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

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

Specifying "pick a number"

- The problem with $x' \in \{1, \ldots, 10\}$ is that it allows anything to happen to variables other than $x$.
- We can strengthen it to require that only $x$ be modified:

$$x' \in \{1, \ldots, 10\} \land \nu' = \nu$$

- This idiom is so common we invent specific notation for it:

$$x : [x' \in \{1, \ldots, 10\}]$$

The outer $x$ says, "modifying $x$ only,"
- The notation $x :[\ldots]$ is called a (specification) "frame".

Formally introducing frames

- Once again, we extend our predicate language, with specification frames (n.s.)

$$\text{Pred} ::= \ldots | \bar{x} : [P]$$

- It asserts $P$, and that any variable not in $\bar{x}$ is unchanged:

$$\langle \text{frame-def} \rangle \quad \bar{x} : [P] \quad \equiv \quad P \land \nu' = \nu, \quad A = \{\bar{x}, \bar{x}', \nu, \nu'\}$$

- Not be confused with $[P]$, the universal closure of $P$, or $x : T$, asserting that $x$ has type $T$, or $P[e/x]$ where $e$ replaces free $x$ in $P$. 
Satisfying Specifications

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

Implication is not Satisfaction

- Consider \( y := 99 \Rightarrow x : [x' \in \{1, \ldots, 10\}] \)
- Informally, this fails (is False) if started in a state where:
  - \( x \) is not already in range 1,\ldots,10,
  - or \( y \) is not already equal to 99.
- Formally (assuming only \( x \) and \( y \) in scope):
  - \( \neg (y := 99 \Rightarrow x : [x' \in \{1, \ldots, 10\}] ) \)
  - "\( \langle \langle:=\text{-}\text{def}\rangle, \langle \text{frame-def}\rangle \rangle \)"
  - \( \neg (x' = x \land y' = 99 \Rightarrow x' \in \{1, \ldots, 10\} \land y' = y) \)
  - "\( \langle \langle:=\text{-}\text{def}\rangle, \langle \text{deMorgan}\rangle \text{ (twice) } \rangle \rangle \)"
  - \( x' = x \land y' = 99 \land (x' \notin \{1, \ldots, 10\} \lor y' \neq y) \)
  - "equality substitution (?) — see later"
  - \( x' = x \land y' = 99 \land (x \notin \{1, \ldots, 10\} \lor 99 \neq y) \)

Refinement

- Given specification \( S \) and program \( P \):
  - all before/after-relationships resulting from \( P \) must satisfy those required by \( S \)
  - so \( P \) must imply \( S \) for all possible variable before- and after-values
- So we require \textit{universal} implication, saying that \( P \) satisfies \( S \) iff
  \[
  [P \Rightarrow S]
  \]
- Once more, we add special predicate notation, saying that \( S \) is “refined by” \( P \), written \( S \subseteq P \)

\[
\text{Pred} \quad ::= \\ S \subseteq P \\
\text{\textbar def} \quad S \subseteq P \quad \equiv \quad [P \Rightarrow S]
\]
Example: Assignment as Frame refinement
Consider $x : [P] \sqsubseteq x := e$, assuming $x$ and $y$ in scope,

\[
[x := e \Rightarrow x : [P]] = \text{"<zs-def>, frame-def"} \\
[x' = e \land y' = y \Rightarrow P \land y' = y] = \text{"shunting"} \\
[x' = e \Rightarrow (y' = y \Rightarrow P \land y' = y)] = \text{"(A \Rightarrow B \land A) \equiv (A \Rightarrow B)"} \\
[x' = e \Rightarrow (y' = y \Rightarrow P)] = \text{"<[]-lpt>"} \\
[y' = y \Rightarrow P[e/x']][y/y'] = \text{"<[]-lpt>"} \\
[P[e/x'][y/y']] = \text{" substitution "}
\]

Example: wrong-Assignment as Frame refinement
Consider $y : [P] \sqsubseteq y := e$, assuming $x$ and $y$ in scope,

\[
[y := e \Rightarrow y : [P]] = \text{"<zs-def>, frame-def"} \\
[x' = x \land y' = e \Rightarrow P \land y' = y] = \text{"shunting"} \\
[x' = x \Rightarrow (y' = e \Rightarrow P \land y' = y)] = \text{"<[]-lpt>"} \\
[(y' = e \Rightarrow P \land y' = y)[x'/x]] = \text{"<[]-lpt>"} \\
[(P \land y' = y)[x'/x][e/y']] = \text{" substitution "} \\
[(P[x/x']][e/y] \land e = y)]
\]

