We want to formalise our notion of healthiness.

Remember, the following are our conditions:

- **H1**  
  \[ P = \text{ok} \Rightarrow P \]

- **H2**  
  \[ P = P; \ j \]

- **H3**  
  \[ P = P; \ \text{skip} \]

- **H4**  
  \[ P; \ \text{true} = \text{true} \]

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 \( \text{True} \) if the predicate is **H1**-healthy:
  
  \[ \text{isH1}(P) \equiv P = (\text{ok} \Rightarrow P) \]

- We refine this further by introducing another function that given a predicate returns an appropriately modified predicate:
  
  \[ \text{mkH1}(P) \equiv \text{ok} \Rightarrow P \]
  \[ \text{isH1}(P) \equiv P = \text{mkH1}(P) \]

- 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 \( \text{about} \) predicates (e.g. \( \text{isH1} \))
  - We are introducing functions that \( \text{transform} \) predicates (e.g. \( \text{mkH1} \))
  - UTP also allows quantifier variables to range over predicates
### 2nd-Order Predicates

- A first-order predicate $P$ is a function over environment variables, returning $\text{True}$ or $\text{False}$:
  
  $$[P] : \text{Env} \rightarrow \mathbb{B}$$

- A second-order predicate $P$ is a function from predicates to $\text{True}$ or $\text{False}$:
  
  $$[P] : \text{Pred} \rightarrow \mathbb{B}$$

- We can expand this as:
  
  $$[P] : (\text{Env} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

- So $\text{isH1}$ is just such a 2nd-order predicate.

### Predicate Transformers

- A predicate-transformer $F$ is a function over predicates, returning a predicate as a result:
  
  $$[F] : \text{Pred} \rightarrow \text{Pred}$$

- We can expand this also as
  
  $$[F] : (\text{Env} \rightarrow \mathbb{B}) \rightarrow (\text{Env} \rightarrow \mathbb{B})$$

- Function $\text{mkH1}$ is a predicate transformer

### Changing the Game (I)

- We add a new category to our language: that of a higher-order definition

  $$P, Q, R, S \in \text{PVar} \quad \text{predicate (meta-)variables}$$

  $$\text{H, F} \in \text{HOF} \quad \text{higher order functions}$$

  $$\text{HOFDef} ::= H(P) \triangleq \text{body} \quad \text{HOF definition}$$

- We extend our notion of predicate to allow the application of a HOF to a predicate argument

  $$P \in \text{Pred} ::= \ldots | H(P)$$

### Changing the Game (II)

- Given $H(P) \triangleq \text{body}$, we have a new law:

  $$H(\text{MyPred}) = \text{body}[\text{MyPred} / P]$$

  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

  $$\text{Pred} ::= \ldots | \forall P \cdot H(P)$$

  we get “full 2nd-order logic”
2nd-order Logic

- 2nd-order logic is strictly more powerful than 1st-order
- When quantifying over a predicate, it is interpreted as a function from the environment to boolean
  - So
  \[ \forall P \cdot H(P) \]
  is interpreted as meaning:
  “for all functions \( F : \text{Env} \rightarrow \mathbb{B} \), the predicate \( H(F) \) evaluates to true in any environment”

- UTP is based on full 2nd-order logic
  - needed to treat recursion properly
  - needed to reason about healthiness.

Revisiting Healthiness

- We can now formalise our healthiness conditions as follows:
  \[
  \begin{align*}
  \text{mkH1}(P) & \triangleq \text{ok} \Rightarrow P \\
  \text{mkH2}(P) & \triangleq P; J \\
  \text{mkH3}(P) & \triangleq P; \text{skip} \\
  \text{isHi}(P) & \triangleq P = \text{mkHi}(P), \quad i \in 1, 2, 3 \\
  \text{isH4}(P) & \triangleq P; \text{true} = \text{true}
  \end{align*}
  \]

- Apart from \( \text{H4} \), we have the same pattern:
  - a predicate transformer \( \text{mkH} \)
  - a predicate condition \( \text{isH} \), defined in terms of the former
- We shall take a closer look at \( \text{mkH} \).

