Frame Specifications with $\text{ok}$, $\text{ok}'$

- What is the semantics of a frame specification $\vec{x} : [S]$ in our new theory?
- In the old theory it was

$$\vec{x} : [S] \equiv S \land \nu' = \nu$$

where $\nu$ were all variables not in $\vec{x}$.
- How do we fit in $\text{ok}$, $\text{ok}'$?

### Frame Syntax change

- To fit our new theory, frames now have to explicitly distinguish pre and post conditions:

$$\vec{x} : [\text{pre}, \text{Post}]$$

- We also need to note that if $\text{ok} = \text{False}$ at the outset, then we can modify any variables, not just those in the frame.

The definition we get is therefore:

$$\langle \textit{frame-def} \rangle \quad \vec{x} : [\text{pre}, \text{Post}] \equiv \text{ok} \land \text{pre} \Rightarrow \text{ok}' \land \text{Post} \land \nu' = \nu$$

where $\nu$ are all alphabet variables not mentioned in $\vec{x}$.

### Upgrading frame specs to new theory

- An old specification of the form

$$w : [P]$$

where $P$ is not an implication, becomes

$$w : [\text{true}, P]$$

- An old specification of the form

$$w : [\text{pre} \Rightarrow \text{Post}]$$

becomes

$$w : [\text{pre}, \text{Post}]$$
Refinement with $ok$, $ok'$

- How does adding in $ok$ and $ok'$ affect refinement laws?
- e.g. (old rules)

\[
\langle \langle \text{prog-strengthen} \rangle \rangle \quad P \sqsubseteq P \land Q
\]

\[
\langle \langle \text{subst} \rangle \rangle \quad (S \sqsubseteq P \land x = e) \\
\equiv (S[e/x] \sqsubseteq P \land x = e)
\]

\[
\langle \langle = \rangle \rangle \quad x' = e \sqsubseteq x := e
\]

- Can we use these as is?
- Are there more useful variants of these?

Using $\langle \langle \text{prog-strengthen} \rangle \rangle$ with $ok$, $ok'$

- 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

\[
\langle \langle \text{prog-strengthen} \rangle \rangle \quad (P \Rightarrow Q) \sqsubseteq (P \Rightarrow Q \land R)
\]

- In particular, often, $P$ will be $ok \land pre$, and $Q$ will be $ok' \land Post$.

Using $\langle \langle \text{subst} \rangle \rangle$ with $ok$, $ok'$

- The rule is still valid, but again we now have those extra implications
- The following variant is useful

\[
\langle \langle \text{subst} \rangle \rangle \quad (S \sqsubseteq P \land x = e) \\
\equiv (S[e/x] \sqsubseteq P \land x = e)
\]

- As before, we often find that $P$ will be $ok \land pre$, and $Q$ will be $ok' \land Post$.

Using $\langle \langle = \rangle \rangle$ with $ok$, $ok'$

- The specification $x' = e$ is no longer valid — it does not mention $ok$ and $ok'$ in a valid way
- We get useful variants if we change the specification to fit our new theory:

\[
\langle \langle = \rangle \rangle \quad \vec{w} : [pre, x' = e] \sqsubseteq x := e, \quad x \in \vec{w}
\]

Note here that $p$ does not mention $x'$. 
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
\[ \text{HSpec} \sqsubseteq \text{HProg} \]
using new version of our theory, with \(ok\) and \(ok'\)

Refining Hoare’69: revisited (1)

We must show \(\text{HSpec} \sqsubseteq r, q : [\text{true}, \text{hinv'}, y' > r']\)
\[ r, q : [\text{true}, y' > r' \land x' = r' + y' \ast q'] \sqsubseteq r, q : [\text{true}, x' = r' \ast q' \ast y' \land y'> r'] \]
= \(\triangleleft \text{-refl}\)
= \(\text{true}\)

Refining Hoare’69: revisited (2)

