Loop Refinement Example 1 (Floyd ’67)

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

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

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

Prove
\[\text{ASpec} \subseteq \text{AProg}\]

Refining Floyd’67: process

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

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 \(s + \sum_{j=k}^{n} a_j = A(n)\).

\[a_{inv} \triangleq s + \sum_{j=i}^{n} a_j = A(n)\]
Refining Floyd’67: process (1x)

We introduce a shorthand, with laws:

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

Now the invariant is more concise

\[
ainv \triangleq s + A(i, n) = A(n)
\]

Refining Floyd’67: process (2)

We must show \( A\text{Spec} \subseteq s, i : [ainv \land i' > n'] \)

\[
\begin{align*}
s, i : [s' = A(n)] & \subseteq s, i : [s' + A(i', n') = A(n') \land i' > n'] \\
= & \quad \"\text{frame-def}\" \\
s' & = A(n) \land n' = n \subseteq s' + A(i', n') = A(n') \land i' > n' \land n' = n \\
= & \quad \"\text{prog-strengthen}\" \\
s' & = A(n') \land n' = n \subseteq s' = A(n') \land i' > n' \land n' = n
\end{align*}
\]

true

Refining Floyd’67: process (3)

We must show \( s, i : [ainv'] \subseteq i, s := 1, 0 \)

\[
\begin{align*}
s, i : [s' = A(i', n')] & = A(n') \subseteq s, i := 1, 0 \\
= & \quad \"\text{sim:-def}\" \\
s' + A(i', n') & = A(n') \land n' = n \subseteq i' = 1 \land s' = 0 \land n' = n \\
= & \quad \"\text{prog-strengthen}\" \\
0 + A(1, n') & = A(n') \land n' = n \subseteq i' = 1 \land s' = 0 \land n' = n \\
= & \quad \"\text{law of A(n)} \" \\
true \land n' = n & \subseteq i' = 1 \land s' = 0 \land n' = n \\
= & \quad \"\text{A-unit}\" \\
true
\end{align*}
\]

Refining Floyd’67: process (4)

We must show \( s, i : [i \leq n \land ainv \Rightarrow ainv'] \subseteq s, i := s + a_i, i + 1 \)

\[
\begin{align*}
s, i : [i \leq n \land s + A(i, n) = A(n) \Rightarrow s' + A(i', n') = A(n')] \\
\subseteq s, i := s + a_i, i + 1 \\
= & \quad \"\text{sim:-def}\" \\
s, i : [i \leq n \land s + A(i, n) = A(n) \Rightarrow s' + A(i', n') = A(n')] \\
\subseteq s' = s + a_i \land i' = i + 1 \land n' = n \\
= & \quad \"\text{prog-strengthen}\" \\
s, i : [i \leq n \land s + A(i, n) = A(n) \Rightarrow s + A(i + 1, n) = A(n)] \\
\subseteq s' = s + a_i \land i' = i + 1 \land n' = n \\
= & \quad \"\text{law of A(n)} \" \\
s, i : [i \leq n \land s + A(i, n) = A(n) \Rightarrow s + A(i, n) = A(n)] \\
\subseteq s' = s + a_i \land i' = i + 1 \land n' = n
\end{align*}
\]
Refining Floyd'67: process (4 cont.)

\[
\begin{align*}
  s, i : & [i \leq n \land s + A(i, n) = A(n) \Rightarrow s + A(i, n) = A(n)] \\
  \models & s' = s + a_i \land i' = i + 1 \land n' = n \\
  = & \quad \text{"} A \land B \Rightarrow B \text{"} \\
  s, i : & [\text{true}] \\
  \models & s' = s + a_i \land i' = i + 1 \land n' = n \\
  = & \quad \text{"} \langle \text{frame-def} \rangle \text{"} \\
  \text{true} \\
\end{align*}
\]

Refining Hoare'69: process

We need to

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

Loop Refinement Example 2 (Hoare '69)

Given

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

Prove

\( HSpec \sqsubseteq HProg \)

Looking at loop execution:

\[
\begin{array}{c|c|c|c|c|c|c}
  r & x & x - y & x - 2y & x - k \cdot y & x - n \cdot y \\
  q & 0 & 1 & 2 & k & n \\
\end{array}
\]

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

\( \text{hin}v \triangleq x = r + q \cdot y \)
Refining Hoare’69: process (2)