The Nature of \( \text{mkH} \)

- Consider the application \( \text{mkH}(P) \)
- We can interpret \( \text{mkH} \) as a “healthifying” function (i.e. it makes predicates healthy)
- What if \( P \) is already healthy?
  - Then it satisfies \( \text{isH} \), which says \( P = \text{mkH}(P) \)
  - So \( \text{mkH} \) should not change an already healthy predicate
- We are led to the following requirement for any “healthifier” \( \text{mkH} \):
  \[
  \begin{align*}
  \text{mkH}\left(\text{mkH}(P)\right) & = \text{mkH}(P) \\
  \text{mkH} \circ \text{mkH} & = \text{mkH}
  \end{align*}
  \]
  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.

A Notational Convention (obvious, yet confusing!)

- We have definitions provided for \( \text{mkH} \)
- We define \( \text{isH} \) as \( P = \text{mkH}(P) \)
- It is standard practise in the UTP literature to use \( \text{H} \) to denote both of these functions
- Which of \( \text{mkH} \) or \( \text{isH} \) is meant can (usually) be deduced from context
- We shall adopt this convention from now on.
**H1 is idempotent**

We want to show that $H1 \circ H1 = H1$

Reduce Lhs to Rhs

1. $H1(H1(P))$
   - " defn. $H1$ (a.k.a $mkH1$)"
   - $ok \Rightarrow (ok \Rightarrow P)$
   - " $\iff$-def""
   - $\neg ok \lor \neg ok \lor P$
   - " $\lor$-idem""
   - $\neg ok \lor P$
   - " $\iff$-def""
   - $ok \Rightarrow P$

2. " defn. $H1$"

$H1(P)$

---

**H2 is idempotent**

We want to show that $H2 \circ H2 = H2$

Reduce Lhs to Rhs

1. $H2(H2(P))$
   - " defn. $H2$"
   - $(P; J): J$
   - " $\triangleright$-assoc"
   - $P; (J; J)$
   - " Lemma: $J; J = J$"
   - $P; J$
   - " defn. $H2$"

2. $H2(P)$

---

**Lemma Proof (I)**

Goal: $J; J = J$

Reduce lhs to rhs

1. $J; J$
   - " defn. $J$"
   - $((ok \Rightarrow ok') \land \nu' = \nu)\land ((ok \Rightarrow ok') \land \nu' = \nu)$
   - " $\iff$-def, $\triangleright$-def, substitution"
   - $\exists ok_m, \nu_m :$
     - $\neg ok \lor ok_m \land \nu_m = \nu \land (\neg ok_m \lor ok') \land \nu' = \nu_m$
   - " $\exists$-1pt, $\nu_m = \nu$"
   - $\exists ok_m : (\neg ok \lor ok_m) \land (\neg ok_m \lor ok') \land \nu' = \nu$
   - " shrink scope, $\land\lor$-distr, $\lor\land$-distr"
   - $\nu' = \nu \land (\exists ok_m :$
     - $\neg ok \land \neg ok_m \lor \neg ok \land ok'$
     - $\lor ok_m \land \neg ok_m \lor ok_m \land ok')$

---

**Lemma Proof (II)**

$\nu' = \nu \land (\exists ok_m :$

1. $\neg ok \land \neg ok_m \lor \neg ok \land ok'$
2. $\lor ok_m \land \neg ok_m \lor ok_m \land ok')$

- " witness, simplify"

$\nu' = \nu \land (\neg ok \land \neg ok \lor ok'$

- " $\lor$-absorb"

$\nu' = \nu \land (\neg ok \land ok')$

- " $\iff$-def"

$\nu' = \nu \land (ok \Rightarrow ok')$

- " defn. $J$"
Aside: the (un-)Healthiness of $J$

- We have just seen that $J; J = J$, i.e. that $J$ is $H_2$-healthy.
- What about $H_1(J)$ and $H_3(J)$?
- Careful calculation shows:

\[
\begin{align*}
H_1(J) &= \text{ok} \Rightarrow J = \text{skip} \\
H_3(J) &= J; \text{skip} = \text{skip}
\end{align*}
\]

- So $J$ is not $H_1$ or $H_3$, and attempts to “healthify” it using either $H_1$ or $H_3$ turn it into $\text{skip}$.

