \) to override part or all of another \( (\rho) \), indicating this by \( \rho \oplus \rho' \).

The bindings in the second map, extend and take precedence over those in the first map — e.g.:

\[
\{ a \mapsto 1, b \mapsto 2, c \mapsto 3 \} \oplus \{ c \mapsto 33, d \mapsto 44 \} = \{ a \mapsto 1, b \mapsto 2, c \mapsto 33, d \mapsto 44 \}
\]

The Meaning of Quantifiers (I)

We can now give the meaning for the main two quantifiers as:

\[ [\forall x : T \bullet P]_\rho = \text{ for all values } k \text{ in } T \] 
we have \([P]_{\rho \oplus \{ x \mapsto k \}} = \text{True}\)

“\( \forall x : T \bullet P \) is true if \( P \) is true for all \( x : T \)”

\[ [\exists x : T \bullet P]_\rho = \text{ for at least one value } k \text{ in } T \] 
we have \([P]_{\rho \oplus \{ x \mapsto k \}} = \text{True}\)

“\( \exists x : T \bullet P \) is true if \( P \) is true for at least one \( x : T \)”
Example

Let $T = \{1, 3, 5\}$ and evaluate:

$$[\forall x : T \cdot x < y]_{\{x\mapsto2,y\mapsto5\}}$$

For all values $k$ in $\{1, 3, 5\}$

- $k = 1$ we have $[x < y]_{\{x\mapsto1,y\mapsto5\}} = 1 < 5 = True$
- $k = 3$ we have $[x < y]_{\{x\mapsto3,y\mapsto5\}} = 3 < 5 = True$
- $k = 5$ we have $[x < y]_{\{x\mapsto5,y\mapsto5\}} = 5 < 5 = False$

Overall result: $False$.

The Meaning of Quantifiers (II)

We can now give the meaning for the other two quantifiers as:

$$[[\exists x : T \cdot P]]_{\rho} \triangleq for\ exactly\ one\ k\ in\ T$$
$$\text{we have } [P]_{\rho\oplus\{x\mapsto k\}} = True$$

"$\exists x : T \cdot P$ is true if $P$ is true for exactly one $x : T$"

$$[[P]]_{\rho} \triangleq for\ any\ \rho'\ such\ that\ \text{dom }\rho' = \text{dom }\rho$$
$$\text{we have } [P]_{\rho'} = True$$

"$[P]$ is true if $P$ is true for all values of all its variables"

Evaluating Quantifiers

- Consider:

$$\forall n : \mathbb{N} \cdot \text{prime}(n) \land n > 2 \Rightarrow \neg \text{prime}(n + 1)$$

True, or False?

- To evaluate a quantifier, we may need to generate all possible environments $\rho'$ involving the bound variable — tricky if type has an infinite number of values.

- In general, in order to reason about quantifiers we need to use Axioms and Inference Rules.
Under what circumstances is the following true?

\[(x < 3 \land x > 5) \Rightarrow x = x + 1\]

Example I Commentary

\[\begin{align*}
(x < 3 \land x > 5) & \Rightarrow x = x + 1 \\
= \quad \text{"definition of } \Rightarrow" \\
\neg (x < 3 \land x > 5) \lor x &= x + 1 \\
= \quad \text{"5 < x < 3 clearly impossible"} \\
\neg (\text{false}) \lor x &= x + 1 \\
= \quad \text{"negation"} \\
\text{true} \lor x &= x + 1 \\
= \quad \text{"logical-OR"} \\
\text{true} \\
\end{align*}\]

It is always true!

Implication Truth-Table

The truth table for implication is as follows:

<table>
<thead>
<tr>
<th>(P)</th>
<th>(Q)</th>
<th>(P \Rightarrow Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Simply put, for \(P \Rightarrow Q\), if \(P\) is false, then the whole thing is true, regardless of \(Q\).
Logic Example II

The following statement is true:

\[ \forall n : \mathbb{N} \cdot n > 2 \Rightarrow \text{prime}(n) \Rightarrow \neg \text{prime}(n + 1) \]

How might we argue this?

Example II Commentary

\[ \forall n : \mathbb{N} \cdot n > 2 \Rightarrow (\text{prime}(n) \Rightarrow \neg \text{prime}(n + 1)) \]

▶ Case \( n \leq 2 \) : \( False \Rightarrow \ldots \) is true.
▶ Case \( n > 2 \land \neg \text{prime}(n) \) : \( True \Rightarrow (False \Rightarrow \ldots) \) is true.
▶ Case \( n > 2 \land \text{prime}(n) \) : 2 is only even prime, so here \( n \) is odd, so \( n + 1 \) is even and therefore not prime.

Quantifier Expansion

▶ We can view quantifiers as (approximate) shorthand for repeated logical-and/or:

\[ \forall n : \mathbb{N} \cdot P(n) = P(0) \land P(1) \land P(2) \land P(3) \land \ldots \]
\[ \exists n : \mathbb{N} \cdot P(n) = P(0) \lor P(1) \lor P(2) \lor P(3) \lor \ldots \]

▶ The equality is exact if the type quantified over is finite:

\[ \forall b : \mathbb{B} \cdot P(b) = P(False) \land P(True) \]
\[ \exists b : \mathbb{B} \cdot P(b) = P(False) \lor P(True) \]

Logic Example III

Assuming all variables are natural numbers, is the statement

\[ \forall x \cdot \exists x \cdot x < y \]

equivalent to

\[ \forall x \cdot x < y ; \]

or

\[ \exists z \cdot z < y ; \]

or neither?

(What precisely do the three statements actually mean?)

What about \( \exists z \cdot z < x \)?
Example III Commentary (1)

\[ \forall x \cdot \exists x \cdot x < y \] 
\[ \rho = " by our semantics " \]
\[ \exists x \cdot x < y \] 
\[ \rho \oplus \{ x \mapsto \rightarrow k \}, \text{ for all } k \]
\[ [x < y]_{\rho \oplus \{ x \mapsto \rightarrow k \}}, \text{ for all } k, \text{ for some } k' \]
\[ [x < y]_{\rho \oplus \{ x \mapsto \rightarrow k' \}}, \text{ for all } k, \text{ for some } k' \]
\[ [0 < y]_{\rho} \]
\[ = " true for natural number } y \text{ if non-zero } " \]
\[ y \neq 0 \]

This is a statement about } y \text{, not about } x \text{, and the outer } \forall x \text{ is masked by inner } \exists x \text{.}

Example III Commentary (2)

\[ [\forall x \cdot x < y]_{\rho} \]
\[ = " by our semantics " \]
\[ [x < y]_{\rho \oplus \{ x \mapsto \rightarrow k \}}, \text{ for all } k \]
\[ = " false if we choose } k = \rho(y) + 1 " \]
\[ False \]
\[ [\exists z \cdot z < y]_{\rho} \]
\[ = " by our semantics " \]
\[ [z < y]_{\rho \oplus \{ z \mapsto \rightarrow k \}}, \text{ for some } k \]
\[ = " choose } k = 0 " \]
\[ [0 < y]_{\rho} \]
\[ = " true for natural number } y \text{ if non-zero } " \]
\[ y \neq 0 \]

Example III Commentary (3)

\[ [\exists z \cdot z < x]_{\rho} \]
\[ = " by our semantics " \]
\[ [z < x]_{\rho \oplus \{ z \mapsto \rightarrow k \}}, \text{ for some } k \]
\[ = " choose } k = 0 " \]
\[ [0 < x]_{\rho} \]
\[ = " true for natural number } x \text{ if non-zero } " \]
\[ x \neq 0 \]

This is a statement about } x \text{, not } y \text{, or } z \text{.}

Logic Example IV

Define the following in terms of other predicates:

\[ \exists_1 x \cdot P(x) \]

Here } P(x) \text{ indicates explicitly that } P \text{ mentions } x \text{.}

A reminder:

\[ [\exists_1 x : T \cdot P]_{\rho} \cong \text{ for exactly one } k \text{ in } T \]
\[ \text{we have } [P]_{\rho \oplus \{ x \mapsto \rightarrow k \}} = True \]

\[ " \exists_1 x : T \cdot P \text{ is true if } P \text{ is true for exactly one } x : T" \]
Example IV Commentary

\[ \exists x \cdot P(x) \]
\[ \equiv (\exists x \cdot P(x)) \land (\forall x \forall y \cdot (P(x) \land P(y)) \Rightarrow x = y) \]

There exists only one \( x \) satisfying \( P \), iff
there is at least one \( x \) satisfying \( P \)
and for all \( x \) and \( y \)
if both \( x \) and \( y \) satisfy \( P \) then it must the case that \( x = y \).

Example V Commentary

\[ \exists x \cdot x = y + z \land x < 2 \ast y \]

“ The only way this can be true is when \( x = y + z \) ”
“ so replace \( x \) by that. ”
\[ \exists x \cdot y + z = y + z \land y + z < 2 \ast y \]

“ \( x \) is not mentioned, so \( \exists \) is now redundant ”
\[ y + z = y + z \land y + z < 2 \ast y \]

“ \( y + z = y + z \) is clearly true ”
\[ y + z < 2 \ast y \]

“ arithmetic ”
\[ z < y \]

Logic Example V

Simplify the following predicate:
\[ \exists x \cdot x = y + z \land x < 2 \ast y \]

Formal Methods: early History

  - Floyd described a technique for annotating flowcharts with predicates describing what should be true at every point in the execution.
  - Hoare introduced the notation known today as a “Hoare triple”:
    \[ Pre \{ prog \} Post \]
    “if \( Pre \) holds at the start and \( prog \) terminates, then \( Post \) will be true at the end”.
Floyd/Hoare as a foundation

- The framework we shall explore is known as “Unifying Theories of Programming” (UTP)
- It is a direct descendant of the Floyd/Hoare approach
- UTP aims to link theories of different kinds of programming language:
  imperative, concurrent, functional, logical, dataflow, assembler, …
- C.A.R. (Tony) Hoare is one of the prime movers behind UTP
  - he spent most of time in Oxford at the Programming Research Group (PRG)
  - now with Microsoft Research, Cambridge
  - most famous outside the formal methods community for QuickSort!

Mini-Exercise 1

Q1.1 Evaluate $A \cap \text{elems}(\text{trace}) = \emptyset$

in environment $\{A \mapsto \{2, 4, 6\}, \text{trace} \mapsto \langle 1, 2, 1, 3 \rangle\}$

- $\text{elems}$ takes a list and returns the set of values in that list

Q1.2 Is $\exists n : \mathbb{N} \cdot n \geq 3 \land \text{prime}(n) \Rightarrow \text{prime}(n - 1)$ true or false? Justify your answer

Q1.3 Write one paragraph about a software failure (not yours) that you found most vexing.

(handwritten, due in at start of 2pm class, Fri. 4th October)
Class 4

Key Concept: Free and Bound Variables

- Expressions, Formula, Propositions and Predicates usually involve a number of variables.
- Depending on context, a variable occurrence can be free or bound.
- If a variable occurrence is bound, that is because there is something in the context in which it occurs that is binding it.
- A binding occurrence is something (a piece of syntax) that mentions a variable in a special way, usually introducing it to stand for something specific.
- If nothing is binding an occurrence of a variable, then it is free.
- Free occurrences of variables are usually there to denote a wide range of possible values (of a given type).

Examples of Binding — Definitions

- The definition (read 'x is defined to be 3'):
  \[ x \triangleq 3 \] (1)
  binds the variable \( x \) to 3 wherever the definition is in scope.
- The following recursive definition of factorial:
  \[
  \begin{align*}
  fac(0) & \triangleq 1 \\
  fac(n) & \triangleq n \ast fac(n - 1)
  \end{align*}
  \]
  binds two variables:
  - The variable \( fac \) is bound wherever this definition is in scope
  - \( n \) is bound in the righthand-side of the second line
  - The binding occurrence of \( n \) is in the lefthand-side.

Examples of Binding — Quantifiers

- The Logic Quantifiers \( \forall, \exists \) are examples of binding constructs.
- For example: \( \exists x : \mathbb{Q} \ast x^2 = 2 \) has
  - a binding occurrence of \( x \) immediately after \( \exists \),
  - and it is bound in the expression to the right of \( \ast \).
- Consider:
  \[ \forall x \ast x \geq 3 \Rightarrow y = 0 \land \exists y \ast y > 4 \land z = 10 \]
  Which variable occurrences are free, bound or binding?
Answer

- We shall colour-code as follows:
  
<table>
<thead>
<tr>
<th>Type</th>
<th>Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td>Green</td>
</tr>
<tr>
<td>Binding</td>
<td>Red</td>
</tr>
<tr>
<td>Bound</td>
<td>Orange</td>
</tr>
</tbody>
</table>

- \( \forall x \cdot x \geq 3 \Rightarrow y = 0 \land \exists y \cdot y > 4 \land z = 10 \)

- the first occurrence of \( y \) is free, whilst the second and third are binding and bound respectively.
- This is why we only talk about occurrences of variables as being free or bound.

Examples of Binding — Programs

- In a program block, local variable declarations are binding
- For example, code to swap \( x \) and \( y \), using local variable \( \text{temp} \):

```plaintext
{ int temp ;
  temp = x ;
  x = y ;
  y = temp ;
}
```

Manipulating Free and Bound Variables

- The key operation we use to manipulate free and bound variables is that of substitution.
- This takes two forms:
  - \( \alpha \)-Substitution, or Renaming of Bound Variables
  - Replacing all Free Occurrences of a Variable by an Expression.
- We change our substitution notation, instead of \( E[x := F] \), we write \( E[F/x] \) to denote the substitution of \( F \) for every free occurrence of \( x \) in \( E \).
- This notation extends to cover multiple simultaneous substitutions:

\[
E[F_1, F_2, \ldots, F_n/x_1, x_2, \ldots, x_n]
\]

Replacing Free Variables by Expressions (I)

- We often want to replace free occurrences of variables by expressions (of the same type).
- We denote the replacement by \( F \) of all occurrences of free variable \( a \) in expression \( E \), by the notation \( E[F/a] \).
- It is defined if none of the free variables in \( F \) have binding occurrences in \( E \).
- Example:

\[
(\forall x \cdot x \geq 3 \Rightarrow y = 0 \land \exists y \cdot y > 4 \land z = 10) \left[ a + 5/y \right] = \forall x \cdot x \geq 3 \Rightarrow a + 5 = 0 \land \exists y \cdot y > 4 \land z = 10
\]

Note that only the free occurrence of \( y \) is affected.
Replacing Free Variables by Expressions (II)

- Substituting \(x - 10\) for \(y\) does not work — we get a phenomenon known as *name capture*:
  \[
  (\forall x \cdot x \geq 3 \Rightarrow y = 0 \land \exists y \cdot y > 4 \land z = 10) \mid \frac{x - 10}{y}
  = \forall x \cdot x \geq 3 \Rightarrow x - 10 = 0 \land \exists y \cdot y > 4 \land z = 10
  \]
  - The first free occurrence of \(y\) (in \(y = 0\)) is replaced by \(x - 10\), where the free \(x\) becomes bound!
  - In \(E[F/x]\), the free/bound status of variables in \(F\) should not change.

Renaming Bound Variables (I)

- We can rename bound variables in a consistent way (binding, and all bound occurrences) without changing the meaning of the underlying expression.
- We denote the replacement of bound variable \(a\) by \(b\) in \(E\) by \(E(a \mapsto b)\) (read as “\(a\) becomes \(b\)”).
- It is defined if \(b\) does not occur free in \(E\).
- It is defined in terms of free variable substitution:
  \[
  \begin{align*}
  &\langle\langle \forall \alpha\text{-rename} \rangle\rangle (\forall x \cdot P)\{x \alpha \mapsto y\} = \forall y \cdot (P[y/x]) \\
  &\langle\langle \exists \alpha\text{-rename} \rangle\rangle (\exists x \cdot P)\{x \alpha \mapsto y\} = \exists y \cdot (P[y/x])
  \end{align*}
  \]

Renaming Bound Variables (II)

- Example:
  \[
  (\forall y \cdot x \geq 3 \Rightarrow y = 0 \land \exists y \cdot y > 4 \land z = 10)\{y \alpha \mapsto w\}
  = \forall w \cdot x \geq 3 \Rightarrow w = 0 \land \exists y \cdot y > 4 \land z = 10
  \]
  - Note that only the instances of \(y\) outside the inner \(\exists y\) were replaced.

Renaming Bound Variables (III)

- Substituting \(x\) for \(y\) is illegal:
  \[
  (\forall y \cdot x \geq 3 \Rightarrow y = 0 \land \exists y \cdot y > 4 \land z = 10)\{y \alpha \mapsto x\}
  = \forall x \cdot x \geq 3 \Rightarrow x = 0 \land \exists z \cdot z > 4 \land z = 10
  \]
  - We see that \(x\) becomes bound!
Using Both Substitutions

- *alpha*-substitution is often used to get around the restriction associated with replacing free variables.
- Let's say we really want to replace free \( y \) by \( x - 10 \) in the following

\[
\forall x \bullet x \geq 3 \Rightarrow y = 0 \land \exists y \bullet y > 4 \land z = 10
\]

- First, we use \( \alpha \)-substitution to rename the binding and bound occurrences of \( x \) to something different (\( a \), say) — \( \{ x \mapsto a \} \)

\[
\forall a \bullet a \geq 3 \Rightarrow y = 0 \land \exists y \bullet y > 4 \land z = 10
\]

- Then we do our desired substitution \( [x - 10/y] \):

\[
\forall a \bullet a \geq 3 \Rightarrow x - 10 = 0 \land \exists y \bullet y > 4 \land z = 10
\]

Function Application

- Consider a function, defined as,

\[
f(x) \equiv E_{\text{rhs}}
\]

- If applied to an argument \( E_{\text{arg}} \), then the result is given by replacing all free occurrences of \( x \) in \( E_{\text{rhs}} \) by \( E_{\text{arg}} \):

\[
f(E_{\text{arg}}) = E_{\text{rhs}}[E_{\text{arg}}/x]
\]

- Remember to use \( \alpha \)-substitution to avoid name-capture, if necessary.

Use in Logic — an example

- If we know that for any \( s : T \) that \( P[s/x] \) is true, then we can conclude that

\[
\forall x : T \bullet P
\]

- This inference rule is often written as

\[
\frac{s : T \quad P[s/x]}{\forall x : T \bullet P}
\]

- Alternatively, we have the converse law:

\[
\frac{\forall x : T \bullet P \quad s : T}{P[s/x]}
\]
Rule-based Manipulation

- So far, we have seen ad-hoc ways of reasoning about properties.
- Calculation/Evaluation
  - when we had precise information about environment values
  - doing symbolic evaluation
- “Handwaving”
  - reasoning about a range of environments
  - reasoning over infinite cases
- We need a way to reason through symbol manipulation (the “formal approach”).

The Rules of Formal Logic

- There are many ways to formulate the laws of logic.
- We shall adopt the formulation of Equational Logic.
  - Logic proofs handled “equation-style”
  - Four inference laws (Liebniz, Transitivity, Substitution, Equanimity)
  - Many axioms

Inference Rules

- Inference rules describe how we may deduce the truth of one statement (the consequent, $C$) from the truth of a collection of other statements (the antecedents $A_i$), subject to some side-condition $S$.

  Rule Name: $\frac{A_1, \ldots, A_n}{C}$ $S$

- “From the truth of $A_1$ and … and $A_n$ we can deduce $C$ (provided $S$ holds),”
- Side-conditions $S$ are not predicates themselves, but are properties of predicates.
Inference meta-Notation

- Sometimes both antecedents and consequents in inference rules use extra (meta-)notation not part of the formal language.
- For example, notation
  \[ P[v := Q] \]
  (Textual Substitution)
  describes predicate \( P \) where all instances (if any) of variable \( v \) have been replaced by \( Q \).
- e.g. \((x < y + z)[y := 3 \ast n]\) denotes \( x < 3 \ast n + z \)

Equational Axioms of Propositional Calculus

- Axioms are laws taken as true, as a starting point.
- Different sets of axioms are possible.
- G&S define their own set, of which the following are just a few:

  - ⟨≡-assoc⟩ \((p \equiv q) \equiv (q \equiv r)\) (3.1)
  - ⟨≡-comm⟩ \(p \equiv q \equiv q \equiv p\) (3.2)
  - ⟨false-def⟩ false \equiv \neg true (3.8)
  - ⟨∨-comm⟩ \(p \lor q \equiv q \lor p\) (3.24)
  - ⟨golden-rule⟩ \(p \land q \equiv q \equiv p \lor q\) (3.35)
  - ⟨⇒-join⟩ \(p \Rightarrow q \equiv p \lor q \equiv q\) (3.57)

  The numbers in parentheses are those used in G&S, but the names are different.

The inference rules of Equational Logic

- Equanimity: \( P, P = Q \) \( \longrightarrow \) \( Q \)
- Transitivity: \( P = Q, Q = R \) \( \longrightarrow \) \( P = R \)
- Substitution: \( P \) \( \longrightarrow \) \( P[r := Q] \) [no capture]
- Leibniz: \( F = G \) \( \longrightarrow \) \( E[v := F] = E[v := G] \) [\( v \) a variable]
Inference Rule: Equanimity

\[ P, \quad P = Q \]

\[ Q \]

- If \( P \) is true, and we know that \( P = Q \) then, we can deduce that \( Q \) must be true also.
- Simple, really!
- Not actually mentioned in Gries & Schneider (1994), but used in that text!

Inference Rule: Transitivity

\[ P = Q, \quad Q = R \]

\[ P = R \]

- If \( P \) equals \( Q \) and \( Q \) equals \( R \), then \( P \) equals \( R \).
- This key property of equality allows us to “linearise” our proofs.
- Assume we know that \( A = B, B = C, C = D \) and \( D = E \), and we want to prove that \( A = E \).
- We get branching — we can use the first two equalities to give us \( A = C \), and the second two to get \( C = E \), and then combine these.

Proving with Transitivity (I)

- We can use our inference rule notation to illustrate the proof:

\[
\begin{align*}
A &= B \\
B &= C \\
C &= D \\
D &= E \\
\hline
A &= C \\
C &= E \\
\hline
A &= E
\end{align*}
\]

- Very unwieldy!

Proving with Transitivity (II)

- However, this very transitivity property allows us to re-write the proof as

\[
\begin{align*}
A &= \text{ “ why } A = B \text{ ”} \\
B &= \text{ “ why } B = C \text{ ”} \\
C &= \text{ “ why } C = D \text{ ”} \\
D &= \text{ “ why } D = E \text{ ”} \\
E &= \text{ } \\
\end{align*}
\]

- Much better!
Predicate textual substitution

- Consider the consequent of Substitution: $P[r := Q]$
- We do not expect $P$ to appear with an explicit substitution (remember, $[r := Q]$ is not in the language).
- Instead $P[r := Q]$ denotes the case where:
  - $P$ contains an occurrence of $Q$ within it.
  - We use variable $r$ to refer to that occurrence of $Q$.

Inference Rule: Substitution

- If predicate $P$ is true, then so is $P$ with all instances of any boolean variable ($r$) replaced by an arbitrary predicate $Q$, (provided $r$ does not occur inside any quantifier binding any free variable of $Q$ — no capture).
- Example:
  
  $$p \lor \neg p$$
  
  $$(p \lor \neg p)[r := (x \leq y + z)]$$
  
  or (performing the substitution)
  
  $$(x \leq y + z) \lor \neg (x \leq y + z)$$

Inference Rule: Leibniz

- $F = G$
  
  $$E[v := F] = E[v := G]$$
  
  $[v$ a variable$]$

  - If $F = G$ then, in any predicate/expression $E$ mentioning $F$, we can replace that mention by $G$.
  - a.k.a. “Substitution of equals for equals”.
  - Here, $E$, $F$ and $G$ stand for arbitrary predicates and/or expressions.
  - Example:
    
    $$x+x = 2x$$
    
    $$(y - v)[v := x + x] = (y - v)[v := 2x]$$
    
    or
    
    $$x+x = 2x$$
    
    $$y - (x + x) = y - (2x)$$

Theorems and Proofs

- **Theorems** are predicates that are logical consequences of the axioms and inferences rules.
- A predicate is shown to be a theorem by providing a proof.
- A proof is either:
  - an axiom
  - or a consequence of an inference rule whose premises are axioms or pre-existing theorems.
  - in practice, we chain these together using the linear form of the Transitivity inference rule.
Example: $R \equiv R \equiv S \equiv S$, see G&S,p43

\[
\begin{align*}
\equiv\text{-assoc} & \quad (p \equiv q) \equiv (p \equiv (q \equiv r)) \quad (3.1) \\
\equiv\text{-comm} & \quad p \equiv q \equiv q \equiv p \quad (3.2)
\end{align*}
\]

$R \equiv R \equiv S \equiv S$

= “ $\equiv\text{-comm}$ with $p = R$ and $q = S$ ”

$R \equiv R$

= “ $\equiv\text{-comm}$ with $p = R$ and $q = S$ ”

$R \equiv S \equiv S \equiv R$

“ Axiom $\equiv\text{-comm}$, so we are done ”

Some Terminology

Conjecture a predicate we believe is true, but with no proof (yet).

Theorem a conjecture for which we have a proof.

Law a theorem or an axiom (i.e. a predicate either given as true, or proven to be so).

Theorems of Propositional Calculus

- We build a collection of proven theorems, from axioms, using inferences.
- We can also use these theorems in other proofs
- G&S proves many, such as :

\[
\begin{align*}
\equiv\text{-assoc} & \quad (p \equiv q) \equiv (p \equiv (q \equiv r)) \quad (3.1) \\
\equiv\text{-comm} & \quad p \equiv q \equiv q \equiv p \quad (3.2) \\
\not\equiv\text{-comm} & \quad (p \not\equiv q) \equiv (q \not\equiv p) \quad (3.16) \\
\lor\text{-unit} & \quad p \lor \text{false} \equiv p \quad (3.30) \\
\land\text{-assoc} & \quad (p \land q) \land r \equiv p \land (q \land r) \quad (3.37) \\
\lor\text{-exclusive-or} & \quad p \not\equiv q \equiv (\lnot p \land q) \lor (p \land \lnot q) \quad (3.53) \\
\Rightarrow\text{-meet} & \quad p \Rightarrow q \equiv p \land q \equiv p \quad (3.60)
\end{align*}
\]

And many, many more.
Adding flexibility

▶ In principle, a proof of $P$ uses the 4 inference rules and axioms to transform $P$ into axioms (e.g. proof of $R \equiv R \equiv S \equiv S$ above).
▶ We can also transform, using laws, to any law, to get a proof.
  i.e if $P$ and $Q$ are theorems, then so is $P \equiv Q$
▶ However, we can also derive new inference rules (a.k.a. derived rules)
  ▶ allows better structuring of proofs
  ▶ in principle can be converted to proof using original four rules.

Derived Proof Techniques (I)

▶ **Reduction**: Prove $P \equiv Q$ by converting $P$ to $Q$ (or v.v.)
▶ **Deduction**: Prove $P \Rightarrow Q$ by assuming $P$, and proving $Q$ careful: variables in $P$ must be treated as constants!
▶ **Boolean Case Analysis**:
  Prove $Q[b := P]$ by proving $Q[b := True]$ and $Q[b := False]$

  Boolean Case Analysis: $
  \begin{array}{c}
  Q[b := True] \\
  Q[b := False]
  \end{array}
  

▶ **Case Analysis**: Prove $P$ by finding $Q_i, i \in 1 \ldots n$ such that we can prove $(Q_1 \lor \ldots \lor Q_n)$ and $Q_i \Rightarrow P$ for all $i$

  Case Analysis: $
  \begin{array}{c}
  (Q_1 \lor \ldots \lor Q_n) \quad (Q_i \Rightarrow P)_{i \in 1\ldots n}
  \end{array}
  

Derived Proof Techniques (II)

▶ **Mutual Implication**: Prove $P \equiv Q$ by proving $P \Rightarrow Q$ and $Q \Rightarrow P$

  Mutual Implication: $
  \begin{array}{c}
  P \Rightarrow Q \\
  Q \Rightarrow P
  \end{array}
  \quad \Rightarrow \quad
  P \equiv Q
  
▶ **Contradiction**: Prove $P$ by proving $\neg P \Rightarrow false$

  Contradiction: $
  \begin{array}{c}
  \neg P \Rightarrow False
  \end{array}
  \quad \Rightarrow \quad
  P
  
▶ **Contrapositive**: Prove $P \Rightarrow Q$ by proving $\neg Q \Rightarrow \neg P$

  Contrapositive: $
  \begin{array}{c}
  \neg Q \Rightarrow \neg P
  \end{array}
  \quad \Rightarrow \quad
  P \Rightarrow Q
  
Mutual Transitivity

▶ Transitivity works when we mix $\equiv$ and $\Rightarrow$

  $\equiv\Rightarrow$-trans: $(p \Rightarrow q) \land (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$ (3.82)
  $\equiv\Rightarrow\equiv$-trans: $(p \equiv q) \land (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$ (3.82)
  $\equiv\Rightarrow\Rightarrow$-trans: $(p \Rightarrow q) \land (q \equiv r) \Rightarrow (p \Rightarrow r)$ (3.82)

▶ We can mix these in Reduction proofs (here that $P \Rightarrow R$):

  $P$
  $\Rightarrow$
  $\quad " \text{why} \ P \Rightarrow Q "$
  $Q$
  $\equiv$
  $\quad " \text{why} \ Q \equiv R "$
  $R$

▶ We call this $\Rightarrow$-Reduction
▶ Also possible is $\prec$-Reduction, using $=, <$ and $\leq$. 
Example: $x + y \geq 2 \Rightarrow (x \geq 1 \lor y \geq 1)$ (G&S)

- We apply *Contrapositive* to get:
  $$\neg (x \geq 1 \lor y \geq 1) \Rightarrow \neg (x + y \geq 2)$$
  $$= \neg (x \geq 1) \land \neg (y \geq 1) \Rightarrow \neg (x + y \geq 2)$$

- We now use *Deduction*: assume $x < 1$ and $y < 1$ as axioms, to show $x + y < 2$
- We now use *<Reduction>*:
  $$x + y$$
  $$< \text{ “ assumption: } x < 1, y < 1, \text{ arithmetic ”}$$
  $$1 + 1$$
  $$= \text{ “ arithmetic ”}$$
  $$2$$
Mini-Exercise 1

Q1.1 Evaluate $A \cap \text{elems}(\text{trace}) = \emptyset$ in environment $\{A \mapsto \{2, 4, 6\}, \text{trace} \mapsto \langle 1, 2, 1, 3 \rangle\}$ where
   - `elems` takes a list and returns the set of values in that list

Q1.2 Is $\exists n : N \cdot n \geq 3 \land \text{prime}(n) \Rightarrow \text{prime}(n - 1)$ true or false? Justify your answer

Q1.3 Write one paragraph about a software failure (not yours) that you found most vexing.

(handwritten, due in at start of 2pm class, Fri. 4th October)

Quantifier Axioms

\[
\begin{align*}
\text{∀-lpt} & : (\forall x \cdot x = E \Rightarrow P) \equiv P[E/x], \quad x \notin E \\
\text{∃-lpt} & : (\exists x \cdot x = E \land P) \equiv P[E/x], \quad x \notin E \\
\text{∀-distr} & : (\forall x \cdot P) \land (\forall x \cdot Q) \equiv (\forall x \cdot P \land Q) \\
\text{∃-distr} & : (\exists x \cdot P) \lor (\exists x \cdot Q) \equiv (\exists x \cdot P \lor Q) \\
\text{∀-swap} & : (\forall x \cdot \forall y \cdot P) \equiv (\forall y \cdot \forall x \cdot P) \\
\text{∃-swap} & : (\exists x \cdot \exists y \cdot P) \equiv (\exists y \cdot \exists x \cdot P) \\
\text{∀-nest} & : (\forall x, y \cdot P) \equiv (\forall x \cdot \forall y \cdot P) \\
\text{∃-nest} & : (\exists x, y \cdot P) \equiv (\exists x \cdot \exists y \cdot P) \\
\text{∀-rename} & : (\forall x \cdot P) \equiv (\forall y \cdot P[y/x]), \quad y \notin P \\
\text{∃-rename} & : (\exists x \cdot P) \equiv (\exists y \cdot P[y/x]), \quad y \notin P \\
\text{∀-∀-distr} & : P \land (\forall x \cdot Q) \equiv (\forall x \cdot P \land Q) \quad x \notin P \\
\text{gen-deMorgan} & : (\exists x \cdot P) \equiv \neg (\forall x \cdot \neg P)
\end{align*}
\]
Moving Bindings Around

- the following axioms move bindings around
  
  $\forall\text{-swap} \quad (\forall x \bullet y \bullet P) \equiv (\forall y \bullet \forall x \bullet P)$
  
  $\exists\text{-swap} \quad (\exists x \bullet \exists y \bullet P) \equiv (\exists y \bullet \exists x \bullet P)$
  
  $\forall\text{-nest} \quad (\forall x, y \bullet P) \equiv (\forall x \bullet \forall y \bullet P)$
  
  $\exists\text{-nest} \quad (\exists x, y \bullet P) \equiv (\exists x \bullet \exists y \bullet P)$

- often we implicitly use these laws, ignoring binding, nesting, or ordering
- Note these only apply when the quantifier is the same!
  
  $(\forall x \bullet \exists y \bullet P) \not\equiv (\exists y \bullet \forall x \bullet P)$

Quantifier Distributivity

- We can distribute a quantifier through the appropriate binary propositional operator:
  
  $\forall\text{-distr} \quad (\forall x \bullet P) \land (\forall x \bullet Q) \equiv (\forall x \bullet P \land Q)$
  
  $\exists\text{-distr} \quad (\exists x \bullet P) \lor (\exists x \bullet Q) \equiv (\exists x \bullet P \lor Q)$

- To understand these, consider them as repeated and/or:
  
  $(P(x_1) \land P(x_2) \ldots) \land (Q(x_1) \land Q(x_2) \ldots)$
  
  $\equiv ((P(x_1) \land Q(x_1)) \land (P(x_2) \land Q(x_2)) \land \ldots)$
  
  $(P(x_1) \lor P(x_2) \ldots) \lor (Q(x_1) \lor Q(x_2) \ldots)$
  
  $\equiv ((P(x_1) \lor Q(x_1)) \lor (P(x_2) \lor Q(x_2)) \lor \ldots)$

- c.f. “Quantifier Expansion” slide, Class 2.

Mixed Quantifier Distributivity

- We can distribute a quantifier through the “other” binary propositional operator, under certain circumstances:
  
  $\forall\text{-}\forall\text{-distr} \quad P \lor (\forall x \bullet Q) \equiv (\forall x \bullet P \lor Q) \quad x \not\in P$
  
  $\forall\text{-}\exists\text{-distr} \quad P \land (\exists x \bullet Q) \equiv (\exists x \bullet P \land Q) \quad x \not\in P$

- Essentially we can merge $P$ as shown above if it does not mention $x$.
- Law $\forall\text{-}\exists\text{-distr}$ is a theorem, not an axiom (proof depends on $\forall\text{-gen-deMorgan}$, so let’s defer this)

Renaming

- We have already covered this
  
  $\forall\text{-rename} \quad (\forall x \bullet P) \equiv (\forall y \bullet P[y/x]), \quad y \not\in P$
  
  $\exists\text{-rename} \quad (\exists x \bullet P) \equiv (\exists y \bullet P[y/x]), \quad y \not\in P$

- We can uniformly replace binding and bound occurrences of one variable by another, provided the new variable is “fresh”.
- Known for historical reasons as $\alpha$-renaming.
Generalised de-Morgan

- de-Morgan's laws apply to quantifiers:
  \[ \langle \text{gen-deMorgan} \rangle \; (\exists x \bullet P) \iff (\forall x \bullet \neg P) \]
  \[ \neg (\exists x \bullet P) \iff (\forall x \bullet P) \]
  \[ (\exists x \bullet P) \iff (\forall x \bullet \neg P) \]
  \[ (\exists x \bullet \neg P) \iff (\forall x \bullet P) \]

- the first law above is an axiom, the rest are theorems
- Proving the three theorems above involves the axiom, and the law of involution:
  \[ \langle \neg \; -\text{invol} \rangle \; \neg \neg p \iff p \]
  (voluntary exercises !)