We must show $HSpec \sqsubseteq r, q : [hinv' \land y' > r']$
\[
\begin{align*}
    r, q : [y' > r' \land x' = r' + y' \land y' > r'] \\
    \sqsubseteq r, q : [x' = r' + q' \land y' > r'] \\
    = \quad \text{``prog-strengthen''}
\end{align*}
\]
true

Refining Hoare’69: process (3)

We must show $r, q : [hinv'] \sqsubseteq r, q := x, 0$
\[
\begin{align*}
    r, q : [x' = r' + q' \land y'] \\
    \sqsubseteq r, q := x, 0 \\
    = \quad \text{``prog-strengthen''}
\end{align*}
\]
\[
\begin{align*}
    x' = r' + q' \land x' = x \land y' = y \\
    \sqsubseteq r' = x \land q' = 0 \land x' = x \land y' = y \\
    = \quad \text{``arithmetic''}
\end{align*}
\]
true

Refining Hoare’69: process (4)

We must show $r, q : [y \leq r \land \text{hinv} \Rightarrow \text{hinv}] \sqsubseteq r, q := r - y, 1 + q$
\[
\begin{align*}
    r, q : [y \leq r \land x = r + q \land y \Rightarrow x' = r' + q' \land y'] \\
    \sqsubseteq r, q := r - y, 1 + q \\
    = \quad \text{``prog-strengthen''}
\end{align*}
\]
\[
\begin{align*}
    y \leq r \land x = r + q \land y \Rightarrow x' = r' + q' \land y' = y \\
    \sqsubseteq r' = r - y \land q' = 1 + q \land x' = x \land y' = y \\
    = \quad \text{``prog-strengthen''}
\end{align*}
\]
\[
\begin{align*}
    y \leq r \land x = r + q \land y \Rightarrow x = r - y + (q + 1) \land y' = y \\
    \sqsubseteq r' = r - y \land q' = 1 + q \land x' = x \land y' = y \\
    = \quad \text{``arithmetic''}
\end{align*}
\]
true

Refining Hoare’69: process (4 cont.)

\[
\begin{align*}
    y \leq r \land x = r + q \land y \Rightarrow x = r - y \land y' = y \\
    \sqsubseteq r' = r - y \land q' = 1 + q \land x' = x \land y' = y \\
    = \quad \text{``prog-strengthen''}
\end{align*}
\]
true
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
  - \( p : a_p' = w < w \in a < p' = 0 \)

Searching an Array

\[
\begin{align*}
    a & : \text{array } 1 \ldots N \text{ of } T \\
    w & : T \\
    p & : \mathbb{N} \\
\end{align*}
\]

\[
\begin{align*}
    \text{ASSpec} & \equiv p : [a_p' = w < w \in a \triangleright p' = 0] \\
    \text{ASProg} & \equiv p := N ; (a_p \neq w \land p > 0) \triangleright p := p - 1 \\
\end{align*}
\]

The Proof obligations

1. find an appropriate invariant \( \text{asinv} \)
2. Prove \( \text{ASSpec} \sqsubseteq p : [\text{asinv'} \land \neg (a_p' \neq w \land p' > 0)] \)
3. Prove \( p : [\text{asinv'}] \sqsubseteq p := N \)
4. Prove \( p : [a_p \neq w \land p > 0 \land \text{asinv} \Rightarrow \text{asinv'}] \sqsubseteq p := p - 1 \)
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:
  \[
  w \in a \equiv \exists k \bullet 1 \leq k \leq N \land a_k = w
  \]
- We also want to talk about membership in a subset of an array:
  \[
  \begin{align*}
  w \in a_{\ell \ldots u} & \equiv \exists k \bullet \ell \leq k \leq u \land a_k = w, \quad 1 \leq \ell \leq u \leq N \\
  a_{\ell \ldots u} = a_{\ell+1 \ldots u} & \equiv w = a_{\ell} \lor w \in a_{\ell+1 \ldots u}, \quad \ell \leq u \\
  w \in a_{\ell \ldots u} = \text{false}, \quad \ell > u
  \end{align*}
  \]
Picking the invariant

- An overview of a run of the algorithm looks like:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>k</th>
<th>k+1</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a_1</td>
<td>a_2</td>
<td>···</td>
<td>a_k</td>
<td>a_{k+1}</td>
<td>···</td>
</tr>
</tbody>
</table>

- 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 \( asinv = 0 \leq p \leq N \land w \notin a_{p+1...N} \)

The Proof obligations (revisited)

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

Wrong Choice?