H3 is idempotent

We want to show that $H_3 \circ H_3 = H_3$.

Reduce Lhs to Rhs:

\[
\begin{align*}
H_3(H_3(P)) &= \"\text{defn. } H_3 \" (P; \text{skip}; \text{skip} \\
&= \"\text{;}\text{-assoc} \" P; (\text{skip}; \text{skip}) \\
&= \"\text{skip;}\text{-unit}, with } P = \text{skip } \" P; \text{skip} \\
&= \"\text{defn. } H_3 \" H_3(P)
\end{align*}
\]

Designs: a final comment

- Given $D$ which we know is a design (because we have shown it to be $H_1$ and $H_2$) can we write it in the form $P \vdash Q$?
- Can we determine $P$ and $Q$, given $D$?
- Yes:
  - If $D$ is $H_1$- and $H_2$-healthy, then $D = (\neg D') \vdash D'$

  - The precondition is those situations that do not lead to $\text{ok}' = \text{False}$, i.e. $\neg D'[\text{False}/\text{ok}']$
  - The post condition is those situations that end with $\text{ok}' = \text{True}$, i.e. $D[\text{True}/\text{ok}']$

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.

\[
\begin{align*}
H_1 \circ H_2 &= H_2 \circ H_1 \\
H_1 \circ H_3 &= H_3 \circ H_1 \\
H_3 \circ H_2 &= H_2 \circ H_3
\end{align*}
\]

We say that $H_1$, $H_2$ and $H_3$ “commute”. 

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 $H_b(H_a(P))$ 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)

\[
H_1(H_2(P)) \\
= \text{ " defns., } H_1, H_2 \text{ "}
\]

\[
ok \Rightarrow (P; J) \\
= \text{ " } \Leftrightarrow\text{-def } \text{ "}
\]

\[
\neg ok \lor (P; J) \\
= \text{ " } \neg ok \text{ (a.k.a. miracle) is a design and hence } H_2 \text{ "}
\]

\[
H_2(\neg ok) \lor (P; J)
\]

Proof that $H_1$ and $H_2$ commute (RHS)

\[
H_2(H_1(P)) \\
= \text{ " defns., } H_1, H_2 \text{ "}
\]

\[
(ok \Rightarrow P); J \\
= \text{ " } \langle \lor; \text{-distr}\rangle \text{ (see below) "}
\]

\[
(\neg ok; J) \lor (P; J) \\
= \text{ " } \text{ defn. } H_2 \text{ "}
\]

\[
H_2(\neg ok) \lor (P; J)
\]

\[
\square
\]

We have used $\langle \lor; \text{-distr}\rangle$: $(P \lor Q); R \equiv (P; R) \lor (Q; R)$ 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, for all predicates $P$ and $Q$
  
  \[
  (P \sqsubseteq Q) \Rightarrow (F(P) \sqsubseteq F(Q))
  \]

- Note how monotonicity is 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: $c \ast P$
  - We can consider this a function of $P$ (how?)
  - Simple, define
    \[ WLOOP_c(P) \equiv c \ast P \]
- It turns out that $WLOOP_c$ is monotonic
- So if $P \sqsubseteq Q$ then $WLOOP_c(P) \sqsubseteq WLOOP_c(Q)$, i.e. $c \ast P \sqsubseteq c \ast Q$

2-place HOFs

- With 2-place HOFs we can define the other language constructs
  - \[
  SEQ(P, Q) \equiv P : Q
  \]
  \[
  COND_c(P, Q) \equiv P < c > Q
  \]
- We say a 2-place HOF $F(\_ , \_ )$ is:
  - Monotonic in 1st arg. if $P \sqsubseteq Q \Rightarrow F(P, R) \sqsubseteq F(Q, R)$
  - Monotonic in 2nd arg. if $P \sqsubseteq Q \Rightarrow F(R, P) \sqsubseteq F(R, Q)$
- $F(\_ , \_ )$ 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 \( F \circ G \)

\[
\begin{align*}
P & \subseteq Q \\
\Rightarrow & \quad \text{"} G \text{ is monotonic "} \\
G(P) & \subseteq G(Q) \\
\Rightarrow & \quad \text{"} F \text{ is monotonic "} \\
F(G(P)) & \subseteq F(G(Q))
\end{align*}
\]