Proof of theorem \( \langle \land-\exists\text{-distr} \rangle \)

- **Goal:** \( P \land (\exists x \bullet Q) \equiv (\exists x \bullet P \land Q), \; x \notin P \)
- **Strategy:** Reduce left-hand side (lhs) to right-hand side (rhs)
- **Proof:**
  \[
  \begin{align*}
  P \land (\exists x \bullet Q) & = \quad \langle \text{gen-deMorgan} \rangle \quad "\" \\
  P \land (\forall x \bullet \neg Q) & = \quad "\quad \langle \neg \; -\text{invol} \rangle \quad "\" \\
  \neg P \land (\forall x \bullet \neg Q) & = \quad "\quad \langle \text{deMorgan} \rangle \quad "\" \\
  \neg (\forall x \bullet \neg P \lor \neg Q) & = \quad "\quad \langle \lor \; \forall \text{-distr} \rangle \quad "\\notag
  \end{align*}
  \]

- **The “one-point” laws**
  - The “one-point” laws are very important.
    \[
    \langle \forall\text{-1pt} \rangle \; (\forall x \bullet x = E \Rightarrow P) \equiv P[E/x], \; x \notin E
    \]
    \[
    \langle \exists\text{-1pt} \rangle \; (\exists x \bullet x = E \land P) \equiv P[E/x], \; x \notin E
    \]
  - These allow us to get rid of quantifiers.
  - The key intuition behind them is that having the condition \( x = E \) in the quantifier allows us to focus on that specific value of \( x \), rather than considering them all.
  - Remember “Logic Example V” in Class 2 ?
  - Law \( \langle \exists\text{-1pt} \rangle \) will play a key role in the UTP theories we will explore later on.
One-point examples

- \( \forall n \bullet n = 5 \Rightarrow \text{prime}(n) \) = " \( \forall\)-1pt" = \text{prime}(5) = " substitution " = " prime number theory " = true

- \( \exists m \bullet \exists n \bullet \text{prime}(m) \land n = m + 2 \land \text{prime}(n) \) = " \( \exists\exists\)-1pt" = \( \exists\exists\)-1pt, given that \( x \not\in E \)

Simultaneous one-point

- We can do multiple "points" at once — goal:
  \( (\exists x, y \bullet x = E \land y = F \land P) \equiv P[E, F/x, y], \quad x, y \not\in E, F \)

- Proof (strategy, reduce lhs to rhs):
  \( \exists x, y \bullet x = E \land y = F \land P \) = " \( \exists\)-1pt" with \( x \), given that \( x \not\in E \)
  \( \exists y \bullet (y = F \land P)[E/x] \) = " substitution, given that \( x \not\in F \)
  \( \exists y \bullet y = F \land (P[E/x]) \) = " \( \exists\)-1pt", given that \( y \not\in F \)
  \( (P[E/x])[F/y] \) = " can substitute simultaneously, given that \( y \not\in E \)
  \( P[E, F/x, y] \)

Proof using 1-pt (many, many times …)

- Goal:
  \( (\exists t_1, t_2, x_1, x_2, y_1, y_2 \bullet \) t_1 = x \land x_1 = x \land y_1 = y \land t_2 = t_1 \land x_2 = y_1 \land y_2 = y_1 \land x' = x_2 \land y' = t_2 \) = \( \exists \text{prime}(t_1, t_2, x_1, x_2, y_1, y_2) \bullet \)

- Where did this strange example come from?
  - Remember \{ int t; t=x; x=y; y=t \}? 
  - Strategy: reduce lhs to rhs

Proof using 1-pt, cont.

- Proof:
  \( \exists t_1, t_2, x_1, x_2, y_1, y_2 \bullet \) t_1 = x \land x_1 = x \land y_1 = y \land t_2 = t_1 \land x_2 = y_1 \land y_2 = y_1 \land x' = x_2 \land y' = t_2 = " \( \exists\)-1pt", t_1, x_1, y_1 \not\in x, y, t_2, x_2, y_2, substitute "
  \( \exists t_2, x_2, y_2 \bullet \) t_2 = x \land x_2 = y \land y_2 = y \land x' = x_2 \land y' = t_2 = " \( \exists\)-1pt", t_2, x_2, y_2 \not\in x, y, substitute "
  x' = y \land y' = x
Mini-Exercise 2

Q2.1 For the following predicate, state for every variable occurrence if it is free, binding or bound:

\[(\forall a \bullet (\forall b \bullet (\exists c \bullet a \land c \lor b) \land (c \lor \neg a)) \lor b)\]

Q2.2 Perform the following (simultaneous, double) substitution:

\[(\exists t, t' \bullet t' - t = tr' - tr) \land (t \land tr, t' \land tr' / tr, tr')\]

Here \(tr\) and \(tr'\) are different variables.

Q2.3 The proof on the slide with title “A Theorem ! Proof ?” has no justifications—fill them in. Use the Proof-Section format described in class. (due in next Fri, 2pm, in class)
A Simple Imperative Language

- We now turn our attention to a simple (terse) imperative language, with assignment, sequencing, conditionals and a while-loop:

\[
p, q \in \text{Prog} \ ::= \begin{align*}
\text{skip} & \quad \text{do nothing} \\
 v := e & \quad \text{assignment} \\
p; q & \quad \text{do p, then do q} \\
p \begin{array}{c}
\text{c} \\
\text{q}
\end{array} & \quad \text{do p if c, else do q} \\
c \otimes p & \quad \text{while c do p}
\end{align*}
\]

- \text{skip} does nothing, but is useful to have around.
- Assignment uses := (“becomes”) rather than =.
- ; is an operator that joins programs together.
- For now, we ignore variable declarations.

Program Examples

- Swapping the values of \( x \) and \( y \), using \( t \) as temporary:

\[
t := x; \ x := y; \ y := t
\]

- Setting \( a \) and \( b \) to the maximum and minimum of \( a \) and \( b \) respectively:

\[
\text{skip} \triangleleft a \gtrless b \triangleright (t := a; \ a := b; \ b := t)
\]

- Setting \( f \) to factorial of \( n \), using \( x \) as temporary

\[
f := 1; \ x := n; \ (x > 1) \otimes (f := f \times x; \ x := x - 1)
\]

The Meaning of an Imperative Program

- The meaning of an expression (or predicate), is a function from an environment to a value of the appropriate type:

\[
[E] : \ Env \rightarrow \text{Value} \\
[P] : \ Env \rightarrow \mathbb{B}
\]

- For a program, the environment maps program variables to values, and is often called the program state.
- The meaning of an (deterministic ?) imperative program, is a function from an initial environment/state to a final environment/state

\[
[p] : \ Env \rightarrow \Env
\]

- Often we refer to a program as a state-transformer.
Denotational Semantics

Defining the meaning of a program as a function from program text to some mathematical structure, in a compositional way, is known as a denotational semantics, i.e., a function that returns what a program denotes.

\[
[-] : \text{Prog} \rightarrow \text{Env} \rightarrow \text{Env}
\]

\[
[\text{skip}]_\rho := \rho
\]

\[
[v := e]_\rho := \rho \oplus \{v \mapsto [e]_\rho\}
\]

\[
[p; q]_\rho := [q][p]_\rho
\]

\[
[p < c > q]_\rho := \begin{cases} [p]_\rho, & \text{if } [c]_\rho \\ [q]_\rho, & \text{if } [c]_\rho
\end{cases}
\]

\[
[c \oplus p] := \mu W \cdot \lambda \rho \cdot \text{if } [c]_\rho \text{ then } (W \circ [p])_\rho \text{ else } \rho
\]

The last definition is a mystery left for later.

In every case the meaning of a composite is given in terms of the meanings of its components (compositionality).

Program Meaning Example I

- Swapping the values of \( x \) and \( y \), using \( t \) as temporary:
  \[
  t := x; \; x := y; \; y := t
  \]

- The after-value of \( x \) is the before-value of \( y \), and vice-versa
- Using our environment notation:
  \[
  [t := x; \; x := y; \; y := t]_\rho
  = \rho \oplus \{t \mapsto \rho(x), x \mapsto \rho(y), y \mapsto \rho(x)\}
  \]

Program Meaning Example II

- Setting \( a \) and \( b \) to the maximum and minimum of \( a \) and \( b \) respectively:
  \[
  \text{skip} \triangleleft a \geq b \triangleright (t := a; \; a := b; \; b := t)
  \]

- The after-value of \( a \) is the maximum of the before-values of both \( a \) and \( b \), whilst the after value of \( b \) is the minimum.
- Using environments
  \[
  [\text{skip} \triangleleft a \geq b \triangleright (t := a; \; a := b; \; b := t)]_\rho
  = \rho \oplus \{t \mapsto \rho(a), a \mapsto \rho(b), b \mapsto \rho(a)\}, \; a < b
  = \rho, \; a \geq b
  \]

Program Meaning Example III

- Setting \( f \) to factorial of \( n \), using \( x \) as temporary
  \[
  f := 1; \; x := n; \; (x > 1) \oplus (f := f \times x; \; x := x - 1)
  \]

- If before-value of \( n \) is negative, then the after values of \( f \) and \( x \) are 1 and \( n \) respectively, otherwise:
  - The after-value of \( f \) is the factorial of the before-value of \( n \)
  - The after-value of \( n \) is the same as before
  - The after-value of \( x \) is 1, except if the before-value of \( n \) was 0, in which case so is \( x \)'s after-value.
- With environments
  \[
  [f := 1; \; x := n; \; (x > 1) \oplus (f := f \times x; \; x := x - 1)]_\rho
  = \rho \oplus \{f \mapsto 1, x \mapsto \rho(n)\}, \; \rho(n) \in \{0, 1\}
  = \rho \oplus \{f \mapsto \rho(n)!, x \mapsto 1\}, \; \rho(n) > 1
  = \rho \oplus \{f \mapsto 1, x \mapsto \rho(n)\}, \; \rho(n) < 0
  \]
Too much $\rho$!

- The use of explicit environments is precise, but verbose
  - we have to distinguish between variable $x$ and its value $\rho(x)$
  - we have a language syntax $x := e$ that is different to that used to express its meaning: $\rho \oplus \{x \mapsto [e]_\rho\}$.
- To reduce clutter, we adopt some key ideas
  - Programs are predicates
  - We denote the before-value of program variable $x$ by predicate variable $x$.
  - We denote the after-value of program variable $x$ by predicate variable $x'$.
  - Programs are now predicates that relate before and after values.

Programs are Predicates

- Eric Hehner pushed this idea, inspiring Tony Hoare to write “Programs are predicates”, *Mathematical logic and programming languages*, Royal Society of London, pp141–155, 1984.
- Many approaches to programming language semantics keep the programming language and the semantics language separate
  
  $$\text{Meaning}[\text{program}] = \text{mathematical stuff}$$

  What’s inside the brackets ($[[]]$) is syntax, outside is semantics
- This distinction is considered by many today as artificial and unnecessary.

Extending our Predicate Language

- We are going to extend our predicate language to include our program language:

  $$P, Q \in \text{Pred} ::= \ldots$$
  $$| \quad \text{skip}$$
  $$| \quad v := E$$
  $$| \quad P; Q$$
  $$| \quad P \triangleleft c \triangleright Q$$
  $$| \quad c \odot P$$

  $c$ denotes a condition (boolean expression)
- In effect, we shall define our programming constructs as shorthand for equivalent predicates

skip is a predicate

- Program skip leaves that state unchanged
  
  $$\text{skip} \models x' = x \land y' = y \land z' = z$$

  (here assuming that our only variables are $x$, $y$ and $z$!)
- We have a little problem here — we need to know which variables are in scope.
Alphabets

We associate the set of variables in scope with every predicate $P$:

- this set is called the alphabet of the predicate ($\alpha P$).
- if variable $x$ is in scope, the alphabet contains $x$ and $x'$
- all variables mentioned in $P$ must be in $\alpha P$.
- we denote all the un-dashed alphabet variables by $in\alpha P$
- we denote all the dashed alphabet variables by $out\alpha P$

Alphabet example

> consider the assignment

$$x := e$$

in a program where $x$, $y$ and $z$ are in scope.

> $e$ can only mention these three variables

> We have

\[
\begin{align*}
\alpha(x := e) &= \{x, x', y, y', z, z'\} \\
\text{in}\alpha(x := e) &= \{x, y, z\} \\
\text{out}\alpha(x := e) &= \{x', y', z'\}
\end{align*}
\]

Alphabet labelling

We can label language constructs with their alphabet ($A$, say)

- $skip_A$ and $x :=_A e$
- We can infer alphabets for other language constructs
- We often omit alphabets where obvious from context
- We are talking about using “alphabetised predicates” to give meanings to programs

The “rest”

> Frequently we single out a few variables (zero or more) and just want to refer to “the rest”.

> We shall use meta-variable $S$ to refer to the set of all program (script) variable before values, and $S'$ for after-values.

> $S' = S$ means $x'_1 = x_1 \land \ldots \land x'_n = x_n$ where $x_1 \ldots x_n$ are the variables in $S$.

> If we want to talk about all script variables except $x$ and $y$ (say) we write $S \setminus x, y$

> For now, $S$ and $S'$ correspond to $in\alpha$ and $out\alpha$, respectively.
skip as an alphabetised predicate

- Given \( A = \{ S, S' \} \):
  \[
  \langle \text{skip-def} \rangle \quad \text{skip}_A \triangleq S' = S
  \]
- All the variables are unchanged
- Example, if \( A = \{ i, j, k, i', j', k' \} \), then
  \[
  \text{skip}_A = (i' = i \land j' = j \land k' = k)
  \]

; as a binary predicate operator

- \( P; Q \) is the **sequential composition** of \( P \) with \( Q \).
- How is this expressed using our predicate notation?
- We phrase its behaviour as follows:
  “\( P; Q \) maps before-state \( S \) to after-state \( S' \), when there exists a mid-state \( S_m \), such that \( P \) maps \( S \) to \( S_m \), and \( Q \) maps \( S_m \) to \( S' \).”
- We get the following definition:
  \[
  \langle ; - \text{def} \rangle \quad P; Q \triangleq (\exists S_m \cdot P[S_m/S] \land Q[S_m/S])
  \]
- The alphabets involved must satisfy the following conditions:
  \[
  \begin{align*}
  \text{out}_{\alpha} P &= (\text{in}_\alpha Q)', \quad \{ x, \ldots, z \}' = \{ x', \ldots, z' \} \\
  \text{in}_\alpha(P; Q) &= \text{in}_\alpha P \\
  \text{out}_\alpha(P; Q) &= \text{out}_\alpha Q
  \end{align*}
  \]

Sequential Composition Example

- Consider program \( t := x; x := y \), with \( t, x \) and \( y \) in scope.
- We calculate:
  \[
  \begin{align*}
  t &:= x; x := y \\
  &= \quad \langle ; - \text{def} \rangle \\
  t' &= x \land x' = x \land y' = y; x := y \\
  &= \quad \langle ; - \text{def} \rangle \\
  t' &= x \land x' = x \land y' = y; t' = t \land x' = y \land y' = y \\
  &= \quad \exists \text{tm, x}_m, y_m \cdot \\
  &\quad \left( t' = t \land x' = y \land y' = y \right) \left( \text{tm, x}_m, y_m, t', x', y' \right) \\
  &= \quad \exists \text{tm, x}_m, y_m \cdot \\
  &\quad \left( t' = t \land x' = y \land y' = y \right) \left( \text{tm, x}_m, y_m, t, x, y \right)
  \end{align*}
  \]

\( x := e \) as an alphabetised predicate

- Given \( A = \{ S, S' \} \):
  \[
  \langle ; - \text{def} \rangle \quad x := A e \triangleq x' = e \land S'_{x'} = S_{x}
  \]
- the after-value of \( x \) takes on the value of \( e \), evaluated in the before-state.
- Example, if \( A = \{ i, j, k \} \), then
  \[
  j := A i + k = (i' = i \land j' = i + k \land k' = k)
  \]
Sequential Composition Example, cont.

- We continue:

\[ \exists t_m, x_m, y_m \cdot t_m = x \land x_m = x \land y_m = y \land t' = t_m \land x' = y_m \land y' = y_m \]

\[ t' = x \land x' = y \land y' = y \]

- We see that the net effect is like the simultaneous assignment \( t, x : = x, y \),
  \( t \) now has the value of \( x \), and \( x \) has the value of \( y \).

\[ \langle\langle \exists-1pt \rangle\rangle, t_m, x_m, y_m \not\in x, y \]

\[ t' = x \land x' = y \land y' = y \]

as a ternary predicate operator

- \( P \triangleleft c \triangledown Q \), behaves like \( P \), if \( c \) is true, otherwise it behaves like \( Q \).
- Definition:

\[ \langle\langle \triangleleft\triangledown\rangle\rangle \quad P \triangleleft c \triangledown Q \triangleleft \quad c \land P \lor \neg c \land Q \]

- Alphabet constraints

\[ \text{in}_\alpha(P \triangleleft c \triangledown Q) = \alpha c = \text{in}_\alpha P = \text{in}_\alpha Q \]

\[ \text{out}_\alpha(P \triangleleft c \triangledown Q) = \text{out}_\alpha P = \text{out}_\alpha Q \]

Example

- Program: \( \text{skip} \triangleleft a \geq b \triangledown (t : = a; \quad a : = b; \quad b : = t) \), with alphabet \( A = \{a, a', b, b', t, t'\} \)
- Meaning:

\[ \text{skip}_A \triangleleft a \geq b \triangledown t : = a; \quad a : = b; \quad b : = t \]

\[ a \geq b \land \text{skip}_A \]

\[ \lor \]

\[ \neg (a \geq b) \land (t : = a; \quad a : = b; \quad b : = t) \]

\[ a \geq b \land a' = a \land b' = b \land t' = t \]

\[ \lor \]

\[ a < b \land a' = b \land b' = a \land t' = a \]

as a binary predicate operator

- Program \( c \circ P \) checks \( c \), and if true, executes \( P \), and then repeats the whole process
- We won’t define it yet, instead we give a law that describes how a while-loop can be “unrolled” once:

\[ \langle\langle \circ\rangle\rangle\text{-unroll} \quad c \circ P = (P; \quad c \circ P) \triangleleft c \triangledown \text{skip} \]

If \( c \) is \textit{False}, we \textit{skip}, otherwise we do \( P \) followed by the whole loop \( (c \circ P) \) once more.

- Alphabet constraints

\[ \text{in}_\alpha(c \circ P) = \alpha c = \text{in}_\alpha P \]

\[ \text{out}_\alpha(c \circ P) = \text{out}_\alpha P = (\text{in}_\alpha P)' \]
Loop example

\[
f := 1; x := n; (x > 1) \circ (f := f \ast x; x := x - 1)
\]

\[
= \quad \text{":=def"}
\]

\[
n' = n \land f' = 1 \land x' = x
\]

\[
x := n; (x > 1) \circ (f := f \ast x; x := x - 1)
\]

There must be a better way !!
all the other variables …

- Consider computing the semantics of

\[ x := x + y; \ y := x - y; \ x := x - y \]

in a scope with variables \( s, t, u, v, w, x, y, z \).

- The program only mentions \( x \) and \( y \), but we have to carry equalities around for \( s, t, u, w \) and \( z \).

- We can tidy-up using the 1-pt law, but this is tedious and error prone.

- Can we show that the above program swaps the values of \( x \) and \( y \), without mentioning the other variables?

- What we need are laws that work at the programming language level
   (a.k.a. “Laws of Programming”)

(0me) Laws of Programming

- Here \( P, Q \) and \( R \) stand for arbitrary programs, \( A \) for an arbitrary alphabet, \( x, y \) and \( z \) for arbitrary variables, and \( c, e \) and \( f \) for arbitrary expressions

\[
\begin{align*}
\langle\langle \text{skip-alt} \rangle\rangle & \quad \text{skip}_A = \ x :=_A x, \quad x \in A \\
\langle\langle \text{skip-unit} \rangle\rangle & \quad \text{skip}; \ P = P \\
\langle\langle \text{-skip-unit} \rangle\rangle & \quad P; \ \text{skip} = P \\
\langle\langle \text{-assoc} \rangle\rangle & \quad P; (Q; R) = (P; Q); R \\
\langle\langle \text{-seq} \rangle\rangle & \quad x := e; \ x := f = x := f[e/x] \\
\langle\langle \text{-swap} \rangle\rangle & \quad x := e; \ y := f = y := f[e/x]; \ x := e, \ y \notin e \\
\langle\langle \text{-true} \rangle\rangle & \quad P \triangleright True \triangleright Q = P \\
\langle\langle \text{-false} \rangle\rangle & \quad P \triangleright False \triangleright Q = Q \\
\langle\langle \text{-seq} \rangle\rangle & \quad (P \triangleright c \triangleright Q); \ R = (P; R) \triangleright c \triangleright (Q; R)
\end{align*}
\]
Proof of \langle\langle \text{skip-}\text{alt} \rangle\rangle

- Goal: \( \text{skip}_A = x :=_A x \)
- Strategy: reduce rhs to lhs
- Proof:

\[
\begin{align*}
x :=_A x &= \text{\langle\langle \text{-def} \rangle\rangle}, \ A = \{S, S'\} \\
x' &= x \land S' \setminus x' = S \setminus x \\
   &= \text{\langle\langle \text{merge} \rangle\rangle} \\
S' &= S \\
   &= \text{\langle\langle \text{skip-}\text{-def} \rangle\rangle}, \ A = \{S, S'\} \\
   \square
\end{align*}
\]

Proof of \langle\langle \text{skip-};\text{-unit} \rangle\rangle

- Goal: \( \text{skip}; P = P \)
- Strategy: reduce lhs to rhs
- Proof:

\[
\begin{align*}
\text{skip}; P &= \text{\langle\langle \text{-def} \rangle\rangle} \\
S &= \{S, S'\} \\
\exists S_m \bullet S_m = S \land P[S_m/S] &= \text{\langle\langle \exists-1\text{pt} \rangle\rangle} \\
(\exists S_m \bullet S_m)[S/S_m] &= \text{\langle\langle \text{substitution \ 'inverse'} \rangle\rangle}
\end{align*}
\]

Laws of Substitution

- We have used some “laws of substitution” in our proofs.
- These too have a rigorous basis.
- First, we promote substitution to be part of our predicate language:

\[
\begin{align*}
\text{Pred} &\ ::= \ldots \mid P[e_1, \ldots, e_n/v_1, \ldots, v_n]
\end{align*}
\]

We assume that substitution has highest precedence (binds tighter) than all other constructs
- Next we define its effect on predicates.

Defining Single Substitution

\[
\begin{align*}
k[e/x] &\triangleq k \\
v[e/x] &\triangleq v, \ v \neq x \\
x[e/x] &\triangleq e \\
(e_1 + e_2)[e/x] &\triangleq e_1[e/x] + e_2[e/x] \\
(\neg P)[e/x] &\triangleq \neg P[e/x] \\
(P \land Q)[e/x] &\triangleq P[e/x] \land Q[e/x] \\
(\forall v \bullet P)[e/x] &\triangleq \forall v \bullet P[e/x], \ v \neq x, \ v \notin e \\
(\forall v \bullet P)[e/x] &\triangleq \forall w \bullet (P[w/v])[e/x], \ v \neq x, \ v \notin e, w \notin P, e, x
\end{align*}
\]

The constructs not mentioned above follow the same pattern. We shall now look at the last four lines in more detail.
Substitution and Quantifiers (I)

$$(\forall x \bullet P)[e/x] \equiv \forall x \bullet P$$

The simplest case: $x$ is simply not free in $\forall x \bullet P$, so nothing changes.

Substitution and Quantifiers (II)

$$(\forall \nu \bullet P)[e/x] \equiv \forall \nu \bullet P[e/x], \quad \nu \neq x, \nu \notin e$$

- We are not substituting for $\nu$, and $\nu$ does occur in $e$, so there is no possibility of name capture.
- We simply recurse to the body predicate $P$.

Substitution and Quantifiers (III)

$$(\forall \nu \bullet P)[e/x] \equiv \forall w \bullet (P[w/\nu])[e/x], \quad \nu \neq x, \nu \in e, w \notin P, e, x$$

- The tricky case, where $\nu$ occurs in $e$.
- To avoid name capture of $\nu$ in $e$:
  - Pick a fresh variable $w$ — i.e. one not currently in use.
  - $\alpha$-rename the bound and binding occurrences of $\nu$ to $w$
  - Then recurse into the now $\alpha$-renamed body $P$.

Laws of single substitution

- We can identify a number of useful laws
  - $P[x/y][y/x] = P, x \notin P$
  - $P[e/x][f/x] = P[e[f/x]/x]$
  - $P[e/x][f/y] = P[f/y][e/x], x \notin f, y \notin e$

- We shall not prove these laws at this point
  - to do so requires induction,
  - over the grammar (?)! of our predicate language
**Substitution Law examples**

- ⟨⟨subst-inv⟩⟩: \( P[x/y][y/x] = P, \ x \notin P \)
  - \((x+y)[z/x][x/z] = (z+y)[x/z] = x + y\)
  - The side-condition matters:
  \( (x+y)[y/x][x/y] = (y+y)[x/y] = x + x \)
  \( (x+y)[x/y][x/y] = (x+x)[y/x] = y + y \)

- ⟨⟨subst-comp⟩⟩: \( P[e/x][f/x] = P[e[f/x]/x] \)
  - \( x[x + y/x][z/x] = (x + y)[z/x] = z + y \)
  - \( x[(x+y)[z/x]/x] = x[z+y/x] = z + y \)

**Simultaneous Substitution**

- Simultaneous substitution does several replacements at once:
  \( (x+y)[y^2/k+x/x, y] = y^2 + k + x \)

- In general, it is not the same as doing each replacement one at a time:
  \( (x+y)[y^2/x][k+x/y] = (y^2+y)[k+x/y] = (k+x)^2+k+x \)

- When the first substitution does not introduce the “target” of the second, then we can do them one at a time.

**Laws of Simultaneous Substitution**

- Order of substitutions does not matter:
  ⟨⟨subst-comm⟩⟩ \( P[e,f/x,y] = P[f,e/y,x] \)

- We can merge in later substitutions if they don’t act on earlier replacements:
  ⟨⟨subst-seq⟩⟩ \( P[e/x][f/y] = P[e,f/x,y], \ y \notin e \)

- The above law generalises to many substitutions:
  \( P[e_1, \ldots, e_n/x_1, \ldots, x_n][f/y] = P[e_1, \ldots, e_n, f/x_1, \ldots, x_n, y] \)
  (provided \( y \notin e_1, \ldots, e_n \))

**Proof of ⟨⟨; -assoc⟩⟩**

- Goal: \( P; (Q; R) = (P; Q); R \)
- Strategy: reduce both lhs and rhs to same predicate
- Proof: hold on tight!
Proof \( \llcorner; \text{-assoc} \) (lhs)

\[
P: (Q : R) \quad = \quad \langle\quad \text{def} \quad \rangle, \quad \{S, S'\} = \text{out}_Q \cup \text{in}_R \quad \langle\quad \text{def} \quad \rangle
\]

\[
P : (\exists S_m \bullet Q[S_m/S'] \land R[S_m/S])
\]

\[
\exists S_n \bullet P[S_n/S'] \land \exists S_m \bullet Q[S_m/S'] \land R[S_m/S] [S_n/S]
\]

\[
\text{defn. of substitution, } S \neq S_m
\]

\[
\exists S_n, S_m \bullet P[S_n/S'] \land Q[S_m/S'] [S_n/S] \land R[S_m/S] [S_n/S]
\]

\[
\text{\langle\quad \text{def} \quad \rangle, \quad S_m \notin P[S_n/S'] \langle\quad \text{def} \quad \rangle
\]

\[
\exists S_n, S_m \bullet P[S_n/S'] \land Q[S_m/S'] [S_n/S] \land R[S_m/S] [S_n/S]
\]

\[
\exists S_n, S_m \bullet P[S_n/S'] \land Q[S_m, S_n, S', S] \land R[S_m/S'] [S_n/S]
\]

\[
\exists S_n, S_m \bullet P[S_n/S'] \land Q[S_m, S_n, S', S] \land R[S_m/S] [S_n/S]
\]

\[
\exists S_m, S_n \bullet P[S_n/S'] [S_m/S'] \land Q[S_n/S] [S_m/S'] \land R[S_m/S]
\]

\[
\exists S_m, S_n \bullet P[S_n/S'] [S_m/S'] \land Q[S_n, S_m, S, S'] \land R[S_m/S]
\]

\[
\text{lhs=rhs, with minor re-orderings}
\]

Proof \( \llcorner; \text{-assoc} \) (rhs)

\[
(P : Q) : R
\]

\[
\langle\quad \text{def} \quad \rangle, \quad \{S, S'\} = \text{out}_Q \cup \text{in}_R \langle\quad \text{def} \quad \rangle
\]

\[
\langle\quad \text{def} \quad \rangle, \quad \{S, S'\} = \text{out}_Q \cup \text{in}_R
\]

\[
\exists S_m \bullet P[S_m/S'] \land Q[S_m/S] : R
\]

\[
\exists S_m \bullet P[S_m/S'] \land Q[S_m/S] \land R[S_m/S]
\]

\[
\exists S_m, S_n \bullet P[S_n/S'][S_m/S'] \land Q[S_n/S] [S_m/S'] \land R[S_m/S]
\]

\[
\exists S_m, S_n \bullet P[S_n/S'][S_m/S'] \land Q[S_n, S_m, S, S'] \land R[S_m/S]
\]

\[
\text{lhs=rhs, with minor re-orderings}
\]

Proof using Laws (example)

- Goal:

\[
(f := 1; x := n; f := f \cdot x; x := x - 1)
\]

= \[
(f := n; x := n - 1)
\]

- Strategy: reduce lhs to rhs

- Proof:

\[
f := 1; x := n; f := f \cdot x; x := x - 1
\]

= \[
" \llcorner;\text{-swap} \quad \text{\langle\quad \text{def} \quad \rangle, \quad f \neq n \langle\quad \text{def} \quad \rangle
\]

\[
f := 1; f := f \cdot n; x := n; x := x - 1
\]

= \[
" \llcorner;\text{-seq} \quad \langle\quad \text{def} \quad \rangle
\]

\[
f := 1; n; x := n - 1
\]

= \[
" \text{arithmetic} \quad \langle\quad \text{def} \quad \rangle
\]

\[
f := n; x := n - 1
\]

Subtleties with Substitution (I)

Consider the following proof of \( \llcorner;\text{-seq} \) reducing LHS to RHS:

\[
x := e; x := f
\]

= \[
" \llcorner;\text{-def} \quad \text{\langle\quad \text{def} \quad \rangle, \quad A = \{S, S'\} \langle\quad \text{def} \quad \rangle
\]

\[
x' = e \land S'_x = S_x \quad ; \quad x' = f \land S'_x = S_x
\]

= \[
" \llcorner;\text{-def} \quad \text{\langle\quad \text{def} \quad \rangle, \quad A = \{S, S'\} \langle\quad \text{def} \quad \rangle
\]

\[
\exists S_m \bullet (x' = e \land S'_{x'} = S_x) [S_m/S]
\]

\[
\land (x' = f \land S'_{x'} = S_x) [S_m/x, S]
\]

= \[
" \text{substitution, noting that } e \text{ has no dashed vars} \quad \langle\quad \text{def} \quad \rangle
\]

\[
\exists S_m \bullet x_m = e \land S_m/x_m = S_{x_m}
\]

\[
\land x' = f[S_m/S] \land S'_{x'} = S_{x_m}
\]

= \[
" \llcorner;\text{-1pt} \quad \text{\langle\quad \text{def} \quad \rangle}
\]

\[
x' = f[e, S \setminus x / x, S_x] \land S'_{x'} = S_x
\]

= \[
" \text{ignore } [S \setminus x / S_x] \quad \langle\quad \text{def} \quad \rangle
\]

\[
x := f[e/x]
\]

\[
\]
We might have been tempted to apply \(\langle\langle\cdot\rangle\rangle\) first, “do” the substitution, and then use \(\langle\langle=\rangle\rangle\):

\[
\begin{align*}
x &:= e; \ x := f \\
\exists S_m \cdot (x := e)[S_m/S'] \\
& \quad \land (x := f)[S_m/S]
\end{align*}
\]

“\(\langle\langle=\rangle\rangle\), twice”

\[
\begin{align*}
\exists S_m \cdot x' &:= e \\
& \quad \land x'_m := f[S_m/S]
\end{align*}
\]

What is \(x'_m\)? We seem to have a problem!

Sequential composition is designed to work with predicates using \(x, S\) for before-variables, and \(x'\) and \(S'\) for after variables.

In \(x := e\), the variable \(x\) stands for the program variable, and not its initial value.

We cannot do substitutions safely until we have expanded its definition.

In fact, we cannot determine what its free variables are until it has been expanded.

Some of our new predicate (programming) constructs are non-substitutable (n.s.).

Substitution can only be applied to these once their definitions have been expanded.

We have assignment as one example, but there are others:

\[
\text{skip} \quad P, \ Q \quad c * P
\]

Of the new constructs so far, only conditional is substitutable:

\[
(P \triangleleft c \triangleright Q)[e/x] = P[e/x] \triangleleft c[e/x] \triangleright Q[e/x]
\]
Simultaneous Assignment

- We introduce a further extension to predicate syntax, simultaneous assignment:

  \[ \text{Pred} ::= \ldots \]

  \[ | \quad x_1, \ldots, x_n := e_1, \ldots, e_n \]

- We introduce shorthands: \( \vec{x} \) and \( \vec{e} \) for \( x_1, \ldots, x_n \) and \( e_1, \ldots, e_n \) resp.

- Its meaning is that the expressions \( e_1 \) through \( e_n \) are evaluated, and then all the \( x_i \) are updated simultaneously:

  \[ \langle \text{sim-}:=\text{-def} \rangle \quad \vec{x} :=_A \vec{e} \quad \overset{\text{def}}{=} \quad \begin{cases} x'_1 = e_1 & \wedge \ldots & x'_n = e_n & \wedge \ S \setminus \vec{x}' = S \setminus \vec{x} \end{cases}, \quad A = \{S, S'\}, \quad x_1, \ldots, x_n \in S \]

Swapping “trick” revisited

- Re-consider computing the semantics of

  \[ x := x + y ; \quad y := x - y ; \quad x := x - y \]

  in a scope with variables \( s, t, u, v, w, x, y, z \)

- We’d like to use the laws of programming, so we can ignore \( s, t, u, v, w \) and \( z \).

- We can’t use \( \langle :=\text{-}swap \rangle \), because the side-condition does not hold.

- We can’t use \( \langle :=\text{-}seq \rangle \), because we don’t have two assignments to the same variable one after the other.