Example: $x : [x' \in \{1, \ldots, 10\}] \sqsubseteq x := 2$

\[
x : [x' \in \{1, \ldots, 10\}] \sqsubseteq x := 2 = \text{" previous slide"} \\
[(x' \in \{1, \ldots, 10\}][2/x'][y/y']] = \text{" substitution"} \\
[2 \in \{1, \ldots, 10\}] = \text{" set theory"} \\
true
\]

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

\[
x : [x' \in \{1, \ldots, 10\}] \not\sqsubseteq y := 99 = \text{" previous slide"} \\
[(x' \in \{1, \ldots, 10\}][x/x'][99/y'] \land 99 = y)] = \text{" substitution"} \\
[x \in \{1, \ldots, 10\} \land 99 = y] = \text{" <[]>-split"} \\
[x \in \{1, \ldots, 10\} \land 99 = y] \land (x \in \{1, \ldots, 10\} \land 99 = y)[0, 0/x, y] = \text{" substitution"} \\
[x \in \{1, \ldots, 10\} \land 99 = y] \land 0 \in \{1, \ldots, 10\} \land 99 = 0 = \text{" set-theory, arithmetic, logic, hand-waving"} \\
false
\]
Equality Substitution

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

$$\langle \langle \equiv \text{-subst} \rangle \rangle \ x = e \land P \equiv x = e \land P[e/x]$$

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

Refinement Example: MinMax (I)

- Consider the following specification:

$$S \equiv x, y : [x, y := \max(x, y), \min(x, y)]$$

- “changing only $x$ and $y$, ensure that $x$ ends up as the maximum of the two, whilst $y$ is the minimum.”
- We posit the following “code” as a solution:

$$Q \equiv \text{skip} \land x \geq y \land x, y := y, x$$

- Does $Q$ refine $S$?

Mini-Exercise 4

Q4.1 Prove $S \sqsubseteq P \lor Q \equiv (S \sqsubseteq P) \land (S \sqsubseteq Q)$

$\langle \equiv \text{-prog-alt} \rangle$

Q4.2 Show that $x := -y$; $y := -x$ refines $x := [x' := -y]$

General Note: refinement laws have had $\equiv$ replaced by $\equiv$ in some cases. For predicates, we can assume the following definition of equals:

$$P = Q \equiv [P \equiv Q]$$

(due in next Thursday, 12noon, in class)

$S \sqsubseteq Q$? Solution (Ia)

$$x, y : [x, y := \max(x, y), \min(x, y)]$$

$$\sqsubseteq \text{skip} \land x \geq y \land x, y := y, x$$

$$= \langle \equiv \text{-def} \rangle$$

$$\begin{bmatrix}
\text{skip} \land x \geq y \land x, y := y, x \\
\Rightarrow x, y : [x, y := \max(x, y), \min(x, y)]
\end{bmatrix}$$

$$= \langle \equiv \text{-def} \rangle$$

$$\begin{bmatrix}
x \geq y \land \text{skip} \lor x < y \land x, y := y, x \\
\Rightarrow x, y : [x, y := \max(x, y), \min(x, y)]
\end{bmatrix}$$

$$= \langle A \lor B \Rightarrow C \rangle \equiv (A \Rightarrow C) \land (B \Rightarrow C)$$

$$\begin{bmatrix}
x \geq y \land \text{skip} \\
\Rightarrow x, y : [x, y := \max(x, y), \min(x, y)]
\end{bmatrix}$$

$$\land (x < y \land x, y := y, x$$

$$\Rightarrow x, y : [x, y := \max(x, y), \min(x, y)]$$
S ⊑ Q? Solution (Ib)

= "\langle [], \wedge \text{-distr} \rangle"
\[
\begin{align*}
& x \geq y \land \text{skip} \\
& \Rightarrow x, y : [x, y := \text{max}(x, y), \text{min}(x, y)] \\
\end{align*}
\]
\&
\[
\begin{align*}
& x < y \land x, y := y, x \\
& \Rightarrow x, y : [x, y := \text{max}(x, y), \text{min}(x, y)] \\
\end{align*}
\]