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:
    \[
    \forall P, Q \ (P \subseteq Q) \Rightarrow (F(P) \subseteq F(Q))
    \]
  - 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 it
  occurrences are said to be mixed.
  - passing through either argument of \( \equiv \) results in a mixed
    occurrence.
- Saoithín keeps track of a location’s polarity as focus is
  moved into a predicate.

Anti-Monotonicity

- HOF \( F \) is anto-monotonic if, for all predicates \( P \) and \( Q \)
  \[
  (P \subseteq Q) \Rightarrow (F(Q) \subseteq F(P))
  \]
- \( F(P) \) is anti-monotonic if every occurrence of \( P \) is in a
  negative location.
Monotonicity Example I

Is $F(P) \equiv P \land (\exists x \cdot Q \Rightarrow P)$ monotonic?

$P \land (\exists x \cdot Q \Rightarrow P)$

"mark occurrence polarity"

$P^+ \land (\exists x \cdot Q^- \Rightarrow P^+)^+$

"both $P$ are labelled with $+$"

$F$ confirmed monotonic

Monotonicity Example II

Is $F(Q) \equiv P \land (\exists x \cdot Q \Rightarrow P)$ monotonic?

$P \land (\exists x \cdot Q \Rightarrow P)$

"mark occurrence polarity"

$P^+ \land (\exists x \cdot Q^- \Rightarrow P^+)^+$

"the sole $Q$ is labelled with $-$"

$F$ confirmed non-monotonic

Monotonicity Example III

Is $F(P) \equiv P \land \lnot (\exists x \cdot P \Rightarrow Q)$ monotonic?

$P \land \lnot (\exists x \cdot P \Rightarrow Q)$

"mark occurrence polarity"

$P^+ \land \lnot (\exists x \cdot P^+ \Rightarrow Q^-)^-$

"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"

$P^+ \land (P^+ \equiv Q^-)^+$

"so, not monotonic then..."

$= P \land Q$

"logic (exercise)"

$P^+ \land Q^+$

"mark occurrence polarity"

"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.

Anti-monotonicity

- A function $F$ is anti-monotonic if
  
  $$(P \sqsubseteq Q) \Rightarrow (F(Q) \sqsubseteq F(P))$$

- A function $F(P) = \ldots$ is anti-monotonic if $P$ occurs only at negative locations.

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
      
      $$(S \sqsubseteq D) \land (D \sqsubseteq P) \Rightarrow S \sqsubseteq P$$

      We can refine $S$ first to $D$, and then refine that to $P$.
    - Monotonicity: we want to work on sub-parts
      
      $$(D \sqsubseteq P) \Rightarrow F(D) \sqsubseteq F(P)$$

      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 ($SEQ$, $COND_c$, $WLOOP_c$) are monotonic.
- What about specification frames ($w : [P, Q]$), or designs ($P \vdash Q$)?
- Lets 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 $SPEC_w$ and $DSGN$:

\[
SPEC_w(P, Q) = ok \land P \Rightarrow ok' \land Q \land \nu' = \nu
\]

\[
DSGN(P, Q) = ok \land P \Rightarrow ok' \land Q
\]

- 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.

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 $c\perp$ is a (specification) statement that a condition holds.
  - if $c$ is true, then $c\perp$ behaves like $\text{Skip}$
  - if $c$ id false, then $c\perp$ behaves like $\text{Chaos}$

\[
c\perp \equiv \text{Skip} < c \triangleright \text{Chaos}
\]

- Often written as as $\{c\}$ in the literature

- In UTP we can reason about them directly (e.g.):

\[
b\perp; c\perp = (b \land c)\perp
\]

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 $\text{pre}_\perp; \text{prog}; \text{post}_\perp$:
  - we can automatically derive a VC involving $\text{pre}_\perp$ and $\text{post}_\perp$
  - 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).

