We showed that $S_{\text{Loop}} \sqsubseteq P_{\text{Loop}}$

- 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

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

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

Loops and Invariants

- If \( \text{inv} \) is an invariant for \( c \cdot P \), then the following holds:
  - If \( c \) and \( \text{inv} \) are true, then execution of the loop body will leave \( \text{inv}' \) true.
    \[ \langle \langle \text{rel-is-invariant} \rangle \rangle \text{\v} : [c \land \text{inv} \Rightarrow \text{inv}'] \sqsubseteq P \]
  - If \( \text{inv} \) is true beforehand, then on termination of the loop, \( \text{inv}' \) is true, as is \( \neg c' \)
    \[ \langle \langle \text{rel-\Leftrightarrow} \rangle \rangle \text{\v} : [\text{inv} \Rightarrow \text{inv}' \land \neg c'] \sqsubseteq c \cdot (\text{\v} : [c \land \text{inv} \Rightarrow \text{inv}']) \]

- Note that we have to carry the same frame \( \text{\v} : [\ldots] \) through the refinement.

- So, we have two refinement checks:
  - \( \langle \langle \text{rel-is-invariant} \rangle \rangle \) checks that \( \text{inv} \) is an invariant for \( c \cdot P \)
  - \( \langle \langle \text{rel-\Leftrightarrow} \rangle \rangle \) states how we can refine such a loop

Invariants

- For a while-loop \( c \cdot P \), 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 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 (I)

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

\[
\begin{align*}
\text{\v} : [c \land \text{inv} \Rightarrow \text{inv}'] \sqsubseteq P \\
\text{\v} : [\text{inv} \Rightarrow \text{inv}' \land \neg c'] \sqsubseteq c \cdot (\text{\v} : [c \land \text{inv} \Rightarrow \text{inv}']) \\
\text{\v} : [\text{inv} \Rightarrow \text{inv}' \land \neg c'] \sqsubseteq c \cdot P
\end{align*}
\]

- In other words, in order to show that \( S \sqsubseteq c \cdot P \), we need to:
  - Look at \( c \) and \( P \) to find an appropriate invariant \( \text{inv} \), satisfying \( \langle \langle \text{rel-is-invariant} \rangle \rangle \).
  - Show \( S \) is refined by \( \text{inv} \Rightarrow \text{inv}' \land \neg c' \).
  - Show that \( \langle \langle \text{rel-\Leftrightarrow} \rangle \rangle \) holds.
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 $\langle \langle \text{rel-init-loop-}\sqsubseteq \rangle \rangle$ allows us to couple both initialisation and the loop:

  $\vec{v} : [\text{pre} \Rightarrow inv' \land c']$
  $\sqsubseteq \vec{v} : [\text{pre} \Rightarrow inv'] ; c \ast \vec{v} : [c \land inv \Rightarrow inv']$

- Given refinement check $S \sqsubseteq c \ast P$, 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

  \[
  s, i : [s' = S(n)]
  \]

  \[
  \sqsubseteq s, i : [s = 0, n; (i > 0) \ast (s, i := s + i, i - 1)]
  \]

  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:

\[
\begin{align*}
  s, i & : [s' = S(n)] \\
  = & \quad \langle \Rightarrow \text{-l-unit} \rangle \\
  s, i & : [\text{true} \Rightarrow s' = S(n)] \\
  \sqsubseteq & \quad \langle \text{strengthen-post} \rangle, \text{using } x : [A] \sqsubseteq \vec{x} : [A \land B] \\
  s, i & : [\text{true} \Rightarrow s' = S(n) \land i = 0] \\
  = & \quad \langle \text{-rel-init-loop-} \rangle \quad \langle \Rightarrow \text{-l-unit} \rangle \\
  s, i & : [\text{true} \Rightarrow s' + S(i') = S(n') \land i' = 0] \\
  = & \quad \langle \text{rel-init-loop-} \rangle \quad \langle \text{-l-unit} \rangle \\
  s, i & : [\text{true} \Rightarrow s' + S(i') = S(n')]
\end{align*}
\]

\[
\begin{align*}
  ; (i > 0) \ast s, i & : [i > 0 \land s + S(i) = S(n) \Rightarrow s' + S(i') = S(n')] \\
\end{align*}
\]

At this point we can refine the two statements individually.

More about “equality substitution”

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

  \[
  x = e \land P \quad \equiv \quad x = e \land P[e/x]
  \]

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

  \[
  \begin{align*}
  \langle \Rightarrow \Rightarrow \text{-subst} \rangle & \quad (x = e \land P \Rightarrow Q) \quad \equiv \quad (x = e \land P \Rightarrow Q[e/x]) \\
  \langle \Rightarrow \Rightarrow \text{-subst} \rangle & \quad (S \sqsubseteq P \land x = e) \quad \equiv \quad (S[e/x] \sqsubseteq P \land x = e)
  \end{align*}
  \]

- Proofs: easy exercises in propositional calculus
The Simple example (redone — III)

The first statement

\[ s, i = 0, n \implies s' + S(i') = S(n') \]

\[ s, i = s + i, i - 1 \]

\[ s' = s + i \land i' = i - 1 \land n = n \]

\[ s' = s + i \land i' = i - 1 \land n = n \]

Refining by Initialised While-Loops: recap

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

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

\[ \vec{v} : [pre \implies inv'] \subseteq \text{init} \]

\[ \vec{v} : [c \land inv \implies inv'] \subseteq P \]

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.
Pre and post conditions

- 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, 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 \( x < 0 \)?