= "\langle [], \wedge \text{-distr} \rangle"
\[
\begin{align*}
& x \geq y \land x' = x \land y' = y \\
& \Rightarrow x' = \text{max}(x, y) \land y' = \text{min}(x, y) \\
\end{align*}
\]
\&
\[
\begin{align*}
& x < y \land x' = y \land y' = x \\
& \Rightarrow x' = \text{max}(x, y) \land y' = \text{min}(x, y) \\
\end{align*}
\]

S ⊑ Q? Solution (Ic)

= "\langle \text{shunting}(2) \rangle"
\[
\begin{align*}
& x' = x \land y' = y \Rightarrow x \geq y \\
& \Rightarrow x' = \text{max}(x, y) \land y' = \text{min}(x, y) \\
\end{align*}
\]
\&
\[
\begin{align*}
& x' = y \land y' = x \Rightarrow x < y \\
& \Rightarrow x' = \text{max}(x, y) \land y' = \text{min}(x, y) \\
\end{align*}
\]

= "\langle \text{shunting}(2) \rangle",
\[
\begin{align*}
& \text{\text{-1pt}}(2), \text{replacing } x' \text{ and } y' \\
\end{align*}
\]
\[
\begin{align*}
& x' = x \land y' = y \Rightarrow \\
& x \geq y \Rightarrow x = \text{max}(x, y) \land y = \text{min}(x, y) \\
\end{align*}
\]
\&
\[
\begin{align*}
& x' = y \land y' = x \Rightarrow \\
& x < y \Rightarrow y = \text{max}(x, y) \land x = \text{min}(x, y) \\
\end{align*}
\]

S ⊑ Q? Solution (Id)

= " properties of max and min "
\[
\begin{align*}
& [x' = x \land y' = y \Rightarrow \text{true]} \\
\end{align*}
\]
\&
\[
\begin{align*}
& [x' = y \land y' = x \Rightarrow \text{true]} \\
\end{align*}
\]

= "\langle \text{\text{-r-zero}} \rangle"
\[
\begin{align*}
& [\text{true]} \land [\text{true}] \\
\end{align*}
\]

= "\langle \text{\text{-true}} \rangle"
\[
\begin{align*}
& \text{true} \land \text{true} \\
\end{align*}
\]

= " logic "
\[
\begin{align*}
& \text{true} \\
\end{align*}
\]

Refinement Example: MinMax (II)

• Given the following (familiar) program
\[
\begin{align*}
& P \ \triangleq \ x := x + y ; \ y := x - y ; \ x := x - y \\
\end{align*}
\]

• Can we replace \( x, y := y, x \) in \( Q \) by \( P \) and still have it refine \( S \)?
re-examining Refinement

- Consider the following laws:
  - $\boxempty$-refl: $P \sqsubseteq P$
  - $\boxempty$-trans: $(S \sqsubseteq Q) \land (Q \sqsubseteq P) \Rightarrow (S \sqsubseteq P)$
  - $\boxempty$-anti: $(P = Q) \equiv (P \sqsubseteq Q) \land (Q \sqsubseteq P)$

  What do they say about refinement?
  - It is a partial order.

Partial Orders (reminder)

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

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

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

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

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

$$\quad (x \cap y = x) \equiv (x \sqsubseteq y) \equiv (x \cup y = y)$$

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

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

\begin{align*}
\langle \sqcap \text{-def} \rangle & \quad P \sqcap Q \equiv P \lor Q \\
\langle \sqcup \text{-def} \rangle & \quad P \sqcup Q \equiv P \land Q
\end{align*}

- From these we can deduce the following laws:

\begin{align*}
\langle \sqcap \text{-join} \rangle & \quad (P \sqsubseteq Q) \equiv (P \sqcup Q \equiv Q) \\
\langle \sqcup \text{-meet} \rangle & \quad (P \sqsubseteq Q) \equiv (P \sqcap Q \equiv P)
\end{align*}

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

Complete Lattices

- A lattice is Complete if meets and joins (maxima/minima) can be found for arbitrary sets of elements:

\begin{align*}
\sqcap S & \quad \text{the minimum of all the elements of } S \\
\sqcup S & \quad \text{the maximum of all the elements of } S
\end{align*}

- Not all of our examples are complete lattices:

\begin{itemize}
  
  \item \( (\mathbb{N}, \leq) \) is not complete — what is the maximum element of the set of all even naturals?
  