An annotation example

The integer division algorithm from [Hoare69] with annotations:

\[
\begin{align*}
true \perp; \\
r &:= x; \\
q &:= 0; \\
(y \leq r) \ast \\
(x = r + y \ast q) \perp; \\
r &:= r - y; \\
q &:= q + 1; \\
(x = r + (y \ast q) \land r < y) \perp
\end{align*}
\]

Generating VCs

• VCs are generated for all the program statements
• We shall define VC generation recursively over program structure

\[ genVC : Program \rightarrow \mathbb{P}VC \]

VC generation (; )

• Given that the last statement is an assignment

\[
\begin{align*}
genVC(p_\perp; P_1; \ldots; P_{n-1}; v := e; q_\perp) \\
= \quad genVC(p_\perp; P_1; \ldots; P_{n-1}; q[e/v]_\perp)
\end{align*}
\]

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

\[
\begin{align*}
genVC(p_\perp; P_1; \ldots; P_{n-1}; r_\perp; r_\perp; P_n; q_\perp) \\
= \quad genVC(r_\perp; P_n; q_\perp) \\
\cup \quad genVC(p_\perp; P_1; \ldots; P_{n-1}; r_\perp)
\end{align*}
\]

We process the last statement and the recursively treat the rest of the sequence.
VC generation (:=)

\[
\text{genVC}(p; x := e; q) \triangleq \{ p \Rightarrow q[e/v] \}
\]

The pre-condition must imply the post-condition with \( v \) replaced by \( e \).

Example:

\[
\text{genVC}(p; r := r - y; q) \triangleq \{ p \Rightarrow q\[r - y/r] \}
\]

So, given \((x = r + y * q)\); \( r := r - y \), what assertion at the end will work?

- Not \((x = r + y * q)\), because
  \[
x = r + y * q \implies x = r - y + y * q
  \]

- Assertion \(x = r + y * (q + 1)\) does work, because
  \[
x = r + y * q \implies x = r - y + y * (q + 1)
  \]

VC generation (≤)

\[
\text{genVC}(p; P; c \cdot i; q) \triangleq \{ p \Rightarrow i, i \wedge \neg c \Rightarrow q \}
\]

These correspond closely to our proof technique for loops:

- \( p \Rightarrow i \) — pre-condition sets up invariant
- \( i \wedge \neg c \Rightarrow q \) — invariant and termination satisfies postcondition
- \( (i \wedge c) \wedge P; i \) — invariant and loop-condition preserve the invariant

VCs generated by our example.

- Do in class
- Solution
  \[
  \begin{align*}
  \text{true} & \Rightarrow x = x \land 0 = 0 \\
  r = x \land q = 0 & \Rightarrow x = r + (y * q) \\
  x = r + y * q \land (y \leq r) & \Rightarrow x = r + (y * q) \land r < y \\
  x = r + y * q \land y \leq r & \Rightarrow x = r - y + (y * (q + 1))
  \end{align*}
  \]

- These are simple enough to do by hand (or with Saoithin).
- 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 \( \text{pre} \sqsubseteq \text{post} \), 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 \( \text{prog} \)

\[
\begin{align*}
\text{pre} \vdash \text{post}' & \sqsubseteq \text{prog} \\
& = [\text{prog} \Rightarrow \text{pre} \vdash \text{post}'] \\
& = [\text{prog} \Rightarrow (\text{ok} \wedge \text{pre} \Rightarrow \text{ok}' \wedge \text{post}')] \\
& = [\text{ok} \wedge \text{pre} \Rightarrow (\text{prog} \Rightarrow \text{ok}' \wedge \text{post}')] \\
& = [\text{ok} \wedge \text{pre} \Rightarrow \forall \text{ok}_m, \nu_m \bullet \text{prog} \Rightarrow \text{ok}' \wedge \text{post}'] \\
& = [\text{ok} \wedge \text{pre} \Rightarrow \exists \text{ok}_m, \nu_m \bullet \text{prog} \wedge (\text{ok}' \Rightarrow \neg \text{post}')] \\
& = [\text{ok} \wedge \text{pre} \Rightarrow \neg (\text{prog} \land \neg \text{post}')] \\
\end{align*}
\]