We must show \(\text{true} : [r, q, \text{hinv'}] \sqsubseteq r, q := x, 0\)
\[ r, q : [\text{true}, x' = r' \ast q' \ast y' \sqsubseteq r, q := x, 0 \]
= \("\text{frame-def}\", \("\text{sim-:=def}\")\)
\[ ok \Rightarrow ok' \land x' = r' \ast q' \ast y' \land x' = x \land y' = y \]
= \("\sqsubseteq-\text{-subst}\" \((\text{new variant})\)\)
\[ ok \Rightarrow ok' \land x = x + 0 \ast y \land x' = x \land y' = y \]
= \("\sqsubseteq\"\)
\[ ok \Rightarrow ok' \land x = x + q' \land 0 \land x' = x \land y' = y \]
= \("\text{arithmetic}\")
\[ ok \Rightarrow ok' \land \text{true} \land x' = x \land y' = y \]
\[ \sqsubseteq ok \Rightarrow ok' \land x = x \land q' = 0 \land x' = x \land y' = y \]
\[ = \("\text{prog-strengthen}\" \((\text{new variant})\)\)
\[ = \text{true} \]
We have “proved” something False !!

- We have proved that if the Hoare’69 program starts ok, then it terminates, establishing \( y' > r' \land x' = y' \ast q' + r' \)
- The proof of correctness was very similar to before
- We had to handle the extra complexity of \( \text{ok} \land \ldots \Rightarrow \text{ok}' \land \ldots \)
- What have we gained ?
  - in fact, things aren’t just pointless, they are worse than that !

- 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 \( \vec{v} : [\text{pre}, \text{Post}] \) by \( \text{init} : c \ast P \) we need to:

1. Find an appropriate invariant \( \text{inv} \) and variant \( V \)
2. Show that
   \[
   \vec{v} : [\text{pre}, \text{Post}] \sqsubseteq \vec{v} : [\text{pre}, \text{inv'} \land \lnot c']
   \]
3. Show
   \[
   \vec{v} : [\text{pre}, \text{inv'}] \sqsubseteq \text{init}
   \]
4. Show
   \[
   \vec{v} : [c \land \text{inv'}, \text{inv'} \land 0 \leq V' < V] \sqsubseteq P
   \]

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
  \[
  r, q : [y \leq r \land x = q \ast y + r, \quad x' = q' \ast y' + r' \land 0 \leq r' \ominus y' < r \ominus y] \sqsubseteq r, q := r - y, q + 1
  \]

Attempting to prove that \( r \ominus y \) is a variant (I)

- Rather than prove the whole thing, we focus on the variant part
  \[
  r, q : [y \leq r, 0 \leq r' \ominus y' < r \ominus y] \sqsubseteq r, q := r - y, q + 1
  \]
- We expand the lhs using \(\text{frame-def}\):
  \[
  ok \land y \leq r \Rightarrow ok' \land 0 \leq r' \ominus y' < r \ominus y \land x' = x \land y' = y
  \]
- We expand the rhs using \(\text{sim} \equiv \text{def}\):
  \[
  ok \Rightarrow ok' \land r' = r - y \land q' = q + 1 \land x' = x \land y' = y
  \]
Attempting to prove that \( r \ominus y \) is a variant (II)

- We use the rhs, and \( \Leftarrow \text{subst} \) (new version) to transform the lhs:
  \[
  ok \land y \leq r \Rightarrow ok' \land 0 \leq (r - y) \ominus y < r \ominus y \land x = x \land y = y
  \]

- Again we ignore the variables \( ok, x \), 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

\[
\begin{align*}
\text{HSpec} & \triangleq r, q : [y > 0, y' > r' \land x' = r' + y' \ast q'] \\
\text{HProg} & \triangleq r, q := x, 0 ; (y \leq r) \ast (r, q := r - y, 1 + q)
\end{align*}
\]

We can now prove

\[
\text{HSpec} \sqsubseteq \text{HProg}
\]

by choosing invariant

\[
\text{inv} \triangleq x = q \ast y + r
\]

and variant

\[
\text{V} \triangleq r \ominus y
\]

within the new version of our theory, with \( ok \) and \( ok' \).

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)
A Theory of Sets

Axioms:

\[ \langle \langle -\text{in}\{\}\rangle \rangle \quad x \notin \{\} \]
\[ \langle \text{in-singleton} \rangle \quad x \in \{y\} \equiv x = y \]
\[ \langle \text{set-extensionality} \rangle \quad S = T \equiv \forall x \bullet x \in S \equiv x \in T \]

Not a “traditional” axiomatization of set-theory!