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

This doesn't reduce to \( \text{true} \) as expected.
- We get an execution of \( P \) in an arbitrary starting state!

Wrong Choice?

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

This works — but we ruled it out because it refines any specification.
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 **true** (a.k.a. **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".

Observing (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** variable.
  - We assume no program has such a variable
- 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 \( ok \) and \( ok' \)

- Consider the specification: \( pre \Rightarrow Post \)
  - i.e. if \( pre \) holds at start, the \( Post \) relates start and end
    (provided the program stops)
- We shall now replace the above by:
  \[ ok \land pre \Rightarrow ok' \land Post \]
- if started in a stable state \( ok \)
- and \( pre \) holds at start
- then, we end in a stable state \( ok' \),
- and the relation \( Post \) holds of starting and ending states.
- For sequential imperative programs, stability is termination
- **Important**: We assume that neither \( pre \) nor \( Post \) mention \( ok \) or \( ok' \).
- Also, variables \( ok \) and \( ok' \) are added to the alphabet of every language construct.

Program Semantics using \( ok \) and \( ok' \)

- We can now give our program language semantics in this style:
  \[
  \begin{align*}
  \langle \langle \text{skip-def} \rangle \rangle & \quad \text{skip} \triangleq ok \Rightarrow ok' \land \nu' = \nu \\
  \langle \langle ;=\text{-def} \rangle \rangle & \quad x := e \triangleq ok \Rightarrow ok' \land x' = e \land \nu' = \nu \\
  \langle \langle ;=\text{-def} \rangle \rangle & \quad P; Q \triangleq \exists \nu_m, ok_m \cdot \\
 & \quad P[\nu_m, ok_m/\nu', ok'] \\
 & \quad \land Q[\nu_m, ok_m/\nu, ok]
  \end{align*}
  \]
- The rules regarding alphabets are unchanged, remembering that \( ok \) and \( ok' \) 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.

The Meaning of \( ok \land pre \Rightarrow ok' \land Post \)

Given this specification, then for any run of a satisfying program:
- If \( ok = False \), i.e. we started in a divergent state
  (previous program diverged) and then any subsequent behaviour was acceptable.
- If \( pre = False \), then anything was also allowed.
- If \( ok = pre = True \), then \( ok' \) must have been true (the program terminated without diverging), and relation \( Post \) was satisfied.

(Some) Laws of Programming

- The followings laws, seen before, all still hold.
  \[
  \begin{align*}
  \langle \langle \text{skip-alt} \rangle \rangle & \quad \text{skip}_A = x :=_A x \\
  \langle \langle ;=\text{-assoc} \rangle \rangle & \quad P; (Q; R) = (P; Q); R \\
  \langle \langle ;=\text{-seq} \rangle \rangle & \quad x := e; x := f = x := f[e/x] \\
  \langle \langle ;=\text{-swap} \rangle \rangle & \quad x := e; y := f = y := f[e/x] ; x := e, \\
 & \quad y \not\in e \\
  \langle \langle \text{<true>} \rangle \rangle & \quad P \triangleq True \triangleright Q = P \\
  \langle \langle \text{<false>} \rangle \rangle & \quad P \triangleq False \triangleright Q = Q \\
  \langle \langle \text{<seq>} \rangle \rangle & \quad (P \triangleq c \triangleright Q); R = (P; R) \triangleq c \triangleright (Q; R)
  \end{align*}
  \]
Laws requiring the new form

- If $P$ has the (new) form
  \[ ok \land pre \Rightarrow ok' \land Post \]
  then the following laws also hold:
  \[
  \langle \langle \Skip; -\Unit \rangle \rangle \Skip; P = P \\
  \langle \langle \Skip \rangle \rangle \Skip; P = P \\
  \]

- However, they are no longer true for arbitrary $P$.
- This occurs with laws involving assignment and $\Skip$ whose definitions have changed.