- In fact it’s not clear what laws would work: after the first two assignments we have

  \( x' = x + y \wedge y' = x \wedge S \setminus \vec{x}, \vec{y}' = S \setminus \vec{x}, \vec{y} \) whilst at the end we get

  \( x' = y \wedge y' = x \wedge S \setminus \vec{x}, \vec{y}' = S \setminus \vec{x}, \vec{y} \).

Laws of Simultaneous Assignment (I)

- A single assignment to \( y \) can always be merged with a preceding simultaneous assignment to \( \vec{x} \), provided \( y \notin \vec{x} \):

  \[ \langle \text{sim-}:=\text{-merge} \rangle \quad \vec{x} :=\vec{e} ; \quad y := f \quad = \quad \vec{x}, y := (\vec{e}, f[\vec{e}/\vec{x}]) \]

- Proof, reducing lhs to rhs:

  \[ \begin{align*}
  \vec{x} :=\vec{e} ; \quad y := f \\
  &\quad \langle \text{sim-}:=\text{-def} \rangle, \langle :=\text{-def} \rangle \quad \\
  \vec{x}' &= \vec{e} \wedge S \setminus \vec{x}' = S \setminus \vec{x} \quad \wedge \ y' = f[S_m/S] \wedge S \setminus \vec{y}' = S_m \setminus \vec{y} \\
  &\quad \langle :=\text{-def} \rangle, \text{ and substitute } \\
  \exists S_m \bullet \vec{x}_m = \vec{e} \wedge S_m \setminus \vec{x}_m = S \setminus \vec{x} \\
  \wedge \ y' &= f[S_m/S] \wedge S \setminus \vec{y}' = S_m \setminus \vec{y} \\
  &\quad \langle \exists\text{-1pt} \rangle, \text{ noting } y_m = y \\
  \vec{x}' &= \vec{e} \wedge \ y' = f[\vec{e}, S \setminus \vec{e} / \vec{y}, S \setminus \vec{x}] \wedge S \setminus \vec{x}', \vec{y}' = S \setminus \vec{x}, \vec{y} \\
  &\quad \langle \text{sim-}:=\text{-def} \rangle, \text{ ignoring } [y, S \setminus \vec{y} / y, S \setminus \vec{y}] \\
  \vec{x}, y &= \vec{e}, f[\vec{e}/\vec{x}] 
  \end{align*} \]
Laws of Simultaneous Assignment (II)

- If \( y \in \vec{x} \), the sequencing law has to be slightly different
  \[
  \langle \text{sim}::=\text{seq} \rangle \quad \vec{x}, y := \vec{e}, f ; y := g = \vec{x}, y := \vec{e}, g[\vec{e}, f / \vec{x}, y] 
  \]

- Proof, reducing LHS to RHS:
  \[
  \vec{x}, y := \vec{e}, f ; y := g = \langle \text{sim}::=\text{def} \rangle, \langle \text{def} \rangle \\
  \vec{x}' = \vec{e} \land y' = f \land S'_{\vec{x}, y} = S_{\vec{x}, y} : y' = g \land S'_{\vec{x}, y} = S_{\vec{x}, y}
  \]
  \[
  \exists S_m \bullet \vec{x}_m = \vec{e} \land y_m = f \land S_m[\vec{x}_m, y_m] = S_{\vec{x}, y} \land y' = g[y_m, S_m[\vec{x}_m] / y, S_{\vec{x}, y}] \land S'_{\vec{x}, y} = S_{\vec{x}, y}
  \]
  \[
  \vec{x}' = \vec{e} \land y' = g[\vec{e}, f, S'_{\vec{x}, y} / \vec{x}, y, S'_{\vec{x}, y}] \land S' = S
  \]
  \[
  \vec{x}, y := (\vec{e}, g)[\vec{e}, f / \vec{x}, y] 
  \]

Messing with notation

- Is simultaneous assignment a programming language construct?
- Depends on the language:
  - in languages like C, Java, it is not allowed
  - in Handel-C it is allowed, as it targets hardware and so we have real parallelism
- It does not matter!
  - It is a predicate with a sensible meaning
  - It is convenient for certain proofs
  - It can describe outcomes concisely that are not possible using only single assignments, e.g. \( x, y := y, x \).
- Not all predicate language extensions have to be “code”.

Swapping Trick Proof

- Goal:
  \[
  x := x + y ; y := x - y ; x := x - y = x, y := y, x 
  \]
- Proof, reducing LHS to RHS:
  \[
  x := x + y ; y := x - y ; x := x - y = \langle \text{sim}::=\text{merge} \rangle \\
  x, y := x + y ; (x - y)[x + y / x] ; x := x - y
  \]
  \[
  \text{do substitution} \\
  x, y := x + y ; (x - y)[x + y - y / x, y, (x + y) - y
  \]
  \[
  \text{do substitution} \\
  x, y := x + y - (x + y) - y, (x + y) - y
  \]
  \[
  \text{arithmetic} \\
  x, y := y, x 
  \]

Programs as Predicates

- If programs are predicates, then we can join them up using predicate notation.
  - e.g \( \text{prog}_1 \wedge \text{prog}_2 \)
  - e.g \( \text{prog}_1 \lor \text{prog}_2 \)
  - e.g \( \text{prog}_1 \Rightarrow \text{prog}_2 \)
- We can also mix them with non-program predicates
  - e.g \( \text{prog} \wedge \text{pred} \)
  - e.g \( \text{prog} \lor \text{pred} \)
  - e.g \( \text{prog} \Rightarrow \text{pred} \)
- Do these make sense? If so, how?
- Are any of these useful?
Consider the following:

\[(x := 2) \land (x := 3)\]
\[(x := 2) \land (y := 3)\]
\[(x := 2) \land x = 2\]
\[(x := 2) \land y = 3\]

What behaviour do these describe?

Examining \[(x := 2) \land (x := 3)\]

\[(x := 2) \land (x := 3)\]

\[= \text{"}\langle\langle \:\equiv\text{-def}\rangle\rangle\text{, twice}\] \]
\[x' = 2 \land S_{\backslash x}^\prime = S_{\backslash x} \land x' = 3 \land S_{\backslash x}^\prime = S_{\backslash x}\]

\[= \text{"}2 \neq 3\] \false

Not unexpectedly, we get \false
we cannot assign 2 and 3 to \(x\) (at the same time)

Examining \[(x := 2) \land (y := 3)\]

\[(x := 2) \land (y := 3)\]

\[= \text{"}\langle\langle \:\equiv\text{-def}\rangle\rangle\text{, twice}\] \]
\[x' = 2 \land S_{\backslash y}^\prime = S_{\backslash y} \land y' = 3 \land S_{\backslash y}^\prime = S_{\backslash y}\]

\[= \text{"}2 \neq 3\] \false

We get a predicate stating that the assignment occurred, in a starting state where \(x\) had value 2.

It is the same as \Skip \land x = 2

true only if \(x\) and \(y\) start as 2 and 3 resp., and do not change.
Examining \((x := 2) \land y = 3\)

- \((x := 2) \land y = 3 = "\llcorner := \text{def} \lrcorner\ "\)  
  \[x' = 2 \land S\setminus_{x'} = S\setminus_x \land y = 3\]

- We get a predicate stating that the assignment occurred, in a starting state where \(y\) had value 3.

Programs and Conjunction—Comment

- Conjoining two programs \((\text{prog}_1 \land \text{prog}_2)\) easily leads to contradiction
- Predicate \(\text{prog} \land \bar{x} = \bar{e}\) describes a run of \(\text{prog}\) that started in a state where variables \(\bar{x}\) had values \(\bar{e}\).
- Remember, \(x := e\) means \(x\) is changed, and that all other variables are left unchanged.

Examining \((x := 2) \lor (x := 3)\)

- Consider the following:
  \[\begin{align*}
  (x := 2) \lor (x := 3) \\
  (x := 2) \lor (y := 3) \\
  (x := 2) \lor x = 2
  \end{align*}\]

- \((x := 2) \lor (x := 3) = "\llcorner := \text{def} \lrcorner, \text{twice} \ "\)  
  \[x' = 2 \lor x' = 3 \land S\setminus_{x'} = S\setminus_x \land x = 2 \lor y = 3 = "\llcorner \land \lor \text{distr} \lrcorner \ "\)  
  \[x' = 2 \lor x' = 3 \land S\setminus_{x'} = S\setminus_x\]

- Variable \(x\) ends up having either value 2 or 3, and all other variables are unchanged.
- The choice between 2 or 3 is arbitrary — nothing here states how that choice is made.
Examining \((x := 2) \lor (y := 3)\)

\[
\begin{align*}
(x := 2) \lor (y := 3) & = \text{"\(\langle\langle :=\text{-def}\rangle\), twice"}\} \\
n & = (x') = 2 \land y' = y \land S' \setminus x', y' = S \setminus x, y \\
& \lor x' = x' = x \land y' = 3 \land S' \setminus x', y' = S \setminus x, y \\
& = \text{"\(\langle\langle \land\land\text{-distr}\rangle\rangle\), twice"}\} \\
(x' = 2 \land y' = y \lor x' = x \land y' = 3) \land S' \setminus x', y' = S \setminus x, y
\end{align*}
\]

- Either \(x\) ends up having value 2, or \(y\) ends up equal to 3, and all other variables are unchanged.
- The choice between changing \(x\) or \(y\) is arbitrary — nothing here states how that choice is made.

Programs and Disjunction—Comment

- Disjoining two programs \((\text{prog}_1 \lor \text{prog}_2)\) denotes an arbitrary choice between the two behaviours.
- Predicate 

\[
\text{\text{prog}} \lor \vec{x} = \vec{e}
\]
tells us that we get:

- the behaviour of \text{prog}, if initial values of \(\vec{x}\) are \textit{not all} equal to \(\vec{e}\);
- arbitrary behaviour, if they are all equal.

Programs and Negation

- How do we “not” assign something?

\[
\neg (x := e)
\]

- Let’s calculate:

\[
\begin{align*}
\neg (x := e) & = \text{"\(\langle\langle :=\text{-def}\rangle\rangle\), twice"}\} \\
& = (x' = e) \land S' \setminus x' = S \setminus x \\
& \lor \neg (x' = e) \land S' \setminus x' = S' \setminus x \\
& = \text{"\(\langle\langle \land\land\text{-distr}\rangle\rangle\rangle\), twice"}\} \\
& = \text{"\(\langle\langle \land\land\text{-distr}\rangle\rangle\rangle\), twice"}\} \\
(x' \neq e) \lor S' \setminus x' \neq S \setminus x
\end{align*}
\]

- So, \(\neg (x := e)\) is any situation where

- either \(x\) does not end up with value \(e\)
- or some other variable gets changed
Programs and Implication

Consider the following:

\[(x := 2) \Rightarrow (x := 3)\]

\[(x := 2) \Rightarrow (y := 3)\]

\[(x := 2) \Rightarrow (x := 3)\]

\[(x := 2) \Rightarrow x' = 2\]

\[(x := 2) \Rightarrow x' \in \{1, \ldots, 10\}\]

Complicated!

Examining \((x := 2) \Rightarrow (y := 3)\)

\[(x := 2) \Rightarrow (y := 3)\]

\[\langle\langle (x := 2) \Rightarrow (y := 3) \rangle\rangle, \text{ twice }\]

\[x' = 2 \land S_{x', y'} \land S_{x, y} \land S_{y, x} \]

\[\Rightarrow x' = x \land y' = 3 \land S_{x', y'} \land S_{x, y} \land S_{y, x} \]

\[\langle\langle \Rightarrow \rangle\rangle, \text{ given that } S' \neq S\]

\[x' \neq 2 \lor S' \neq S \lor x' = 3 \land S_{x, y} \land S_{y, x} \]

\[\langle\langle \Rightarrow \rangle\rangle, \text{ with } p \text{ matched to } S' \neq S\]

\[x' \neq 2 \lor S' \neq S \lor x' = 3\]

This can only be true if the assignment \(x := 2\) does not happen.

Examining \((x := 2) \Rightarrow (x := 3)\)

\[(x := 2) \Rightarrow (x := 3)\]

\[\langle\langle (x := 2) \Rightarrow (x := 3) \rangle\rangle, \text{ twice }\]

\[x' = 2 \land S_{x', x} \land S_{x, x} \Rightarrow x' = 3 \land S_{x', x} \land S_{x, x} \]

\[\langle\langle \Rightarrow \rangle\rangle, \text{ given that } x' = 3 \Rightarrow x' \neq 2\]

\[x' \neq 2 \lor S' \neq S\]

This is always true: if we assign 2 to \(x\), then the final value of \(x\) is 2.

Examining \((x := 2) \Rightarrow x' = 2\)

\[(x := 2) \Rightarrow x' = 2\]

\[\langle\langle (x := 2) \Rightarrow x' = 2 \rangle\rangle, \text{ twice }\]

\[x' = 2 \land S_{x, x} \land S_{x, x} \Rightarrow x' = 2\]

\[\langle\langle \Rightarrow \rangle\rangle, \text{ given that } x' = 2 \Rightarrow x' \neq 2\]

\[x' \neq 2 \lor S' \neq S \lor x' = 3\]

\[\langle\langle \Rightarrow \rangle\rangle, \text{ with } p \text{ matched to } S' \neq S\]

\[x' \neq 2 \lor S' \neq S \lor x' = 3\]

\[\text{true} \lor S_{x', x} \neq S_{x, x} \]

\[\text{true}\]
Examining \((x := 2) \Rightarrow x' \in \{1, \ldots, 10\}\)

\[
(x := 2) \Rightarrow x' \in \{1, \ldots, 10\} \equiv
\begin{align*}
\langle \leftarrow-def \rangle & \quad \langle \leftarrow-def, \langle \text{deMorgan} \rangle \rangle \\
\overline{x'} \neq 2 \lor S' \setminus x' \neq S' \setminus x \lor x' \in \{1, \ldots, 10\} & \quad \langle \text{excluded-middle} \rangle \\
\text{true} & \quad \langle \lor-zero \rangle \\
\text{true}
\end{align*}
\]

This is always true: if we assign 2 to \(x\), then the final value of \(x\) is between 1 and 10.

Programs and Implication—Comment

- Predicate \(\text{prog}_1 \Rightarrow \text{prog}_2\) is true if
  - the before-after relationship described by \(\text{prog}_1\) does not hold, or …
  - \(\text{prog}_1\) holds, and the behaviour described by \(\text{prog}_2\) somehow includes the behaviour of \(\text{prog}_1\)
- Predicate \(\text{prog} \Rightarrow \text{pred}\) is true if
  - the before-after relationship described by \(\text{prog}\) does not hold, or …
  - \(\text{prog}\) holds, and the situation described by \(\text{pred}\) is covered by the behaviour of \(\text{prog}\)

Mini-Exercise 3

Q3.1 Prove

\[
(\forall x \bullet P) \equiv \neg (\exists x \bullet \neg P)
\]

using only the following axioms:

\[
\langle \text{gen-deMorgan} \rangle \quad (\exists x \bullet P) \equiv \neg (\forall x \bullet \neg P)
\]

\[
\langle \neg \text{-invol} \rangle \quad \neg \neg p \equiv p
\]

Q3.2 Prove

\[
\langle ;\text{-skip-unit} \rangle \quad P; \text{skip} = P
\]

(due in next Friday, 2pm, in class)
Taking Stock

- We are playing a BIG formal game
- We have a well-defined extensible language:
  \[ \text{Pred ::= true} \mid \ldots \mid \forall x \bullet P \mid \ldots \]
- We have given it a well-defined meaning:
  \[ \exists x \bullet P \rho = \ldots \]
- We have provided rules to do proofs:
  \[ F = G \quad \frac{E[v := F]}{E[v := G]} \]
- Does it all make sense?

Truth and Provability

- **Statements**
  We are interested in statements of the following form:
  
  Whenever predicates \( P_1, \ldots, P_n \) are true, then so is \( Q \).

  (Let us use \( \Gamma \) as a shorthand for \( P_1, \ldots, P_n \)).

- **Truth** — \( \Gamma \models Q \)
  The above statement is *true* when, for every environment \( \rho \) that makes all the \( P_i \) true, we have \( [Q]_\rho = \text{True} \).

- **Provability** — \( \Gamma \vdash Q \)
  The above statement is *provable* when, given the \( P_i \) as assumptions, we can prove \( Q \) as a theorem.

Truth vs. Provability

- **Truth and Provability are not the same thing!**
- **Soundness**
  A proof system is sound if whenever we can prove something, it is also true:
  
  \( \Gamma \vdash Q \) means that \( \Gamma \models Q \)

- **Completeness**
  A proof system is complete if whenever something is true, it can be proved:
  
  \( \Gamma \models Q \) means that \( \Gamma \vdash Q \)

- The “Holy Grail” of formal systems is a sound and complete formal system
Soundness

- Soundness is a critical property
- Unsound proof systems allow "proofs" of false statements
  Very undesirable!
- Unsoundness can arise in one of two ways:
  - **Inconsistency**
    - A proof system is inconsistent if anything can be proved using it
    - In particular if we can prove false, we can prove anything.
  - **Environment Mismatch**
    Even if consistent, a proof system can be unsound if our axioms and inference rules are incorrect, and fail to capture the truth properly.
- Ensuring soundness require great care in determining axioms and inference rules.

Completeness

- Completeness is a “nice-to-have” property
  - In principle a complete proof-system can be made fully automatic.
- Incompleteness simply means there are some true properties for which there are no proofs in our formal system.
- Gödel's Incompleteness Theorems (1931):
  1. Any proof-system powerful enough for arithmetic cannot be both consistent and complete.
  2. Any proof-system powerful enough to prove theorems about itself (arithmetic !), is inconsistent iff it can prove its own consistency
- Most formal systems we want to use embody arithmetic and so are incomplete as just described.

Meta-mathematics

- Meta-mathematics is the study of mathematics itself, as a "mathematical object"
- A major focus of meta-mathematics is the study of proof-systems
- There is a large body of soundness and completeness results out there for different formal systems.
- This course is based on a (mainly) sound but incomplete logic system
  - Mainly ???
  - Yes, but …
  - We will be extending our language later by adding axioms of our own …

Relating $x := 2$ and $x' \in \{1, \ldots, 10\}$

- $x' \in \{1, \ldots, 2\}$ could be viewed as a specification:
  "pick a number between 1 and 10, call it $x$".
- $x := 2$ is clearly a program that satisfies that specification
- So does $x := 5$ (say)
- More surprisingly, so does the program $x := 2; y := 99$ ! The spec. says nothing about $y$ (or any other variable).
- Interestingly, the “program” $x := 2 \lor x := 5$ also satisfies the specification.
Specifying “pick a number”

- the problem with $x' \in \{1, \ldots, 10\}$ is that it allows anything to happen to variables other than $x$.
- We can strengthen it to require that only $x$ be modified:
  $$x' \in \{1, \ldots, 10\} \land S' \setminus x = S_\setminus x$$
- This idiom is so common we invent specific notation for it:
  $$x : [x' \in \{1, \ldots, 10\}]$$
  The outer $x$ says, “modifying $x$ only,”
- The notation $x : […]$ is called a (specification) “frame”.

Formally introducing frames

- Once again, we extend our predicate language, with 
  **specification frames** (n.s.)
  $$\text{Pred} ::= \ldots \mid \vec{x} : [P]$$
- It asserts $P$, and that any variable not in $\vec{x}$ is unchanged:
  $$\langle\langle \text{frame-def} \rangle\rangle \vec{x} : [P] \equiv P \land S' \setminus \vec{x} = S_\setminus \vec{x}, \ A = \{S, S'\}$$
- Not be confused with $[P]$, the universal closure of $P$
  or $x : T$, asserting that $x$ has type $T$, 
  or $P(e/x)$ where $e$ replaces free $x$ in $P$.

Satisfying Specifications

- Given specification $S \equiv x : [x' \in \{1, \ldots, 10\}]$, how do we check if an offered program $P$ fully satisfies it?
  - $P_2 \equiv x := 2$ clearly satisfies it,
    and we saw previously that $P_2 \Rightarrow S$.
  - Can we conclude then that we want $P \Rightarrow S$ as our acceptance criteria?
- Problem: predicate $P \Rightarrow S$ does not give us a yes/no answer.
- Instead, we get a predicate whose truth depends on the values of variables.

Is Implication Satisfaction?

- Consider $x := 2 \Rightarrow x : [x' \in \{1, \ldots, 10\}]$
- Formally (assuming only $x$ and $y$ in scope):
  $$x := 2 \Rightarrow x : [x' \in \{1, \ldots, 10\}]$$
  = “ $\langle\langle \text{def} \rangle\rangle$, $\langle\langle \text{frame-def} \rangle\rangle$ ”
  $$x' = 2 \land y' = y \Rightarrow x' \in \{1, \ldots, 10\} \land y' = y$$
  = “ equality substitution (?) — see later ”
  $$x' = 2 \land y' = y \Rightarrow 2' \in \{1, \ldots, 10\} \land y = y$$
  = “ set theory, reflexivity of equality ”
  $$x' = 2 \land y' = y \Rightarrow \text{true} \land \text{true}$$
  = “ prop. logic ”
  $$\text{true}$$
- Seems OK.
Implication is not Satisfaction (I)

- Consider $y := 99 \Rightarrow x : [x' \in \{1, \ldots, 10\}]
- Formally (assuming only $x$ and $y$ in scope):

\[
y := 99 \Rightarrow x : [x' \in \{1, \ldots, 10\}]
= \langle\langle \text{:=-def} \rangle, \langle\langle \text{frame-def} \rangle \rangle
x' = x \land y' = 99 \Rightarrow x' \in \{1, \ldots, 10\} \land y' = y
= \text{equality substitution (? — see later)}
\]
\[
x' = x \land y' = 99 \Rightarrow x \in \{1, \ldots, 10\} \land 99 = y
\]

The truth/falsity of this satisfaction question seems to depend on initial $x$ already satisfying the specification, and initial $y$ being 99 already!

Implication is not Satisfaction (II)

- Consider $x := x + 10 \Rightarrow x : [x' > y']
- Formally (assuming only $x$ and $y$ in scope):

\[
x := x + 10 \Rightarrow x : [x' > y']
= \langle\langle \text{:=-def} \rangle, \langle\langle \text{frame-def} \rangle \rangle
x' = x + 10 \land y' = y \Rightarrow x' > y' \land y' = y
= \text{equality substitution (? — see later)}
\]
\[
x' = x + 10 \land y' = y \Rightarrow x + 10 > y \land y = y
= \text{logic, arithmetic}
\]
\[
x' = x + 10 \land y' = y \Rightarrow x > y - 10
\]

- Program $x := x + 10$ only satisfies specification $x : [x' > y']$
  if $x$ is greater than $y - 10$ to begin with.
Full Satisfaction (a.k.a. "Refinement")

- Given specification $S$ and program $P$:
  - all before/after-relationships resulting from $P$ must satisfy those required by $S$
  - so $P$ must imply $S$ for all possible variable before- and after-values
- So we require universal implication, saying that $P$ satisfies $S$ iff
  $$[P \Rightarrow S]$$

- Once more, we add special predicate notation, saying that $S$ is "refined by" $P$, written $S \sqsubseteq P$

Universal Closure re-visited

- Reminder: notation $[P]$ asserts that $P$ is true for all values of its variables
- $[P] \equiv \forall x_1, \ldots, x_n \bullet P$, $\forall \forall P \subseteq \{x_1, \ldots, x_n\}$
- It satisfies the following laws:

  - $\llbracket \text{true} \rrbracket \equiv \text{true}$
  - $\llbracket \text{false} \rrbracket \equiv \text{false}$
  - $\llbracket P \land Q \rrbracket \equiv [P] \land [Q]$
  - $\llbracket [P] \rrbracket \equiv [P]$
  - $\llbracket \forall x_1, \ldots, x_n \bullet P \rrbracket \equiv |P|
  - $\llbracket x = e \Rightarrow P \rrbracket \equiv [P[e/x]]$
  - $\llbracket P \rrbracket \equiv [P] \land P[\vec{e}/\vec{x}], \alpha P = \{\vec{x}\}$

- It does not distribute over logical-or $[P \lor Q] \neq [P] \lor [Q]$

Example: Assignment as Frame refinement

Consider $x : [P] \sqsubseteq x := e$, assuming $x$ and $y$ in scope,

$[x := e \Rightarrow x : [P]]$

$\equiv\ " \llbracket \text{:=-def}, \llbracket \text{frame-def} \rrbracket "$

$[x' = e \land y' = y \Rightarrow P \land y' = y]$

$\equiv\ " \llbracket \text{shunting} \rrbracket "$

$[x' = e \Rightarrow (y' = y \Rightarrow P \land y' = y)]$

$\equiv\ " (A \Rightarrow \land B) \equiv (A \Rightarrow B) "$

$[x' = e \Rightarrow (y' = y \Rightarrow P)]$

$\equiv\ " \llbracket \text{=1pt} \rrbracket "$

$[y' = y \Rightarrow P[e/x']]$

$\equiv\ " \llbracket \text{=1pt} \rrbracket "$

$[P[e/x'][y/y']$
Example: \( x : [x' \in \{1, \ldots, 10\}] \sqsubseteq x := 2 \)

\[
x : [x' \in \{1, \ldots, 10\}] \sqsubseteq x := 2
= \text{ " previous slide "}
\]

\[
[ (x' \in \{1, \ldots, 10\})[2/x'][y/y'] ]
= \text{ " substitution "}
\]

\[
x \in \{1, \ldots, 10\}
= \text{ " set theory "}
\]

\[
\text{false}
= \text{ " set-theory, arithmetic, logic, hand-waving "}
\]

Example: \( x : [x' \in \{1, \ldots, 10\}] \not\sqsubseteq y := 99 \)

\[
x : [x' \in \{1, \ldots, 10\}] \not\sqsubseteq y := 99
= \text{ " previous slide "}
\]

\[
[ (x' \in \{1, \ldots, 10\})[x/x'][99/y'] \land 99 = y ) ]
= \text{ " substitution "}
\]

\[
x \in \{1, \ldots, 10\} \land 99 = y
= \text{ " [\text{-split} "}
\]

\[
x \in \{1, \ldots, 10\} \land 99 = y
\land (x \in \{1, \ldots, 10\} \land 99 = y )[0, 0/x, y]
= \text{ " substitution "}
\]

\[
x \in \{1, \ldots, 10\} \land 99 = y \land 0 \in \{1, \ldots, 10\} \land 99 = 0
= \text{ " set-theory, arithmetic, logic, hand-waving "}
\]

false

Example: \( x : [P] \sqsubseteq y := e \), assuming \( x \) and \( y \) in scope,

\[
[y := e \Rightarrow x : [P]]
= \text{ " [\text{-def}, \text{frame-def} "}
\]

\[
[x' = x \land y' = e \Rightarrow P \land y' = y]
= \text{ " [shunting "}
\]

\[
[x' = x \Rightarrow (y' = e \Rightarrow P \land y' = y)]
= \text{ " [\text{-1pt} "}
\]

\[
[(y' = e \Rightarrow P \land y' = y)[x'/x]]
= \text{ " substitution "}
\]

\[
[(P \land y' = y)[x/x'][e/y']]
= \text{ " substitution "}
\]

\[
(P[x/x'][e/y'] \land e = y)
= \text{ " substitution "}
\]

false

Example: wrong-Assignment as Frame refinement

Equality Substitution

- Earlier, we used a law referred to as "equality substitution".
- If we assert a number of equalities, and something else, it allows us to use those equalities in the "something else"
- Formally:

\[
\langle\langle \text{\text{-\Land-subst}} \rangle\rangle
\]

\[
(x = e \land P \equiv x = e \land P[e/x])
\]

- Proof is by induction over the structure of \( P \), so we omit it (consider it an axiom).
- The law still holds if we only substitute \( e \) for \( x \) in part of \( P \), rather than all of it. We also have the following theorem:

\[
\langle\langle \text{\text{-\To-subst}} \rangle\rangle
\]

\[
(x = e \land P \Rightarrow Q) \equiv (x = e \land P \Rightarrow Q[e/x])
\]
Refinement Example: MinMax (I)

- Consider the following specification:
  \[ S \triangleq x, y :: [x := \max(x, y), \min(x, y)] \]

- "changing only \( x \) and \( y \), ensure that \( x \) ends up as the maximum of the two, whilst \( y \) is the minimum."

- We posit the following "code" as a solution:
  \[ Q \triangleq \text{skip} \implies x \geq y \implies x := y, x \]

- Does \( Q \) refine \( S \)?

\[ S \sqsubseteq Q? \text{ Solution (Ia)} \]

- \[ x, y :: [x, y := \max(x, y), \min(x, y)] \]
  \[ \sqsubseteq \text{skip} \implies x \geq y \implies x := y, x \]
  \[ = \quad " \text{sim-def}\) " \]
  \[ \text{skip} \implies x \geq y \implies x := y, x \]
  \[ \implies x, y :: [x, y := \max(x, y), \min(x, y)] \]
  \[ = \quad " \text{<def>-def}\) " \]
  \[ x \geq y \land \text{skip} \lor x < y \land x := y, x \]
  \[ \implies x, y :: [x, y := \max(x, y), \min(x, y)] \]
  \[ = \quad " (A \lor B \Rightarrow C) \equiv (A \Rightarrow C) \land (B \Rightarrow C) " \]
  \[ (x \geq y \land \text{skip}) \]
  \[ \implies x, y :: [x, y := \max(x, y), \min(x, y)] \]
  \[ \land (x < y \land x, y := y, x) \]
  \[ \implies x, y :: [x, y := \max(x, y), \min(x, y)] \]

\[ S \sqsubseteq Q? \text{ Solution (Ib)} \]

- \[ x \geq y \land \text{skip} \]
  \[ \implies x, y :: [x, y := \max(x, y), \min(x, y)] \]
  \[ = \quad " \text{sim-def}\} , \text{<frame>-def}\{(2), \text{<sim>-def}\} " \]
  \[ x \geq y \land x' = x \land y' = y \]
  \[ \implies x', y :: [x, y := \max(x, y), \min(x, y)] \]
  \[ = \quad " \text{<def>-def}\} , \text{<frame>-def}\{(2), \text{<sim>-def}\} " \]
  \[ x \geq y \land x' = x \land y' = y \]
  \[ \implies x', y :: [x, y := \max(x, y), \min(x, y)] \]
  \[ \land x < y \land x', y :: y \land y' = x \]
  \[ \implies x', y :: [x, y := \max(x, y), \min(x, y)] \]

\[ S \sqsubseteq Q? \text{ Solution (Ic)} \]

- \[ x' = x \land y' = y \implies x \geq y \]
  \[ \implies x' = \max(x, y) \land y' = \min(x, y) \]
  \[ = \quad " \text{<def>-def}\} , \text{<frame>-def}\{(2), \text{<sim>-def}\} " \]
  \[ x' = y \land y' = x \implies x < y \]
  \[ \implies x' = \max(x, y) \land y' = \min(x, y) \]
  \[ = \quad " \text{<def>-def}\} , \text{<frame>-def}\{(2), \text{<sim>-def}\} " \]
  \[ x' = x \land y' = y \implies x \geq y \]
  \[ \implies x = \max(x, y) \land y = \min(x, y) \]
  \[ \land x < y \implies x' = \max(x, y) \land y = \min(x, y) \]
Refinement Example: MinMax (II)

Given the following (familiar) program

\[ P \triangleq x := x + y ; y := x - y ; x := x - y \]

Can we replace \( x, y := y, x \) in \( Q \) by \( P \) and still have it refine \( S \)?
Mini-Issues (1–3)

A number of issues have been raised by class members while working through the mini-exercises:

- Precisely how to interpret laws, when doing matching against them.
- The strange notation $P(b)$.

We address these briefly.

Pattern Matching Laws

Consider, for example, the following laws:

$$
(\exists x \cdot P) \equiv (\neg (\forall x \cdot \neg P))
$$

$$
\nu := e \equiv \nu' = e \land S' = S
$$

$x, \nu'$ are regular observation variables, $P$ is a predicate variable, $e$ is an expression variable, and $\nu$ is a program variable.

In general these match anything of the corresponding type.

The exception is when such a variable has been defined elsewhere to mean something specific. In such a case a variable can only match itself.

So far the only example we have seen of this is when we have defined a specific alphabet. $S, S'$ are list-variables that match themselves, or the alphabet, if known.

What is $P(b)$?

Previously, we saw a law of the form

$$
(\forall b : B \cdot P(b)) \equiv P(False) \land P(True)
$$

Here, the intended meaning of $P(b)$ is that $P$ is a predicate that (possibly) mentions $b$, as a free variable.

Then $P(True)$ is $P$ with all such free $b$ replaced by $True$.

It is a convenient shorthand, more formally written as

$$
(\forall b : B \cdot P) \equiv P(False/b) \land P(True/b)
$$
re-examining Refinement

Consider the following laws:

- ▶ \( P \subseteq P \) (\( \sqsubseteq \)-refl)
- ▶ \( (S \subseteq Q) \land (Q \subseteq P) \Rightarrow (S \subseteq P) \) (\( \sqsubseteq \)-trans)
- ▶ \( P = Q \equiv (P \subseteq Q) \land (Q \subseteq P) \) (\( \sqsubseteq \)-anti)

What do they say about refinement?

- ▶ It is a partial order.

Partial Orders (reminder)

- ▶ A set \( S \), with a binary relation \( \triangleleft \) between its elements, is called a partial order (p.o.) iff:
  - the relation is reflexive: \( \forall x : S \ni x \triangleleft x \)
  - the relation is transitive: \( \forall x, y, z : S \ni x \triangleleft y \land y \triangleleft z \Rightarrow x \triangleleft z \)
  - the relation is anti-symmetric: \( \forall x, y : S \ni x \triangleleft y \land y \triangleleft x \Rightarrow x = y \)

- ▶ Examples:
  - \( (\mathbb{N}, \leq) \), natural numbers under usual numeric ordering.
  - \( (\mathbb{P}, \subseteq) \), sets, under the subset relation.
  - \( (\text{Pred}, \Rightarrow) \), predicates, under the implication relation.
  - \( (\text{Pred}, \sqsubseteq) \), predicates, under the refinement relation.

- ▶ Partiality: in general some of these orders are partial, in that, given arbitrary \( x \) and \( y \), it may not be the case that either \( x \triangleleft y \) or \( y \triangleleft x \), e.g. \( \{1, 2\} \not\subseteq \{3\} \) and \( \{3\} \not\subseteq \{1, 2\} \)

Meet and Join (a.k.a. Min and Max)

- ▶ Given two elements \( x \) and \( y \) in a p.o., we can ask if their minimum or maximum exists under that ordering.
- ▶ We refer to the minimum as the “meet” (\( \sqcap \)), and the maximum as the “join” (\( \sqcup \)).
- ▶ If \( x \triangleleft y \) then the meet of \( x \) and \( y \) is \( x \), and their join is \( y \).

\[
(x \sqcap y = x) \equiv (x \triangleleft y) \equiv (x \sqcup y = y)
\]

- ▶ Meets and joins exists for all our examples so far:
  - \( (\mathbb{N}, \leq) \), meet is minimum, join is maximum
  - \( (\mathbb{P}, \subseteq) \), meet is intersection, join is union.
  - \( (\text{Pred}, \Rightarrow) \), meet is logical-and, join is logical-or.