  \item The following are complete lattices:
    
    \begin{itemize}
      
      \item \( \mathcal{P} T, \subseteq \), take \( \bigcap \) and \( \bigcup \).
      
      \item \( \langle \text{Pred, } \Rightarrow \rangle \), take \( \forall \), \( \exists \) (?).
    \end{itemize}
\end{itemize}

Complete Lattices: a Key Property

- A complete lattice \( (S, \preceq) \) has overall maximum and minimum elements
  
  \begin{itemize}
    
    \item The minimal element (\( \bot \) or “bottom”) is simply
      
      \[ \bot \equiv \sqcap S \]
    
    \item The maximal element (\( \top \) or “top”) is simply
      
      \[ \top \equiv \sqcup S \]
  \end{itemize}

- Clearly, both top and bottom satisfy the following laws:

\begin{align*}
\bot \preceq x, & \quad \text{for all } x \in S \\
x \preceq \top, & \quad \text{for all } x \in S
\end{align*}

Refinement is a Complete Lattice

- The set of all predicates under the refinement relation forms a complete lattice

\begin{align*}
P \sqcap Q & \equiv P \lor Q \\
P \sqcup Q & \equiv P \land Q \\
\sqcap S & \equiv \lor S \\
\sqcup \{P_1, \ldots, P_n\} & \equiv P_1 \lor \ldots \lor P_n \\
\sqcup S & \equiv \land S \\
\sqcup \{P_1, \ldots, P_n\} & \equiv P_1 \land \ldots \land P_n
\end{align*}

- The minimal events:

\begin{align*}
\bot & \equiv \text{true} \\
\text{true} & \equiv P \\
\top & \equiv \text{false} \\
P & \equiv \text{false}
\end{align*}
What are true and false?

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

- **false** is capable of satisfying any specification!
- We shall call it “**miracle**” (a.k.a. “magic”).
  \( \text{miracle} \equiv \text{false} \)

Implication Lattice

relating predicates with universal implication (\( \_ \Rightarrow \_ \)):

\[
\begin{align*}
\text{false} & \quad \text{strong} \\
\text{false} & \quad \text{false} \quad x' = 3 \\
\text{true} & \quad \text{weak} \\
\text{true} & \quad \text{true} \quad x' = 6 \\
\text{true} & \quad \text{false} \quad x' = 3 \lor x' = 6 \\
\text{true} & \quad \text{false} \quad x' \in \{1 \ldots 10\} \\
\text{true} & \quad \text{true} \\
\end{align*}
\]

Chaos spec. \( \sqsubseteq \) P \( \sqsubseteq \) Miracle prog.

Refinement Lattice

relating programs/specifications with refinement (\( \sqsubseteq \)):

\[
\begin{align*}
\text{miracle} & \quad \text{prog.} \\
x' = 3 & \quad x' = 6 \\
x' = 3 \lor x' = 6 & \\
x' \in \{1 \ldots 10\} & \\
\text{Chaos} & \quad \text{spec.} \\
\end{align*}
\]

Refinement

- The notion of Refinement, relating a specification to any program that satisfies it, is key in formal methods.
- In our formalism, UTP, it (\( \sqsubseteq \)) is defined as “universally-closed (\( [\_] \)) reverse implication (\( \sqsubseteq \)):
  \[
  \langle [\_]-\text{def} \rangle \quad S \sqsubseteq P \quad \equiv \quad [S \Leftarrow P]
  \]

- The key thing to remember is the reversal of the implication, e.g.
  \[
  S \sqsubseteq P \quad \equiv \quad [P \Rightarrow S]
  \]


**Laws of Implication**

- There are many laws of implication:
  