Proof of $\langle \langle \Skip; -\Unit \rangle \rangle$ (I)

\[
\exists \nu_m, ok_m \bullet \ \\
(\nu \Rightarrow ok' \land Post)_{\nu_m, ok_m, \nu, ok} \\
= \ \\
\text{substitution, noting that } ok, ok' \notin \text{pre, Post} \\
(\exists \nu_m, ok_m \bullet \ \\
(\nu \Rightarrow ok_m \land \nu = \nu) \\
\land (ok_m \land \text{pre}_{\nu_m, \nu} \Rightarrow ok' \land Post[\nu_m/\nu]))
\]

We have something of the form $\exists \xi \bullet (A \Rightarrow B) \land (C \Rightarrow D)$, which we now transform.

Proof of $\langle \langle \Skip; -\Unit \rangle \rangle$ (II)

\[
\exists \xi \bullet (A \Rightarrow B) \land (C \Rightarrow D) \\
= \ \\
\langle \langle \Rightarrow\text{-def} \rangle \rangle \\
\exists \xi \bullet (\neg A \lor C) \lor (\exists \xi \bullet \neg A \land D) \lor B \lor C \lor B \land D \\
= \ \\
\langle \langle \exists\text{-distr}, \text{several times} \rangle \rangle \\
(\exists \xi \bullet \neg A \land C) \lor (\exists \xi \bullet \neg A \land D) \lor \\
(\exists \xi \bullet B \lor C) \lor (\exists \xi \bullet B \land D)
\]

If we apply this to our proof we get

\[
\exists \nu_m, ok_m \bullet \neg ok \land (ok_m \land \text{pre}_{\nu_m, \nu}) \lor \\
\exists \nu_m, ok_m \bullet \neg ok \land ok' \land \text{Post}_{\nu_m, \nu} \lor \\
(\exists \nu_m, ok_m \bullet ok_m \land \nu = \nu \land (ok_m \land \text{pre}_{\nu_m, \nu}) \lor \\
(\exists \nu_m, ok_m \bullet ok_m \land \nu = \nu \land ok' \land \text{Post}_{\nu_m, \nu})
\]
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 \( \langle \langle \text{skip} - \text{-unit} \rangle \rangle \)):

  - \((S;U.1)\) \(\exists \nu_m, ok_m \cdot \neg ok \land \neg (ok_m \land \text{pre}[\nu_m, /\nu])\)
  - \((S;U.2)\) \(\exists \nu_m, ok_m \cdot \neg ok \land ok' \land \text{Post}[\nu_m /\nu]\)
  - \((S;U.3)\) \(\exists \nu_m, ok_m \cdot ok_m \land \nu_m = \nu \land \neg (ok_m \land \text{pre}[\nu_m, /\nu])\)
  - \((S;U.4)\) \(\exists \nu_m, ok_m \cdot ok_m \land \nu_m = \nu \land ok' \land \text{Post}[\nu_m /\nu]\)

Proof of \( \langle \langle \text{skip} - \text{-unit} \rangle \rangle \) (IV, simplifying S;U.1)

\[
\exists \nu_m, ok_m \cdot \neg ok \land \neg (ok_m \land \text{pre}[\nu_m, /\nu])
\]

- "move quantifier in"
- \(\neg ok \land \exists \nu_m, ok_m \cdot \neg (ok_m \land \text{pre}[\nu_m, /\nu])\)
- "witness: \(ok_m = False\) makes existential body true"
- \(\neg ok \land \text{true}\)
- "simplify"
- \(\neg ok\)

Proof of \( \langle \langle \text{skip} - \text{-unit} \rangle \rangle \) (V, simplifying S;U.2)

\[
\exists \nu_m, ok_m \cdot \neg ok \\land \neg (ok_m \land \text{pre}[\nu_m, /\nu])
\]

- "move quantifier in"
- \(\neg ok \land \exists \nu_m, ok_m \cdot \neg (ok_m \land \text{pre}[\nu_m, /\nu])\)
- "witness: \(ok_m = False\) makes existential body true"
- \(\neg ok \land \text{true}\)
- "simplify"
- \(\neg ok\)

Proof of \( \langle \langle \text{skip} - \text{-unit} \rangle \rangle \) (VI, simplifying S;U.3)

\[
\exists \nu_m, ok_m \cdot ok_m \land \nu_m = \nu \land \neg (ok_m \land \text{pre}[\nu_m, /\nu])
\]