Meet and Join in Refinement

- ▶ We use the general symbols \( \sqcap \) and \( \sqcup \) to stand for meet and join with respect to the refinement ordering.
- ▶ Meet and Join for refinement have simple definitions.
  - ▶ \( (\sqcap \sqsubseteq) \) \( P \sqcap Q \sqsubseteq P \land Q \)
  - ▶ \( (\sqcup \sqsubseteq) \) \( P \sqcup Q \sqsubseteq P \lor Q \)

- ▶ From these we can deduce the following laws:
  - ▶ \( (\sqsubseteq \sqcap) \) \( P \subseteq Q \equiv (P \sqcap Q \equiv Q) \)
  - ▶ \( (\sqsubseteq \sqcup) \) \( P \subseteq Q \equiv (P \sqcup Q \equiv P) \)

- ▶ A p.o. in which meets and joins exists for all pairs of elements is called a Lattice.
Complete Lattices

- A lattice is Complete if meets and joins (maxima/minima) can be found for arbitrary sets of elements:
  - $\bigcap S$ — the minimum of all the elements of $S$
  - $\bigcup S$ — the maximum of all the elements of $S$
- Not all of our examples are complete lattices:
  - $(\mathbb{N}, \leq)$ is not complete — what is the maximum element of the set of all even naturals?
- The following are complete lattices:
  - $\mathcal{P}(T), \subseteq$, take $\cap$ and $\cup$.
  - $(\text{Pred}, \Rightarrow)$, take $\forall, \exists(?)$.

Complete Lattices: a Key Property

- A complete lattice $(S, \preceq)$ has overall maximum and minimum elements
  - The minimal element ($\bot$ or “bottom”) is simply $\bot \equiv \bigcap S$
  - The maximal element ($\top$ or “top”) is simply $\top \equiv \bigcup S$
- Clearly, both top and bottom satisfy the following laws:
  - $\bot \preceq x$, for all $x \in S$
  - $x \preceq \top$, for all $x \in S$

Refinement is a Complete Lattice

- The set of all predicates under the refinement relation forms a complete lattice
  - $P \cap Q \equiv P \lor Q$
  - $P \cup Q \equiv P \land Q$
  - $\bigcap S \equiv \bigvee S$
  - $\bigcap \{P_1, \ldots, P_n\} \equiv P_1 \lor \ldots \lor P_n$
  - $\bigcup S \equiv \bigwedge S$
  - $\bigcup \{P_1, \ldots, P_n\} \equiv P_1 \land \ldots \land P_n$
- The minimal events:
  - $\bot \equiv \text{true}$
  - $\text{true} \subseteq P$
  - $P \subseteq \top$
  - $P \subseteq \text{false}$

What are $\text{true}$ and $\text{false}$?

- So, we have $\text{true} \subseteq P \subseteq \text{false}$ for any predicate $P$.
- In terms of our interpretation of predicates as programs and/or specifications, what do $\text{true}$ and $\text{false}$ denote?
- $\text{true}$ is the weakest possible specification (“whatever”) or the most badly behaved program (“do anything”)
- We shall call it “Chaos” (a.k.a. “abort”)
  - $\text{Chaos} \equiv \text{true}$
- $\text{false}$ is capable of satisfying any specification!
- We shall call it “miracle” (a.k.a. “magic”).
  - $\text{miracle} \equiv \text{false}$
Implication Lattice

relating predicates with universal implication (~⇒~):  

false  \quad strong  
\begin{align*}  
x' = 3 & \quad x' = 6 \\  \therefore x' = 3 \lor x' = 6 \\  \therefore x' \in \{1\ldots10\} \\  true & \quad weak  
\end{align*}

Refinement Lattice

relating programs/specifications with refinement (~⊆~):  

\begin{align*}  
miracle & \quad prog. \\  \begin{array}{c}  
x := 3 \\  x := 6 \\  x := 3 \land x := 6 \\  x \colon [x' \in \{1\ldots10\}] \\  Chaos \quad \text{spec.} 
\end{array}  
\end{align*}

The Theology of Computing

\begin{align*}  
miracle & \quad prog. \\  \begin{array}{c}  
x := 3 \\  x := 6 \\  x := 3 \land x := 6 \\  x \colon [x' \in \{1\ldots10\}] \\  Chaos \quad \text{spec.} 
\end{array}  
\end{align*}

Refinement

- The notion of Refinement, relating a specification to any program that satisfies it, is key in formal methods.
- In our formalism, UTP, it (~⊆~) is defined as “universally-closed (~[~]) reverse implication (~⇐~):
  \[ \text{⟨⟨~[~]-def~⟩⟩ } S \subseteq P \iff [S \leftarrow P] \]
- The key thing to remember is the reversal of the implication, e.g.
  \[ S \subseteq P \equiv [P \Rightarrow S] \]
Laws of Implication

- There are many laws of implication:

  - \( \Leftrightarrow\)-refl: \( P \Rightarrow P \equiv \text{true} \)
  - \( \Leftrightarrow\)-trans: \( (P \Rightarrow Q) \land (Q \Rightarrow R) \Rightarrow (P \Rightarrow R) \)
  - \( \equiv\Rightarrow\)-trans: \( (P \equiv Q) \land (Q \Rightarrow R) \Rightarrow (P \Rightarrow R) \)
  - \( \equiv\Rightarrow\)-trans: \( (P \Rightarrow Q) \land (Q \equiv R) \Rightarrow (P \Rightarrow R) \)
  - \( \text{strengthen-ante} \): \( P \land Q \Rightarrow P \)
  - \( \text{weaken-cnsq} \): \( P \Rightarrow P \lor Q \)
  - \( \text{ante}\lor\text{-distr} \): \( P \lor Q \Rightarrow R \equiv (P \Rightarrow R) \land (Q \Rightarrow R) \)
  - \( \text{cnsq}\lor\text{-distr} \): \( P \Rightarrow Q \land R \equiv (P \Rightarrow Q) \land (P \Rightarrow R) \)

- As refinement is defined as “universally-closed reverse implication”, we expect to derive many refinement laws from these, and similar laws.

Exploring the Refinement Laws (I)

- \( P \sqsubseteq P \) \( \equiv\)-refl
  Anything refines itself (trivially).

- \( (P = Q) \Rightarrow (P \sqsubseteq Q) \) \( \equiv\)-imp-$\equiv$
  Equality is a (fairly trivial) refinement.

- \( (P = Q) \equiv (P \sqsubseteq Q) \land (Q \sqsubseteq P) \) \( \equiv\)-anti
  Two programs/specifications are equal if they refine each other.

- \( (S \sqsubseteq Q) \land (Q \sqsubseteq P) \Rightarrow (S \sqsubseteq P) \) \( \equiv\)-trans
  If \( Q \) refines \( S \), and \( P \) refines \( Q \), then \( P \) also refines \( S \).

- \( (S = Q) \land (Q \sqsubseteq P) \Rightarrow (S \sqsubseteq P) \) \( \equiv\)-trans

- \( (S \sqsubseteq Q) \land (Q = P) \Rightarrow (S \sqsubseteq P) \) \( \equiv\)-trans
  A refinement step can also be a replacement by something equal.

Exploring the Refinement Laws (II)

- \( S \sqsubseteq P \lor Q \equiv (S \sqsubseteq P) \land (S \sqsubseteq Q) \) \( \equiv\)-spec-split
  Refining by an arbitrary choice of alternatives is the same as refining by both options individually.

- \( (S \sqsubseteq P) \land (T \sqsubseteq P) \) \( \equiv\)-prog-alt
  Satisfying a specification with two mandatory parts is the same as satisfying both parts individually.

- \( S \lor T \sqsubseteq T \) \( \equiv\)-spec-weakening
  A specification offering alternatives can be satisfied by choosing one of those alternatives.

- \( P \sqsubseteq P \land Q \) \( \equiv\)-prog-strengthen
  A specification can be refined by adding in extra constraints.
Proving some refinement laws

- Select and do in class.
- Note the following meta-theorem:
  - If \( P \) is a theorem, then so is \([P]\).
  - Why: because if \( P \) is a theorem, then it is true for all instantiations of variables it contains. So, it remains true if we quantify over the expression variables.

Proof of \(\equiv\)-trans

- Goal: \((S = Q) \land (Q \sqsubseteq P) \Rightarrow (S \sqsubseteq P)\).
- Strategy: reduce to law \(\Rightarrow\)-trans.
- Proof:

\[
(S = Q) \land (Q \sqsubseteq P) \Rightarrow (S \sqsubseteq P) = \quad \langle\langle \equiv\text{-def} \rangle\rangle \\
S = Q \land [P \Rightarrow Q] \Rightarrow [P \Rightarrow S] = \quad \langle\langle \text{equality substitution, } S = Q \rangle\rangle \\
S = Q \land [P \Rightarrow Q] \Rightarrow [P \Rightarrow Q] = \quad \langle\langle \text{strengthen-ante} \rangle\rangle \\
\text{true} \quad \Box
\]

Mini-Exercise 4

Q4.1 Prove law \(\vec{x} : [A] \sqsubseteq \vec{x} : [A \land B]\)
(due in next Friday, 2pm, in class)
Consider the addition of a notation for arrays into our expression language.
- If $a$ is an array, then $a_i$ denotes the $i$th element.
- We assume the array elements are indexed from $1 \ldots N$.

Find index of a value
- Consider specification $spec$:
  \[
  p : \{ [a_p = w \land w \in a] \land p' = 0 \}
  \]
  Modifying $p$, set it equal to the index of an element in the array equal to $w$, otherwise zero.
- Consider $prog_1$:
  \[
  p := 1 \\
  \quad ; \quad (a_p \neq w \land p < N) \land p := p + 1 \\
  \quad ; \quad p := 0 \land p = N \land skip
  \]
- Consider $prog_2$:
  \[
  p := N \\
  \quad ; \quad (a_p \neq w \land p > 0) \land p := p - 1
  \]

Running the programs
- Consider starting state
  \[
  w = 3 \land a = [1, 2, 3, 4, 5, 6] \quad (N = 6)
  \]
- Running $prog_1$ results in the final outcome:
  \[
  p' = 3
  \]
  (it searches left-to-right)
- Running $prog_2$ results in the final outcome:
  \[
  p' = 5
  \]
  (it searches right-to-left)
Relating *spec* and the *prog*.

- We see that both programs refine the specification:
  
  \[ spec \sqsubseteq \text{prog}_i, \quad i \in 1, 2 \]

- Both programs, however, have different outcomes.
  
  \[ \text{prog}_1 \neq \text{prog}_2 \]
  
  \((p' = 3 \text{ vs. } p' = 5)\).

- An arbitrary choice between the programs also refines the spec:
  
  \[ spec \sqsubseteq \text{prog}_1 \lor \text{prog}_2 \]
  
  \((p' = 3 \lor p' = 5)\).

Refinement and non-determinism

- The programs are deterministic
  
  \[ \text{prog}_1 \text{ search left-to-right} \]
  \[ \text{prog}_2 \text{ search right-to-left} \]

- The choice between them has some non-determinism
  
  \[ \text{prog}_1 \lor \text{prog}_2 \text{ search either l-to-r, or r-to-l} \]

- The specification has lots of non-determinism
  
  \[ spec \text{ search any which-way} \]

- Refinement is essentially about reducing non-determinism.

Program-Oriented Refinement Laws

- Most of the refinement laws seen to date are derived from the implication laws.

- We have others as well, based on language constructs:
  
  \[
  \begin{align*}
  \langle\langle \text{:=} \rangle\rangle & \quad x' = e \sqsubseteq x := e \\
  \langle\langle \text{:=\sim} \rangle\rangle & \quad \vec{x}' = \vec{e} \sqsubseteq \vec{x} := \vec{e} \\
  \langle\langle \text{:=\lead} \rangle\rangle & \quad P \sqsubseteq x := e; P, \quad x \not\in P \\
  \langle\langle \text{:=\trail} \rangle\rangle & \quad P \sqsubseteq P; x := e, \quad x' \not\in P \\
  \langle\langle \text{sim:=\lead} \rangle\rangle & \quad P \sqsubseteq \vec{x} := \vec{e}; P, \quad \vec{x} \not\in P \\
  \langle\langle \text{sim:=\trail} \rangle\rangle & \quad P \sqsubseteq P; \vec{x} := \vec{e}, \quad \vec{x}' \not\in P \\
  \langle\langle \sqsubseteq \rangle\rangle & \quad (S_1 \sqsubseteq P_1) \land (S_2 \sqsubseteq P_2) \Rightarrow (S_1; S_2) \sqsubseteq (P_1; P_2)
  \end{align*}
  \]

- The conditions \( x \not\in P \) and \( x' \not\in P \) exclude a lot, e.g. all assignment statements (why?)

Proof of \( \langle\langle \text{:=} \rangle\rangle \)

\[
\begin{align*}
\text{x'} = e & \sqsubseteq x := e \\
= & \quad \text{“} \langle\langle \text{:=\def} \rangle\rangle \text{”} \\
x' = e & \sqsubseteq x' = e \land S' \setminus x' = S' \setminus x \\
= & \quad \text{“} \langle\langle \text{:=\sim\def} \rangle\rangle \text{”} \\
[x' = e \land S' \setminus x' = S' \setminus x] & \Rightarrow x' = e \\
= & \quad \text{“prop. law: } A \land B \Rightarrow A \text{”} \\
\text{[true]} & \quad \text{“}\langle\langle \text{-true} \rangle\rangle \text{”} \\
\text{true} & 
\end{align*}
\]
Proof of $\langle \langle \vdash \rangle \rangle$

$P \models P; x := e$

$\Rightarrow \langle \langle \vdash \rangle \rangle$

$[P; x' = e \land S^{\bar{x}'} = S^{\bar{x}} \Rightarrow P]$

$\Rightarrow \langle \langle \vdash \rangle \rangle$

$[(\exists x_m, S_m \setminus x_m \bullet P[x_m, S_m \setminus x_m / x', S^{\bar{x}'}])$

$\land (x' = e \land S^{\bar{x}'} = S^{\bar{x}})[x_m, S_m \setminus x_m / x, S^{\bar{x}}])$

$\Rightarrow P$

$\Rightarrow \langle \langle \vdash \rangle \rangle$

substitution, noting $x' \notin P$

$[(\exists x_m, S_m \setminus x_m \bullet P[x_m, S_m \setminus x_m / x', S^{\bar{x}'}])$

$\land (x' = e \land S^{\bar{x}'} = S^{\bar{x}})[x_m, S_m \setminus x_m / x, S^{\bar{x}}])$

$\Rightarrow P$

$\Rightarrow \langle \langle \vdash \rangle \rangle$

Real Life: BASE Trusted Gateway

Goal — Trusted Gateway for transferring messages between different security levels, for British Aerospace Systems & Equipment.

Approach — Two teams, one conventional, the other using formal methods.

Method — Formal team employed VDM-SL, using IFAD Toolkit.


BASE: Key Results (1)

- Formal approach spent more time up front in System Design (43% as against 34%).
- Formal approach uncovered an implicit special condition from requirements. Informal code had to be re-written at late stage to cope.
- Formal code was less complex (“McCabe Complexity”).
- Formal code one-fifth the size of informal code.
BASE: Key Results (2)

Formal system started up slower
(4 times longer)

1. Formal System Invariant better understood, so more care was taken by resulting initialisation code.
2. Not a big issue as the system is meant to stay up and running.

BASE: Key Results (3)

Formal system throughput higher
(almost 14 times faster)

1. The informal system had to have a last-minute fix, so the code speed got worse.
2. If code is formally verified, then you don’t need so many run-time checks (array bounds, etc.)

Reasoning about Loops

So far we have one law regarding loops, that “unrolls” once:

\[ \langle c \text{-unroll} \rangle \quad c \otimes P = (P \; c \otimes P) \triangleleft c \triangleright \text{skip} \]

This has limited utility, as we often want to reason about arbitrary numbers of “un-rollings”.

Loop Semantics

- What is the meaning of \( c \otimes P \)?
- Let us call it \( W \).
- It must satisfy the un-rolling law:

\[ W = (P; \; W) \triangleleft c \triangleright \text{skip} \]

- It must also be the the least predicate that satisfies this law:

\[ (X = (P; \; X) \triangleleft c \triangleright \text{skip}) \Rightarrow W \sqsubseteq X \]

- These results come from a branch of mathematics called “fixpoint theory”.
  - The “meaning” of recursion is (typically) the least fixed point of an appropriate higher-order function.
  - We shall not concern ourselves with this at present.
Infinite Loops

- What is the meaning of $\text{true \# skip}$?
- It is an infinite loop, so call it $\text{Forever}$.
- It must satisfy the un-rolling law:
  $$\text{Forever} = (\text{skip}; \text{Forever}) \triangleleft \text{true} \triangleright \text{skip}$$
  $$= \text{skip}; \text{Forever}$$

- However, all of the following satisfy this instance of the un-rolling law:
  - $\text{miracle} = \text{skip}; \text{miracle}$
  - $\text{skip} = \text{skip}; \text{skip}$
  - $\text{Chaos} = \text{skip}; \text{Chaos}$
  - $P = \text{skip}; P$ for any $P$ (It's law $\langle \langle \text{skip}; \cdot \rangle \rangle$) !
- Why pick the least of these as the meaning of $\text{Forever}$?

Forever is bad

- Every predicate satisfies the $\text{Forever}$ unrolling law, including $\text{miracle}$ and $\text{Chaos}$, our two extremes.
- It makes no sense to pick a predicate “in the middle”, so which extreme point should we use?
- If we use $\text{miracle}$, then $\text{Forever}$ refines any specification, which is not desirable at all.
- It makes sense to choose $\text{Chaos}$, as it is least in our ordering, and the only thing refining it is itself.
- So $\text{Forever} = \text{Chaos} = \text{true}$
  So $\text{Chaos}$ covers all unpredictable/undesirable behaviour, including non-termination.
Loop Reasoning

Given a loop $c \triangleright P$, and possible candidate $W$:

- checking the unroll-law is not too bad:
  $$W = (P; W) \triangleleft c \triangleright \text{skip}$$

- checking for the least such fixpoint is painful:
  $$(X = (P; X) \triangleleft c \triangleright \text{skip}) \Rightarrow W \sqsubseteq X, \text{ for any } X$$

- Fortunately, in many cases, the fixpoint is unique.
- We won’t characterise the unique cases here (quite technical).
- Instead we shall just use the unroll check.
- A challenge still remains: finding $W$!

A Simple example (I)

- Our specification: summing (natural) numbers between 1 and $n$:
  $$SSum \equiv s, i : \{ s' = \sum_{j=1}^{n} j \} .$$

- Our program:
  $$PSum \equiv s := 0; i := n; (i > 0) \triangleright (s := s + i; i := i - 1)$$

- Our goal, to prove the program satisfies our specification:
  $$SSum \sqsubseteq PSum$$

- We assume all variables are natural numbers\(^1\).

\(^1\)a.k.a. “unsigned int”
A Simple example (II)

- Our strategy, to split the spec and program into two parts: initialisation and the loop, and do these separately:

\[
\begin{align*}
S\text{Init} & \sqsubseteq P\text{Init} \\
S\text{Loop} & \sqsubseteq P\text{Loop}
\end{align*}
\]

- Here, we have

\[
\begin{align*}
P\text{Init} & \triangleq s := 0; i := n \\
P\text{Loop} & \triangleq (i > 0) \land (s := s + i; i := i - 1)
\end{align*}
\]

- We then use the following law of refinement (\(\sqsubseteq\)) to complete:

\[
(S_1 \sqsubseteq P_1) \land (S_2 \sqsubseteq P_2) \Rightarrow (S_1 ; S_2) \sqsubseteq (P_1 ; P_2)
\]

- The question now is, what are \(S\text{Init}\) and \(S\text{Loop}\)?

A Simple example (III — \(S\text{Init} \sqsubseteq P\text{Init}\))

- We take a simple approach here, setting \(S\text{Init} = P\text{Init}\)

\[
\begin{align*}
P\text{Init} & = \text{" defn. "} \\
s := 0; i := n \\
P\text{Loop} & = \text{" sim-:=-merge"} \\
s, i := 0, n \\
& = \text{" sim-:=-def"} \\
s' = 0 \land i' = n \land n' = n \\
& = \text{" by design "}
\end{align*}
\]

A Simple example (IV — \(S\text{Loop} \sqsubseteq P\text{Loop}\))

- Our loop specification says we sum on top of starting \(s\) value:

\[
S\text{Loop} \triangleq s, i[; s' = s + S(i)]
\]

- But what is \(P\text{Loop}\)?
- We need to find \(W\) such that

\[
W = (s := s + i; i := i - 1; W) < i > 0 \triangleright \text{skip}
\]

- Do we guess? informed guess? informed by what?

A Simple example (V — “guessing” \(W\))

- We suggest the following definition for \(W\):

\[
W \triangleq i' = 0 \land s' = s + S(i) \land n' = n
\]

- It iterates until \(i = 0\), so hence \(i' = 0\).
- It sums from starting \(i\) down to 1
- It adds on top of the starting value of \(s\)
- It does not change \(n\).
A Simple example (VI.a — simplifying \( W \))

- It is useful to do some pre-computation:
  \[
  i = 0 \land W \\
  = \quad \text{"defn. } W \text{"} \\
  i = 0 \land i' = 0 \land s' = s + S(i) \land n' = n \\
  = \quad \text{"equality substitution"} \\
  i = 0 \land i' = 0 \land s' = s + S(0) \land n' = n \\
  = \quad \text{"arithmetic"} \\
  i = 0 \land i' = 0 \land s' = s \land n' = n \\
  = \quad \text{"skip-def"} \\
  i = 0 \land skip
  \]

A Simple example (VI.b — simplifying \( W \))

- More pre-computation (here ignoring \( n \) and \( n' \) for brevity):
  \[
  i > 0 \land (s := s + i; i := i - 1; W) \\
  = \quad \text{"defn. } W \text{"} \\
  i > 0 \land (s' = s + i \land i' = i - 1; \\
  \quad \text{if } i' = 0 \land s' = s + S(i)) \\
  = \quad \text{"\langle \langle \text{def}, \ substitute \rangle \rangle"} \\
  i > 0 \land i' = 0 \land s' = s + i \land i_m = i - 1; \\
  \quad \text{if } i' = 0 \land s' = s_m + S(i_m)) \\
  = \quad \text{"\langle \langle \exists-1pt \rangle \rangle"} \\
  i > 0 \land i' = 0 \land s' = s + i + S(i - 1) \\
  = \quad \text{"arithmetic"} \\
  i > 0 \land i' = 0 \land s' = s + S(i) \\
  = \quad \text{"defn. } W \text{"} \\
  i > 0 \land W
  \]

A Simple example (VII — checking \( W \))

- We need to check our \( W \) is a fixed-point:
  \[
  \langle s := s + i; i := i - 1; W \rangle \land i > 0 \triangleright skip \\
  = \quad \text{"\langle \langle \triangleright \rangle \rangle-def, \ sim:=:merge"} \\
  i > 0 \land (s, i := s + i, i - 1; W) \lor i = 0 \land skip \\
  = \quad \text{"previous calculations"} \\
  i > 0 \land W \lor i = 0 \land W \\
  = \quad \text{"\langle \langle \lor \rangle \rangle-distr"} \\
  (i > 0 \lor i = 0) \land W \\
  = \quad \text{"\langle \langle excluded-middle \rangle \rangle, noting } i : \mathbb{N} \text{"} \\
  \textbf{true} \land W \\
  = \quad \text{"\langle \langle \text{unit} \rangle \rangle"} \\
  W
  \]

A Simple example (VIII — \( SLoop \subseteq W \))

- We need to show that \( W \), our semantics for \( PLoop \) does in fact refine the loop specification.
  \[
  SLoop \subseteq W \\
  = \quad \text{"defns."} \\
  \left[ i' = 0 \land s' = s + S(i) \land n' = n \Rightarrow \right. \left[ s, i : [s' = s + S(i)] \right] \\
  = \quad \text{"\langle \langle \text{frame-def} \rangle \rangle"} \\
  \left[ i' = 0 \land s' = s + S(i) \land n' = n \Rightarrow \right. \left[ s' = s + S(i) \land n' = n \right] \\
  = \quad \text{"prop. law } [A \land B \Rightarrow A] \text{"} \\
  \textbf{true}
  \]

CS4003 Formal Methods
A Simple example (IX — final assembly)

- We have $S_{\text{Init}} = P_{\text{Init}}$ and $S_{\text{Loop}} \subseteq P_{\text{Loop}}$, so we can conclude by $\langle\subseteq;\rangle$ that $S_{\text{Init}}; S_{\text{Loop}} \subseteq P_{\text{Init}}; P_{\text{Loop}}$.
- However, we need to show that $SSum = S_{\text{Init}}; S_{\text{Loop}}$

$$S_{\text{Init}}; S_{\text{Loop}} = \text{“ defns. ”}$$
$$s' = 0 \land i' = n \land n' = n; s' = s + S(i) \land n' = n$$
$$= \text{“ \{ -def \}, subst. ”}$$
$$\exists s_m, i_m, n_m \bullet s_m = 0 \land i_m = n \land n_m = n$$
$$\land s' = s_m + S(i_m) \land n' = n_m$$
$$= \text{“ \{\exists-1pt\} ”}$$
$$s' = 0 + S(n) \land n' = n$$
$$= \text{“ arithmetic, \{ frame-def \} ”}$$
$$s, i : [s' = S(n)]$$

Uuuhh !

- We showed that $S_{\text{Loop}} \subseteq P_{\text{Loop}}$
- It was an ad-hoc approach
- Complicated, long-winded
- Is there a better, more systematic way？

A Simple example (revisited — I)

- Consider two cases:
  1. the loop is continuing, as $i > 0$, so we have
     $$i > 0 \Rightarrow W = s := s + i; i := i - 1; W$$
  
  2. the loop has terminated, as $i = 0$, so we have
     $$i = 0 \Rightarrow W = \text{skip}$$

- Whilst the loop is continuing, we see that $W$ is in some sense doing the same thing each time round
- Somehow $W$ must embody a property true before and after each iteration: the loop invariant ($inv$).

A Simple example (revisited — II)

- Consider partway through, where $0 < i < n$.
  - We have summed from $n$ down to $i + 1$.
  - So the final sum should be the current value of $s$, plus the remaining numbers to be added, namely from $i$ down to 1.
  - We posit: $inv \triangleq S(n) = s + S(i)$

- At the start, $i = n \land s = 0$, so $inv$ holds
- At the end, $i = 0 \land inv$, so $s = S(n)$ holds
A Simple example (revisited — III)

- From where did we magic this invariant?

\[ s + S(i) = S(n) \]

- Considering execution snapshots often helps

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
i & n & n-1 & n-2 & k & 0 \\
\hline
s & 0 & n & \quad & \quad & \quad \\
\hline
S(i) & S(n) & S(n-1) & S(n-2) & S(k) & S(0) \\
\hline
\end{array}
\]

As \( i \) shrinks, \( s \) increases

Loops and Invariants

- If \( \text{inv} \) is an invariant for \( c \circ P \), then the following holds:
  - If \( c \) and \( \text{inv} \) are true, then execution of the loop body will leave \( \text{inv}' \) true.

\[ \langle \text{rel-is-invariant} \rangle \quad \vec{v} : [c \land \text{inv} \Rightarrow \text{inv}'] \subseteq P \]

  - If \( \text{inv} \) is true beforehand, then on termination of the loop, \( \text{inv}' \) is true, as is \( \neg c' \)

\[ \langle \text{rel-\neg} \rangle \quad \vec{v} : [\text{inv} \Rightarrow \text{inv}' \land \neg c'] \subseteq c \odot (\vec{v} : [c \land \text{inv} \Rightarrow \text{inv}']) \]

  - Note that we have to carry the same frame \( \vec{v} : [\ldots] \) through the refinement.
  - So, we have two refinement checks:
    - \( \langle \text{rel-is-invariant} \rangle \) checks that \( \text{inv} \) is an invariant for \( c \circ P \)
    - \( \langle \text{rel-\neg} \rangle \) states how we can refine such a loop

Loop Refinement (I)

- We can put two refinements together into a refinement inference rule:

\[
\begin{align*}
\vec{v} : [c \land \text{inv} \Rightarrow \text{inv}'] & \subseteq P \\
\vec{v} : [\text{inv} \Rightarrow \text{inv}' \land \neg c'] & \subseteq c \odot (\vec{v} : [c \land \text{inv} \Rightarrow \text{inv}']) \\
\vec{v} : [\text{inv} \Rightarrow \text{inv}' \land \neg c'] & \subseteq c \odot P
\end{align*}
\]

  - In other words, in order to show that \( S \circ c \circ P \), we need to:
    1. Look at \( c \) and \( P \) to find an appropriate invariant \( \text{inv} \), satisfying \( \langle \text{rel-is-invariant} \rangle \).
    2. Show \( S \) is refined by \( \text{inv} \Rightarrow \text{inv}' \land \neg c' \)
    3. Show that \( \langle \text{rel-\neg} \rangle \) holds.

  - Proof of these rules is (slightly) beyond the scope of this course.

Invariants

- For a while-loop \( c \odot P \), an invariant \( \text{inv} \) is:
  - a condition (undashed variables only)
  - required to be true before we execute the loop
  - will be true after each iteration around the loop
  - will be true on exit from the loop

  - Such conditions are called “invariants” because they state a property that does not \textit{not} change as the loop executes.

  - Surprisingly, the loop invariant is key to proving that a loop does the right thing!

  - A “variant” predicate, describing how \( P \) changes things in each iteration does not help in this regard!!
**Loop Refinement (II)**

- However, using invariants requires that $inv$ holds at loop start.
  We need to ensure that initialization brings this about.
- Another refinement rule $\langle\langle \text{rel-init-loop} \rangle\rangle$ allows us to couple both initialization and the loop:

$$\vec{v} : \{\text{pre} \Rightarrow inv \wedge \neg c'\} 
\sqsubseteq \vec{v} : \{\text{pre} \Rightarrow inv'\} ; c \otimes \vec{v} : \{c \wedge inv \Rightarrow inv'\}$$

- Given refinement check $S \sqsubseteq c \otimes P$, we no longer look for an appropriate $W$.
- Instead we have to find $inv$, which is simpler.

**The Simple example (redone — I)**

- Reminder: we want to show

$$s, i : \{s' = S(n)\} \sqsubseteq \begin{cases} s, i := 0, n : (i > 0) \otimes (s, i := s + i, i - 1) \end{cases}$$

using $inv = s + S(i) = S(n)$.
- We need to get the specification into the correct form, which requires the very useful, very general refinement inference rule $\langle\langle \text{strengthen-post} \rangle\rangle$:

$$\vec{v} : \{\text{Post}_1\} \sqsubseteq \vec{v} : \{\text{Post}_2\} \sqsubseteq \vec{v} : \{\text{pre} \Rightarrow \text{Post}_1\} \sqsubseteq \vec{v} : \{\text{pre} \Rightarrow \text{Post}_2\}$$

(The rationale for this law will follow shortly)

**The Simple example (redone — II)**

We can now proceed:

$$s, i : \{s' = S(n)\} = \langle\langle \Rightarrow\text{-1-unit}\rangle\rangle$$

$$s, i : \{\text{true} \Rightarrow s' = S(n)\} \sqsubseteq \langle\langle \text{strengthen-post} \rangle\rangle, \text{using } \vec{x} : \{A\} \sqsubseteq \vec{x} : \{A \wedge B\}$$

$$s, i : \{\text{true} \Rightarrow s' = S(n) \wedge i' = 0\} = \langle\langle \text{rel-init-loop} \rangle\rangle$$

$s, i : \{\text{true} \Rightarrow s' + S(i') = S(n') \wedge i' = 0\}$

$s, i : \{\text{true} \Rightarrow s' + S(i') = S(n')\}$

: $(i > 0) \otimes s, i : \{i > 0 \wedge s + S(i) = S(n) \Rightarrow s' + S(i') = S(n')\}$

At this point we can refine the two statements individually.

**Reminder: “equality substitution”**

- We have the law $\langle\langle \Rightarrow\text{-A-subst}\rangle\rangle$:

$$x = e \wedge P \equiv x = e \wedge P[e/x]$$

- From this we can derive many useful laws, e.g.

$$\langle\langle \Rightarrow\text{-\neg-subst}\rangle\rangle (x = e \wedge P \Rightarrow Q) \equiv (x = e \wedge P \Rightarrow Q[e/x])$$

$$\langle\langle \Rightarrow\text{-\neg-subst}\rangle\rangle (S \sqsubseteq P \wedge x = e) \equiv (S[e/x] \sqsubseteq P \wedge x = e)$$

- Remember also, that the laws above also apply even if we don’t replace all free $x$ with $e$, but instead only change some.
Intuition for “equality substitution”

- In essence the equality substitution laws just mentioned express when we can propagate an identity \((e_1 = e_2)\) around the rest of its containing predicate.
- by “propagating \(e_1 = e_2\)” we mean identifying places in the containing predicate where we can replace \(e_1\) by \(e_2\), or v.v.
- Identities propagate through logical-and \((\land)\).
- Identities propagate leftwards through implication \((\Rightarrow)\), i.e. following the arrow.

The Simple example (redone — III)

The first statement

\[
\begin{align*}
\text{true} & \Rightarrow s' + S(i') = S(n') \\
\iff & \quad s, i := 0, n \\
& = \quad \text{sim:-def} \\
& \quad s' + S(i') = S(n') \land n' = n \\
& \iff \quad 0 \land i' = n \land n' = n \\
& = \quad \text{arith} \\
& \quad s' = 0 \land i' = n \land n' = n \\
& \Rightarrow \quad 0 + S(n) = S(n) \land n = n \\
& = \quad \text{arith} \\
& \quad s' = 0 \land i' = n \land n' = n \Rightarrow \text{true} \\
& = \quad \text{arith-r-zero} \\
& \text{true}
\end{align*}
\]

The Simple example (redone — IV)

The second statement

\[
\begin{align*}
& s, i \mid [i > 0 \land s + S(i) = S(n) \Rightarrow s' + S(i') = S(n')] \\
& \iff s, i := s + i, i - 1 \\
& = \quad \text{frame-def, sim:-def} \\
& (i > 0 \land s + S(i) = S(n) \Rightarrow s' + S(i') = S(n')) \land n' = n \\
& \iff s' = s + i \land i' = i - 1 \land n' = n \\
& = \quad \text{arith} \\
& \quad (i > 0 \land s + S(i) = S(n) \Rightarrow s + i + S(i - 1) = S(n)) \land n = n \\
& \iff s' = s + i \land i' = i - 1 \land \text{true} \\
& = \quad \text{arith}, \text{esp laws of } S(x) \\
& (i > 0 \land s + S(i) = S(n) \Rightarrow s + S(i) = S(n)) \\
& \iff s' = s + i \land i' = i - 1 \land n' = n \\
& = \quad \text{prop. calc } A \land B \Rightarrow A \\
& \text{true} \iff s' = s + i \land i' = i - 1 \land n' = n \\
& = \quad \text{true (Chaos) is refined by anything} \\
& \text{true}
\end{align*}
\]

Refining by Initialised While-Loops: recap

To refine \(\vec{v} : [S] \subseteq \text{init} : c \land P\) we need to:

1. find an appropriate invariant \(\text{inv}\)
2. find an appropriate pre-condition \(\text{pre}\) (often \text{true} will do)
3. show that \(\vec{v} : [S] \subseteq \vec{v} : [\text{pre} \Rightarrow \text{inv'} \land \neg c']\)
4. Then it remains to verify both

\[
\begin{align*}
\vec{v} : [\text{pre} \Rightarrow \text{inv'}] & \subseteq \text{init} \\
\vec{v} : [c \land \text{inv} \Rightarrow \text{inv'}] & \subseteq P
\end{align*}
\]
Invariants: the last words

- Invariants capture a property that is true before and after every iteration of the loop body
- However, the existence of an invariant, does not mean that nothing changes going round the loop
- Clearly something must change or the loop would not achieve its goal (or terminate).
- The trick in verifying loops is finding the appropriate invariant.
We have seen a few specifications now of the form $P \Rightarrow Q$.