Definitions:

\[ \langle \langle -\text{in}\cup \rangle \rangle \quad x \in (S \cup T) \equiv x \in S \lor x \in T \]
\[ \langle \langle -\text{in}\cap \rangle \rangle \quad x \in (S \cap T) \equiv x \in S \land x \in T \]
\[ \langle \langle -\text{in}\backslash \rangle \rangle \quad x \in (S \backslash T) \equiv x \in S \land x \notin T \]
\[ \langle \text{def}\subseteq \rangle \quad S \subseteq T \equiv \forall x \bullet x \in S \Rightarrow x \in T \]

What is the rôle of \textit{ok} and \textit{ok'}

- We could have used the variant idea with the old theory
- It would then have been a theory of total correctness
- But we would still have the same problem with

\[ \text{Forever} \equiv \text{True} \land \text{skip} \]

being \textit{true}, and the following being the case:

\[ P; \text{Forever} \neq \text{Forever} \neq \text{Forever}; P \]

- It is this problem, getting \textit{Forever} to obey the right laws, that is solved by adding \textit{ok} and \textit{ok'}.

Designs: Notation and Definition

- We are going to introduce the notion of a certain class of predicates, that we will call \textit{Designs}.
- Every design \textit{D} can be written in the form

\[ \text{ok} \land P \Rightarrow \text{ok'} \land Q \]

- so our new theory is all about \textit{Designs}.
- We introduce a shorthand for designs:

\[ \langle \text{def}\Rightarrow \rangle \quad P \Rightarrow Q \equiv \text{ok} \land P \Rightarrow \text{ok'} \land Q \]

This notation is \textit{n.s} (non-substitutable).
Design Semantics

- We now require the semantics of all our program language statements to be designs.
- We only need to give semantics for key building blocks:

  \[
  \begin{align*}
  \langle\langle \text{skip-def} \rangle \rangle & \quad \text{skip} \overset{A}{=} \text{true} \vdash \nu' = \nu \quad A = \{\nu, \nu'\} \\
  \langle\langle \text{=def} \rangle \rangle & \quad x := e \overset{A}{=} \text{true} \vdash x' = e \land \nu' = \nu \quad A = \{x, x', \nu, \nu'\} \\
  \langle\langle \text{frame-def} \rangle \rangle & \quad \vec{x} : [P, Q]_A \overset{A}{=} P \land Q \land \nu' = \nu \quad A = \{\vec{x}, x', \nu, \nu'\}
  \end{align*}
  \]

Here, \(A\) is the alphabet of the whole construct.

Design Lattice

- Designs themselves form a lattice, with refinement as the ordering.
- The lattice bottom is \text{false} \vdash \text{true}
  - It simplifies to \text{true}
  - It is called \text{Chaos}
- The lattice top is \text{true} \vdash \text{false}
  - It simplifies to \neg \text{ok}
  - It is called \text{miracle}
- If \(Q_1 \preceq Q_2\), then \((P \vdash Q_1) \preceq (P \vdash Q_2)\)
  (strengthen postcondition)
- If \(P_2 \preceq P_1\), then \((P_1 \vdash Q) \preceq (P_2 \vdash Q)\)
  (weaken precondition)

Design Lattice

\[
\begin{array}{c}
\text{true} \vdash x' = 3 \\
\text{true} \vdash x' = 6 \\
\text{true} \vdash (x' = 3 \lor x' = 6) \\
\text{true} \vdash x' \in \{1 \ldots 10\} \\
\text{false} \vdash \text{true}
\end{array}
\]

Design-Predicate Lattice

\[
\begin{array}{c}
\neg \text{ok} \\
\neg \text{ok} \Rightarrow \text{ok'} \land x' = 3 \\
\neg \text{ok} \Rightarrow \text{ok'} \land x' = 6 \\
\text{ok} \Rightarrow \text{ok'} \land (x' = 3 \lor x' = 6) \\
\text{ok} \Rightarrow \text{ok'} \land x' \in \{1 \ldots 10\} \\
\text{true} \vdash \text{false}
\end{array}
\]
Using Designs

How are the techniques we learnt up to this point affected by the Design notion?

1. We have to think in terms of pre-post condition pairs (but this is OK — it’s a natural way to think about programs).
2. Most programming laws are unchanged, except that loop proofs now require a variant as well.
3. We can provide a collection of design laws, so the user need rarely use $\vdash\text{def}$, or see the $\text{ok}$ and $\text{ok}'$ variables.