Mini-Exercise 5

Q5 Given

- \( n, f, x : \mathbb{N} \)
- \( fac(0) = 1 \)
- \( fac(n) = n \ast fac(n−1), \ n > 0 \)

\[
FSpec \triangleq f, x : [f' = fac(n)]
\]
\[
FProg \triangleq f, x := 1, 2 \ ; (x \leq n) \ast f, x := f \ast x, x + 1
\]

5.1 State clearly the proofs that need to be done.
5.2 Determine a suitable invariant.
5.3 Prove that statement (one of the proofs in the answer to 5.1) that \( FSpec \) is refined appropriately.

(due in next Thursday, 12noon, 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 Saoithín

- Saoithín is a theorem proving assistant designed for UTP
- It is under development here at TCD
- A version tailored for CS3001 is now available
  - Saoithín0.90.a4
- Open-source, has been built on Linux; Binaries for Windows

Getting and installing Saoithín

- Follow links from class web-page
- (Windows) Download the ZIP file
- Copy everything into a new folder
- If using Unix, you can build from source
  http://www.scss.tcd.ie/Andrew.Butterfield/Saoithin/
Running Saoithín

- For this course, you need to run it under Windows, logged in under your TCD username.
- 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.

Saoithín Top-Level Theory Stack

- Related definitions, axioms and laws form a “Theory”
- Saoithín 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.
Saoithín User-Interface

- 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

Saoithín Theories

Theories consist of a series of tables:

**OBS** *Name → Type* Observation Variables
**LANGUAGE** Theory-specific language constructs
**PRECEDENCE** *Name → Type* Binary Operator Precedences
**TYPEdef** *Name → Type* Type Definitions
**CONSTdef** *Name → Expr* Constant Definitions
**EXPRdef** *Name → Expr* Expression Definitions
**PREDdef** *Name → Pred* Predicate Definitions
**TYPES** *Var → Type* Types
**CONJ.** *Name → Law* Conjectures
**THEOREMS** *Name → Proof* Theorems
**LAWS** *Name → Law* Laws

Saoithín Top Level Window (TYPES in Theory)

Saoithín Top Level Window (Laws)
Saoithín Top Level Window (Conjectures)

<table>
<thead>
<tr>
<th>Name</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>P == P</td>
</tr>
<tr>
<td>0'</td>
<td>P' == P'</td>
</tr>
<tr>
<td>0''</td>
<td>P'' == P''</td>
</tr>
<tr>
<td>0x</td>
<td>IF P THEN (\neg P) END</td>
</tr>
<tr>
<td>0x+</td>
<td>IF P THEN TRUE END</td>
</tr>
<tr>
<td>0x'</td>
<td>IF P THEN (\neg P) END</td>
</tr>
<tr>
<td>0x''</td>
<td>IF P THEN (\neg P) END</td>
</tr>
<tr>
<td>0x'x'</td>
<td>IF P THEN (\neg P) END</td>
</tr>
<tr>
<td>0x'x''</td>
<td>IF P THEN (\neg P) END</td>
</tr>
<tr>
<td>0x''x''</td>
<td>IF P THEN (\neg P) END</td>
</tr>
<tr>
<td>0x'x''x''</td>
<td>IF P THEN (\neg P) END</td>
</tr>
<tr>
<td>0x'x''x''x''</td>
<td>IF P THEN (\neg P) END</td>
</tr>
</tbody>
</table>

[Diagram of the Saoithín Top Level Window (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 ≡) 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 ⇒) 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

Distribution Files

- Binaries
  - Saoithin.exe
  - wxc-msw2.8.10-0.11.1.2.dll
  - Saoithin-*.wav
  - keep together
- Documentation
  - *.txt
  - read!
- Helpfile
  - Needs to be moved to application data directory
  - Saoithín tells you where on first use.

Workspaces

All your working files live in a workspace:
- A directory holding your files
- Saoithín 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 Theory Stack Window
- Never click the red X button in the top right corner of a window
  (exception: do this to end the “Proof Key Shortcuts” window)
Mini-Exercise 6

Q6.0 Install Saoithin (v0.90α4)
Q6.1 Use it to prove conjectures 1–10 (1–4 done in class, figure out 5–10)

Paper Submission .txt files generated (per proof).
Electronic Submission .txt files generated (per proof), plus final version of GS3001.teoric

(due in next teaching-week Thursday, 12noon, in class)