Typically $P$ has been a condition (no dashed-variables) whilst $Q$ may have had both dashed and non-dashed variables.

We shall use lower-case names for predicates that are conditions:

- $p$ : a pre-condition (no dashed variables)
- $p'$ : a post-condition (only dashed variables)
- $P$ : a pre-post-relation (mixed variables)

Interpreting Implications (I)

- The specification $pre \Rightarrow post'$
  - is interpreted as "if $pre$ holds at the start, the we must ensure that $post$ holds at the end".
  - relates a "pre-snapshot" to a "post-snapshot"
- The specification $pre \Rightarrow Rel$
  - is interpreted as "if $pre$ holds at the start, then we must ensure that start and end states are related by $Rel$".
  - relates a "pre-snapshot" to a relationship between before and after states

Interpreting Implications (II)

- Given specification $pre \Rightarrow Rel$ what should our refinement do, in situations were $pre$ is $False$?
- Guiding principle: guarantees and commitments
- The pre-condition $pre$ is a guarantee that some other agent will set things up appropriately.
- Given that guarantee has been met, then $Rel$ is the commitment that any refinement must make.
- But, but, if the guarantee $isn't$ met ?
- If the pre-condition is $False$ we can do anything.
  - in general we cannot offer anything better than that consider: $x, r : \mathbb{R} \land x \geq 0 \Rightarrow (r')^2 = x$
  - what can we do if $x < 0$?
Loop Refinement Example 1 (Floyd ’67)

Given

\[ \begin{align*}
    a & : \text{array } 1 \ldots n \text{ of } \mathbb{R} \\
    A(0) & = 0 \\
    A(x) & = a_x + A(x - 1), \ x > 0
\end{align*} \]

\[ A_{\text{Spec}} \triangleq s, i : [s' = A(n)] \]

\[ \alpha A_{\text{Prog}} = \{n, i, s, n', i', s'\} \]

\[ A_{\text{Prog}} \triangleq i, s := 1, 0; (i \leq n) \ast (s, i := s + a_i, i + 1) \]

Prove

\[ A_{\text{Spec}} \subseteq A_{\text{Prog}} \]

Refining Floyd’67: process

We need to
1. find appropriate invariant \( a_{\text{inv}} \)
2. show \( A_{\text{Spec}} \subseteq s, i : [a_{\text{inv}}' \land i' > n'] \)
   (precondition is \textbf{true}, so we ignore it)
3. show \( s, i : [a_{\text{inv}}'] \subseteq i, s := 1, 0 \)
4. show \( s, i : [i \leq n \land a_{\text{inv}} \Rightarrow a_{\text{inv}}'] \subseteq s, i := s + a_i, i + 1 \)

Refining Floyd’67: process (1)

Looking at loop execution:

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>( k )</th>
<th>( n )</th>
<th>( n + 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s )</td>
<td>0</td>
<td>( a_1 )</td>
<td>( a_1 + a_2 )</td>
<td>( a_1 + \cdots a_{k-1} )</td>
<td>( a_1 + \cdots a_{n-1} )</td>
<td>( a_1 + \cdots a_n )</td>
</tr>
<tr>
<td>( A(0) )</td>
<td>( A(1) )</td>
<td>( A(2) )</td>
<td>( A(k - 1) )</td>
<td>( A(n - 1) )</td>
<td>( A(n) )</td>
<td></td>
</tr>
</tbody>
</table>

We can see that for \( i = k \) that \( s + \sum_{j=k}^{n} a_j = A(n) \).

\[ a_{\text{inv}} \triangleq s + \sum_{j=i}^{n} a_j = A(n) \]

Refining Floyd’67: process (1x)

We introduce a shorthand, with laws:

\[ \begin{align*}
    A(\ell, u) & \triangleq \sum_{j=\ell}^{u} a_j \\
    A(k, k) & = a_k \\
    A(\ell, u) & = 0, \ \ell > u \\
    A(\ell, u) & = a_\ell + A(\ell + 1, u), \ \ell \leq u \\
    A(1, n) & = A(n), \ n > 0
\end{align*} \]

Now the invariant is more concise

\[ a_{\text{inv}} \triangleq s + A(i, n) = A(n) \]
Refining Floyd’67: process (2)

We must show $\text{ASpec} \subseteq s, i : [\text{ainv'} \land i' > n']$

\[
s, i : [s' = A(n)] \subseteq s, i : [s' + A(i', n') = A(n') \land i' > n'] \]

- “\text{frame-def}”
\[
s' = A(n) \land n' = n \subseteq s' + A(i', n') = A(n') \land i' > n' \land n' = n
\]

- “$A(i', n') = 0$ if $i' > n'$”
\[
s' = A(n') \land n' = n \subseteq s' = A(n') \land i' > n' \land n' = n
\]

- “\text{prog-strengthen}”
\[
s' = A(n') \land n' = n \subseteq s' = A(n') \land i' > n' \land n' = n
\]

- “\text{prog-strengthen}”
\[
\text{true}
\]

Refining Floyd’67: process (3)

We must show $s, i : [\text{ainv'}] \subseteq i, s := 1, 0$

\[
s, i : [s' + A(i', n') = A(n')] \subseteq i, s := 1, 0
\]

- “\text{frame-def}”
\[
s' + A(i', n') = A(n') \land n' = n \subseteq i' = 1 \land s' = 0 \land n' = n
\]

- “\text{sim-def}”
\[
0 + A(1, n') = A(n') \land n' = n \subseteq i' = 1 \land s' = 0 \land n' = n
\]

- “\text{true}”
\[
\text{true}
\]

Refining Floyd’67: process (4)

We must show $s, i : [i \leq n \land \text{ainv} \Rightarrow \text{ainv'}]$  $\subseteq s, i := s + a_i, i + 1$

\[
s, i : [i \leq n \land s + A(i, n) = A(n) \Rightarrow s' + A(i', n') = A(n')] \subseteq s, i := s + a_i, i + 1
\]

- “\text{sim-def}”
\[
s, i : [i \leq n \land s + A(i, n) = A(n) \Rightarrow s' + A(i', n') = A(n')] \subseteq s' = s + a_i \land i' = i + 1 \land n' = n
\]

- “\text{prog-subst}”
\[
s, i : [i \leq n \land s + A(i, n) = A(n) \Rightarrow s + A(i + 1, n) = A(n)] \subseteq s' = s + a_i \land i' = i + 1 \land n' = n
\]

- “\text{true}”
\[
s, i : [i \leq n \land s + A(i, n) = A(n) \Rightarrow s + A(i, n) = A(n)] \subseteq s' = s + a_i \land i' = i + 1 \land n' = n
\]

Refining Floyd’67: process (4 cont.)

We must show $s, i : [i \leq n \land s + A(i, n) = A(n) \Rightarrow s + A(i, n) = A(n)]$

\[
s, i : [i \leq n \land s + A(i, n) = A(n) \Rightarrow s + A(i, n) = A(n)] \subseteq s' = s + a_i \land i' = i + 1 \land n' = n
\]

- “\text{true}”
\[
\text{true}
\]
Loop Refinement Example 2 (Hoare ’69)

Given

\[ HSpec \equiv r, q :: [y' > r' \land x' = r' + y' \cdot q'] \]
\[ HProg \equiv r, q := x, 0 ; (y \leq r) \ast (r, q := r - y, 1 + q) \]

Prove

\[ HSpec \sqsubseteq HProg \]

Refining Hoare’69: process

We need to

1. find appropriate invariant \( hinv \)
2. show \( HSpec \sqsubseteq r, q :: [hinv' \land y' > r'] \)
   (precondition is \( \text{true} \), so we ignore it)
3. show \( r, q :: [hinv'] \sqsubseteq r, q := x, 0 \)
4. show \( r, q :: [y \leq r \land hinv \Rightarrow hinv'] \sqsubseteq r, q := r - y, 1 + q \)

Refining Hoare’69: process (1)

Looking at loop execution:

<table>
<thead>
<tr>
<th></th>
<th>( r )</th>
<th>( x )</th>
<th>( x - y )</th>
<th>( x - 2y )</th>
<th>( x - k \cdot y )</th>
<th>( x - n \cdot y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>( k )</td>
<td>( n )</td>
<td></td>
</tr>
</tbody>
</table>

We can see that for \( q = k \) that \( r = x - k \cdot y \).

\[ hinv \equiv x = r + q \cdot y \]

Refining Hoare’69: process (2)

We must show \( HSpec \sqsubseteq r, q :: [hinv' \land y' > r'] \)

\[ r, q :: [y' > r' \land x' = r' + y' \cdot q'] \]
\[ r, q :: [x' = r' + q' \cdot y' \land y' > r'] \]
\[ \sqsubseteq \text{“} \triangleleft \text{-refl} \text{”} \]
\[ \text{true} \]
Refining Hoare’69: process (3)

We must show \( r, q \vdash \langle \text{hinv} \rangle \subseteq r, q := x, 0 \)

\[
\begin{align*}
  r, q \vdash [x'] & = r' + q' \ast y' \subseteq r, q := x, 0 \\
  & = \text{"@frame-def@, @sim:=@def"} \\
  x' & = r' + q' \ast y' \land x' = x \land y' = y \\
  & \subseteq r' = x \land q' = 0 \land x' = x \land y' = y \\
  & = \text{"arithmetic"} \\
  \text{true} & \land x' = x \land y' = y \\
  & \subseteq r' = x \land q' = 0 \land x' = x \land y' = y \\
  & = \text{"@prog-strengthen@"} \\
  \text{true} & 
\end{align*}
\]

Refining Hoare’69: process (4 cont.)

\[
\begin{align*}
  (y \leq r \land x & = r + q \ast y) \\
  \Rightarrow x & = r - y + (q + 1) \ast y \land x' = x \land y' = y \\
  & \subseteq r' = r - y \land q' = 1 + q \land x' = x \land y' = y \\
  & = \text{"arithmetic"} \\
  (y \leq r \land x & = r + q \ast y) \land x' = x \land y' = y \\
  & \subseteq r' = r - y \land q' = 1 + q \land x' = x \land y' = y \\
  & = \text{"A \land B \Rightarrow B"} \\
  \text{true} & \land x' = x \land y' = y \\
  & \subseteq r' = r - y \land q' = 1 + q \land x' = x \land y' = y \\
  & = \text{"@prog-strengthen@"} \\
  \text{true} & 
\end{align*}
\]

Refining Hoare’69: process (4)

We must show \( r, q \vdash y \leq r \land \text{hinv} \Rightarrow \text{hinv} \subseteq r, q := r - y, 1 + q \)

\[
\begin{align*}
  r, q \vdash [y \leq r \land x & = r + q \ast y \Rightarrow x' = r' + q' \ast y'] \\
  & \subseteq r, q := r - y, 1 + q \\
  & = \text{"@frame-def@, @sim:=@def"} \\
  (y \leq r \land x & = r + q \ast y \Rightarrow x' = r' + q' \ast y') \land x' = x \land y' = y \\
  & \subseteq r' = r - y \land q' = 1 + q \land x' = x \land y' = y \\
  & = \text{"@prog-strengthen@"} \\
  (y \leq r \land x & = r + q \ast y \Rightarrow x = r - y + (q + 1) \ast y) \land x' = x \land y' = y \\
  & \subseteq r' = r - y \land q' = 1 + q \land x' = x \land y' = y 
\end{align*}
\]

One More Loop Example

- We shall finish off with one more different loop example
- A loop whose termination is not pre-determined
  - e.g., searching an array
  - \( p : [a_p = w < w \in a \Rightarrow p' = 0] \)
Searching an Array

\[
a : \text{array } 1 \ldots N \text{ of } T
\]
\[
w : T
\]
\[
p : N
\]
\[
\text{ASSpec} \triangleq p : [a_p = w \land w \in a \land p' = 0]
\]
\[
\text{ASProg} \triangleq p := N ; (a_p \neq w \land p > 0) * p := p - 1
\]
\[
\text{ASSpec} \sqsubseteq \text{ASProg}
\]

The Proof obligations

1. find an appropriate invariant \( asinv \)
2. Prove \( \text{ASSpec} \sqsubseteq p : [asinv' \land \neg (a_p \neq w \land p' > 0)] \)
3. Prove \( p : [asinv'] \sqsubseteq p := N \)
4. Prove \( p : [a_p \neq w \land p > 0 \land asinv \Rightarrow asinv'] \sqsubseteq p := p - 1 \)
5. For simplicity, we shall assume \( w' = w \) and \( a' = a \) throughout, so we can ignore the frames.

Array Stuff

- We need to have some laws/definitions regarding array membership:
  \[
w \in a \triangleq \exists k \cdot 1 \leq k \leq N \land a_k = w
\]
- We also want to talk about membership in a subset of an array:
  \[
w \in a_{\ell \ldots u} \triangleq \exists k \cdot \ell \leq k \leq u \land a_k = w, \quad 1 \leq \ell \leq u \leq N
\]
  \[
w \in a_{\ell \ldots u} = w = a_\ell \lor w \in a_{\ell + 1 \ldots u}, \quad \ell \leq u
\]
  \[
w \in a_{\ell \ldots u} = \text{false}, \quad \ell > u
\]

Picking the invariant

- An overview of a run of the algorithm looks like:
  \[
  \begin{array}{cccccccc}
  0 & 1 & 2 & k & k + 1 & \cdots & N \\
  a_1 & a_2 & \cdots & a_k & a_{k + 1} & \cdots & a_N
  \end{array}
  \]
- When \( p = k \), we have come down from \( N \), without finding \( w \).
- We also know that \( p \) will range from \( N \) down to 0
- We propose \( asinv = 0 \leq p \leq N \land w \notin a_p + 1 \ldots N \)
The Proof obligations (revisited)

1. We drop frames, assume $a = a'$ and $w = w'$ throughout
2. Prove
   \[
   \text{ASSpec} \quad \subseteq \quad 0 \leq p' \leq N \land w \notin a_{p' + 1} \ldots N \land (a_{p'} \neq w \land p' > 0)
   \]
3. Prove $0 \leq p' \leq N \land w \notin a_{p' + 1} \ldots N \subseteq p := N$
4. Prove
   \[
   a_p \neq w \land p > 0 \land 0 \leq p \leq N \land w \notin a_{p + 1} \ldots N \\
   \Rightarrow 0 \leq p' \leq N \land w \notin a_{p' + 1} \ldots N \\
   \subseteq p := p - 1
   \]
5. Left as exercise for the very keen!

Wrong Choice?

- Maybe we chose wrong earlier — should $\text{Forever}$ be $\text{miracle}$?

- $\text{miracle} : P$
  - “defn. $\text{miracle}$”
- $\text{false} : P$
  - “{;};-def”
  - $\exists S_m \bullet \text{false}[S_m/S'] \land P[S_m/S]$
  - “substitution”
  - $\exists S_m \bullet \text{false} \land P[S_m/S]$
  - “{;};-zero”
  - $\exists S_m \bullet \text{false}$
  - “{exists-free}, defn. $\text{miracle}$”
  - $\text{miracle}$

- This works — but we ruled it out because it refines any specification.

Uh-Oh! Heouston, we have a problem ...

- Clearly $\text{Forever} : P$ = $\text{Forever} = P$, right? (Remember, $\text{Forever}$ is $\text{true} \neq \text{skip}$.)

  - $\text{Forever} : P$
    - “semantics of $\text{Forever}$”
    - $\text{true} : P$
      - “{;};-def”
      - $\exists S_m \bullet \text{true}[S_m/S'] \land P[S_m/S]$
      - “substitution”
      - $\exists S_m \bullet \text{true} \land P[S_m/S]$
      - “{;};-unit”
      - $\exists S_m \bullet P[S_m/S]$
      - “{exists-free}, defn. $\text{miracle}$”
      - $\exists S \bullet P$

- This doesn’t reduce to $\text{true}$ as expected.
- We get an execution of $P$ in an arbitrary starting state!
Partial vs. Total Correctness

- We have a theory of *partial correctness*.
- Right now, a proof of
  \[ \text{Spec} \sqsubseteq \text{Prog} \]
  proves that \( \text{Prog} \) satisfies \( \text{Spec} \), if it terminates.
- We want a theory of *total correctness*.
- In other words, a proof of
  \[ \text{Spec} \sqsubseteq \text{Prog} \]
  should prove that \( \text{Prog} \) terminates and satisfies \( \text{Spec} \).

What’s the problem regarding termination?

- We can’t observe it!
- All our theory allows us to do is:
  - observe variable values at the start \((A)\)
  - observe variable values at the end \((B)\)
  - relate these observations \((C)\)
- For \((B)\) and \((C)\) it is necessary that the program terminates.
- In other words, a hidden (wrong!) assumption in our theory so far is that all programs terminate!

When things go bad: Divergence

- A program execution is divergent if
  - a serious error has occurred
  - the error is unrecoverable
  - future system behaviour is effectively unpredictable
- In our theory to date, the predicate \textbf{true} (a.k.a. \textit{Forever}) captures such unpredictability
- In our theory so-far of (sequential) programming, non-termination is an instance of divergence.
- Divergence is sometimes also referred to as "Instability".
Observe (non-)Divergence

- We shall extend our theory by allowing divergence/non-divergence to be an observable notion.
- We introduce a new variable: \( \text{ok} \)
  - \( \text{ok} = \text{True} \) program is non-divergent (a.k.a. “stable”)
  - \( \text{ok} = \text{False} \) program is diverging (a.k.a. “unstable”)
- Variable \( \text{ok} \) is not a program variable
  - it is an auxiliary or model variable.
  - We assume no program has such a variable
  - \( \text{ok} \notin S \)
- As with program variable, we distinguish before- and after-execution:
  - \( \text{ok} \) — stability/non-divergence at start
    (i.e. of “previous” program)
  - \( \text{ok}' \) — stability/non-divergence at end

Using \( \text{ok} \) and \( \text{ok}' \)

- Consider the specification: \( \text{pre} \Rightarrow \text{Post} \)
  - i.e. if \( \text{pre} \) holds at start, the \( \text{Post} \) relates start and end
    (provided the program stops)
- We shall now replace the above by:
  - \( \text{ok} \land \text{pre} \Rightarrow \text{ok}' \land \text{Post} \)
  - if started in a stable state \( \text{ok} \)
    - and \( \text{pre} \) holds at start
    - then, we end in a stable state \( \text{ok}' \)
      - and the relation \( \text{Post} \) holds of starting and ending states.
- For sequential imperative programs, stability is termination
  - Important: We assume that neither \( \text{pre} \) nor \( \text{Post} \) mention \( \text{ok} \) or \( \text{ok}' \).
- Also, variables \( \text{ok} \) and \( \text{ok}' \) are added to the alphabet of every language construct.

The Meaning of \( \text{ok} \land \text{pre} \Rightarrow \text{ok}' \land \text{Post} \)

Given this specification, then for any run of a satisfying program:
- If \( \text{ok} = \text{False} \), i.e. we started in a divergent state
  (previous program diverged)
  and then any subsequent behaviour was acceptable.
- If \( \text{pre} = \text{False} \), then anything was also allowed.
- If \( \text{ok} = \text{pre} = \text{True} \), then \( \text{ok}' \) must have been true
  (the program terminated without diverging), and relation \( \text{Post} \) was satisfied.

Program Semantics using \( \text{ok} \) and \( \text{ok}' \)

- We can now give our program language semantics in this style:
  - \( \langle \langle \text{skip-def} \rangle \rangle \) \( \text{skip} \triangleq \text{ok} \Rightarrow \text{ok}' \land \text{S}' = \text{S} \)
  - \( \langle \langle \text{:=def} \rangle \rangle \) \( x := e \triangleq \text{ok} \Rightarrow \text{ok}' \land x' = e \land \text{S}' \backslash x = \text{S} \backslash x \)
  - \( \langle \langle \text{;def} \rangle \rangle \) \( P; Q \triangleq \exists m, \text{ok}_m \bullet P[S_m, \text{ok}_m / S', \text{ok}'] \land Q[S_m, \text{ok}_m / S, \text{ok}] \)
- The rules regarding alphabets are unchanged, remembering that \( \text{ok} \) and \( \text{ok}' \) now belong.
- All other language constructs are defined as before, except for the while-loop, which has some differences.
- Many of the laws remain unchanged.
(Some) Laws of Programming

- The followings laws, seen before, all still hold.
  - `<skip-alt>` \( \text{skip}_A = x := x \)
  - `<;:assoc>` \( P ; (Q ; R) = (P ; Q) ; R \)
  - `<;:seq>` \( x := e ; x := f = x := f[e/x] \)
  - `<;:swap>` \( x := e ; y := f = y := f[e/x] ; x := e, y \notin e \)
  - `<(/>-true>` \( P \triangleleft \text{True} \triangleright Q = P \)
  - `<(/>-false>` \( P \triangleleft \text{False} \triangleright Q = Q \)
  - `<(/>-seq>` \( (P \triangleleft c \triangleright Q) ; R = (P ; R) \triangleleft c \triangleright (Q ; R) \)

Laws requiring the new form

- If \( P \) has the (new) form \( \text{ok} \land \text{pre} \Rightarrow \text{ok}' \land \text{Post} \)

  then the following laws also hold:
  - `<skip:-unit>` \( \text{skip}; P = P \)
  - `<;:skip-unit>` \( P ; \text{skip} = P \)

- However, they are no longer true for arbitrary \( P \).
- This occurs with laws involving assignment and `skip` whose definitions have changed.

Proof of `<skip;-unit>` (Setup)

- Goal \( \text{skip} ; P = P \),
  given new definition for `skip` and \( P \) of the form \( \text{ok} \land \text{pre} \Rightarrow \text{ok}' \land \text{Post} \)
- Strategy: reduce lhs to rhs
  \[ \text{skip} ; \text{ok} \land \text{pre} \Rightarrow \text{ok}' \land \text{Post} = " \text{proof steps ...} " \]
  \[ \text{ok} \land \text{pre} \Rightarrow \text{ok}' \land \text{Post} \]

Proof of `<skip;-unit>` (I)

- \( \text{skip}; \text{ok} \land \text{pre} \Rightarrow \text{ok}' \land \text{Post} \)
  \[ = " \text{new} \langle \text{skip}-\text{def} \rangle \" \]
  \[ \text{ok} \Rightarrow \text{ok}' \land S' = S ; \text{ok} \land \text{pre} \Rightarrow \text{ok}' \land \text{Post} \]
  \[ = " \text{new} \langle ;:-\text{def} \rangle \" \]
  \[ \exists S_m , ok_m \bullet \]
  \[ \text{ok}' \land S' = S ; \text{ok} \land \text{pre} \Rightarrow \text{ok}' \land \text{Post} \]
  \[ = " \text{substitution, noting that ok, ok}' \land \text{pre, Post} \" \]
  \[ \exists S_m , ok_m \bullet \]
  \[ \text{ok}' \land S_m = S ; \text{ok}' \land \text{pre}[S_m/S] \Rightarrow \text{ok}' \land \text{Post}[S_m/S] \]

We have something of the form \( \exists x \bullet (A \Rightarrow B) \land (C \Rightarrow D) \),
which we now transform.
Proof of $\langle \langle \text{skip} - \text{unit} \rangle \rangle$ (II)

$$\exists \text{xs} \cdot (A \Rightarrow B) \land (C \Rightarrow D)$$

"$\Rightarrow$-def"

$$\exists \text{xs} \cdot (\neg A \lor B) \land (\neg C \lor D)$$

"Distributivity laws"

$$\exists \text{xs} \cdot \neg A \land \neg C \lor \neg A \land D \lor B \land \neg C \lor B \land D$$

"$\exists$-distr, several times"

If we apply this to our proof we get

$$(\exists \text{sm}, \text{ok}_m \cdot \neg \text{ok} \land \neg (\text{ok}_m \land \text{pre}[\text{sm}, /\text{sm}])) \lor$$

$$(\exists \text{sm}, \text{ok}_m \cdot \neg \text{ok} \land \text{ok}' \land \text{Post}[\text{sm} /\text{sm}]) \lor$$

$$(\exists \text{sm}, \text{ok}_m \cdot \text{ok}_m \land S_m = S \land \neg (\text{ok}_m \land \text{pre}[\text{sm}, /\text{sm}])) \lor$$

$$(\exists \text{sm}, \text{ok}_m \cdot \text{ok}_m \land S_m = S \land \text{ok}' \land \text{Post}[\text{sm} /\text{sm}])$$

Proof of $\langle \langle \text{skip} - \text{unit} \rangle \rangle$ (III)

- We shall now simplify each of the four components obtained above
- For convenience we give them labels (using "S;U" as short for $\langle \langle \text{skip} - \text{unit} \rangle \rangle$):

  (S;U.1) $\exists \text{sm}, \text{ok}_m \cdot \neg \text{ok} \land \neg (\text{ok}_m \land \text{pre}[\text{sm}, /\text{sm}])$

  (S;U.2) $\exists \text{sm}, \text{ok}_m \cdot \neg \text{ok} \land \text{ok}' \land \text{Post}[\text{sm} /\text{sm}])$

  (S;U.3) $\exists \text{sm}, \text{ok}_m \cdot \text{ok}_m \land S_m = S \land \neg (\text{ok}_m \land \text{pre}[\text{sm}, /\text{sm}])$

  (S;U.4) $\exists \text{sm}, \text{ok}_m \cdot \text{ok}_m \land S_m = S \land \text{ok}' \land \text{Post}[\text{sm} /\text{sm}])$

Proof of $\langle \langle \text{skip} - \text{unit} \rangle \rangle$ (IV, simplifying S;U.1)

$$\exists \text{sm}, \text{ok}_m \cdot \neg \text{ok} \land \neg (\text{ok}_m \land \text{pre}[\text{sm}, /\text{sm}])) \lor$$

"move quantifier in"

$$\neg \text{ok} \land \exists \text{sm}, \text{ok}_m \cdot \neg (\text{ok}_m \land \text{pre}[\text{sm}, /\text{sm}]).$$

"witness: $\text{ok}_m = \text{False}$ makes existential body true"

$$\neg \text{ok} \land \text{true}$$

"simplify"

$$\neg \text{ok}$$

Proof of $\langle \langle \text{skip} - \text{unit} \rangle \rangle$ (V, simplifying S;U.2)

$$\exists \text{sm}, \text{ok}_m \cdot \neg \text{ok} \land \text{ok}' \land \text{Post}[\text{sm} /\text{sm}])$$

"move quantifier in"

$$\neg \text{ok} \land \text{ok}' \land \exists \text{sm}, \text{ok}_m \cdot \text{Post}[\text{sm} /\text{sm}])$$

"$\text{ok}_m \notin \text{Post}"

$$\neg \text{ok} \land \text{ok}' \land \exists \text{sm} \cdot \text{Post}[\text{sm} /\text{sm}])$$
Proof of \langle \langle \text{skip}; \text{-unit} \rangle \rangle (VI, simplifying S;U.3)

\[ \exists S_m, ok_m \bullet ok_m \land S_m = S \land \neg (ok_m \land \text{pre}[S_m,/S]) \]

\[ = \quad \text{“ de-Morgan ”} \]

\[ \exists S_m, ok_m \bullet ok_m \land S_m = S \land (\neg ok_m \lor \neg \text{pre}[S_m,/S]) \]

\[ = \quad \text{“ distributivity ”} \]

\[ \exists S_m, ok_m \bullet ok_m \land S_m = S \land \neg ok_m \lor \text{pre}[S_m,/S] \]

\[ = \quad \text{“ contradiction ”} \]

\[ \exists S_m, ok_m \bullet \text{False} \lor ok_m \land S_m = S \land \neg \text{pre}[S_m,/S] \]

\[ = \quad \text{“ simplify, one-point ”} \]

\[ \exists S_m, ok_m \bullet \neg \text{pre} \quad \text{“ witness, ok}_m = \text{True ”} \]

\[ \neg \text{pre} \]

\[ \boxempty \]

Proof of \langle \langle \text{skip}; \text{-unit} \rangle \rangle (VII, simplifying S;U.4)

\[ \exists S_m, ok_m \bullet ok_m \land S_m = S \land ok' \land \text{Post}[S_m/S] \]

\[ = \quad \text{“ one-point law ”} \]

\[ \exists ok_m \bullet ok_m \land ok' \land \text{Post} \]

\[ = \quad \text{“ witness: ok}_m = \text{true ”} \]

\[ ok' \land \text{Post} \]

“Witness”?

- What is meant by the proof steps labelled “witness”?
- Remember, \( \exists x \bullet P \) is true if any value of \( x \) exists that makes \( P \) true. Such a value is is a “witness” (to the truth of the existential).
- We have a law that states: \( P[e/x] \Rightarrow \exists x \bullet P \)
- by \( \Rightarrow \)-join this becomes \( P[e/x] \lor \exists x \bullet P \equiv \exists x \bullet P \)
- In effect, knowing that \( P[e/x] \) is true, we can replace \( \exists x \bullet P \) by true:
  \[ \exists x \bullet P \]
  \[ = \quad \text{“ law above ”} \]
  \[ P[e/x] \lor (\exists x \bullet P) \]
  \[ = \quad \text{“ we know (can show) that } P[e/x] = \text{true ”} \]
  \[ \text{true} \lor (\exists x \bullet P) \]
  \[ = \quad \text{“ simplify ”} \]
  \[ \text{true} \]
Frame Specifications with \( ok, ok' \)

- What is the semantics of a frame specification \( \bar{x} : \bar{S} \) in our new theory?
- In the old theory it was

\[
\bar{x} : \bar{S} \overset{\triangleq}{=} S \land S^\perp = S^\perp
\]

- How do we fit in \( ok, ok' \)?

Frame Syntax change

- To fit our new theory, frames now have to explicitly distinguish pre and post conditions:

\[
\bar{x} : [pre, Post]
\]

- We also need to note that if \( ok = False \) at the outset, then we can modify any variables, not just those in the frame.
- The definition we get is therefore:

\[
\langle \text{frame-def} \rangle \bar{x} : [pre, Post] \overset{\triangleq}{=} ok \land pre \Rightarrow ok' \land Post \land S^\perp = S^\perp
\]

Upgrading frame specs to new theory

- An old specification of the form

\[
w : [P]
\]

where \( P \) is not an implication, becomes

\[
w : [true, P]
\]

- An old specification of the form

\[
w : [pre \Rightarrow Post]
\]

becomes

\[
w : [pre, Post]
\]

Refinement with \( ok, ok' \)

- How does adding in \( ok \) and \( ok' \) affect refinement laws?
- e.g. (old rules)

\[
\langle \text{prog-strengthen} \rangle \cdot P \sqsubseteq P \land Q
\]

\[
\langle \text{subst} \rangle \cdot (S \sqsubseteq P \land x = e) \equiv (S[e/x] \sqsubseteq P \land x = e)
\]

\[
\langle \text{assign} \rangle \cdot x' = e \sqsubseteq x := e
\]

- Can we use these as is?
- Are there more useful variants of these?
Using ⟨⟨prog-strengthen⟩⟩ with ok, ok’

⟨⟨prog-strengthen⟩⟩ :  \( P \subseteq P \land Q \)

- The rule is still valid
- However we are less likely to be able to use it, as the \( P \land Q \) part will often occur on the rhs of an implication.
- The following variant is useful

\[
\langle⟨ \text{prog-strengthen} \rangle\rangle (P \Rightarrow Q) \subseteq (P \Rightarrow Q \land R)
\]

- In particular, often, \( P \) will be \( \text{ok} \land \text{pre} \), and \( Q \) will be \( \text{ok’} \land \text{Post} \).

Using ⟨⟨\text{=}\text{-}\text{subst}⟩⟩ with ok, ok’

⟨⟨\text{=}\text{-}\text{subst}⟩⟩ :  \( (S \subseteq P \land x = e) \equiv (S[e/x] \subseteq P \land x = e) \)

- The rule is still valid, but again we now have those extra implications
- The following variant is useful

\[
\langle⟨ \text{=}\text{-}\text{subst} \rangle\rangle (S \subseteq p \Rightarrow Q \land x' = e) \equiv (S[e/x'] \subseteq p \Rightarrow Q \land x' = e)
\]

Note here that \( p \) does not mention \( x' \).
- As before, we often find that \( P \) will be \( \text{ok} \land \text{pre} \), and \( Q \) will be \( \text{ok’} \land \text{Post} \).

Using ⟨⟨\text{-}\text{:=}⟩⟩ with ok, ok’

⟨⟨\text{-}\text{:=}⟩⟩ :  \( x' = e \subseteq x := e \)

- The specification \( x' = e \) is no longer valid — it does not mention \( \text{ok} \) and \( \text{ok’} \) in a valid way
- We get useful variants if we change the specification to fit our new theory:

\[
\langle⟨ \text{-}\text{:=} \rangle\rangle \vec{w} [:\text{pre}, x' = e] \subseteq x := e, \quad x \in \vec{w}
\]

Revisiting Loop Refinement Example 2 (Hoare ’69)

Given

\[
\text{HSpec} \equiv r, q : [\text{true}, y' > r' \land x' = r' + y' \ast q']
\]

\[
\text{HProg} \equiv r, q := x, 0; (y \leq r) \ast (r, q := r - y, 1 + q)
\]

Prove

\[
\text{HSpec} \subseteq \text{HProg}
\]

using new version of our theory, with \( \text{ok} \) and \( \text{ok’} \)
Refining Hoare’69: revisited

We need to
1. find appropriate invariant $hinv$
   ▶️ this is still $x = r + q \ast y$, as before
2. show $HSpec \sqsubseteq r, q : [true, \ hinv \land y' > r']$
3. show $r, q : [true, hinv] \sqsubseteq r, q := x, 0$
4. show $r, q : [y \leq r \land hinv, hinv'] \sqsubseteq r, q := r - y, 1 + q$

Refining Hoare’69: revisited (1)

We must show $HSpec \sqsubseteq r, q : [true, hinv' \land y' > r']$

$r, q : [true, y' > r' \land x' = r' + y' \ast q']$

$\sqsubseteq r, q : [true, x' = r' + q' \ast y' \land y' > r']$

"(≡-refl)"

true

Refining Hoare’69: revisited (2)

We must show $true : [r, q, hinv'] \sqsubseteq r, q := x, 0$

$r, q : [true, x' = r' + q' + y'] \sqsubseteq r, q := x, 0$

"(≡-frame-def)\\(\equiv \text{sim} \equiv \text{def})"$

ok $\Rightarrow$ ok' $\land$ $y' = x' + 1 + q' \land x' = x \land y' = y$

$\sqsubseteq$ ok $\Rightarrow$ ok' $\land$ $r' = x \land q' = 0 \land x' = x \land y' = y$

"(≡-subst) (new variant)"

ok $\Rightarrow$ ok' $\land$ $x = x + 0 \ast y \land x' = x \land y' = y$

$\sqsubseteq$ ok $\Rightarrow$ ok' $\land$ $r' = x \land q' = 0 \land x' = x \land y' = y$

"arithmetic"

ok $\Rightarrow$ ok' $\land$ $true \land x' = x \land y' = y$

$\sqsubseteq$ ok $\Rightarrow$ ok' $\land$ $r' = x \land q' = 0 \land x' = x \land y' = y$

"(equiv-strengthen) (new variant)"

true

Refining Hoare’69: revisited (3)

We must show $r, q : [y \leq r \land hinv, hinv'] \sqsubseteq r, q := r - y, 1 + q$

$r, q : [y \leq r \land x = r + q \ast y, x' = r' + q' \ast y']$

$\sqsubseteq r, q : r - y, 1 + q$

"(≡-frame-def)\\(\equiv \text{sim} \equiv \text{def})"$

ok $\land$ $y \leq r \land x = r + q \ast y$

$\Rightarrow$ ok' $\land$ $x' = r' + q' \ast y' \land x' = x \land y' = y$

$\sqsubseteq$ ok $\Rightarrow$ ok' $\land$ $r' = r - y \land q' = 1 + q \land x' = x \land y' = y$

"(≡- subst) (new variant)"

ok $\land$ $y \leq r \land x = r + q \ast y$

$\Rightarrow$ ok' $\land$ $x = r - y + (q + 1) \ast y \land x = x \land y = y$

$\sqsubseteq$ ok $\Rightarrow$ ok' $\land$ $r' = r - y \land q' = 1 + q \land x' = x \land y' = y
Refining Hoare’69: revisited (3 cont.)

\[ ok \land y \leq r \land x = r + q \times y \]
\[ \Rightarrow ok' \land x = r - y + (q + 1) \times y \land x = x \land y = y \]
\[ \subseteq ok' \Rightarrow ok' \land r' = r - y \land q' = 1 + q \land x' = x \land y' = y \]
\[ = \text{“arithmetic, and } e = e \equiv \text{true”} \]
\[ ok \land y \leq r \land x = r + q \times y \Rightarrow ok' \land x = r + q \times y \]
\[ \subseteq ok' \Rightarrow ok' \land r' = r - y \land q' = 1 + q \land x' = x \land y' = y \]
\[ = \text{“(A } \land B \Rightarrow C \land B) \equiv (A \land B \Rightarrow C)”} \]
\[ ok \land y \leq r \land x = r + q \times y \Rightarrow ok' \]
\[ \subseteq ok' \Rightarrow ok' \land r' = r - y \land q' = 1 + q \land x' = x \land y' = y \]
\[ = \text{“(A } \land B \Rightarrow C) \subseteq (A \Rightarrow C \land D)”} \]
\[ \text{true} \]

(Exercise (voluntary): prove the last two laws used).