Some Design Laws

1. $\text{(export-precondition)}$
   
   
   $$(P \vdash Q) = (P \vdash P \land Q)$$

2. $\text{(design-refinement)}$

   $$(P_1 \vdash Q_1) \subseteq (P_2 \vdash Q_2) = (P_2 \subseteq P_1) \land (Q_1 \subseteq P_1 \land Q_2)$$

3. $\text{($\forall$-distr)}$

   $$(P_1 \vdash Q_1) \cap (P_2 \vdash Q_2) = (P_1 \land P_2 \vdash Q_1 \lor Q_2)$$

4. $\text{($\forall$-distr)}$

   $$(P_1 \vdash Q_1) \triangleright b \triangleright (P_2 \vdash Q_2) = (P_1 \triangleright b \triangleright P_2 \vdash Q_1 \triangleright b \triangleright Q_2)$$

Designs and Sequential Composition

1. For sequential composition, we have the following, somewhat complex law $\langle; \vdash\text{-distr}\rangle$:

   $$(P_1 \vdash Q_1); (P_2 \vdash Q_2) = \neg (\neg P_1; \text{true}) \land \neg (Q_1 \land Q_2) \vdash Q_1; Q_2$$

   Why such a complex pre-condition?

   1. First, note that in a design $P \vdash Q$, nothing so far says that either $P$ or $Q$ have to be proper (pre-/post-)conditions.
   2. Remember, a pre-(post-)condition only has undashed (dashed) variables.
   3. In particular, the so-called "pre-condition" $P$ can in general mention both before- and after-variables.
   4. It is this fact that leads to some of the complexity above.

Explaining $\neg (\neg P_1; \text{true})$

1. If we simplify $\neg (\neg P_1; \text{true})$ it reduces to $\forall \nu' \cdot P_1$
2. This simply states that condition $P_1$ must be true, no matter what the final variable values are.
3. If $P_1$ is actually a pre-condition (no dashed variables), then this reduces further, to $P_1$ itself.
Explaining \( \neg (Q_1; \neg P_2) \)

- Predicate \( Q_1; \neg P_2 \) describes a situation where
  - \( Q_1 \) has executed, and
  - afterwards, the situation described by \( P_2 \) does not hold
- In other words it describes situations where running \( Q_1 \) leads to “precondition” \( P_2 \) not being satisfied
- We negate it in order to rule out such situations from our overall “precondition”.

“preconditions” vs. pre-conditions (I)

- As already stated, in design \( P \vdash Q \), the “precondition” \( P \) may have dashed variables, and hence not be a true condition
- However, in many (most!) cases, \( P \) will be such a condition, only referring to before-values of variables
- In fact, in practise, in real specifications, a pre-condition should almost always be a true condition!
  - If our spec says \( P \vdash Q \), with \( P \) mentioning after variables, then the programmer can cheat.
  - Simply write a program that sets up final variables to make \( P \) false — that satisfies the specification!

“preconditions” vs. pre-conditions (II)

- We shall use a lower-case symbol (e.g. \( p \)) to denote a condition predicate.
  i.e. only mentions before-variables.
- The \( \langle ; \neg \rangle \text{-distr} \) law is simplified somewhat if the first “precondition” really is a condition:

\[
(p_1 \vdash Q_1); (p_2 \vdash Q_2) = p_1 \land \neg (Q_1; \neg P_2) \vdash Q_1; Q_2
\]

true pre-condition preservation

- Consider the following case where both pre-conditions are true pre-conditions.

\[
(p_1 \vdash Q_1); (p_2 \vdash Q_2)
\]

- Our simplified \( \langle ; \neg \rangle \text{-distr} \) law says this is equivalent to

\[
p_1 \land \neg (Q_1; \neg p_2) \vdash Q_1; Q_2
\]

- We have a combined “precondition” \( p_1 \land \neg (Q_1; \neg p_2) \)
- It is a pre-condition, despite \( Q_1 \)’s dashed variables.

\[
Q_1; \neg p_2 \quad \text{“ using } \langle ; \text{-def} \rangle \text{”}
\]

\[
= \exists \nu_m \cdot Q_1[\nu_m/\nu] \land \neg p_2[\nu_m/\nu]
\]