- "de-Morgan"
- \(\exists \nu_m, ok_m \cdot ok_m \land \nu_m = \nu \land \neg (ok_m \land \text{pre}[\nu_m, /\nu])\)
- "distributivity"
- \(\exists \nu_m, ok_m \cdot ok_m \land \nu_m = \nu \land \neg ok_m \lor \text{pre}[\nu_m, /\nu]\)
- "contradiction"
- \(\exists \nu_m, ok_m \cdot False \lor ok_m \land \nu_m = \nu \land \neg \text{pre}[\nu_m, /\nu]\)
- "simplify, one-point"
- \(\exists ok_m \cdot ok_m \land \neg \text{pre}\)
- "witness, \(ok_m = True\"
- \(\neg \text{pre}\)
Proof of \(\langle \langle \text{skip}; \text{-unit} \rangle \rangle\) (VII, simplifying S;U.4)

\[ \exists \nu_m, ok_m \implies ok_m \land \nu_m = \nu \land ok' \land \text{Post}[\nu_m/\nu] \]

\[ = \text{ " one-point law "} \]
\[ \exists ok_m \implies ok_m \land ok' \land \text{Post} \]

\[ = \text{ " witness: } ok_m = \text{ true } \]

\[ ok' \land \text{Post} \]

We now merge our four simplifications, and continue:

\[ \neg \text{ ok } \lor \neg \text{ ok } \land \text{ ok}' \land \exists \nu_m \implies \text{Post}[\nu_m/\nu] \lor \neg \text{ pre } \lor \text{ ok}' \land \text{Post} \]

\[ = \text{ " } \langle \langle \lor \land \text{-absorb} \rangle \rangle \text{ "} \]
\[ \neg \text{ ok } \lor \neg \text{ pre } \lor \text{ ok}' \land \text{Post} \]

\[ = \text{ " deMorgan "} \]
\[ \neg (\text{ ok } \land \text{ pre }) \lor \text{ ok}' \land \text{Post} \]

\[ = \text{ " } \langle \langle \Rightarrow \text{-def} \rangle \rangle \text{ "} \]
\[ \text{ ok } \land \text{ pre } \Rightarrow \text{ ok}' \land \text{Post} \]

\[ \Box \]

“Witness”?

- What is meant by the proof steps labelled “witness”?
- Remember, \( \exists x \implies P \) is true if any value of \( x \) exists that makes \( P \) true. Such a value is a “witness” (to the truth of the existential).
- We have a law that states: \( P[e/x] \Rightarrow \exists x \implies P \)
- by \( \langle \langle \Rightarrow \text{-join} \rangle \rangle \) this becomes \( P[e/x] \lor \exists x \implies P \equiv \exists x \implies P \)
- In effect, knowing that \( P[e/x] \) is true, we can replace \( \exists x \implies P \) by true:

\[ \exists x \implies P \]

\[ = \text{ " law above "} \]
\[ P[e/x] \lor (\exists x \implies P) \]

\[ = \text{ " we know (can show) that } P[e/x] = \text{ true } \]
\[ \text{true} \lor (\exists x \implies P) \]

\[ = \text{ " simplify "} \]
\[ \text{true} \]

Mini-Exercise 6

Q6.0 Install Saoithín (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)
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?

“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:** $e, f \in \text{Expr} ::= \ldots \mid e_1 = e_2 \mid e_1 \neq e_2$

**Axioms:**

\[
\begin{align*}
&\text{[[refl]]} & e &= e \\
&\text{[[comm]]} & (e = f) &\equiv (f = e) \\
&\text{[[trans]]} & (e = f) \land (f = g) &\Rightarrow (e = g) \\
&\text{[[def]]} & (e \neq f) &\equiv \neg (e = f)
\end{align*}
\]

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

$$m, n \in \text{Expr} ::= \ldots$$

- numeric constants
- basic operations
- additive operators
- multiplicative operators
- exponential operators

We use $r$ and $s$ for atomic predicates over numeric values

$$r, s \in \text{Expr} ::= \ldots$$

- equalities
- comparisons (strict)
- comparisons (non-strict)
- numeric type assertions ($T \in \mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$)