What was the point ?

- We have proved that if the Hoare’69 program starts ok, then it terminates, establishing \( y' > r' \land x' = y' \times q' + r' \).
- The proof of correctness was very similar to before.
- We had to handle the extra complexity of \( ok \land \ldots \Rightarrow ok' \land \ldots \).
- What have we gained ?
- in fact, things aren’t just pointless, they are worse than that !

We have “proved” something False !!

- This is a cardinal error — it means our proof system is **unsound**!
- A proof system is unsound if we can use it to give a “proof” of a property, that, by some other means, we know is in fact False.
- Unsoundness means there is a mismatch between our proof method and our intended meaning for the predicates we are manipulating.
- Hold on ! How do we know the property *is* False ?

Hoare’69 can fail to terminate

Consider the following execution of the program:

- **start** \( x=3, y=0 \)
- **init** \( x=3, y=0, r=3, q=0, y \leq r \)
- **loop1** \( x=3, y=0, r=3, q=1, y \leq r \)
- **loop2** \( x=3, y=0, r=3, q=2, y \leq r \)
- **loop3** \( x=3, y=0, r=3, q=3, y \leq r \)
- **loop4** \( x=3, y=0, r=3, q=4, y \leq r \)
- and so on ...

So our proof technique is flawed.
Fixing our Proof System

- So the addition of `ok` and `ok'` hasn’t completely solved our need to reason about total correctness.
- The problem is that invariants tell us about loop correctness, if they terminate.
- We need something extra to show that loops do indeed terminate.
- We need to show progress towards termination, by using a “variant”.

Loop Variants

- A Variant $V$ for a loop $c \ast P$, is:
  - An expression over program variables.
  - Has a finite numeric (integer) value
  - Is always greater than or equal to zero
  - Always decreases by at least one on each loop iteration.
- The existence of an expression $V$ with the above properties proves that a loop terminates.

How does a variant guarantee termination?

- as the loop starts $V$ will have some value $k_0$
- after the 1st iteration it will have value $k_1$, with $k_1 < k_0$
- after at most $k_0$ iterations it will have value 0
- the loop must have terminated at this point (Why?)
- it is hasn’t terminated, then either
  - the next iteration does not decrease $V$, or
  - the value of $V$ becomes negative
- in both cases, we do not have a variant property, so $V$ cannot have been a variant after all.
Revising our loop proof method

To refine $\vec{v} : \text{pre, Post}$ by $\text{init} : c \ast P$ we need to:

1. Find an appropriate invariant $\text{inv}$ and variant $V$
2. Show that
   
   $\vec{v} : [\text{pre, Post}] \subseteq \vec{v} : [\text{pre, inv'兰c'}]$ 

3. Show
   
   $\vec{v} : [\text{pre, inv'}] \subseteq \text{init}$

4. Show
   
   $\vec{v} : [c兰inv兰inv'兰0 \leq V' < V] \subseteq P$

This is the key difference: we now have to show the variant is reduced, but never below zero.

Revisiting Hoare ’69 (one more time)

- We need to determine a variant for Hoare’69 $V = f(x, y, r, q)$
- Should be zero when loop terminates $(r < y)$
  
  \[ r < y \Rightarrow f(x, y, r, q) = 0 \]

- Should decrease by at least once per iteration $r, q := r - y, q + 1$
  
  \[ f(x, y, r - y, q + 1) < f(x, y, r, q) \]

- Intuitively, the variant measures the “distance to go” towards termination.

A Variant for Hoare’69

- We shall try the following variant:
  
  \[ V \triangleq r \ominus y \]

  where $a \ominus b = a - b$, if $b \leq a$, and equals zero otherwise.

- The proof obligation involving the variant is now
  
  $r, q : [y \leq r \land x = q \ast y + r,$
  
  $x' = q' \ast y' + r' \land 0 \leq r' \ominus y' < r \ominus y]$

  $\subseteq r, q := r - y, q + 1$

Attempting to prove that $r \ominus y$ is a variant (I)

- Rather than prove the whole thing, we focus on the variant part
  
  $r, q : [y \leq r, 0 \leq r' \ominus y' < r \ominus y] \subseteq r, q := r - y, q + 1$

- We expand the Lhs using $\langle\langle \text{frame-def} \rangle\rangle$:
  
  \[ ok \land y \leq r \Rightarrow ok' \land 0 \leq r' \ominus y' < r \ominus y \land x' = x \land y' = y \]

- We expand the rhs using $\langle\langle \text{sim+:=:def} \rangle\rangle$:
  
  \[ ok \Rightarrow ok' \land r' = r - y \land q' = q + 1 \land x' = x \land y' = y \]
Attempting to prove that $r \ominus y$ is a variant (II)

- We use the rhs, and $\mathfrak{R}_{\ominus \text{subst}}$ (new version) to transform the lhs:

$$\text{ok} \land y \leq r \Rightarrow \text{ok}' \land 0 \leq (r - y) \ominus y < r \ominus y \land x = x \land y = y$$

- Again we ignore the variables $\text{ok}$, $x$, and their dashed variants and focus on the key variant part:

$$y \leq r \Rightarrow 0 \leq (r - y) \ominus y < r \ominus y$$

- The first part $0 \leq (r - y) \ominus y$ is always true because $\ominus$ never returns a negative result

- For the second part, having $y \leq r$ ensures that $r - y$ is not negative, but we cannot guarantee that $(r - y) \ominus y < r \ominus y$, unless $y > 0$!

Revising Hoare '69

We now know that there was a missing pre-condition: $y > 0$!

Given a revised specification

$$\text{HSpec} \triangleq r, q : [y > 0, y' > r' \land x' = r' + y' * q']$$

$$\text{HProg} \triangleq r, q := x, 0; (y \leq r) * (r, q := r - y, 1 + q)$$

We can now prove

$$\text{HSpec} \sqsubseteq \text{HProg}$$

by choosing invariant

$$\text{inv} \triangleq x = q * y + r$$

and variant

$$\text{V} \triangleq r \ominus y$$

within the new version of our theory, with $\text{ok}$ and $\text{ok}'$.

Debugging by Proof

- The Hoare'69 example wasn’t just about a pre-condition that we forgot/ignored.
- It showed another strength of reasoning by proof
- In attempting to prove what we believed was a correct result, we uncovered an error in our initial assumptions.
- We discovered/calculated an important pre-condition
- “95% of theorems are false” (John Rushby, PVS Theorem Prover, 1990s)

Formal Methods: recent Research

- (2003–present, w/Jim Woodcock) Using UTP to model synchronous Circus. (a.k.a. “slotted-Circus”)
Formal Methods: current Research

- (2003–present, w/Jim Woodcock)
  Using UTP to model synchronous Circus. (a.k.a. “slotted-Circus”)
  - (2008–2013, w/Arshad Beg)
    Formal translation from Circus to CSP
- (2007–∞, w/Arshad Beg)
  Developing the U·(TP)² proof assistant
- (2011–2014, w/Lero)
  Formal aspects of modelling software development processes
- (2012–2013, w/David Sanán, Lero & ESA)
  Investigating techniques to formally model and verify a secure partitioning microkernel.

Circus, Flash

- GC6 is a computing “grand-challenge” to develop libraries of verified software, currently with a number of “pilot projects”:
  - Mondex Smart Card (Natwest Bank)
  - POSIX filestore (NASA JPL)
- Circus is a formal language that combines imperative programming (variables, assignment) with concurrent systems (message-passing) (see http://www.cs.york.ac.uk/circus/).
- work between TCD and York has focussed on formal models of Flash Memory

Handel-C, slotted-Circus

- Handel-C is the C language extended with message-passing and parallelism, that compiles directly to hardware (FPGAs).
  (see http://www.mentor.com/products/fpga/handel-c/)
  - it is based on notion of synchronously clocked time-slots.
- “slotted-Circus” is a generic framework for adding discrete time to Circus, suitable for modelling time-slot languages like Handel-C.
  - it is currently be worked on here at TCD, using UTP as the semantic framework.

Kernel Verification

- ESA wants to put multiple applications on one computer
  - currently each application (propulsion, telemetry, experiment, sensor array) has its own computing hardware
  - want integrated modular avionics (IMA) to save weight
- This requires a time-space partitioning kernel (hypervisor) to ensure non-interference between applications.
- We are doing a feasibility study:
  - Develop a Reference Specification.
  - Formalise it (Using Isabelle/HOL).
  - Explore feasibility of verifying the correctness of actual kernel code.
Mini-Exercise 5

Q5 Given

\[
\begin{align*}
n, f, x & : \mathbb{N} \\
fac(0) & = 1 \\
fac(n) & = n \cdot fac(n-1), \quad n > 0
\end{align*}
\]

\[
\begin{align*}
FSpec & \triangleq f, x : [f' = fac(n)] \\
FProg & \triangleq f, x \leftarrow 1, 2; (x \leq n) \bowtie f, x \leftarrow f \cdot x, x + 1
\end{align*}
\]

Using the partial correctness theory only:

5.1 State clearly the proofs that need to be done.
5.2 Determine a suitable invariant.
5.3 Prove the statement that states that \( FSpec \) is refined appropriately (Hint: one of the proofs in the answer to 5.1).

(due in next Friday week, 2pm, in class)
In the proofs so far, we have seen proof-step justifications like

- “arithmetic”
- “defn. of $\leq$”
- “$5 < x < 3$ clearly impossible”

Such justifications are “hand-waving” and not formally rigorous.

How do we formalise these proof steps so they are part of our formal game?

A “Theory” as an organising Principle

- In order to formalise arithmetic, say, we have to:
  - extend our language syntax to cover arithmetic notation
  - extend our axioms to cover the new notation
  - build a collection of useful theorems from those axioms
- A “Theory” is a collection of all the above pieces of information, along with any necessary support material.
- We shall briefly explore two theories:
  - Arithmetic
  - Lists

A small theory of equality

It will prove useful to have a theory of equality

- Syntax: $e, f \in Expr ::= \ldots \mid e_1 = e_2 \mid e_1 \neq e_2$
- Axioms:

  - $\langle\equiv\text{-refl}\rangle$ $e = e$
  - $\langle\equiv\text{-comm}\rangle$ $(e = f) \equiv (f = e)$
  - $\langle\equiv\text{-trans}\rangle$ $((e = f) \land (f = g)) \Rightarrow (e = g)$
  - $\langle\neq\text{-def}\rangle$ $(e \neq f) \equiv \neg (e = f)$

  Equality is an equivalence relation (reflexive, symmetric, transitive)

- We will extend this theory as needed.
A Theory of Arithmetic (Syntax)

We use $m, n$ for numeric expressions

$$m, n \in \text{Expr} ::= \ldots$$

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>\ldots</th>
<th>numeric constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{succ } n$</td>
<td>$\text{pred } n$</td>
<td>basic operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m + n$</td>
<td>$m - n$</td>
<td>additive operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m \times n$</td>
<td>$m \div n$</td>
<td>multiplicative operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m^n$</td>
<td>\ldots</td>
<td>exponential operators</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We use $r$ and $s$ for atomic predicates over numeric values

$$r, s \in \text{Expr} ::= \ldots$$

<table>
<thead>
<tr>
<th>$m = n$</th>
<th>$m \neq n$</th>
<th>equalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m &lt; n$</td>
<td>$m &gt; n$</td>
<td>comparisons (strict)</td>
</tr>
<tr>
<td>$m \leq n$</td>
<td>$m \geq n$</td>
<td>comparisons (non-strict)</td>
</tr>
<tr>
<td>$n : T$</td>
<td>numeric type assertions</td>
<td></td>
</tr>
</tbody>
</table>

($T \in \mathbb{N, Z, Q, R, C}$)

Some definitions for arithmetic

<table>
<thead>
<tr>
<th>$0$-def</th>
<th>$0 + m = m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{succ}$-def</td>
<td>$(\text{succ } n) + m = \text{succ}(n + m)$</td>
</tr>
<tr>
<td>$0$-mul-def</td>
<td>$0 \times m = 0$</td>
</tr>
<tr>
<td>$\text{succ}$-mul-def</td>
<td>$(\text{succ } n) \times m = m + (n \times m)$</td>
</tr>
<tr>
<td>$^0$-def</td>
<td>$m^0 = 1$</td>
</tr>
<tr>
<td>$^\text{succ}$-mul-def</td>
<td>$m^{\text{succ } n} = m \times m^n$</td>
</tr>
<tr>
<td>$1$-def</td>
<td>$1 = \text{succ } 0$</td>
</tr>
<tr>
<td>$2$-def</td>
<td>$2 = \text{succ } 1$</td>
</tr>
<tr>
<td>$3$-def</td>
<td>$3 = \text{succ } 2$</td>
</tr>
<tr>
<td>\vdots</td>
<td></td>
</tr>
</tbody>
</table>

(Strictly speaking we need to model digit strings and their valuations in order to do the last three definitions properly)

Some theorems for arithmetic

<table>
<thead>
<tr>
<th>$+$-0-unit</th>
<th>$n + 0 = n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$-succ-comm</td>
<td>$(\text{succ } n) + m = n + (\text{succ } m)$</td>
</tr>
<tr>
<td>$+$-0-comm</td>
<td>$n + m = m + n$</td>
</tr>
<tr>
<td>$+$-assoc</td>
<td>$\ell + (m + n) = (\ell + m) + n$</td>
</tr>
<tr>
<td>$*$-0-comm</td>
<td>$n \times m = m \times n$</td>
</tr>
<tr>
<td>$*$-0-def</td>
<td>$n \times 0 = 0$</td>
</tr>
<tr>
<td>$*$-assoc</td>
<td>$(\ell \times (m \times n)) = (\ell \times m) \times n$</td>
</tr>
<tr>
<td>$*$-distr</td>
<td>$\ell \times (m + n) = \ell \times m + \ell \times n$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$1+1$-equals-2</th>
<th>$1 + 1 = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2+2$-equals-4</td>
<td>$2 + 2 = 4$</td>
</tr>
</tbody>
</table>

Some axioms for arithmetic

Axioms cover the basic building blocks

- Zero is a natural number
  - $\langle \text{0-is-N} \rangle \quad 0 : \mathbb{N}$
- If $n$ is a natural number then so is the successor of $n$
  - $\langle \text{succ-is-N} \rangle \quad (n : \mathbb{N}) \Rightarrow (\text{succ } n : \mathbb{N})$
- The successor of a natural number is never zero
  - $\langle \text{succ-not-0} \rangle \quad \text{succ } n \neq 0$
- Successor maps different values to different results
  - $\langle \text{succ-injective} \rangle \quad (n = m) \Leftrightarrow (\text{succ } n = \text{succ } m)$
- If $P(0)$ is true, and whenever $P(n)$ is true, we can show that $P(n + 1)$ is true, then $P(x)$ is true for all $x : \mathbb{N}$
  - $\langle \text{N-induction} \rangle \quad P[0/x] \wedge (\forall n \bullet P[n/x]) \Rightarrow P[\text{succ } n/x]$
### Proof of `<1+1-equals-2>`

**Goal:** 1 + 1 = 2  
**Strategy:** reduce to true

\[
1 + 1 = 2 = \langle\langle 1\text{-def}, 2\text{-def} \rangle \rangle \\
(succ 0) + (succ 0) = succ 1 = \langle\langle succ\text{-+}\text{-def}, 1\text{-def} \rangle \rangle \\
succ(0 + succ 0) = succ(succ 0) = \langle\langle 0\text{-+}\text{-def} \rangle \rangle \\
succ(succ 0) = succ(succ 0) = \langle\langle \Rightarrow\text{-refl} \rangle \rangle \\
true
\]

\[\square\]

### Proof of `<+-0-unit>`

**Goal:** n + 0 = n  
**Strategy:** ???

- A traditional proof of this uses induction, on n.
- We show it is true for \(n = 0\), i.e. that \(0 + 0 = 0\).
- We then assume it (i.e. \(n + 0 = n\)) and show it is true for \(succ\ n\).

\[
succ\ n + 0 = succ\ n
\]

- We have an induction principle (law \(\langle\langle N\text{-induction} \rangle \rangle\)), but how do we use it in our proof system?

### Developing an Induction Strategy

- The induction law has the general form \(A \Rightarrow B\).
- Any such law suggests that if we want to show \(B\) to be true, then one such way is to prove \(A\).
- Given \(A \Rightarrow B\), the use of \(\langle\langle \Rightarrow\text{-join} \rangle \rangle\) transforms this to \(B \equiv B \lor A\).
- Now, consider our proof of \(B\):

\[
B = \text{" by alternate form of law above "} \\
B \lor A = \text{" given a proof of } A \text{ "} \\
B \lor true = \text{" } \langle\langle \lor\text{-zero} \rangle \rangle \text{ "} \\
true
\]

- So from a proof of \(A\), we may construct a proof of \(B\).

### The Induction Strategy

- Assume an Induction Principle, i.e., a law of the form \(Q_1 \land Q_2 \land \ldots Q_n \Rightarrow \forall x \bullet P\).
- In order to prove \(\forall x \bullet P\), we now know it suffices to prove individually each of the \(Q_i\).
- So, given goal \(\forall x \mid x : N \bullet P\), an inductive proof allows us to prove it via the following two sub-goals:

\[
P[0/x] \\
P[n/x] \Rightarrow P[succ\ n/x]
\]
∀ and Theoremhood

- What does it mean to say that $P$ is a theorem (or axiom/law) ?
- It means that $P$ evaluates to true, regardless of the environment.
- This is the same as stating that $[P]$ is a theorem.
- Remember: $[P]$ is true only if $P$ is true for all environments.
- We have quantified over nothing ($P$), and everything ($[P]$), but what about $\forall xs \cdot P$ ?
- If $P$ is a theorem, then so is $\forall xs \cdot P$.
- It turns out, regardless of what the $xs$ are, that if any of the following is a theorem, they all are:

$$P \quad (\forall xs \cdot P) \quad [P]$$

Proof of $\langle + \cdot 0 \cdot \text{unit} \rangle$ (again)

- Goal: $n + 0 = n$
- Strategy: Induction on $n$ using $\langle \mathbb{N} \cdot \text{induction} \rangle$.
- 1st sub-goal (a.k.a. “the base case”)

$$0 + 0 = 0$$

- 2nd sub-goal (a.k.a. “the inductive step”)

$$\forall n \quad (n + 0 = n) \Rightarrow (\text{succ } n) + 0 = \text{succ } n$$

- However showing the above a theorem is the same as showing the following to be a theorem

$$\forall n \quad (n + 0 = n) \Rightarrow (\text{succ } n) + 0 = \text{succ } n$$

Proof of $\langle + \cdot 0 \cdot \text{unit} \rangle$ (base case)

(sub-)Goal: $0 + 0 = 0$
(sub-)Strategy: reduce lhs to rhs

$$0 + 0 = " \langle + \cdot 0 \cdot \text{unit} \rangle " = 0$$

□

Proof of $\langle + \cdot 0 \cdot \text{unit} \rangle$ (inductive step)

(sub-)Goal: $(n + 0 = n) \Rightarrow (\text{succ } n) + 0 = \text{succ } n$
(sub-)Strategy: assume antecedent $\langle + \cdot 0 \cdot \text{def.hyp} \rangle$ $n + 0 = n$,
Show consequent (reduce to true):

$$\text{succ } n + 0 = \text{succ } n$$

= " $\langle \text{succ} \cdot + \cdot \text{def} \rangle "$

$$\text{succ } (n + 0) = \text{succ } n$$

= " $\langle + \cdot 0 \cdot \text{def.hyp} \rangle "$

$$\text{succ } n = \text{succ } n$$

= " $\langle = \cdot \text{refl} \rangle "$

true

□
Arithmetic Theory: summary

- Induction using $\langle\mathbb{N}\text{-induction}\rangle$ is the main proof technique for the laws of arithmetic.
- Most if not all of the laws shown are done this way.
- For example, $\langle\text{+-succ\text{-comm}}\rangle$ can be proven by induction on $n$, and using $\langle\text{+-0\text{-unit}}\rangle$ to assist in the base-case.
We can also define numeric ordering, and supply some theorems.

**Definitions**

- $0 \leq n$
- $(\text{succ } n) \leq (\text{succ } m) \equiv n \leq m$
- $\neg (\text{succ } n \leq 0)$

**Theorems**

- $1 \leq 2$
- $\neg (2 \leq 1)$
- $n \leq n + m$
- $n + m \leq n \Rightarrow m = 0$

---

**Definitions and Theorems for Subtraction**

**Definitions:**

- $n - 0 = n$
- $(\text{succ } n) - (\text{succ } m) = n - m$

**Theorems**:

- $(m + n) - n = m$
- $(m - n) + n = m$
- $(m - n) \leq m$
- $3 - 2 = 1$
- $1 - 3 \leq 1$

**Proof of $3 \text{-} 2 \text{-} \text{is} \text{-} 1$**

**Goal:** $3 - 2 = 1$

**Strategy:** reduce LHS to RHS.$^3$

\[
3 - 2 = \begin{cases} 
\text{``3-def'', ''2-def''} & \\
(\text{succ } 2) - (\text{succ } 1) & \\
\text{``--succ-def''} \end{cases} \\
2 - 1 = \begin{cases} 
\text{``2-def'', ''1-def''} & \\
(\text{succ } 1) - (\text{succ } 0) & \\
\text{``--succ-def''} \end{cases} \\
1 - 0 = \begin{cases} 
\text{``--0-def''} & \\
1 & \\
\text{□} \end{cases} \\
\]

$^3$ works for theorems using $=$ as well as $\equiv$.
Proof of \(\langle\langle + - - \langle - \text{inv} \rangle \rangle\rangle\)

- Goal: \((m + n) - n = m\)
- Strategy: Induction on \(n\)
- Base-Case: \((m + 0) - 0 = m\)
- Inductive Step:
  \[\((m + n) - n = m\) \Rightarrow ((m + (\text{succ} n)) - (\text{succ} n) = m)\]

Proof of \(\langle\langle + - - \langle - \text{inv} \rangle \rangle\rangle\), Base Case

Goal: \((m + 0) - 0 = m\)
Strategy: reduce lhs to rhs

\[
\begin{align*}
(m + 0) - 0 &= "\langle\langle + - \langle 0 \text{-unit} \rangle \rangle\rangle" \\
&= m - 0 \\
&= "\langle\langle - \langle 0 \text{-def} \rangle \rangle\rangle" \\
&= m \\
\end{align*}
\]
\[\square\]

Proof of \(\langle\langle + - - \langle - \text{inv} \rangle \rangle\rangle\), Inductive Step

Goal: \[((m + n) - n = m) \Rightarrow ((m + (\text{succ} n)) - (\text{succ} n) = m)\]
Strategy: assume antecedent \(\langle\langle + - - \langle - \text{ind-hyp} \rangle \rangle\rangle\)
\((m + n) - n = m\)
and reduce consequent lhs to rhs

\[
\begin{align*}
(m + (\text{succ} n)) - (\text{succ} n) &= "\langle\langle + - \langle \text{comm} \rangle \rangle\rangle" \\
((\text{succ} n) + m) - (\text{succ} n) &= "\langle\langle \text{succ} + - \langle \text{def} \rangle \rangle\rangle" \\
\text{succ}(n + m) - (\text{succ} n) &= "\langle\langle \text{succ} + - \langle \text{def} \rangle \rangle\rangle" \\
(n + m) - n &= "\langle\langle - \text{succ} - \langle \text{def} \rangle \rangle\rangle" \\
(m + n) - n &= "\langle\langle + - - \langle - \text{ind-hyp} \rangle \rangle\rangle" \\
&= m \\
\end{align*}
\]
\[\square\]

Proof of \(\langle\langle 1 - - 3 \text{-lt-1} \rangle \rangle\)

Goal: \(1 - 3 \leq 1\)
Strategy: reduce to true

\[
\begin{align*}
1 - 3 & \leq 1 \\
&= "\langle\langle 1 - \langle \text{def} \rangle, \langle 3 - \text{def} \rangle \rangle\rangle" \\
&(\text{succ} 0) - (\text{succ} 2) \leq (\text{succ} 0) \\
&= "\langle\langle - \langle \text{succ} - \langle \text{def} \rangle \rangle\rangle" \\
0 - 2 & \leq (\text{succ} 0) \\
&= "??? no law applies ??? " \\
\end{align*}
\]

- It turns out that \(\langle\langle 1 - \text{-lt-3} \text{-lt-1} \rangle \rangle\) is not a theorem!
- Neither is its negation!
- Whazzup?
Definitions for Subtraction (reminder)

- **Definitions:**
  
  - $\langle -0 \text{-def} \rangle \quad n - 0 = n$
  - $\langle -\text{succ-def} \rangle \quad (\text{succ } n) - (\text{succ } m) = n - m$

Subtraction is partial

- Natural number subtraction as just described is **partial**: it is not defined for all values of its arguments.
  - in fact $m - n$ is only defined if $n \leq m$
- This means that the value of $m - n$ is **undefined** if $m < n$
- For example, $\langle -+\text{-inv} \rangle$ as stated $(m - n) + n = m$ is **not a theorem**.
  - If we revise it to $n \leq m \Rightarrow ((m - n) + n = m)$, then it becomes a theorem.
- The big question for us now is how do we handle the issues of partiality and undefinedness?

Handling Undefinedness (I - make it go away)

- Our problems arose because subtraction is partially defined.
- Why don’t we “totalise” it?
  - remember $\ominus$?
- To avoid confusion, we use $\ominus$ to denote the totalised version, which satisfies $\langle -0 \text{-def} \rangle$, $\langle -\text{succ-def} \rangle$ and $\langle \ominus\text{-totalise} \rangle$
  
  - $\ominus$  $0 \ominus m = 0$

- We can now easily prove that $1 \ominus 3 \leq 1$
- Sorted?

Problems with $\ominus$

- The “theorem” $(m \ominus n) + n = m$ is still not true (take $m = 1$, $n = 2$, for example)
  - we still need that side-condition ($n \leq m$)
- In general we find that most laws using $(-)$ requiring side-conditions like $n \leq m$, still require these side-conditions if we use $\ominus$ instead.
- We get more problems with division
  - What should $m/0$ be?
    - Careful! A wrong choice here allows us to prove that “black is white”!
Problems with “Totalisation”

- If a naturally partial function is to be totalised, be careful!
- Laws may become unsound unless side-conditions are added
- The side-conditions will frequently coincide with those that determine when that function is defined.

Handling Undefinedness (II - live with it)

- Division’s partiality is a minor nuisance
  - Just remember to add $y \neq 0$ whenever we see $x/y$
- However in computer science we have a more fundamental problem
- Consider a possibly partial function $f : \mathbb{N} \rightarrow \mathbb{N}$ implemented faithfully as a program $progF$:
  - If $f(x)$ is defined, then $progF$ terminates on input $x$ with the right answer
  - If $f(x)$ is undefined, then $progF$ may not terminate
- The question “is $f$ defined on $x$” is then equivalent to solving the halting problem for $progF$
- The Halting Problem is undecidable — no algorithm can ever solve it in all cases.

Consider this

- Consider the following code, (assuming $n : \mathbb{N}$):

  \[(n > 1) \land n := n/2 < \text{ even } n \land n := (3 \cdot n + 1)/2\]

- Prove it refines the following (total-correctness) specification: $n > 0 \vdash n' = 1$
- Done yet?
- If so, well done, take a bow, leave the class, go to MIT, Stanford, Microsoft Research, wherever …
- It is the so-called Collatz Conjecture (1937): it is known to terminate for all $n \leq 2^{60}$ (May 2011), but no proof that it terminates for all $n$ has been found.

Handling Undefinedness (III - living with it)

- In the computer science domain, undefinedness is a fact of life
- However, there is no universal agreement on how to handle it in formal logics.
- Some approaches:
  - Lots of definedness side-conditions
  - Giving “meanings” to undefined expressions
  - Triple-valued logics
Handling Undefinedness (IV - how we shall live with it)

- The use of a partial function/operator implies its definedness side-condition
e.g. the term \((x + y) - z\) implies a condition \(z \leq x + y\).
- When replacing a term by one that is equal, we shall explicitly add in definedness conditions if the term that required it disappears.
- e.g consider predicate \(v = ((x + y) - z) - ((x + y) - z)\)
  - The term has an implicit side-condition: \(z \leq x + y\)
  - if we replace the term by 0 we get \(v = 0\) but we have lost information about the side-condition.
  - We shall re-introduce the side-condition explicitly, so getting \(v = 0 \land z \leq x + y\).

A Theory of Lists (Syntax)

We use \(\sigma, \tau\)for lists (a.k.a. sequences)

\[
\sigma, \tau \in T^* ::= ...
\]

| empty list | list construction |
| list enumeration | list destruction (1) |
| list concatenation | list length |
| list prefix relation |

Some axioms for lists

Axioms cover the basic building blocks

- \(\langle\langle \text{is-}T^* \rangle\rangle\) \(\langle\rangle : T^*\)
- \(\langle\langle \text{is}-T^* \rangle\rangle\) \(x : T \land \sigma : T^* \Rightarrow (x \circ \sigma) : T^*\)
- \(\langle\langle \text{-not-} \rangle\rangle\) \(\langle\rangle \neq x \circ \sigma\)
- \(\langle\langle \text{-injective} \rangle\rangle\) \((x = y) \land (\sigma = \tau) \Rightarrow (x \circ \sigma) = (y \circ \tau)\)
- \(\langle\langle T^*\text{-Induction} \rangle\rangle\) \(P(\langle\rangle / \sigma) \land (\forall v, \tau \bullet P(v \circ \tau / \sigma) \Rightarrow P[(v \circ \tau) / \sigma])\) 
  \(\Rightarrow (\forall \sigma \mid \sigma : T^* \bullet P)\)

Some definitions for lists (!) (??)

- \(\langle\langle \text{head-def} \rangle\rangle\) \(\text{head}(x \circ \sigma) = x\)
- \(\langle\langle \text{tail-def} \rangle\rangle\) \(\text{tail}(x \circ \sigma) = \sigma\)
- \(\langle\langle \text{enumeration-def} \rangle\rangle\) \((x_1, x_2, \ldots, x_n) = x_1 \circ (x_2 \circ (\ldots (x_n \circ \langle\rangle)))\)
- \(\langle\langle \text{-def} \rangle\rangle\) \(\langle\rangle \circ \tau = \tau\)
- \(\langle\langle \text{-def} \rangle\rangle\) \((x \circ \sigma) \circ \tau = x \circ (\sigma \circ \tau)\)
- \(\langle\langle \#\text{-def} \rangle\rangle\) \(\#(\langle\rangle) = 0\)
- \(\langle\langle \#\text{-def} \rangle\rangle\) \(\#(x \circ \sigma) = 1 + \#\sigma\)
- \(\langle\langle \text{front-single-def} \rangle\rangle\) \(\text{front}(x) = xs\)
- \(\langle\langle \text{front-def} \rangle\rangle\) \(\text{front}(x \circ \sigma) = x \circ \text{front}(\sigma)\)
- \(\langle\langle \text{last-single-def} \rangle\rangle\) \(\text{last}(x) = x\)
- \(\langle\langle \text{last-def} \rangle\rangle\) \(\text{last}(x \circ \sigma) = \text{last}(\sigma)\)

Note that \text{head, tail, front and last} are partial, defined only for non-empty lists.
Some theorems for lists

\[
\text{⟨⟨-r-unit⟩⟩} \quad \sigma \left< \epsilon \right> = \sigma \\
\text{⟨#-morphism⟩} \quad \#(\sigma \left< \tau \right>) = \#\sigma + \#\tau
\]

Proofs in class

Proof of \(\text{⟨⟨-r-unit⟩⟩} \) (I)

- Goal: \( \sigma \left< \epsilon \right> = \sigma \)
- Strategy: induction over \( \sigma \)
  Property \( P(\sigma) \equiv \sigma \left< \epsilon \right> = \sigma \)
  Base Case \( \text{Show } P(\epsilon) \)
  Inductive Step \( \text{Show } P(\tau) \Rightarrow P(x \left< \tau \right>) \)

Proof of \(\text{⟨⟨-r-unit⟩⟩} \) (II)

Base Case \( P(\epsilon) \equiv (\epsilon \left< \epsilon \right> = \epsilon) \)
Strategy: reduce LHS to RHS

\[
\left< \epsilon \right> \left< \epsilon \right> = " \langle -\{\rangle-def \rangle " \\
\epsilon = \epsilon
\]

Proof of \(\text{⟨⟨-r-unit⟩⟩} \) (III)

Inductive Step \( P(\tau) \Rightarrow P(x \left< \tau \right>) \)
Goal: \( (\tau \left< \epsilon \right> = \tau) \Rightarrow ((x \left< \tau \right> \left< \epsilon \right>) = x \left< \tau \right>) \)
Strategy: assume LHS to show RHS

⟨ind-hyp⟩ \( \tau \left< \epsilon \right> = \tau \)

\[
(x \left< \tau \right> \left< \epsilon \right>) = x \left< \tau \right> \\
= " \langle -\{\rangle-def \rangle " \\
x \left< \tau \right> \left< \epsilon \right> = x \left< \tau \right> \\
= " \langle -\{\rangle-def \rangle " \\
x \left< \tau \right> = x \left< \tau \right> \\
= " \langle -\{\rangle-refl \rangle " \\
\text{true}
\]
Some definitions for lists (!) (??)

\[ \text{front} (x) = \langle \rangle \]
\[ \text{front} (x \cdot \sigma) = x \cdot \text{front} (\sigma) \]
\[ \text{last} (x) = x \]
\[ \text{last} (x \cdot \sigma) = \text{last} (\sigma) \]

\text{front} and \text{last} are problematical, but not because they are partial.