\( \neg (\text{prog} : \neg \text{post}') \) 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.

\[
P_1; \ldots; P_{n-2}; P_{n-1}; P_n \text{ wp post}
\]

\[
\equiv P_1; \ldots; P_{n-2}; P_{n-1} \text{ wp } (P_n \text{ wp post})
\]

\[
\equiv P_1; \ldots; P_{n-2} \text{ wp } (P_{n-1} \text{ wp } (P_n \text{ wp post}))
\]

\[
\vdots
\]

\[
\equiv P_1 \text{ wp } (\ldots P_{n-2} \text{ wp } (P_{n-1} \text{ wp } (P_n \text{ wp post})) \ldots)
\]

- We then show that the precondition \textit{pre} implies the overall weakest precondition

\[
\text{pre} \Rightarrow P_1; \ldots; P_{n-2}; P_{n-1}; P_n \text{ wp post}
\]

WP for while loops

- If the user supplies an Invariant \textit{inv} and variant \textit{V}, then we can define \( c \star P \text{ wp } r \)

- If we ignore termination, we can define the following

\[
(c \star P) \text{ wp } r \triangleq \text{ inv}
\]

\[
\land (c \land \text{ inv} \Rightarrow P \text{ wp inv})
\]

\[
\land (\neg c \land \text{ inv} \Rightarrow r)
\]

(technically this is weakest \textit{liberal} precondition — WLP)

- We can automatically put in placeholder names for \textit{inv} (and \textit{V}), but at some stage the user will have to decide what they should be.

The problem with WP

- Where is the while-loop?

- We can show the following for WP and while-loops:

\[
(c \star P) \text{ wp } r \triangleq w \text{ such that}
\]

\[
w \Rightarrow (\neg c \Rightarrow r)
\]

\[
w \Rightarrow c \Rightarrow P \text{ wp } w
\]

\[
w \text{ is the weakest such predicate}
\]

- Calculating \( c \star P \text{ wp } r \) involves some form of a search for such a \textit{w}.

- Automating it successfully is equivalent to solving the Halting Problem (impossible).

CSP: Communicating Sequential Processes


- Despite its title, it covers concurrent processes as well!

- It is a theory of interacting components.
Interacting Components

- A Component, or **Process**, is:
  - independent self-contained entity,
  - with an **interface** with which it interacts with its **environment**.
- We can **compose** processes to form larger systems, which are themselves processes with their own interfaces and interactions.
- This is essentially the CSP view of the world.

Events

- A Process interface will be described as a **set of Events**.
- Event:
  - atomic indivisible action
  - either performed by processes, ...
  - ...or done to processes by the environment
- The **environment** of a process is anything outside a process capable of participating in the events on that process's interface.
- Interfaces constitute a **static specification** of a process.

Process Behaviour

- We need to know about a processes behaviour **on its interface**.
- In other words, the interface defines what is **observable** about the process.
- The emphasis here is on observing the **external activity**, rather than the internal workings.

What's in a Name?

- We need to name two distinct entities: Processes and Events.
- We shall adopt a convention that uses Uppercase for Processes, and lowercase for events.
- Names can have structure:
  - Process State: **ProcessName**(state-values)
  - e.g. a counter process whose state is the current count value: **Count**(5).
  - Event Structure: **a.b.c.d**
  - e.g. value 17 occurred on channel 5: **ch.5.17**.
- Event structure does not conflict with the notion of the event being atomic and indivisible — the whole event occurs, or none of it.
Example: Lift System (I)

- A lift can be at one of five floors, with its doors either open or closed:
  \[ \text{Lift}(\text{floor} : \{1 \ldots 5\}, \text{dooropen} : \mathbb{B}) \]

- The lift can participate in four events: going up, down, opening and closing doors:
  \[ \text{up, down, open, close} \]

- For example a lift with its door open at floor 3 is captured by the process \( \text{Lift}(3, \text{true}) \).