Some definitions for arithmetic

- $0 \vdash \text{def}$: $0 + m = m$
- $\text{succ-}\text{-def}$: $(\text{succ } n) + m = \text{succ}(n + m)$
- $0 \times m = 0$
- $\text{succ}\times\text{-def}$: $(\text{succ } n) \times m = m + (n \times m)$
- $m^0 = 1$
- $\text{-succ-def}$: $m \times \text{succ } n = m \times m^n$
- $1 \vdash \text{def}$: $1 = \text{succ } 0$
- $2 \vdash \text{def}$: $2 = \text{succ } 1$
- $3 \vdash \text{def}$: $3 = \text{succ } 2$

(Some theorems for arithmetic)

- $0 \vdash \text{def}$: $1 + 1 = 2$
- $2 \vdash \text{def}$: $2 + 2 = 4$

(Strictly speaking we need to model digit strings and their valuations in order to do the last three definitions properly)

Some axioms for arithmetic

Axioms cover the basic building blocks

- Zero is a natural number
  $$\langle 0 \text{-is-N} \rangle \quad 0 : \mathbb{N}$$
- If $n$ is a natural number then so is the successor of $n$
  $$\langle \text{succ-}\text{-is-N} \rangle \quad (n : \mathbb{N}) \Rightarrow (\text{succ } n : \mathbb{N})$$
- The successor of a natural number is never zero
  $$\langle \text{succ-}\text{-not-0} \rangle \quad \text{succ } n \neq 0$$
- Successor maps different values to different results
  $$\langle \text{succ-}\text{-injective} \rangle \quad (n = m) \Rightarrow (\text{succ } n = \text{succ } m)$$
- 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 $x : \mathbb{N}$
  $$\langle \text{N}\text{-induction} \rangle \quad P(0) \wedge (\forall n \bullet P[n/x] \Rightarrow P[\text{succ } n/x]) \Rightarrow (\forall x \mid x : \mathbb{N} \bullet P)$$

Some theorems for arithmetic

- $\langle + \text{-0-unit} \rangle \quad n + 0 = n$
- $\langle + \text{-suc}\text{-comm} \rangle \quad (\text{succ } n) + m = n + (\text{succ } m)$
- $\langle + \text{-comm} \rangle \quad n + m = m + n$
- $\langle +\text{-assoc} \rangle \quad \ell + (m + n) = (\ell + m) + n$
- $\langle \ast-\text{comm} \rangle \quad n \times m = m \times n$
- $\langle \ast-\text{-0-def} \rangle \quad n \times 0 = 0$
- $\langle +\text{-assoc} \rangle \quad \ell \times (m \times n) = (\ell \times m) \times n$
- $\langle \ast\text{-distr} \rangle \quad \ell \times (m + n) = \ell \times m + \ell \times n$
Proof of \( \langle +1\text{-equals-2} \rangle \)

Goal: \( 1 + 1 = 2 \)
Strategy: reduce to \textbf{true}

\[
\begin{align*}
1 + 1 &= 2 \\
&= \text{"1-def, 2-def"} \\
(\text{succ } 0) + (\text{succ } 0) &= \text{succ } 1 \\
&= \text{"suc+-def, 1-def"} \\
\text{succ}(0 + \text{succ } 0) &= \text{succ}(\text{succ } 0) \\
&= \text{"0+-def"} \\
\text{succ}(\text{succ } 0) &= \text{succ}(\text{succ } 0) \\
&= \text{"|=refl"} \\
\text{true} &\square
\end{align*}
\]

Proof of \( \langle +0\text{-unit} \rangle \)

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 \( \text{succ } n \), \( \text{succ } n + 0 = \text{succ } n \)
- We have an induction principle (law \( \langle \text{N-induction} \rangle \)), 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 \( \Rightarrow\text{-join} \) transforms this to \( B \equiv B \lor A \)
- Now, consider our proof of \( B \):

\[
\begin{align*}
B &\equiv \text{"by alternate form of law above"} \\
B \lor A &\equiv \text{"given a proof of } A \text{"} \\
B \lor \text{true} &\equiv \text{"\( \lor\text{-zero}\)"} \\
\text{true} &\square
\end{align*}
\]

The Induction Strategy

- Assume an Induction Principle, i.e., a law of the form

\[
Q_1 \land Q_2 \land \ldots \land Q_n \Rightarrow \forall x \bullet P
\]

- In order to prove \( \forall x \bullet P \), we now know it suffices to prove individually each of the \( Q_i \)
- So, given goal \( \forall x \mid x : \text{N} \bullet P \) an inductive proof allows us to prove it via the following two sub-goals:

\[
P[0/x] \\
P[n/x] \Rightarrow P[\text{succ } n/x]
\]
∀ 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 \(\langle +\rangle\) (base case)