  - $\langle\Rightarrow\rangle$-refl: $P \Rightarrow P \equiv \text{true}$
  - $\langle\Rightarrow\rangle$-trans: $(P \Rightarrow Q) \land (Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$
  - $\langle\equiv\Rightarrow\rangle$-trans: $(P \equiv Q) \land (Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$
  - $\langle\Rightarrow\equiv\rangle$-trans: $(P \Rightarrow Q) \land (Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$
  - $\langle\text{strengthen-ante}\rangle$: $P \land Q \Rightarrow P$
  - $\langle\text{weaken-cnsp}\rangle$: $P \Rightarrow P \lor Q$
  - $\langle\text{ante-\lor-distr}\rangle$: $P \lor Q \Rightarrow R \equiv (P \Rightarrow R) \land (Q \Rightarrow R)$
  - $\langle\text{cnsp\-\land-distr}\rangle$: $P \Rightarrow Q \land R \equiv (P \Rightarrow Q) \land (P \Rightarrow R)$

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

**Exploring the Refinement Laws (I)**

- $P \subseteq P$
  - Anything refines itself (trivially).
- $(P = Q) \Rightarrow (P \subseteq Q)$
  - Equality is a (fairly trivial) refinement.
- $(P = Q) \equiv (P \subseteq Q) \land (Q \subseteq P)$
  - Two programs/specifications are equal if they refine each other.
- $(S \subseteq Q) \land (Q \subseteq P) \Rightarrow (S \subseteq P)$
  - If $Q$ refines $S$, and $P$ refines $Q$, then $P$ also refines $S$. We can do refinement in several steps.
- $(S = Q) \land (Q \subseteq P) \Rightarrow (S \subseteq P)$
  - A refinement step can also be replaced by something equal.

**Exploring the Refinement Laws (II)**

- $S \subseteq P \lor Q \equiv (S \subseteq P) \land (S \subseteq Q)$
  - Refining by an arbitrary choice of alternatives is the same as refining by both options individually.
- $S \land T \subseteq P \equiv (S \subseteq P) \land (T \subseteq P)$
  - Satisfying a specification with two mandatory parts is the same as satisfying both parts individually.
- $S \lor T \subseteq T$
  - A specification offering alternatives can be satisfied by choosing one of those alternatives.
- $P \subseteq P \land Q$
  - A specification can be refined by adding in extra constraints.
Proving some refinement laws

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

Proof of $≡-\sqsubseteq\text{-trans}$

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

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

□

Exploring the Refinement Laws (III)

- Consider the addition of a notation for arrays into our expression language.
- If $a$ is an array, then $a_i$ denotes the $i$th element.
- We assume the array elements are indexed from $1 \ldots N$.

Find index of a value

- Consider specification $spec$:

\[
p : [a_p = w \land w \in a \Rightarrow p' = 0] \n\]

Modifying $p$, set it equal to the index of an element in the array equal to $w$, otherwise zero.
- Consider $prog_1$:

\[
p := 1 \\
; (a_p \neq w \land p < N) \Rightarrow p := p + 1 \\
; \text{skip} \Rightarrow a_p = w \Rightarrow p := 0 \n\]

- Consider $prog_2$:

\[
p := N \\
; (a_p \neq w \land p > 0) \Rightarrow p := p - 1 \n\]
Running the programs

- Consider starting state
  \[ w = 3 \land a = 123456 \quad (N = 6) \]

- Running \( \text{prog}_1 \) results in the final outcome:
  \( p' = 3 \)
  (it searches left-to-right)

- Running \( \text{prog}_2 \) results in the final outcome:
  \( p' = 5 \)
  (it searches right-to-left)

Relating \( \text{spec} \) and the \( \text{prog}_i \)

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

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

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

Refinement and non-determinism

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

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

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

- Refinement is essentially about reducing non-determinism.
Most of the refinement laws seen to date are derived from the implication laws.

We have others as well, based on language constructs:

- **⟨⟨ ⊑-:=⟩⟩**: $x' = e$ ⊑ $x := e$
- **⟨⟨ ⊑-sim-:=⟩⟩**: $\vec{x}' = \vec{e}$ ⊑ $\vec{x} := \vec{e}$
- **⟨⟨ :=-lead⟩⟩**: $P$ ⊑ $x := e$; $x \not\in P$
- **⟨⟨ :=-trail⟩⟩**: $P$ ⊑ $P; x := e$; $x' \not\in P$
- **⟨⟨ sim-:=lead⟩⟩**: $P$ ⊑ $\vec{x} := \vec{e}$; $\vec{x} \not\in P$
- **⟨⟨ sim-:=trail⟩⟩**: $P$ ⊑ $P; \vec{x} := \vec{e}$; $\vec{x} \not\in P$
- **⟨⟨ ⊑-;⟩⟩**: $(S_1 \sqsubseteq P_1) \land (S_2 \sqsubseteq P_2) \Rightarrow (S_1; S_2) \sqsubseteq (P_1; P_2)$