Example: Lift System (II)

- How do we capture the fact that \( \text{Lift}(3, \text{true}) \) should only be able to engage in a \text{close} event?
- How do we capture the fact that this lift, once it has performed \text{close}, will then be process \( \text{Lift}(3, \text{false}) \), and will then be able to do \text{open}, \text{up} and \text{down} events?

CSP: Process Description Language

- The CSP Language is designed to describe exactly the kind of situation that arises with our Lift example.
- We first need to describe its syntax.
- We also present an informal description of the behaviour captured by the notation.

CSP: The language

\[ a, b, a.b, a.b.c, \ldots, g \in \text{Events} \]

\[ P, Q, R \in \text{Proc} \]

\[ \text{g} \in \text{Guard} \]

Events (possibly structured)  
- guard predicates
- Processes  
- do nothing, not even terminate
- do nothing and terminate
- do \( a \), then act like \( P \)
- guarded process
- sequential composition
- internal choice
- external choice
- parallel composition
- event hiding
- do anything
Stop

- The simplest process is **Stop**.
- It *never* engages in any events in its interface.
- It *never* terminates.
- The process **Stop** is the canonical representation of **Deadlock**

Skip

- The next simplest process is **Skip**.
- It does not engage in any events, but instead just terminates.
- It's behaviour is very like that of **skip** in imperative programs.

\[ a \rightarrow P \]

- Pronounced “**a then P**”
- A process which can only perform an **a** initially
- Once it has done the **a** event, it then proceeds to behave like **P**.

\[ g & P \]

- Pronounced “**g guard(s) P**”
- If guard **g** is true, then act like **P**
- If guard **g** is false, then act like **Stop**
- Guard **g** is usually defined over the process state.
\[ P; Q \]

- Behave like \( P \) at first
- If \( P \) terminates, then proceed to behave like \( Q \)
- Analogous to \( ; \) in imperative programming, except that \( P \) failing to terminate is not a problem.

\[ P \sqcap Q \]

- Called “internal choice”
  - The process can behave either like \( P \) or \( Q \)
  - The process makes the decision which to do, without reference to the environment.
  - Analogous to the specification notation \( P \sqcap Q \) from the imperative programming world.
  - Also known as
    - non-deterministic choice
    - demonic choice

\[ P \square Q \]

- Called “external choice”
  - The process can behave either like \( P \) or \( Q \)
  - The process makes the decision which to do, always in agreement with the environment.
  - If the environment offers an event that \( P \) is willing to do, then \( P \) runs (similarly for \( Q \)).
  - If the environment offers an event that both \( P \) and \( Q \) are willing to do then it behaves like \( P \sqcap Q \)

\[ P \|_{A} Q \]

- Called “parallel composition”
  - Both \( P \) and \( Q \) run simultaneously
  - They must synchronise on events in \( A \) both must agree to perform such events simultaneously
  - Events not in \( A \) are performed independently by either process.
  - \( P \|_{A} Q \) terminates when both \( P \) and \( Q \) have terminated.
Called “event hiding”
- It behaves like P, except that events in A are hidden (internal events not visible to outside world)
- Used to structure large systems, encapsulating “local” communication.

Chaos
- Does “anything”
- Is the most unpredictable process that does not generate instability
- Willing to perform any event
- May refuse to perform any event
- It might terminate, it might not …
- Bottom of the refinement lattice
- Very similar to Chaos in imperative programs

Recursion
- We use recursion to define iterative/non-terminating processes

\[ M, N \in \text{Names} \]
\[ N \triangleq P \]
\[ N(x, y) \triangleq P \]

- We extend our process notation to allow names, and applications of higher-order functions used in definitions

\[ \text{Proc} ::= \ldots \]
\[ N \]
\[ N(a, b) \]

- Recursion arises in \[ N \triangleq P \]
  if \( P \) mentions \( N \) somewhere.