(sub-)Goal: \(0 + 0 = 0\)
(sub-)Strategy: reduce lhs to rhs

\[
0 + 0 = "\langle +0-unit\rangle" \\
0
\]

□

Proof of \(\langle +\-0-unit\rangle\) (inductive step)

(sub-)Goal: \(n + 0 = n\)
(sub-)Strategy: assume antecedent \(\langle +0-def.hyp\rangle\)

Show consequent (reduce to true):

\[
\begin{align*}
(succ n) + 0 &= succ n \\
&= "\langle succ-use-def\rangle" \\
succ (n + 0) &= succ n \\
&= "\langle +0-def.hyp\rangle" \\
succ n &= succ n \\
&= "\langle =-refl\rangle" \\
true
\end{align*}
\]

□
Arithmetic Theory: summary

- Induction using $\langle \text{N-induction} \rangle$ is the main proof technique for the laws of arithmetic.
- Most if not all of the laws shown are done this way.
- For example, $\langle \text{+ succ- comm} \rangle$ can be proven by induction on $n$, and using $\langle \text{+ 0- unit} \rangle$ to assist in the base-case.

More Arithmetic Theory

We can also define numeric ordering, and supply some theorems

Definitions

- $\langle \text{0-\leq-def} \rangle \quad 0 \leq n$
- $\langle \text{succ-\leq-def} \rangle \quad (\text{succ } n) \leq (\text{succ } m) \equiv n \leq m$
- $\langle \text{0-\leq-bottom} \rangle \quad \neg (\text{succ } n \leq 0)$

Theorems

- $\langle \text{1-lessthan-2} \rangle \quad 1 \leq 2$
- $\langle \text{2-not-lessthan-1} \rangle \quad \neg (2 \leq 1)$
- $\langle \text{+ increases} \rangle \quad n \leq n + m$
- $\langle \text{+ 0- same} \rangle \quad n + m \leq n \Rightarrow m = 0$
Goal: build a portfolio of theories in Saoithin

Method:
1. Rolling deadlines, one per week
2. Each Week develops a new Theory
3. Subsequent Theories build on earlier ones
   - A correct version of each such theory will be made available after each deadline.

Theory Portfolio

- The initial project release with have the following Theories:
  - Logic, Equality
- Due end Week 9: Arithmetic
- Due end Week 10: Lists
- Due end Week 11: Programs (partial correctness)

Submissions:
- Electronic: email, subject CS3001-TheoryName
  - All .teoric files.
- Hardcopy: to CS Office,
  - When: 12noon on Friday

Project Grading

- Marks will be awarded based on the number of conjectures proven (some worth more than others).
- Marks will also be awarded for pointing out genuine errors in theories, along with suggested corrections.
- Also marks will be awarded for identifying bugs (features, even!) in the prover.
- Fixes to theories/software will be issued ASAP.
Hold on! What about \(-\) and \(/\)?

I hoped you wouldn't notice!

Ok, let's have a look at some definitions/theorems.

What's wrong with the simple life anyway?

### Definitions and Theorems for Subtraction

#### Definitions:

- \(\langle \langle -0\text{-def} \rangle \rangle n = 0 = n\)
- \(\langle \langle -\text{succ-def} \rangle \rangle (\text{succ } n) - (\text{succ } m) = n - m\)

#### Theorems:

1. \(\langle \langle +-\text{-inv} \rangle \rangle (m + n) - n = m\)
2. \(\langle \langle +-\text{-inv} \rangle \rangle (m - n) + n = m\)
3. \(\langle \langle \text{-decreases} \rangle \rangle (m - n) \leq m\)
4. \(\langle \langle 3\text{-minus-2-is-1} \rangle \rangle 3 - 2 = 1\)
5. \(\langle \langle 1\text{-minus-3-lt-1} \rangle \rangle 1 - 3 \leq 1\)

Let's prove some of these.