All the \( \nu \) in \( Q_1 \) are bound (no longer free).
So what does \( \neg (Q_1; \neg p_2) \) actually say

- It is a pre-condition: it only talks about the before-state
- It says, about the before state:
  
  \[ \text{It is not one from which an execution of } Q_1 \text{ would lead to a state where } p_2 \text{ does not hold} \]

While-loops are designs

- \( \text{skip}, x := e, w : [p \land Q] \) are designs
- For designs \( D_1 \), we have seen laws that say that \( D_1 \cap D_2 \), \( D_1 \triangleleft c \triangleright D_2 \) and \( D_1; D_2 \) are designs
- It is also the case that if \( D \) is a design, then so is \( c * D \)

- Informally we can appreciate this by noting that
  
  \[
  \begin{align*}
  c * D &= (D; c * D) \triangleleft c \triangleright \text{skip} \\
  &= (D; ((D; c * D) \triangleleft c \triangleright \text{skip})) \triangleleft c \triangleright \text{skip}
  \end{align*}
  \]

Each line is a design if its predecessor is, by laws we already have..

- Writing the design form out explicitly is complicated and not informative
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 \( P \models Q \))

- 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)

H1  \( P = (ok \Rightarrow P) \)

- \( P \) only ‘takes effect’ when \( ok \) is true — i.e. program has started
- When \( \neg ok \), then \( P \) says nothing
- H1 above is equivalent to satisfying both of the following laws:
  \[
  \langle \langle \text{skip;} \text{-unit} \rangle \rangle \text{skip} ; P = P \\
  \langle \langle ; \text{-L-Zero} \rangle \rangle \text{Forever} ; D = \text{Forever}
  \]
- Any design \( P \vdash Q \) satisfies the three laws above.

H1 Example (1)

What about \( \neg ok \wedge ok' \wedge x' = x + 1 \wedge \nu' = \nu \)?

\[
ok \Rightarrow \neg ok \wedge ok' \wedge x' = x + 1 \wedge \nu' = \nu
\]

\[
= \quad \langle \langle \Rightarrow\text{-def} \rangle \rangle
\]

\[
\neg ok \vee \neg ok \wedge ok' \wedge x' = x + 1 \wedge \nu' = \nu
\]

\[
= \quad \langle \langle \vee\wedge\text{-absorb} \rangle \rangle
\]

\[
\neg ok
\]

So it is not H1-healthy

H1 Example (2)

What about \( x := e \)?

\[
ok \Rightarrow x := e
\]

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

\[
ok \Rightarrow (ok \Rightarrow ok' \wedge x' = e \wedge \nu' = \nu)
\]

\[
= \quad \langle \langle \text{shunting} \rangle \rangle
\]

\[
ok \wedge ok \Rightarrow ok' \wedge x' = e \wedge \nu' = \nu
\]

\[
= \quad \langle \langle \wedge\text{-idem} \rangle \rangle
\]

\[
ok \Rightarrow ok' \wedge x' = e \wedge \nu' = \nu
\]

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

\[
x := e
\]

So \( x := e \) is H1-healthy

H1 Example (3)

What about \( P \vdash Q \)?

\[
ok \Rightarrow (P \vdash Q)
\]

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

\[
ok \Rightarrow (ok \wedge P \Rightarrow ok' \wedge Q)
\]

\[
= \quad \langle \langle \text{shunting} \rangle \rangle
\]

\[
ok \wedge ok \wedge P \Rightarrow ok' \wedge Q
\]

\[
= \quad \langle \langle \wedge\text{idem} \rangle \rangle
\]

\[
ok \wedge P \Rightarrow ok' \wedge Q
\]

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

\[
P \vdash Q
\]

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:
  \[ P^b \equiv P[b/ok'] \]

So H2 becomes \[ P^f \Rightarrow P^t \]
(with \( f \) and \( t \) the obvious abbreviations for False and True.