Gotcha!

Explain in Class

Definitions for lists (Corrected, Safe)

\[ \text{head} (x \cdot \sigma) = x \]
\[ \text{tail} (x \cdot \sigma) = \sigma \]
\[ (x_1, x_2, \ldots, x_n) = x_1 \odot (x_2 \odot (\ldots (x_n \odot \langle \rangle))) \]
\[ \odot \tau = \tau \]
\[ (x \odot \sigma) \odot \tau = x \odot (\sigma \odot \tau) \]
\[ \# \langle \rangle = 0 \]
\[ \# (x \cdot \sigma) = 1 + \# \sigma \]
\[ \text{front} (x) = x \]
\[ \text{front} (x \cdot y \cdot \sigma) = x \cdot \text{front} (y \cdot \sigma) \]
\[ \text{last} (x) = x \]
\[ \text{last} (x \cdot y \cdot \sigma) = \text{last} (y \cdot \sigma) \]

Note that \text{head}, \text{tail}, \text{front} and \text{last} are \text{partial}, defined only for non-empty lists.

Lists Theory: summary

- Induction using \( T^* \)-Induction is the main proof technique for the laws of lists.
- Most if not all of the laws shown are done this way
- Care needs to be taken when defining functions ‘by cases’
  - Need to ensure that overlapping cases do not conflict.
A Theory of Sets

Axioms:

⟨¬-in-{}⟩  \quad x \notin \{\}

⟨in-singleton⟩  \quad x \in \{y\} \equiv x = y

⟨set-extensionality⟩  \quad S = T \equiv \forall x \bullet x \in S \equiv x \in T

Not a “traditional” axiomatization of set-theory!

Definitions:

⟨in-∪⟩  \quad x \in (S \cup T) \equiv x \in S \lor x \in T

⟨in-∩⟩  \quad x \in (S \cap T) \equiv x \in S \land x \in T

⟨in-\⟩  \quad x \in (S \setminus T) \equiv x \in S \land x \notin T

⟨def-⊆⟩  \quad S \subseteq T \equiv \forall x \bullet x \in S \Rightarrow x \in T
Mini-Exercise 5

Q5 Given

\[
\begin{align*}
  n, f, x & : \mathbb{N} \\
  \text{fac}(0) &= 1 \\
  \text{fac}(n) &= n \times \text{fac}(n - 1), \quad n > 0
\end{align*}
\]

\[
\begin{align*}
  \	ext{FSpec} & \equiv f, x : [f' = \text{fac}(n)] \\
  \	ext{FProg} & \equiv f, x := 1, 2 ; (x \leq n) \odot f, x := f \times x, x + 1
\end{align*}
\]

Using the partial correctness theory only:

5.1 State clearly the proofs that need to be done.
5.2 Determine a suitable invariant.
5.3 Prove the statement that states that \text{FSpec} is refined appropriately (Hint: one of the proofs in the answer to 5.1).

(due in next Friday week, 2pm, in class)

Tool Support

- Formal Methods sounds like a good idea in theory
- What about in practise?
- Real-world problems result in lots of small, non-trivial proof, each not quite the same as any other.
- Real-world use of formal methods requires tool support to be practical.

Introducing \( U \cdot (TP)^2 \)

- \( U \cdot (TP)^2 \) is a theorem proving assistant designed for UTP
- It is under development here at TCD
- A recent release suitable for CS4003 is now available
  - \( U \cdot (TP)^2 \) 0.97α8
- Open-source, has been built on Linux/Mac OS X; Binaries for Windows

Getting and installing \( U \cdot (TP)^2 \)

- Follow link from Blackboard (“Tool Support” area)
- (Windows) Download the ZIP file
- Copy everything into a new folder
- If using Unix/Mac OS X, you can build from source
    - requires Haskell, wxHaskell (good luck!)
Distribution Files

- Binaries: UTP2.exe
  - wxc-msw2.8.10-0.11.1.2.dll
  - Saoithin-*.wav
    - keep together
- Documentation: *.txt
  - read!

Running $U \cdot (TP)^2$

- Here we give examples on the Windows version Un*x or OS X version will be much the same.
- To run, simply double-click on the .exe file
- A MS-DOS console window appears, followed shortly by the top-level window.
  - the screenshots shown here are from an earlier years version, but essentials are the same.

Workspaces

- All your working files live in a workspace:
  - A directory holding your files
  - $U \cdot (TP)^2$ allows you to give it a nickname
  - You can generate many independent such workspaces

Warnings

- Don’t mess with the files that appear in workspace folders
- Only exit the application using the Quit menu item in the top-level “Theories” Window.
- Avoid clicking the red X button in the top right corner of a window (exception: do this to end the “Proof Key Shortcuts” window)
Related definitions, axioms and laws form a “Theory”
- \( U \cdot (TP)^2 \) maintains a stack of theories, most general at the bottom, most specific at the top
- Higher theories may depend on lower ones
- Theory 1 depends on Theory 2 if a proof in Theory 1 uses a law from Theory 2
- Circular dependencies are not permitted.

We have the usual pull-down menus
- Double-clicking on some items will “open” or “activate” them
- Right-clicking on some items or windows will bring up a context-sensitive menu
Theories consist of a series of tables:

- **LAWS**
  - Name \(\rightarrow\) Law
  - Laws

- **OBS**
  - Name \(\rightarrow\) Type
  - Observation Variables

- **LANGUAGE**
  - Theory-specific language constructs

- **PRECEDENCE**
  - Name \(\rightarrow\) Type
  - Binary Operator
  - Precedences

- **TYPE**
  - Name \(\rightarrow\) Type
  - Type Definitions

- **CONST**
  - Name \(\rightarrow\) Expr
  - Constant Definitions

- **EXPR**
  - Name \(\rightarrow\) Expr
  - Expression Definitions

- **PRED**
  - Name \(\rightarrow\) Pred
  - Predicate Definitions

- **TYPES**
  - Var \(\rightarrow\) Type

- **CONJ.**
  - Name \(\rightarrow\) Law
  - Conjectures

- **THEOREMS**
  - Name \(\rightarrow\) Proof
  - Theorems
Proving Conjectures

- To prove a conjecture, double-click on it
- A proof window appears
- You have to pick a strategy for the proof.
- The goal predicate is shown with the current “focus” underlined
- All prover commands work on the focus
- You move the focus with the arrow keys.

Proof Strategies

- Deduce: Reduce Goal down to true.
- L2R: Transform Goal LHS (of \equiv) into RHS
- R2L: Transform Goal RHS into LHS
- Red. Both: Transform both LHS and RHS of Goal into same thing
- Law: Reduce Transform named Law into Goal
- Assume: Assume Goal LHS (of \Rightarrow) and prove RHS using one of the first 4 strategies above.

Proof Actions

- The current focus in the goal is underlined, e.g. $P \equiv Q$
- Arrow keys allow the focus to be moved, e.g., Down, then Left changes the above focus to $P \equiv Q$.
- Right-click brings up the laws that can be applied to the focus
- Useful Key shortcuts
  (see Help menu in Proof window for more):
  - u undoes the most recent proof step
  - p prints the current proof state to a text file
  - c switches between proof cases
    (only applicable in certain strategies, like Red. Both)

Theorems

- Once a proof is complete it becomes a theorem and is stored in the theorems table.
- The default is also to make the theorem a law, and to save it to a file.
- Right-clicking on a theory in the THEOREMS tab allows you to save the proof in ASCII text or LaTeX form
  - (\LaTeX needs saoithin.sty — in installation zip)
“Design-hood”?

- Consider the following:
  \[ \neg \text{ok} \land \text{ok}' \land x' = x + 1 \land S'_{\downarrow x} = S_{\downarrow x} \]
  - When started in a divergent state, we terminate, increment \( x \), and leave other variables unchanged.
  - Clearly not possible for a real program
  - \( \neg \text{ok} \) means 'not properly/yet started')
- Is this predicate a design?
- Can we re-write it in the form \( \text{ok} \land P \Rightarrow \text{ok}' \land Q \)?
  - if not, how can “prove” it is not a design?

When is a predicate a Design?

- Our theory of total correctness is based on the notion of designs,
  i.e. predicates of the form: \( \text{ok} \land P \Rightarrow \text{ok}' \land Q \)
  (or \( P \vdash Q \))
- However, given an arbitrary predicate, how do we determine if it is a design ?
- Clearly one way is to transform it into the above form.
- This can be quite difficult to do in some cases.
- Is there an easier way?
- Also, just what is it that characterises designs in any case?

Healthiness Conditions

- We view designs as “healthy” predicates, i.e., those that describe real programs and specifications.
- We shall introduce “healthiness conditions” that test a predicate to see if it is healthy.
- These conditions capture key intuitions about how real programs behave
- Some will be mandatory — all programs and specifications will satisfy these.
- Some will be optional — these characterise particularly well-behaved programs
Healthiness Condition 1 (H1)

\[ P = (\text{ok} \Rightarrow P) \]

- \( P \) only 'takes effect' when \( \text{ok} \) is true — i.e. program has started
- When \( \neg \text{ok} \), then \( P \) says nothing
- \( H1 \) above is equivalent to satisfying both of the following laws:
  \[ \langle \langle \text{skip}; \text{-unit} \rangle \rangle \text{skip} \quad P = P \]
  \[ \langle ; \text{-L-Zero} \rangle \text{Forever}; D = \text{Forever} \]

- Any design \( P \vdash Q \) satisfies the three laws above.

H1 Example (1)

What about \( \neg \text{ok} \wedge \text{ok}' \wedge x' = x + 1 \wedge S^\chi_{x'} = S^\chi_{x} \)?

\[ \text{ok} \Rightarrow \neg \text{ok} \wedge \text{ok}' \wedge x' = x + 1 \wedge S^\chi_{x'} = S^\chi_{x} \]

\[ = \quad \langle \neg \Rightarrow \text{-def} \rangle \]
\[ \neg \text{ok} \vee \neg \text{ok} \wedge \text{ok}' \wedge x' = x + 1 \wedge S^\chi_{x'} = S^\chi_{x} \]

\[ = \quad \langle \neg \vee \text{-absorb} \rangle \]
\[ \neg \text{ok} \]

So it is not \( H1 \)-healthy

H1 Example (2)

What about \( x := e \)?

\[ \text{ok} \Rightarrow x := e \]
\[ = \quad \langle := \text{-def} \rangle \]
\[ \text{ok} \Rightarrow (\text{ok} \Rightarrow (\text{ok}') \wedge x' = e \wedge S^\chi_{x'} = S^\chi_{x}) \]
\[ = \quad \langle \text{shunting} \rangle \]
\[ \text{ok} \wedge \text{ok} \Rightarrow (\text{ok}') \wedge x' = e \wedge S^\chi_{x'} = S^\chi_{x} \]
\[ = \quad \langle \wedge \text{-idem} \rangle \]
\[ \text{ok} \Rightarrow (\text{ok}') \wedge x' = e \wedge S^\chi_{x'} = S^\chi_{x} \]
\[ = \quad \langle := \text{-def} \rangle \]
\[ x := e \]

So \( x := e \) is \( H1 \)-healthy

H1 Example (3)

What about \( P \vdash Q \)?

\[ \text{ok} \Rightarrow (P \vdash Q) \]
\[ = \quad \langle \vdash \text{-def} \rangle \]
\[ \text{ok} \Rightarrow (\text{ok} \wedge P \vdash \text{ok}' \wedge Q) \]
\[ = \quad \langle \text{shunting} \rangle \]
\[ \text{ok} \wedge \text{ok} \wedge P \Rightarrow \text{ok}' \wedge Q \]
\[ = \quad \langle \wedge \text{-idem} \rangle \]
\[ \text{ok} \wedge P \Rightarrow \text{ok}' \wedge Q \]
\[ = \quad \langle \vdash \text{-def} \rangle \]
\[ P \vdash Q \]

So any design is \( H1 \)-healthy
Healthiness Condition 2 (H2)

H2 \([P[False/ok'] \Rightarrow P[True/ok']]\)

- No specification can require non-termination.
- If \(P\) is true when \(ok\) is false, then it must also be true if \(ok\) is true.
- We introduce the following abbreviation:
  \[pb \equiv P[b/ok']\]

- so H2 becomes \([P' \Rightarrow P]\)
  (with \(f\) and \(t\) the obvious abbreviations for False and True.

- It is equivalent to satisfying the following law:


\[
\begin{align*}
\langle -H2-R-Unit \rangle & P; J = P \\
\langle J-def \rangle & J \equiv (ok \Rightarrow ok') \land S' = S
\end{align*}
\]

H2 Example (1)

Again, let's try \(\neg ok \land ok' \land x' = x + 1 \land S'\neg x' = S\neg x\).

\[
\begin{align*}
(\neg ok \land ok' \land x' = x + 1 \land S'\neg x' = S\neg x)[False/ok'] \\
\Rightarrow (\neg ok \land ok' \land x' = x + 1 \land S'\neg x' = S\neg x)[True/ok'] \\
= \text{ " substitution "} \\
(\neg ok \land False \land x' = x + 1 \land S'\neg x' = S\neg x) \\
\Rightarrow (\neg ok \land True \land x' = x + 1 \land S'\neg x' = S\neg x) \\
= \text{ " simplify "} \\
False \Rightarrow (\neg ok \land x' = x + 1 \land S'\neg x' = S\neg x) \\
= \text{ " GS3.75"} \\
true
\end{align*}
\]

So it is H2-healthy!

H2 Example (2)

Now, try \(ok \Rightarrow \neg ok'\) (i.e requiring non-termination).

\[
\begin{align*}
(\neg ok')[False/ok'] \Rightarrow (ok \Rightarrow \neg ok')[True/ok'] \\
= \text{ " substitution "} \\
(\neg False) \Rightarrow (ok \Rightarrow \neg True) \\
= \text{ " simplify "} \\
(\neg True) \Rightarrow (ok \Rightarrow False) \\
= \text{ " \(\Rightarrow\)-zero\}, \(\neg\)-def"} \\
\neg ok \Rightarrow \neg ok \\
= \text{ " \(\Rightarrow\)-l-unit"} \\
\neg ok
\end{align*}
\]

So it is not H2-healthy

H2 Example (3)

Finally, what about \(P \vdash Q\), or \(ok \land P \Rightarrow ok' \land Q\)?

\[
\begin{align*}
(ok \land P \Rightarrow ok' \land Q)^1 \Rightarrow (ok \land P \Rightarrow ok' \land Q)^1 \\
= \text{ " substitution, noting P and Q do not mention ok' "} \\
(ok \land P \Rightarrow False \land Q) \Rightarrow (ok \land P \Rightarrow True \land Q) \\
= \text{ " simplify "} \\
(ok \land P \Rightarrow False) \Rightarrow (ok \land P \Rightarrow Q) \\
= \text{ " \(\neg\)-def"} \\
\neg (ok \land P) \Rightarrow (ok \land P \Rightarrow Q) \\
= \text{ " \(\Rightarrow\)-def\}, \(\neg\)-invol"} \\
ok \land P \lor \neg (ok \land P) \lor Q \\
= \text{ " \(\text{excluded-middle}\), simplify "} \\
true
\end{align*}
\]

So designs are H2-healthy.
Designs are $H_1, H_2$ are Designs are $H_1, H_2$ are …

- Any Design satisfies both $H_1$ and $H_2$
- Any predicate satisfying both $H_1$ and $H_2$ is a Design
- So $H_1$ and $H_2$ are our mandatory healthiness conditions.
- All programs and specifications (incl. Chaos and miracle) satisfy these.
- Two more conditions ($H_3, H_4$) characterise more “well-behaved” predicates.

Healthiness Condition 3 ($H_3$)

$\begin{align*}
H_3 & \quad P = P; \text{skip} \\
\end{align*}$

- Running skip afterwards makes no difference
- Not true of all Designs (which fail this?)
- If fails for a design $P \vdash Q$ iff $P$ is not a condition (i.e. has dashed variables).

H3 Example (1)

- Let’s try $x' = 2 \vdash \text{true}$
  expands to $ok \land x' = 2 \Rightarrow ok'$
  “If started, and $x$ ends up equal to 2, then I terminate”

  $\begin{align*}
  ok \land x' = 2 & \Rightarrow ok' \\
  = & \quad \langle\langle \Rightarrow\text{-def} \rangle, \langle\langle \text{deMorgan} \rangle \rangle \text{”} \\
  \neg ok \lor x' \neq 2 & \lor ok' \\
  = & \quad \langle\langle \Rightarrow\text{-def} \rangle \rangle \\
  ok & \Rightarrow x' \neq 2 \lor ok' \\
  \end{align*}$

  “If started, either $x$ ends up different from 2, or I terminate”

- It’s a design, so $H_1,H_2$-healthy

H3 Example (1, cont.)

$\begin{align*}
ok \land x' = 2 & \Rightarrow ok'; \text{skip} \\
= & \quad \langle\langle \text{skip-def} \rangle \rangle \\
ok \land x' = 2 & \Rightarrow ok'; ok \Rightarrow ok' \land S' = S \\
= & \quad \langle\langle \Rightarrow\text{-def} \rangle, \langle\langle \text{def} \rangle, \text{6-way (l) simplification } \rangle \rangle \\
\neg ok & \lor (\exists x_m, ok_m \bullet (x_m = 2) \land \neg ok_m) \lor ok' \lor (\ldots) \\
= & \quad \text{” witness, } x_m = 1, ok_m = \text{False”} \\
\neg ok & \lor \text{true} \lor ok' \lor (\ldots) \\
= & \quad \langle\langle \lor\text{-zero} \rangle \rangle \\
\text{true} & \\
\end{align*}$

Not equal to $ok \land x' = 2 \Rightarrow ok'$, so not $H_3$-healthy
**H3 Example (2)**

Now, try *miracle*, a.k.a. \(\neg ok\)

\[
\neg ok; \text{skip} \\
= \quad " \langle \text{skip-def}, \text{\Rightarrow-def} \rangle " \\
\neg ok; \neg ok \lor ok' \land S' = S \\
= \quad " \langle \lor-\lor-\text{distr}, \exists-\text{distr} \rangle " \\
(\exists ok_m, S_m \cdot \neg ok \land (\neg ok_m \lor ok') \land S' = S_m) \\
= \quad " \text{simplify} " \\
\neg ok \lor \neg ok \land ok' \\
= \quad " \langle \lor-\\land\text{-absorb} \rangle " \\
\neg ok
\]

So *miracle* is H3-healthy

---

**Healthiness Condition 4 (H4)**

\[
\begin{align*}
H4 & \quad P; \quad \text{true} = \text{true} \\
\end{align*}
\]

- A H4-healthy program cannot force termination of what runs afterward.
- All real programs satisfy this property.
- H4 states that a predicate is Feasible.

There is a program that has this behaviour.

---

**H4 Example (1)**

What about *miracle*?

It satisfies any specification?

Is it feasible?

\[
\begin{align*}
\text{miracle; } & \quad \text{true} \\
= & \quad " \text{miracle is } \neg ok " \\
\neg ok; & \quad \text{true} \\
= & \quad " \langle ;\text{-def}, \text{substitution} \rangle " \\
(\exists ok_m, S_m \cdot \neg ok \land \text{true} \\
= & \quad " \text{drop quantifiers, simplify} " \\
\neg ok
\end{align*}
\]

It is not feasible, thankfully!

---

**Healthiness: Summary**

- We have 2 mandatory healthiness conditions for designs

\[
\begin{align*}
\text{H1} & \quad P = \text{ok} \Rightarrow P \\
& \quad (\text{skip; } P = P) \land (\text{true; } P = \text{true}) \\
\text{H2} & \quad [P^f \Rightarrow P^t] \\
& \quad P = P; J
\end{align*}
\]

- We have 2 additional conditions capturing better behaviour

\[
\begin{align*}
\text{H3} & \quad P = P; \text{skip} \\
\text{H4} & \quad P; \quad \text{true} = \text{true}
\end{align*}
\]

- If \(P\) and \(Q\) are Hi-healthy \((i \in 1 \ldots 4)\), then so are

\[
\begin{align*}
P; Q & \quad P \cap Q & \quad P \triangleleft c \triangleright Q & \quad c \circ P
\end{align*}
\]
Design Semantics of \( \text{true} \circ \text{skip} \)

- As before, we try to find something that satisfies the loop-unrolling law for \( \text{Forever} \), namely:
  \[
  X = \text{skip}; X
  \]

- However, now we restrict ourselves to designs only, and use the design definition for \( \text{skip} \).
- Under what conditions does design \( D = (P \vdash Q) \) satisfy the above law?
- All designs satisfy it!
  It is just law \( \langle \langle \text{skip} - \text{unit} \rangle \rangle \), already proven.

Choosing fixpoint for meaning of \( \text{true} \circ \text{skip} \)

- Any design satisfies the loop-unrolling law for \( \text{true} \circ \text{skip} \) (a.k.a. \( \text{Forever} \)).
- We have a good reason to want the least such design, namely \( \text{false} \vdash \text{true} \) (which reduces to \( \text{true} \)).

\[
\langle \langle \text{Forever-def} \rangle \rangle \quad \text{Forever} \equiv \text{true}
\]

- Does it give us the desired laws?

\[
\langle \langle ; \text{-L-Zero} \rangle \rangle \quad \text{Forever} ; D = \text{Forever}
\]

Here \( D \) are designs

Proof of \( \langle \langle ; \text{-L-Zero} \rangle \rangle \), cont.

Goal: \( \text{Forever} ; D = \text{Forever} \)
Strategy: reduce rhs to lhs

\[
\text{Forever} ; D = \langle \langle \text{Forever-def} \rangle, D = P \vdash Q \rangle
\]

\[
\text{true} ; P \vdash Q = \langle \langle ; -\text{-def} \rangle \rangle
\]

\[
\text{true} ; ok \land P \Rightarrow ok' \land Q
\]

\[
\exists S_m, ok_m \cdot \text{true} \land (ok_m \land P[S_m/S] \Rightarrow ok' \land Q[S_m/S])
\]

\[
\exists S_m, ok_m \cdot \neg (ok_m \land P[S_m/S]) \lor ok' \land Q[S_m/S]
\]

\[
\exists S_m, ok_m \cdot \neg ok_m \lor P[S_m/S] \lor ok' \land Q[S_m/S]
\]

\[
\exists S_m, ok_m \cdot \neg ok_m \lor \neg P[S_m/S] \lor ok' \land Q[S_m/S]
\]

Proof of \( \langle \langle ; \text{-L-Zero} \rangle \rangle \), cont.

\[
\exists S_m, ok_m \cdot \neg ok_m \lor \neg P[S_m/S] \lor ok' \land Q[S_m/S]
\]

\[
= \langle \langle ; -\text{-def} \rangle \rangle
\]

\[
\text{true} \lor (\exists S_m \cdot \neg P[S_m/S] \lor ok' \land Q[S_m/S])
\]

\[
= \langle \langle \text{Forever-def} \rangle \rangle
\]

\[
\text{Forever}
\]

\[\square\]
Proof of \(\langle\langle;\text{-R-Zero}\rangle\rangle\)

- **Goal**: \(D; \text{Forever} = \text{Forever}\)
- **Strategy**: reduce rhs to lhs

\[
D; \text{Forever} \\
= \langle\langle \text{Forever} \text{-def}\rangle\rangle \\
D; \text{true}
\]

- But stating this equals \text{Forever}, i.e. \text{true}, is the same as stating that \(D\) is \text{H4}-healthy (feasible)!
- We need to revise our law:

\[
\langle\langle;\text{-R-Zero}\rangle\rangle \quad D \text{ is } \text{H4} \Rightarrow (D; \text{Forever} = \text{Forever})
\]

Design Coda

- You have been introduced to Predicate Calculus as a Formal Language “game”.
- You have seen how the language can be extended so that you can play with the meanings of programs.
- You have seen how the key concepts of loop invariant and variant respectively allow you to reason formally about the correctness and termination of while-loops.
- You have (just) encountered the notion of a **Design**, namely a predicate describing a specification or program as an explicit pre-/post-pair, with termination handled implicitly.
- We have seen that Designs, and interesting sub-classes can be characterised as *healthiness conditions* on predicates.
Using $U \cdot (TP)^2$

- This class is a live demo of the $U \cdot (TP)^2$ proof assistant
- Some proofs will be done in class.
- More will be left as an exercise
  See Blackboard for details
We want to formalise our notion of healthiness

Remember, the following are our conditions

\begin{align*}
    H1 & : P = \text{ok} \Rightarrow P \\
    H2 & : P = P \# J \\
    H3 & : P = P \# \text{skip} \\
    H4 & : P; \text{true} = \text{true}
\end{align*}

They all (except H4) have the same form: $P = H(P)$ where $H$ describes the appropriate “function” of $P$.

We shall now formalise this notion

Formalising H1

We shall define a function that takes a predicate as parameter, and returns True if the predicate is H1-healthy:

\[
isH1(P) \triangleq P = (\text{ok} \Rightarrow P)
\]

We refine this further by introducing another function that given a predicate returns an appropriately modified predicate

\[
\begin{align*}
    \text{mkH1}(P) & \triangleq \text{ok} \Rightarrow P \\
    \text{isH1}(P) & \triangleq P = \text{mkH1}(P)
\end{align*}
\]

What we have done is to start a new formal game altogether!

Higher-Order Logic

Up to now, we have been using “First-Order predicate calculus”

» we have built predicates from basic parts using fixed operators.
» Any functions have only existed inside the expression (sub-)language
» All quantification variables have been limited to expression variables only

Now we are moving towards “Higher-Order Logic”

» We are introducing predicates about predicates (e.g. isH1)
» We are introducing functions that transform predicates (e.g. mkH1)
» UTP also allows quantifier variables to range over predicates
2nd-Order Predicates

- A first-order predicate \( P \) is a function over environment variables, returning \( True \) or \( False \):
  \[
  [P] : Env \to \mathbb{B}
  \]

- A second-order predicate \( P \) is a function from predicates to \( True \) or \( False \):
  \[
  [P] : Pred \to \mathbb{B}
  \]

- We can expand this as:
  \[
  [P] : (Env \to \mathbb{B}) \to \mathbb{B}
  \]

- So \( isH1 \) is just such a 2nd-order predicate.

Predicate Transformers

- A predicate-transformer \( F \) is a function over predicates, returning a predicate as a result:
  \[
  [F] : Pred \to Pred
  \]

- We can expand this also as
  \[
  [F] : (Env \to \mathbb{B}) \to (Env \to \mathbb{B})
  \]

- Function \( mkH1 \) is a predicate transformer.

Changing the Game (I)

- We add a new category to our language: that of a higher-order definition

\[
\begin{align*}
P, Q, R, S & \in PVar \\
H, F & \in HOF \\
HOFDef & ::= H(P) \doteq body
\end{align*}
\]

- We extend our notion of predicate to allow the application of a HOF to a predicate argument
  \[
P \in Pred ::= \ldots \mid H(P)
  \]

Changing the Game (II)

- Given \( H(P) \doteq body \), we have a new law:

\[
H(MyPred) = body[MyPred/P]
\]

Where we now have substitution notation extended to allow predicates to replace predicate variables.

- The addition of higher-order functions (i.e. those that take predicates as arguments) gives "monadic 2nd-order logic"

- If we also allow quantification over predicates

\[
Pred ::= \ldots \mid \forall P \cdot H(P)
\]

we get "full 2nd-order logic"
The Nature of $mkH$

- Consider the application $mkH(P)$
- We can interpret $mkH$ as a “healthifying” function (i.e., it makes predicates healthy)
- What if $P$ is already healthy?
  - Then it satisfies $isH$, which says $P = mkH(P)$
  - So $mkH$ should not change an already healthy predicate
- We are led to the following requirement for any “healthifier” $mkH$: $mkH(mkH(P)) = mkH(P)$
- $mkH \circ mkH = mkH$

i.e. all such HOFs must be idempotent.
- Once a predicate has been “made healthy”, then further attempts to do so should bring about no further change.

Revisiting Healthiness

- We can now formalise our healthiness conditions as follows:
  
  - $mkH1(P) \triangleq ok \Rightarrow P$
  - $mkH2(P) \triangleq P; J$
  - $mkH3(P) \triangleq P; skip$
  - $isH1(P) \triangleq P = mkH1(P)$
  - $isH2(P) \triangleq P = mkH2(P)$
  - $isH3(P) \triangleq P = mkH3(P)$
  - $isH4(P) \triangleq P; true = true$

- Apart from $H4$, we have the same pattern:
  - a predicate transformer $mkH$
  - a predicate condition $isH$, defined in terms of the former
- We shall take a closer look at $mkH$.

A Notational Convention (obvious, yet confusing!)

- We have definitions provided for $mkH$
- We define $isH$ as $P = mkH(P)$
- It is standard practise in the UTP literature to use $H$ to denote both of these functions
- Which of $mkH$ or $isH$ is meant can (usually) be deduced from context
- We shall adopt this convention from now on.
**H1** is idempotent

We want to show that \( H_1 \circ H_1 = H_1 \)

Reduce Lhs to Rhs

\[
H_1(H_1(P)) = \text{" defn. } H_1 \text{ (a.k.a mkH1) "}
\]

\[
ok \Rightarrow (ok \Rightarrow P)
\]

\[
\neg ok \lor \neg ok \lor P
\]

\[
\text{" \lnot\lor\text{-def"}
\]

\[
ok \Rightarrow P
\]

\[
= \text{" defn. } H_1 \ "
\]

**H2** is idempotent

We want to show that \( H_2 \circ H_2 = H_2 \)

Reduce Lhs to Rhs

\[
H_2(H_2(P)) = \text{" defn. } H_2 \ "
\]

\[
(P; J); J
\]

\[
= \text{" \llcorner\lnot\llcorner\text{-assoc} \ "}
\]

\[
P; (J; J)
\]

\[
= \text{" Lemma: } J; J = J \ "
\]

\[
P; J
\]

\[
= \text{" defn. } H_2 \ "
\]

**Lemma Proof (I)**

Goal: \( J; J = J \)

Reduce lhs to rhs

\[
J; J
\]

\[
= \text{" defn. } J \ "
\]

\[
(((ok \Rightarrow ok') \land S' = S); ((ok \Rightarrow ok') \land S' = S)
\]

\[
= \text{" \lnot\lor\text{-def}, \lnot\text{-def}, substitution "}
\]

\[
\exists ok_m, S_m \bullet
\]

\[
(\neg ok \lor ok_m) \land S_m = S \land (\neg ok_m \lor ok') \land S' = S_m
\]

\[
= \text{" \lnot\lor\text{-def}, \exists\text{-1pt}, S_m = S "}
\]

\[
\exists ok_m \bullet (\neg ok \lor ok_m) \land (\neg ok_m \lor ok') \land S' = S
\]

\[
= \text{" shrink scope, \\lnot\lor\text{-distr}, \lor\text{-distr} "}
\]

\[
S' = S \land (\exists ok_m \bullet
\]

\[
\neg ok \land \neg ok_m \lor \neg ok \land ok'
\]

\[
\lor ok_m \land \neg ok_m \lor ok \land ok'
\]

\[
= \text{" contradiction, shrink scope, simplify "}
\]

\[
S' = S \land (\neg ok \land (\exists ok_m \bullet \neg ok_m) \lor \neg ok \land ok')
\]

\[
\lor ok_m \land \neg ok_m \lor ok \land ok'
\]

\[
= \text{" witness, simplify "}
\]

\[
S' = S \land (\neg ok \lor \neg ok \land ok)
\]

\[
\lor ok_m \land \neg ok_m \lor ok \land ok'
\]

\[
= \text{" \lor\land\text{-absorb} "}
\]

\[
S' = S \land (ok \Rightarrow ok')
\]

\[
= \text{" \lnot\lor\text{-def } "}
\]

\[
S' = S \land (ok \Rightarrow ok')
\]

\[
= \text{" defn. } J \ "
\]

**Lemma Proof (II)**

\[
S' = S \land (\exists ok_m \bullet
\]

\[
\neg ok \land \neg ok_m \lor \neg ok \land ok'
\]

\[
\lor ok_m \land \neg ok_m \lor ok \land ok'
\]

\[
= \text{" contradiction, shrink scope, simplify "}
\]

\[
S' = S \land (\neg ok \land (\exists ok_m \bullet \neg ok_m) \lor \neg ok \land ok')
\]

\[
\lor ok_m \land \neg ok_m \lor ok \land ok'
\]

\[
= \text{" witness, simplify "}
\]

\[
S' = S \land (\neg ok \lor \neg ok \land ok)
\]

\[
\lor ok_m \land \neg ok_m \lor ok \land ok'
\]

\[
= \text{" \lor\land\text{-absorb} "}
\]

\[
S' = S \land (ok \Rightarrow ok')
\]

\[
= \text{" \lnot\lor\text{-def } "}
\]

\[
S' = S \land (ok \Rightarrow ok')
\]

\[
= \text{" defn. } J \ "
\]

\[
J
\]
Aside: the (un-)Healthiness of $J$

- We have just seen that $J; J = J$, i.e. that $J$ is $H2$-healthy
- What about $H1(J)$ and $H3(J)$?
- Careful calculation shows
  
  $$
  \begin{align*}
  H1(J) = & \text{ ok } \Rightarrow J = \text{ skip } \\
  H3(J) = & J; \text{ skip } = \text{ skip }
  \end{align*}
  $$

- So $J$ is not $H1$ or $H3$, and attempts to “healthify” it using either $H1$ or $H3$ turn it into $\text{skip}$

H3 is idempotent

We want to show that $H3 \circ H3 = H3$
Reduce Lhs to Rhs

\[
\begin{align*}
H3(H3(P)) &= \text{ “ defn. } H3 \text{ ”} \\
(P; \text{skip})&: \text{skip} \\
&= \text{ “ } \llbracket -\text{assoc} \rrbracket \text{ ”} \\
l; (\text{skip}; \text{skip}) &= \text{ “ } \llbracket \text{skip }; \perp\text{-unit} \rrbracket \text{, with } P = \text{skip } \text{”} \\
l; \text{skip} &= \text{ “ defn. } H3 \text{ ”} \\
H3(P) &= H3(P)
\end{align*}
\]

Other Useful Properties of Healthiness

- We require healthiness transformers to be idempotent
- Another useful property is having independence of healthiness conditions:
  - the order in which most healthiness transformers are applied is immaterial.
- e.g.

\[
\begin{align*}
H1 \circ H2 &= H2 \circ H1 \\
H1 \circ H3 &= H3 \circ H1 \\
H3 \circ H2 &= H2 \circ H3
\end{align*}
\]

We say that $H1$, $H2$ and $H3$ “commute”.