---

**Proof of \(\langle \langle 3\text{-minus-2-is-1} \rangle \rangle**

Goal: \(3 - 2 = 1\)

Strategy: reduce lhs to rhs.

\[
\begin{align*}
3 - 2 &= \langle \langle 3\text{-def} \rangle \rangle, \langle \langle 2\text{-def} \rangle \rangle \\
&= (\text{succ } 2) - (\text{succ } 1) \\
&= \langle \langle -\text{succ-def} \rangle \rangle \\
2 - 1 &= \langle \langle 2\text{-def} \rangle \rangle, \langle \langle 1\text{-def} \rangle \rangle \\
&= (\text{succ } 1) - (\text{succ } 0) \\
&= \langle \langle -\text{succ-def} \rangle \rangle \\
1 - 0 &= \langle \langle -0\text{-def} \rangle \rangle \\
1
\end{align*}
\]

[1]works for theorems using \(=\) as well as \(\equiv\).

---

**Proof of \(\langle \langle +-\text{-inv} \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 \(\langle \langle - -\text{inv} \rangle \rangle\), Base Case**

Goal: \((m + 0) - 0 = m\)
Strategy: reduce lhs to rhs

\[
\begin{align*}
(m + 0) - 0 & = "\langle \langle +\text{-unit} \rangle \rangle" \\
m - 0 & = "\langle \langle -0\text{-def} \rangle \rangle" \\
m & \checkmark
\end{align*}
\]

**Proof of \(\langle \langle - -\text{inv} \rangle \rangle\), Inductive Step**

Goal: \(((m + n) - n = m) \Rightarrow ((m + (\text{succ } n)) - (\text{succ } n) = m)\)
Strategy: assume antecedent
\(\langle \langle +\text{-ind-hyp} \rangle \rangle \quad (m + n) - n = m\)
and reduce consequent lhs to rhs

\[
\begin{align*}
(m + (\text{succ } n)) - (\text{succ } n) & = "\langle \langle +\text{-comm} \rangle \rangle" \\
((\text{succ } n) + m) - (\text{succ } n) & = "\langle \langle \text{succ} +\text{-def} \rangle \rangle" \\
\text{succ}(n + m) - (\text{succ } n) & = "\langle \langle \text{succ} -\text{succ-def} \rangle \rangle" \\
(n + m) - n & = "\langle \langle +\text{-ind-hyp} \rangle \rangle" \\
m & \checkmark
\end{align*}
\]

**Proof of \(\langle \langle 1\text{-minus-3-lt-1} \rangle \rangle\)**

Goal: \(1 - 3 \leq 1\)
Strategy: reduce to true

\[
\begin{align*}
1 - 3 & \leq 1 \\
& = "\langle \langle 1\text{-def} \rangle, \langle 3\text{-def} \rangle \rangle" \\
(succ 0) - (succ 2) & \leq (succ 0) \\
& = "\langle \langle \text{succ} -\text{succ-def} \rangle \rangle" \\
0 - 2 & \leq (succ 0) \\
& = "??? no law applies ??? "
\end{align*}
\]

- It turns out that \(\langle \langle 1\text{-minus-3-lt-1} \rangle \rangle\) is not a theorem!
- Neither is its negation!
- Whazzup?

**Definitions for Subtraction (reminder)**

- Definitions:
  \[\langle \langle -\text{def} \rangle \rangle \quad n - 0 = n\]
  \[\langle \langle \text{succ-def} \rangle \rangle \quad (\text{succ } n) - (\text{succ } m) = n - m\]
**Subtracting 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, \( \langle \langle - - \rangle \rangle \) 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 \( \langle \langle - - \ominus \rangle \rangle \), \( \langle \langle - - suc - \ominus \rangle \rangle \) and \( \langle \langle - - totalise \rangle \rangle \) 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 $\text{prog}_F$:
  - If $f(x)$ is defined, then $\text{prog}_F$ terminates on input $x$ with the right answer
  - If $f(x)$ is undefined, then $\text{prog}_F$ may not terminate
- The question “is $f$ defined on $x$” is then equivalent to solving the halting problem for $\text{prog}_F$
- The Halting Problem is undecidable — no algorithm can ever solve it in all cases.

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

Consider this

- Consider the following code, (assuming $n : \mathbb{N}$):
  $$(n > 1) \ast n := n/2 \land \text{even } n \triangleright n := (3 \cdot n + 1)/2$$

- 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 $n \leq 20 \times 2^{58}$, but no proof that it terminates for all $n$ has been found.

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 $v = (x + y) - z - ((x + y) - z)$ and law $e - e = 0$
  - 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 $v = 0 \land z \leq x + y$.