- It is equivalent to satisfying the following law:
  \[ \langle \text{-H2-R-Unit} \rangle \quad P: J = P \]
  \[ \langle J\text{-def} \rangle \quad J \equiv (ok \Rightarrow ok') \land \nu' = \nu \]

H2 Example (1)

Again, let's try \( \neg ok \land ok' \land x' = x + 1 \land \nu' = \nu \).

\[ (\neg ok \land ok' \land x' = x + 1 \land \nu' = \nu)[False/ok'] \Rightarrow (\neg ok \land ok' \land x' = x + 1 \land \nu' = \nu)[True/ok'] \]

\[ \langle \neg ok \land False \land x' = x + 1 \land \nu' = \nu \rangle \]

\[ \Rightarrow (\neg ok \land True \land x' = x + 1 \land \nu' = \nu) \]

\[ \langle \text{simplify} \rangle \]

\[ False \Rightarrow (\neg ok \land x' = x + 1 \land \nu' = \nu) \]

\[ \langle GS3.75 \rangle \]

\[ \text{true} \]

So it is H2-healthy!

H2 Example (2)

Now, try \( ok \Rightarrow \neg ok' \) (i.e requiring non-termination).

\( (ok \Rightarrow \neg ok')[False/ok'] \Rightarrow (ok \Rightarrow \neg ok')[True/ok'] \]

\[ \langle \text{simplify} \rangle \]

\( (ok \Rightarrow True) \Rightarrow (ok \Rightarrow False) \)

\[ \langle \text{excluded-middle}, \text{simplify} \rangle \]

\[ \text{true} \]

So it is not H2-healthy

H2 Example (3)

Finally, what about \( P \vdash Q \), or \( ok \land P \Rightarrow ok' \land Q \)?

\[ (ok \land P \Rightarrow ok' \land Q)^f \Rightarrow (ok \land P \Rightarrow ok' \land Q)^t \]

\[ \langle \text{simplify} \rangle \]

\[ \neg (ok \land P) \Rightarrow (ok \land P \Rightarrow Q) \]

\[ \langle \text{excluded-middle}, \text{simplify} \rangle \]

\[ \text{true} \]

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 ($H_3$)**

$H_3 \quad P = P; \text{skip}$

- Running $\text{skip}$ 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 $\text{ok} \land x' = 2 \Rightarrow \text{ok}'$
  “If started, and $x$ ends up equal to 2, then I terminate”
- $\text{ok} \land x' = 2 \Rightarrow \text{ok}'$
  “$\Leftrightarrow$-def, deMorgan”
  $\neg \text{ok} \lor x' \neq 2 \lor \text{ok}'$
  “$\Leftrightarrow$-def”
  $\text{ok} \Rightarrow x' \neq 2 \lor \text{ok}'$
  “$\Rightarrow$-def”
  “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.)**

$\text{ok} \land x' = 2 \Rightarrow \text{ok}'$; $\text{skip}$

- “$\text{skip}$-def”
- $\text{ok} \land x' = 2 \Rightarrow \text{ok}'$; $\text{ok} \Rightarrow \text{ok}' \land \nu' = \nu$
- “$\Leftrightarrow$-def”, “$\Leftrightarrow$-def”, 6-way (!) simplification “
- $\neg \text{ok} \lor (\exists x_m, \text{ok}_m \bullet (x_m = 2) \land \neg \text{ok}_m) \lor \text{ok}' \lor (\ldots)$
- “$\Rightarrow$-def”
- $\neg \text{ok} \lor \text{true} \lor \text{ok}' \lor (\ldots)$
- “$\lor$-zero”

$\text{true}$

Not equal to $\text{ok} \land x' = 2 \Rightarrow \text{ok}'$, so not $H_3$-healthy
**H3 Example (2)**

Now, try *miracle*, a.k.a. $\neg ok$

\[
\neg ok; \text{skip} = "\neg skip-def, \neg \Rightarrow-def" \\
\neg ok; \neg ok \lor ok'; \lor' = \nu
\]

\[
\nu = "\neg \Rightarrow-def, \neg substitution" \\
\exists ok_m, \nu_m \cdot \neg ok \land (\neg ok_m \lor ok' \land \lor' = \nu_m)
\]

\[
= "\neg \land-\lor-distr, \neg \exists-distr" \\
(\exists ok_m, \nu_m \cdot \neg ok \land \neg ok_m) \lor \\
(\exists ok_m, \nu_m \cdot \neg ok \land ok' \land \lor' = \nu_m)
\]

\[
= "\text{simplify}" \\
\neg ok \lor \neg ok \land ok'
\]

\[
= "\text{\lor-\land-absorb}" \\
\neg ok
\]

So *miracle* is H3-healthy

---

**Healthiness Condition 4 (H4)**

\[ P; \text{true} = \text{true} \]

- 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?

\[
*miracle*: \text{true} \\
= "*miracle* is \neg ok" \\
\neg ok; \text{true} = "\neg \text{def}, \text{substitution}" \\
\exists ok_m, \nu_m \cdot \neg ok \land \text{true}
\]

\[
= "\text{drop quantifiers, simplify}" \\
\neg ok
\]

It is not feasible, thankfully!