Designs: a final comment

- Given $D$ which we know is a design (because we have shown it to be $H1$ and $H2$) we can write it in the form $P \vdash Q$
- Can we determine $P$ and $Q$, given $D$?
- Yes
  - If $D$ is $H1$- and $H2$-healthy, then
    $$
    D = (\neg D' \vdash D')
    $$

- The precondition is those situations that do not lead to $\text{ok}' = False$, i.e. $\neg D[\text{False/ok}']$
- The post condition is those situations that end with $\text{ok}' = True$, i.e. $D[\text{True/ok}']$
Why Commuting is good

▶ Commuting healthiness is very convenient.
   Knowing $P$ is both $H_1$ and $H_2$ allows us to replace it by either $H_1(P)$ or $H_2(P)$ in a proof.
   ▶ If $H_a$ and $H_b$ (say) don’t commute, then $H_a(H_b(P))$ and $H_b(H_a(P))$ are different.
   ▶ If $P$ is $(H_a \circ H_b)$-healthy, then we can replace $P$ in a proof by
     
     $H_a(P)$ or $H_a(H_b(P))$
     
     but not by $H_b(P)$.

▶ Why/Why not? (do in class)

Proof that $H_1$ and $H_2$ commute

▶ Goal: $H_1 \circ H_2 = H_2 \circ H_1$
   ▶ alternatively
     
     $H_1(H_2(P)) = H_2(H_1(P))$

▶ Strategy: reduce both sides to same

Proof that $H_1$ and $H_2$ commute (LHS)

\[
H_1(H_2(P)) = " \text{ defns., } H_1, H_2" \setminus (ok \Rightarrow (P; J))
\]

Proof that $H_1$ and $H_2$ commute (RHS)

\[
H_2(H_1(P)) = " \text{ defns., } H_1, H_2" \setminus (ok \Rightarrow (P; J))
\]

We have used $\langle \lor; \lor\rangle: (P \lor Q); R \equiv (P; R) \lor (Q; R)$ whose proof is left as a (voluntary) exercise.
### Monotonicity of HOFs

- We have an ordering on predicates based on refinement
  - If $P$ refines $S$, then we view $S$ as having less information than $P$, and we write $S \sqsubseteq P$.
  - We see *Chaos*, the “whatever” specification as least by this ordering
  - We view *miracle*, the “satisfy-anything” (infeasible) program as top-most
- We now introduce the notion of *monotonicity* for HOFs
- HOF $F$ is monotonic if, *forall* predicates $P$ and $Q$

\[
(P \sqsubseteq Q) \implies (F(P) \sqsubseteq F(Q))
\]

- Note how monotonicity can be defined as a 2nd-order predicate!
- This property states that if $F$ is monotonic then refining its argument refines its result.

### Program Language Constructs as HOFs

- We can re-cast much of our program language constructs as HOFs.
- For example, consider the while loop: $c \odot P$
  - We can consider this a function of $P$ (how?)
  - Simple, define
    \[
    WLOOP_c(P) \equiv c \odot P
    \]
- It turns out that $WLOOP_c$ is monotonic
- So if $P \sqsubseteq Q$ then
  \[
  WLOOP_c(P) \sqsubseteq WLOOP_c(Q),
  \]
  i.e. $c \odot P \sqsubseteq c \odot Q$

### 2-place HOFs

- With 2-place HOFs we can define the other language constructs
  
  \[
  SEQ(P, Q) \equiv P \sqcap Q
  \]
  \[
  COND_c(P, Q) \equiv P \sqcap c \sqsupset Q
  \]
- We say a 2-place HOF $F(\_, \_)$ is:
  - Monotonic in 1st arg. if $P \sqsubseteq Q \implies F(P, R) \sqsubseteq F(Q, R)$
  - Monotonic in 2nd arg. if $P \sqsubseteq R \implies F(R, P) \sqsubseteq F(R, Q)$
- $F(\_, \_)$ is simply Monotonic, if it is monotonic in both arguments
- Both $SEQ$ and $COND_c$ are (simply) Monotonic
- The notion of monotonicity extends to $n$-place HOFs in the obvious way.
Monotonicity preserved by composition

If \( F \) and \( G \) are monotonic, then so is \( F \circ G \)

\[
P \sqsubseteq Q \\
\Rightarrow \quad \text{“} \ G \text{ is monotonic “} \\
G(P) \sqsubseteq G(Q) \\
\Rightarrow \quad \text{“} \ F \text{ is monotonic “} \\
F(G(P)) \sqsubseteq F(G(Q))
\]

This generalises to \( n \)-place HOFs as well

Testing for Monotonicity

- There are two ways to test \( F \) to see if it is monotonic.
- The hard (direct) way:
  
  Prove the relevant (2nd-order) theorem:

\[
\forall P, Q \, (P \sqsubseteq Q) \Rightarrow (F(P) \sqsubseteq F(Q))
\]

- An easier (indirect way):
  
  Consider the definition of \( F \), which will look something like:

\[
F(P) \triangleq \text{a predicate mentioning } P \text{ somewhere}
\]

- By analysing the “predicate mentioning \( P \)” we can determine (to a great extent) if \( F \) is monotonic.

Monotonicity Analysis

- \( F(P) \) is monotonic if \( every \) occurrence of \( P \) in its definition is at a “positive” location.
- A location is \( positive \) if we pass down through an \( even \) number of negations to get to it.
  - a negation here is either
    (i) going through an application of \( \neg \), or
    (ii) going down the lhs of \( \Rightarrow \) (why?)
- A location is \( negative \) if we pass down through an \( odd \) number of negations to get to it.
- If \( P \) occurs in both positive and negative locations, then its occurrences are said to be \( mixed \).
  - passing through either argument of \( \equiv \) results in a mixed occurrence.
- \( U \cdot (TP)^2 \) keeps track of a location’s polarity as focus is moved into a predicate.

Anti-Monotonicity

- HOF \( F \) is anti-monotonic if, \( forall \) predicates \( P \) and \( Q \)

\[
(P \sqsubseteq Q) \Rightarrow (F(Q) \sqsubseteq F(P))
\]

- \( F(P) \) is anti-monotonic if every occurrence of \( P \) is in a negative location.
Monotonicity Example I

Is $F(P) \equiv P \land (\exists x \bullet Q \Rightarrow P)$ monotonic?

$p \land (\exists x \bullet Q \Rightarrow P)$
"mark occurrence polarity"
$p^+ \land (\exists x \bullet Q^- \Rightarrow P^+)^+$
"both $P$ are labelled with $+$"
$F$ confirmed monotonic

Monotonicity Example II

Is $F(Q) \equiv P \land (\exists x \bullet Q \Rightarrow P)$ monotonic?

$p \land (\exists x \bullet Q \Rightarrow P)$
"mark occurrence polarity"
$p^+ \land (\exists x \bullet Q^- \Rightarrow P^+)^+$
"the sole $Q$ is labelled with $−$"
$F$ confirmed non-monotonic

Monotonicity Example III

Is $F(P) \equiv P \land (\forall x \bullet P \Rightarrow Q)$ monotonic?

$p \land (\forall x \bullet P \Rightarrow Q)$
"mark occurrence polarity"
$p^+ \land (\forall x \bullet P^+ \Rightarrow Q^+)^+$
"both $P$ are labelled with $+$"
$F$ confirmed monotonic

Monotonicity Example V

Is $F(P) \equiv P \land (P \equiv Q)$ monotonic?

$p \land (P \equiv Q)$
"mark occurrence polarity"
$p^+ \land (P^\pm \equiv Q^\pm)^+$
"so, not monotonic then..."
$p \land Q$
"logic (exercise)"
$p^+ \land Q^+$
"so is monotonic after all !!!"

This example shows the limitations of this technique for testing for monotonicity.
Polarity Testing: preparation

- The last example shows a limitation of polarity marking for assessing monotonicity.
- The problem area was any operator that introduces mixed polarity.
- One strategy is to simplify the predicate by replacing $P \equiv Q$ by either
  \[(P \Rightarrow Q) \land (Q \Rightarrow P)\]
  or
  \[(P \land Q) \lor (\neg P \land \neg Q)\]
- Then do further simplification, down to the “the or-ing of the and-ing of possibly negated atomic predicates”.
- However this is getting almost as complicated as doing a direct proof of monotonicity.

Fixpoints

- For a function $f$ whose input and output types are the same (endofunction), a fixed point (fixpoint) is a solution to the equation
  \[x = f(x)\]
- Fixpoints crop up everywhere in CS theory:
  - any time iteration or recursion is involved
- There are at least two uses of fixpoints in UTP:
  1. defining the meaning of recursion/iteration
  2. healthy predicates are fixpoints of “healthifiers”

Further Significance of Monotonicity

- Knaster-Tarski Fixpoint Theorem (1955)
  - Let $L$ be a complete lattice,
  - and $F : L \rightarrow L$ be a monotonic function,
  - then the fixpoints of $F$ form a complete lattice.
- A complete lattice is non-empty, so at least one fixpoint must exist.
- We must have a least and greatest fixpoint.
- Under the right conditions ($F$ continuous, or $n$ ranging transfinitely), then we can compute the least fixpoint as
  \[\text{lub}\{F^n(\bot) \mid n \in 0 \ldots\}\]

Healthy Lattices

- Healthy Predicates are fixpoints of “healthifier” functions
- If those “healthifiers” are monotonic, then the healthy predicates themselves form a complete lattice w.r.t. refinement.
- Are $H1$, $H2$ and $H3$ monotonic?
### Monotonicity Check: H

**H1**

\[ H_1(P) = \text{ok} \implies P^+ \] (monotonic)

**H2**

\[ H_2(P) = P; J \]
\[ = \exists O_m \bullet P^+[O_m/O'] \land J[O_m/O] \] (monotonic)

**H3**

\[ H_3(P) = P; Skip \]
\[ = \exists O_m \bullet P^+[O_m/O'] \land Skip[O_m/O] \] (monotonic)

### Calculating While-Loop Semantics

So often we can calculate the while-loop semantics as

\[ c \otimes P = \bot \]
\[ \land P; \bot \triangleleft c \triangleright Skip \]
\[ \land P; (P; \bot \triangleleft c \triangleright Skip) \triangleleft c \triangleright Skip \]
\[ \land P; (P; (P; \bot \triangleleft c \triangleright Skip) \triangleleft c \triangleright Skip) \triangleleft c \triangleright Skip \]
\[ \vdots \]

where \( \bot = \text{Chaos} = \text{true} \)

### Recursion in UTP

- Healthy predicates \((H_1, H_2, H_3)\) form a complete lattice under the refinement ordering
- All our programming language HOFs are monotonic
- So we can define the while loop semantics as the least fixed point of an appropriate HOF w.r.t the refinement ordering:

\[ c \otimes P \trianglelefteq \text{lfp}(\lambda W \bullet P; W \triangleleft c \triangleright \text{Skip}) \]

- Our functions are usually continuous, so we can compute the fixpoint:

\[ \text{lfp} F = \bigwedge \{ F^n(\text{Chaos}) \mid n \in 0 \ldots \} \]

### Refinement Revisited

- We are now in a position to look at refinement again
- A key property we want of refinement is that it be compositional:
  - we should be able to refine a specification into code in small steps.
- This breaks into two aspects:
  - **Transitivity**: we want to proceed by stages

\[ (S \sqsubseteq D) \land (D \sqsubseteq P) \implies S \sqsubseteq P \]

We can refine \( S \) first to \( D \), and then refine that to \( P \)
- **Monotonicity**: we want to work on sub-parts

\[ D \sqsubseteq P \implies F(D) \sqsubseteq F(P) \]

We can refine \( F(D) \) by refining component \( D \) into \( P \).
Spec-/Program-Construct Monotonicity

- Having specification and programming constructs that are monotonic is very important in allowing practical refinement.
- We have seen that the program constructs (SEQ, COND<sub>c</sub>, WLOOP<sub>c</sub>) are monotonic.
- What about specification frames (w : [P, Q]), or designs (P ⊢ Q)?
- Let's define HOFs for these:
  
  \[ SPEC_w(P, Q) \triangleq w : [P, Q] \]
  
  \[ DSGN(P, Q) \triangleq P \vdash Q \]

  Are they monotonic?

Designs, Frames and (Anti-)Monotonicity

- Expand our definitions of SPEC<sub>w</sub> and DSGN:
  
  \[ SPEC_w(P, Q) = ok \land P \Rightarrow ok' \land Q \land S' = S \]
  
  \[ DSGN(P, Q) = ok \land P \Rightarrow ok' \land Q \]

- We find that both are monotonic in their second argument, but anti-monotonic in their first.
- This explain why in refinement laws, in order to show that
  
  \((P \vdash Q) \sqsubseteq (R \vdash S)\)

  it is necessary to show \(R \sqsubseteq P\), rather than vice-versa.
- In other words, preconditions appear in a negative position.
Class 26

Live Proof Demo using $U \cdot (TP)^2$

That's all, folks!
Class 27

Correctness via Verification Conditions

- Introduce notion of assertion statements
- Programmer adds assertions to program at appropriate points
- Verification Conditions are automatically extracted
- Verification conditions are proven (automatically?).
- Technique tailored for sequential imperative programs

Assertions

- An assertion $c\bot$ is a (specification) statement that a condition holds.
  - if $c$ is true, then $c\bot$ behaves like $\text{Skip}$
  - if $c$ is false, then $c\bot$ behaves like $\text{Chaos}$
  
  $c\bot \equiv \text{Skip} \triangleleft c \triangleright \text{Chaos}$

- Often written as as $\{c\}$ in the literature
- In UTP we can reason about them directly (e.g.):

  $b\bot; c\bot = (b \land c)\bot$

Verification Conditions

- A verification condition (VC) is a predicate relating two assertions
- Verification conditions depend on the program fragment between the two assertions.
- Given (suitably) annotated program $\text{pre}\bot; \text{prog}; \text{post}\bot$:
  - we can automatically derive a VC involving $\text{pre}\bot$ and $\text{post}\bot$
  - that depends on the structure and contents of $\text{prog}$.
- Hope: VCs are simple enough to be proven automatically.
- Think of them as machine-readable (machine-verifiable?) comments!
Appropriate Annotations

A program is appropriately annotated if
▶ in a sequence $P_1; P_2; \ldots; P_n$ there is an assertion before every statement that is not an assignment or Skip.
▶ in every while-loop there is an (invariant) assertion at the start of the loop body
▶ The first assertion should be a consequence of the pre-condition from the specification
▶ The last assertion should imply the specification post-condition.
▶ Note: this approach works for post-conditions that are conditions (i.e. snapshots of state, and not before-after relations).

Annotation Example

The integer division algorithm from [Hoare69] with annotations:

\[
\begin{align*}
\text{true}_\bot; \\
\mathit{r} & := \mathit{x}; \\
\mathit{q} & := 0; \\
(y \leq r)* & ((\mathit{x} = \mathit{r} + y * \mathit{q})_\bot; \\
\mathit{r} & := \mathit{r} - y; \\
\mathit{q} & := \mathit{q} + 1; \\
(\mathit{x} = \mathit{r} + (y * \mathit{q}) \land \mathit{r} < y)_\bot
\end{align*}
\]

Generating VCs

▶ VCs are generated for all the program statements
▶ We shall define VC generation recursively over program structure

\[
\begin{align*}
genVC : \text{Program} & \Rightarrow \mathcal{P} \text{ VC}
\end{align*}
\]

VC generation (\(\cdot\))

▶ Given that the last statement is an assignment

\[
\begin{align*}
genVC(p_\bot; P_1; \ldots; P_{n-1}; v := e; q_\bot) & = genVC(p_\bot; P_1; \ldots; P_{n-1}; q[e/v]_\bot)
\end{align*}
\]

We drop the assignment and replace all free occurrences of \(v\) in the last assertion by \(e\).

▶ Given that the last statement is an not an assignment

\[
\begin{align*}
genVC(p_\bot; P_1; \ldots; P_{n-1}; r_\bot; P_n; q_\bot) & = genVC(r_\bot; P_n; q_\bot) \\
& \cup genVC(p_\bot; P_1; \ldots; P_{n-1}; r_\bot)
\end{align*}
\]

We process the last statement and the recursively treat the rest of the sequence.
VC generation (\(\equiv\))

\[ \text{genVC}(p \perp; x := e; q \perp) \equiv \{ p \Rightarrow q[e/x] \} \]

The pre-condition must imply the post-condition with \(v\) replaced by \(e\).

Example:

\[ \text{genVC}(p \perp; r := r - y; q \perp) \hat{=} \{ p \Rightarrow q[r - y/r] \} \]

So, given \((x = r + y * q) \perp; r := r - y\), what assertion at the end will work?

- Not \((x = r + y * q) \perp\), because
  \[ x = r + y * q \not\Rightarrow x = r - y + y * q \]

- Assertion \(x = r + y * (q + 1) \perp\) does work, because
  \[ x = r + y * q \Rightarrow x = r - y + y * (q + 1) \]

VC generation (\(<\ Demo>)

\[ \text{genVC}(p \perp; P_1 \text{ if } c \text{ then } P_2; q \perp) \]

\[ = \text{genVC}((p \land c) \perp; P_1; q \perp) \]

\[ \cup \]

\[ \text{genVC}((p \land \neg c) \perp; P_2; q \perp) \]

We add the branch condition to the pre-condition of the then-branch, and its negation to that of the else-branch.

VCs generated by our example.

- Do in class
- Solution

\[
\begin{align*}
\text{true} & \Rightarrow x = x \land 0 = 0 \\
\text{true} & \Rightarrow x = x \land q = 0 \\
r = x \land q = 0 & \Rightarrow x = r + (y \ast q) \\
x = r + y \ast q \land \neg (y \leq r) & \Rightarrow x = r + (y \ast q) \land r < y \\
x = r + y \ast q \land y \leq r & \Rightarrow x = r - y + (y \ast (q + 1))
\end{align*}
\]

- These are simple enough to do by hand (or with \(U \cdot (TP)^2\)).
- Automated provers with good arithmetic facilities should also handle these.
VCs in the “real world”

VC generation and proof is used in a wide range of verification tools:
- SparkADA — dialect of ADA used by Praxis used in the Tokeneer project
- Java/ESC — Extended Static Checker for Java (uses Java Modelling Language for assertion annotations)
- Spec# — Microsoft’s Specification/Verification oriented language

Avoiding Annotations

- Using VCs requires lots of annotations by the programmer
- Can be good discipline for documentation code
- Can it be avoided?
- Can it be automated?

Given \( \text{pre} \vdash \text{post} \), can we generate the internal assertions, and then the VCs?

Another look at refinement

- We want to determine if \( \text{pre} \vdash \text{post}' \) is refined by \( \text{prog} \)

\[
\begin{align*}
\text{pre} \vdash \text{post}' &\subseteq \text{prog} \\
&= [\text{prog} \Rightarrow \text{pre} \vdash \text{post}'] \\
&= [\text{prog} \Rightarrow (\text{ok} \land \text{pre} \Rightarrow \text{ok}' \land \text{post}')] \\
&= [\text{ok} \land \text{pre} \Rightarrow (\text{prog} \Rightarrow \text{ok}' \land \text{post}')] \\
&= [\text{ok} \land \text{pre} \Rightarrow \forall \text{ok}_m, S_m \Rightarrow \text{prog} \Rightarrow \text{ok}' \land \text{post}'] \\
&= [\text{ok} \land \text{pre} \Rightarrow \forall \text{ok}_m, S_m \Rightarrow \text{prog} \land (\text{ok}' \Rightarrow \neg \text{post}')] \\
&= [\text{ok} \land \text{pre} \Rightarrow \neg (\text{prog} \land (\text{post}'))]
\end{align*}
\]

- \( \neg (\text{prog} \land (\neg \text{post}')) \) is the weakest condition under which \( \text{prog} \) is guaranteed to achieve \( \text{post}' \)

Weakest Precondition

- We define a new language construct

\[ \text{prog wp post} \]

It means the weakest pre-condition under which running \( \text{prog} \) will guarantee outcome (condition) \( \text{post} \).

- In UTP we can define it as

\[
Q \text{ wp } r \equiv \neg (Q; \neg r')
\]

Look familiar?

- It can be shown to obey the following laws:

\[
\begin{align*}
x &:= e \text{ wp } r \equiv r[e/x] \\
P; &Q \text{ wp } r \equiv P \text{ wp } (Q \text{ wp } r) \\
(P \triangleleft c \triangleright Q) \text{ wp } r &\equiv (P \text{ wp } r) \triangleleft c \triangleright (Q \text{ wp } r) \\
(P \sqcap Q) \text{ wp } r &\equiv (P \text{ wp } r) \sqcap (Q \text{ wp } r)
\end{align*}
\]
Using WP

- The idea is to start with the post-condition and work backwards, generating weakest conditions.

\[ P_1; \ldots; P_{n-2}; P_{n-1}; P_n \text{ wp post} \equiv P_1; \ldots; P_{n-2}; P_{n-1} \text{ wp } (P_n \text{ wp post}) \]

\[ = P_1; \ldots; P_{n-2} \text{ wp } (P_{n-1} \text{ wp } (P_n \text{ wp post})) \]

\[ = P_1 \text{ wp } (\ldots P_{n-2} \text{ wp } (P_{n-1} \text{ wp } (P_n \text{ wp post}))) \ldots \]

- We then show that the precondition \( \text{pre} \) implies the overall weakest precondition

\[ \text{pre} \Rightarrow P_1; \ldots; P_{n-2}; P_{n-1}; P_n \text{ wp post} \]

The problem with WP

- Where is the while-loop?

- We can show the following for WP and while-loops:

\[(c \ast P) \text{ wp } r \equiv w \text{ such that } \]

\[ w \Rightarrow (\neg c \Rightarrow r) \]

\[ w \Rightarrow c \Rightarrow P \text{ wp } w \]

\[ w \text{ is the weakest such predicate} \]

- Calculating \( c \ast P \text{ wp } r \) involves some form of a search for such a \( w \).

- Automating it successfully is equivalent to solving the Halting Problem (impossible).

WP for while loops

- If the user supplies an Invariant \( \text{inv} \) and variant \( V \), then we can define \( c \ast P \text{ wp } r \)

- If we ignore termination, we can define the following

\[(c \ast P) \text{ wp } r \equiv \text{ inv} \wedge (c \wedge \text{ inv} \Rightarrow P \text{ wp inv}) \wedge (\neg c \wedge \text{ inv} \Rightarrow r) \]

(technically this is weakest \(\text{liberal} \) precondition — WLP)

- We can automatically put in placeholder names for \( \text{inv} \) (and \( V \)), but at some stage the user will have to decide what they should be.
This class was live proofs of the wp laws from the previous class
Real Life: Mondex Electronic Purse

Goal — Smart card as electronic purse (1990s), certified to ITSEC E6. Verified security protocol.
http://www.mondex.com

Approach — required by E6:
▶ formal model of security policy
▶ formal model of security architecture
▶ refinement proof that architecture satisfies policy.

Method — Z, based on predicate calculus and ZF set theory

Who —
Susan Stepney (Logica)
David Cooper (NatWest)
Jim Woodcock (Oxford)

Mondex: Key Results

http://www-users.cs.york.ac.uk/~susan/bib/ss/z/monog.htm

▶ Design was proved correct

▶ Scale (380 pages):
  ▶ abstract model: 20 pages
  ▶ concrete model: 60 pages
  ▶ hand proof: 200 pages
  ▶ support stuff: 100 pages

▶ FM within process:
  ▶ found logging protocol error
  ▶ under budget, ahead of schedule

Grand Challenge Pilot Project

▶ In January 2006, the Mondex verification was introduced as a pilot-project for GC6 (Grand Challenge: Dependable Systems Evolution)

▶ Original Mondex verification used hand proofs.

▶ Idea: ask formal tool developers to apply their tools to the verification problem.

▶ At least 8 took up the challenge:
  ▶ Alloy, ASM, KIV, Event-B, USE, RAISE, Z/Eves, Perfect Developer, π-calculus
Mondex Pilot Project outcomes

- The formalisms and tools used varied a lot in terms of power and expressiveness.
- Surprisingly almost all of the groups got very similar results, finding similar errors.
- Key conclusions:
  - Using tool-support to formally analyse a real-world system like Mondex is quite feasible.
  - The precise choice of tool is often not that important.
- A special issue (Vol. 20, No. 1, Jan 2008) of the Formal Aspects of Computing journal has papers detailing these results.

Real Life: Needham-Schroeder Public Key Protocol

- Want to establish mutual authentication between initiator A and responder B.
- Using public-key protocol:
  - Agents have public keys known to all \((K_a, K_b)\)
  - Agents have secret keys, known only to themselves \((K_a^{-1}, K_b^{-1})\)
  - Agents can generate nonces (random numbers) \((N_a, N_b)\)
- The Needham-Schroeder Public Key Protocol (NS-PKP)
  - published in 1978
  - uses a 7-message sequence to ensure A and B know they are talking to each other.

A Formalism(?) for Describing Protocols

- If agent A includes its own name in a message, we denote that simply as \(A\).
- \(\{M\}_k\) denotes message \(M\) encrypted with key \(k\).
- We denote A’s public/secret keys respectively as \(PK(A), SK(A)\).
- We build up composite messages using dots \((A.b.\{X\}_k)\).
- A message \(m.n\) from A to B is described as \(A \rightarrow B: m.n.o\)

NS-PKP (3-step version)

- We shall focus on a shorter 3-step version
- The protocol:
  1. \(A \rightarrow B: A.B.\{N_a.A\}_{PK(B)}\)
     - A sends B his name and nonce, encoded with B’s public key.
  2. \(B \rightarrow A: B.A.\{N_a.N_b\}_{PK(A)}\)
     - B uses his private key to decode A’s message, and replies with A’s nonce and his own, encrypted for A’s eyes only
  3. \(A \rightarrow B: A.B.\{N_b\}_{PK(B)}\)
     - A decodes the previous message and send B’s nonce back
- At the end both A and B are convinced they are talking to each other, because there is no way anyone else could get at the nonces …
17 years later

Gavin Lowe, then a postdoc at Oxford, encodes NS-PKP in CSP.

\[ a, b, a.b, a.b.c, \ldots \in \text{Events} \]

\[ g \in \text{Guard} \]

\[ P, Q, R, \in \text{Proc} \]

\[ \text{Stop} \] do nothing, not even terminate
\[ \text{Skip} \] do nothing and terminate
\[ a \rightarrow P \] do \( a \), then act like \( P \)
\[ g \& P \] guarded process
\[ P \square Q \] sequential composition
\[ P \sqcap Q \] internal choice
\[ P \sqcup Q \] external choice
\[ P \parallel A \] parallel composition
\[ P \setminus A \] event hiding
\[ \text{Chaos} \] do anything

CSP Tool Support

- A proof by hand of correctness would be hard and error-prone
- Gavin Lowe used a tool called “FDR”
  - “FDR” — Failures-Divergence Refinement
  - Reads an ASCII syntax version of CSP
  - Does exhaustive search to check an assertion
  - See http://www.fsel.com

Failure!

- The check with FDR failed
- NS-PKP was found to be vulnerable to a “man-in-the-middle” attack
- This attack had gone unnoticed for 17 years!

The Attack

- It interleaves two runs \( \alpha \) and \( \beta \), one between \( A \) and \( I \), the other between \( I \) imitating \( A \) to \( B \) (here denoted as \( I(A) \)).
- \[
\begin{align*}
A & \rightarrow I : A.I.\{N_a.A\}_{PK(I)} \\
I(A) & \rightarrow B : A.B.\{N_a.A\}_{PK(B)} \\
B & \rightarrow I(A) : B.A.\{N_a,N_b\}_{PK(A)} \\
I & \rightarrow A : I.A.\{N_a,N_b\}_{PK(A)} \\
A & \rightarrow I : A.I.\{N_b\}_{PK(I)} \\
I(A) & \rightarrow B : A.B.\{N_b\}_{PK(B)}
\end{align*}
\]
Corrections

Gavin Lowe then derived a corrected protocol:

1. \( A \rightarrow B : A.B \{ N_a, A \}^{PK(B)} \)
   \( A \) sends \( B \) his name and nonce, encoded with \( B \)'s public key.
2. \( B \rightarrow A : B.A \{ N_b, N_b, B \}^{PK(A)} \)
   \( B \) uses his private key to decode \( A \)'s message, and replies with \( A \)'s nonce and his own, and his own identity, encrypted for \( A \)'s eyes only
3. \( A \rightarrow B : A.B \{ N_b \}^{PK(B)} \)
   \( A \) decodes the previous message and send \( B \)'s nonce back

He checked it also with FDR — it came up clean.

This protocol known (in educated circles) as the Needham-Schroeder-Lowe protocol.

Real Life: SparkADA

- SparkADA: a subset of ADA with a formal semantics
- Spark Examiner: tool supporting correct refinement to SparkADA
- Developed and marketed by Praxis High Integrity Systems
- http://www.praxis-his.com/sparkada/

The big idea

- SparkADA is full ADA without any language constructs that make formal reasoning (too) difficult
- Tools support a very high degree of rigorous formal development
- Programmer productivity is much higher than normal
- Errors found after product shipped are way lower.

Tokeneer Project

- The Tokeneer project was done by Praxis for the U.S. National Security Agency.
  (biometric access control)
- Key results:
  - lines of code: 9939
  - total effort (days): 260
  - productivity (lines of code per day, overall) : 38
  - productivity (lines of code per day, coding phase) : 203
  - defects discovered by NSA after delivery: 1
- Very high quality outcome by industry standards
Tokeneer Project (ongoing)

- NSA did project to demonstrate that their requirement for formal methods was reasonable and cost-effective.
- NSA release it open-source
  http://www.adacore.com/home/gnatpro/tokeneer/
  (requirements; specification; code; proofs)
- 4 more bugs uncovered in code
- Work by Jim Woodcock uncovered nine requirements-related defects

Recent FM Research at TCD: overview

- Handel-C semantics
- UTP, esp. slotted Circus
- Flash Memory modelling
- Global Computation (Matthew Hennessey, et al.)
- OS kernel verification

Recent Research at TCD: “slotted Circus”

- We are using UTP to model hardware languages like Handel-C
  - program-like syntax (C with parallelism)
  - synchronous clocks — assignments synchronise with clock
  - true parallelism — \( x, y := y, x \) works directly
  - inter-thread channel communication
- The language Circus describes concurrent/shared-variable programs
  - has a UTP semantics
  - key auxiliary observation:
    Traces : sequences of Events (\( tr \), and \( tr' \))
  - Developed by Jim Woodcock and Ana Cavalcanti at York
- In TCD we have slotted-Circus
  - Language has a Wait (for clock tick) primitive.
  - Traces are chopped into discrete \textit{time-slots}.

Recent Research at TCD: “slotted Circus++”

- One project looked at giving a semantics to \textit{priority}
  - Concept of \textit{event} priority is non-trivial and involves time in an essential way
  - PhD, Paweł Gancarski (2011)
- Another project looked at giving a UTP semantics to \textit{probability}
  - Done already, relating starting state to sets of final distributions
  - The trick was to get a homogeneous relations (same start/finish type)
  - We relate start distributions to final ones (harder than it sounds)
  - PhD, Riccardo Bresciani (2012)
- Both projects looked at extensions to slotted Circus.
Recent Research at TCD: Flash Memory

- One of the grand challenge pilot studies was proposed by NASA JPL
  - verify a POSIX filesystem built on Flash Memory
- We looked at modelling the Flash Memory devices themselves (in Z)
- We also looked at modelling the hardware/software interface and verifying Flash Memory internal control FSMs against that interface
- We found errors in the Nand Flash Specification
  - Open Nand Flash interface (ONFi)

Recent Research at TCD: Kernel Verification (I)

- Current spacecraft have many special purpose computers
- ESA wants to merge these into one general purpose computing platform
  - to save weight
  - known as Integrated Modular Avionics (IMA)
- Aircraft did this 20 years ago
- IMA requires an operating system that securely partitions applications
  - each app has its own partition
  - all inter-app communication and access to hardware resources is via o/s kernel

Recent Research at TCD: Kernel Verification (II)

- Our task:
  - develop a (informal) Reference Specification for a separating partitioning µkernel.
  - Scope out and choose a formalism and tools to formally verify a kernel implementation, to CC EAL5+ (probably EAL7+)
  - We chose Isabelle/HOL
  - Formalise the specification using the chosen formalism
  - Do a dry run at the verification to get an idea of feasibility and cost.

FM & FP — a teaching history

- The availability of courses on both formal methods (FM) and functional programming has changed in recent years on the CS B.A. (Mod.), now the ICS.
- Academic year 2010/11 and before:
  - CS3001, Formal Methods, JS Elective, 5 ECTS
  - CS4011, Functional Programming, SS Elective, 10 ECTS
- Academic year 2011/12 and after:
  - CS4003, Formal Methods, SS Elective, 5 ECTS
  - CS3016, Functional Programming, JS Mandatory, 5 ECTS
  - CS4012, Topics in Functional Programming, SS Elective, 5 ECTS
2011/12 — a transition year

- At the start of 2011/12:
  - JS students had no FM, and no choice to do it, and had to do FP.
  - SS students had no FP, but some had chosen it. Also, some had done FM, and wanted more.

- Academic year 2011/12
  - CS4003 was a follow-on to CS3001 from 2010/11
  - CS3016/CS4012 Functional Programming was a joint JS/SS class.

- Academic year 2012/13/14 ...
  - CS4003 is basically CS3001 from 2010/11 and before, called CS3BA31 back in 2008/09
  - CS3016 is a reduced version of CS3016/CS4012 from 2011/12, with some content overlapping with CS4011 in 2010/11.

What’s wait all about?

- For reactive systems (I/O performing) we need to distinguish non-termination from unrecoverable errors.
- Add extra boolean auxiliary variable: wait
  - wait = True — process is waiting for events (not terminated)
  - wait = False — process has terminated, no longer waiting
- We have three valid combinations of ok and wait:
  - ¬ok — divergence has occurred
  - ok ∧ wait — process is stable and still running
  - ok ∧ ¬wait — process has terminated successfully.
Class 30

FM & FP — a teaching history

- The availability of courses on both formal methods (FM) and functional programming has changed in recent years on the CS B.A. (Mod.), now the ICS.
- Academic year 2010/11 and before:
  - CS3001, Formal Methods, JS Elective, 5 ECTS
  - CS4011, Functional Programming, SS Elective, 10 ECTS
- Academic year 2011/12 and after:
  - CS4003, Formal Methods, SS Elective, 5 ECTS
  - CS3016, Functional Programming, JS Mandatory, 5 ECTS
  - CS4012, Topics in Functional Programming, SS Elective, 5 ECTS

2011/12 — a transition year

- At the start of 2011/12:
  - JS students had no FM, and no choice to do it, and had to do FP.
  - SS students had no FP, but some had chosen it. Also, some had done FM, and wanted more
- Academic year 2011/12
  - CS4003 was a follow-on to CS3001 from 2010/11
  - CS3016/CS4012 Functional Programming was a joint JS/SS class.
- Academic year 2012/13/14 …
  - CS4003 is basically CS3001 from 2010/11 and before, called CS3BA31 back in 2008/09
  - CS3016 is a reduced version of CS3016/CS4012 from 2011/12, with some content overlapping with CS4011 in 2010/11.

CS4003: relevant exam papers

- 3BA31 2008/09 (Q4 a bit off-topic)
- CS3001 2009/10
- 3BA31 2010/11

But not CS4003 2011/12!
Good Luck

- CS3001 has been an introduction to Formal Methods
- You have been exposed to key ideas about very rigorous reasoning about program correctness
- Even if you never apply these techniques, hopefully they have given you key intellectual tools for program problem solving:
  - pre/post-conditions
  - loop invariants and variants
  - specification-to-program refinement
- Thank you all for your attention and involvement.