Lift System (III)

We can now define our (5-floor) lift process:

\[ \text{opened} \triangleq \text{True} \]
\[ \text{closed} \triangleq \text{False} \]

\[ \text{Lift}(f, \text{opened}) \triangleq \text{close} \to \text{Lift}(f, \text{closed}) \]
\[ \text{Lift}(f, \text{closed}) \triangleq \big( \text{open} \to \text{Lift}(f, \text{opened}) \big) \]
\[ \quad \big( f < 5 \& \text{up} \to \text{Lift}(f + 1, \text{closed}) \big) \]
\[ \quad \big( f > 1 \& \text{down} \to \text{Lift}(f - 1, \text{closed}) \big) \]
CSP: Semantics

- What does it all mean — precisely?
- How is this be formalised in UTP?
- We shall treat CSP process syntax as “sugar” for UTP predicates.

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 (\(K^{-1}_a, K^{-1}_b\))
  - 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:
  - \(A \rightarrow B : \ A.B.\{N_a.A\}_{PK(B)}\)
    - A sends B his name and nonce, encoded with B’s public key.
  - \(B \rightarrow A : \ B.A.\{N_a.N_b\}_{PK(A)}\)
    - 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
  - \(A \rightarrow B : \ A.B.\{N_b\}_{PK(B)}\)
    - 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.
- System communication is modelled by three channels:
  - $comm$ — regular protocol communication
  - $fake$ — a fake protocol message (receiver unaware of fake)
  - $intercept$ — an intercepted protocol message (sender unaware of intercept)
- Other channels used to model which users are participating in what protocol sessions.

The NS-PKP protocol in CSP

- An intruder-free run of the protocol is modelled as:
  
  $$INITIATOR(a, n_a) \triangleq
  \begin{align*}
  &user.a\times b \to I.running.a.b \to \\
  &comm!Msg1.a.b.Encrypt.key(b).n_a.a \to \\
  &comm!Msg2.b.a.Encrypt.key(a)?n'_a.n_b \to \\
  &n_a = n'_a &
  \end{align*}
  
  $$

- The $user$ channel sets up a session by identifying responder $b$.
- The $I\ldots$ and $commit$ events record details about the protocol run.

Modelling the Intruder

- The intruder can:
  - Overhear/intercept any message
  - Decrypt messages encrypted with his public key
  - Introduce new messages
  - Replay messages, changing plain-text parts.
- Intruder is in the network
- Other agents can/may interact with intruder as if he was a legitimate agent.
- $I(m1s, m2s, m3s, ns)$ models an intruder that so far cannot decrypt step $i$ messages in set $mis$, but knows nonces in $ns$.
- Initially intruder has seen no messages and knows only their own nonce: $I(\emptyset, \emptyset, \emptyset, \emptyset)$

Modelling the System

- Assembling the (good) agents
  
  $$AGENTS \triangleq
  \begin{align*}
  &INITIATOR1 \parallel comm.session.A.B \parallel RESPONDER1 \\
  &\ldots \\
  &SYSTEM \triangleq
  \begin{align*}
  &AGENTS \parallel fake, comm, intercept \parallel INTRUDER \\
  &\ldots \\
  \end{align*}
  \end{align*}
  
  $$

- Agents put in parallel with (bad) intruder

- A specification of correct behaviour ($AIR$) was then developed in CSP
- Plan: prove that $AIR \subseteq SYSTEM$
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 $I(A)$).

<table>
<thead>
<tr>
<th>$A \rightarrow I$</th>
<th>$A.I.{N_a.A}_{PK(I)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I(A) \rightarrow B$</td>
<td>$B.A.{N_a.A}_{PK(B)}$</td>
</tr>
<tr>
<td>$B \rightarrow I(A)$</td>
<td>$B.A.{N_a,N_b}_{PK(A)}$</td>
</tr>
<tr>
<td>$I \rightarrow A$</td>
<td>$I.A.{N_a,N_b}_{PK(A)}$</td>
</tr>
<tr>
<td>$A \rightarrow I$</td>
<td>$A.I.{N_b}_{PK(I)}$</td>
</tr>
<tr>
<td>$I(A) \rightarrow B$</td>
<td>$A.B.{N_b}_{PK(B)}$</td>
</tr>
</tbody>
</table>

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. $B \rightarrow A : B.A.\{N_a,N_b,B\}_{PK(A)}$
     - $B$ 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 known (in educated circles) as the Needham-Schroeder-Lowe protocol.