---

**Healthiness: Summary**

- We have 2 mandatory healthiness conditions for designs

\[
\begin{align*}
\text{H1:} & \quad P = ok \Rightarrow P \\
& \quad (\text{skip}; P = P) \land (\text{true}; P = \text{true}) \\
\text{H2:} & \quad [P^i \Rightarrow P^i] \\
& \quad P = P; J
\end{align*}
\]

- We have 2 additional conditions capturing better behaviour

\[
\begin{align*}
\text{H3:} & \quad P = P; \text{skip} \\
\text{H4:} & \quad P; \text{true} = \text{true}
\end{align*}
\]

- If $P$ and $Q$ are Hi-healthy ($i \in 1 \ldots 4$), then so are

\[
P; Q \quad P \sqcap Q \quad P \triangleleft c \triangleright Q \quad c \ast P
\]
Design Semantics of true * skip

- As before, we try to find something that satisfies the loop-unrolling law for Forever, namely:
  \[ X = \text{skip}; X \]

- However, now we restrict ourselves to designs only, and use the design definition for 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 true * skip

- Any design satisfies the loop-unrolling law for true * skip (a.k.a. Forever).

- We have a good reason to want the least such design, namely \( \text{false} \vdash \text{true} \) (which reduces to true).

\[ \langle \langle \text{Forever-\text{def}} \rangle \rangle \]
\[ \text{Forever} \equiv \text{true} \]

- Does it give us the desired laws?

\[ \langle \langle \text{L-Zero} \rangle \rangle \]
\[ \text{Forever}; D = \text{Forever} \]

\[ \langle \langle \text{R-Zero} \rangle \rangle \]
\[ D; \text{Forever} = \text{Forever} \]

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

\[ \text{Forever}; D \]
\[ = \langle \langle \text{Forever-\text{def}} \rangle \rangle, D = P \vdash Q \rangle \]
\[ \text{true}; P \vdash Q \]
\[ = \langle \langle \vdash \text{def} \rangle \rangle \]
\[ \text{true}; \text{ok} \land P \Rightarrow \text{ok}' \land Q \]
\[ = \langle \langle \vdash \text{def} \rangle \rangle, \text{with substitution} \]
\[ \exists \nu_m, \text{ok}_m \bullet \text{true} \land (\text{ok}_m \land P[\nu_m/\nu] \Rightarrow \text{ok}' \land Q[\nu_m/\nu]) \]
\[ = \exists \nu_m, \text{ok}_m \bullet \nabla \text{ok}_m \lor P[\nu_m/\nu] \lor \text{ok}' \land Q[\nu_m/\nu] \]
\[ = \langle \langle \text{deMorgan} \rangle \rangle \]
\[ \exists \nu_m, \text{ok}_m \bullet \nabla \text{ok}_m \lor P[\nu_m/\nu] \lor \text{ok}' \land Q[\nu_m/\nu] \]
\[ = \langle \langle \text{Forever-\text{def}} \rangle \rangle \]

\[ \text{Forever} \]
\[ \square \]
Proof of \(\langle ;\s R\s-Zero\rangle\)

- **Goal:** \(D; \text{Forever} = \text{Forever}\)
- **Strategy:** reduce rhs to lhs

\[
D; \text{Forever} = \langle \text{Forever-def} \rangle
D; \text{true}
\]

- But stating this equals \text{Forever}, i.e. \text{true}, is the same as stating that \(D\) is H4-healthy (feasible)!
- We need to revise our law:

\[
\langle ;\s R\s-Zero\rangle\quad D \text{ is H4 } \Rightarrow (D; \text{Forever} = \text{Forever})
\]

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.

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, $\pi$-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: 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
  (requirements; specification; code; proofs)
- 4 more bugs uncovered in code
- Work by Jim Woodcock uncovered nine requirements-related defects