**Proof of ⟨⟨ :=-:=⟩⟩**

$x' = e$ ⊑ $x := e$

```
= "⟨⟨ =-def ⟩⟩"
```

$x' = e$ ⊑ $x' = e \land \nu' = \nu$

```
= "⟨⟨ =-def ⟩⟩"
```

$x' = e \land \nu' = \nu \Rightarrow x' = e$

```
= "prop. law: A \land B \Rightarrow A""
```

```
[true]
```

```
= "⟨⟨ []-true ⟩⟩"
```

```
true
```

**Proof of ⟨⟨ :=-trail⟩⟩**

$P$ ⊑ $P; x := e$

```
= "⟨⟨ :=-def, :=-def ⟩⟩"
```

$[P; x' = e \land \nu' = \nu \Rightarrow P]$

```
= "⟨⟨ :=-def ⟩⟩"
```

$[(\exists x_m, \nu_m \bullet P[x_m, \nu_m/x', \nu'] \land (x' = e \land \nu' = \nu)][x_m, \nu_m/x, \nu)]$

```
= "substitution, noting x' \not\in P"
```

```
⇒ P
```

```
⇒ P
```

(cont. overleaf)
Proof of \(\vdash \text{-trail}\) (cont.)

\[
\begin{align*}
(\exists x_m, \nu_m \cdot P[\nu_m/\nu'] \land x' = e[x_m, \nu_m/x, \nu] \land \nu' = \nu_m) \\
\Rightarrow P
\end{align*}
\]

\[
(\exists x_m, \nu_m \cdot P[\nu_m/\nu'][\nu'/\nu_m] \land x' = e[x_m, \nu_m/x, \nu][\nu'/\nu_m]) \\
\Rightarrow P
\]

\[
\text{(subst-inv),}(\land-\exists)-\text{distr} \text{ with } x_m \notin P
\]

\[
[P \land (\exists x_m \cdot x' = e[x_m, \nu'/x, \nu][\nu'/\nu_m]) \Rightarrow P]
\]

\[
\text{prop. law: } A \land B \\ A
\]

\[
\text{true}
\]

Reasoning about Loops

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

\[
\langle \langle \ast \text{-unroll} \rangle \rangle c \ast P = (P; c \ast P) \land c \triangleright skip
\]

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

Loop Semantics

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

\[
W = (P; W) \land c \triangleright skip
\]

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

\[
(X = (P; X) \land c \triangleright skip) \Rightarrow W \subseteq X
\]

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

Infinite Loops

- What is the meaning of \(\text{true} \ast \text{skip}\) ?
- It is an infinite loop, so call it \(\text{Forever}\)
- It must satisfy the un-rolling law:

\[
\text{Forever} = (\text{skip}; \text{Forever}) \land \text{true} \triangleright \text{skip}
\]

- However, all of the following satisfy this instance of the un-rolling law:

\[
\begin{align*}
\text{miracle} &= \text{skip} \land \text{miracle} \\
\text{skip} &= \text{skip} \land \text{skip} \\
\text{Chaos} &= \text{skip} \land \text{Chaos} \\
P &= \text{skip} \land P \text{ for } \text{any } P \text{ (It’s law } \langle \langle \text{skip}; \cdot \text{-unit} \rangle \rangle) \\
\text{Why pick the least of these as the meaning of } \text{Forever} \text{ ?}
\end{align*}
\]
\textbf{Forever} is bad

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

\textbf{Loop Reasoning}

- Given a loop \( c \ast P \), and possible candidate \( W \):
  - checking the unroll-law is not too bad:
    \[ W = (P; W) \downarrow c \uparrow \text{skip} \]
  - checking for the least such fixpoint is painful:
    \[ (X = (P; X) \downarrow c \uparrow \text{skip}) \Rightarrow W \sqsubseteq X, \text{ for any } X \]
- Fortunately, in many cases, the fixpoint is unique.
- We won’t characterise the unique cases here (quite technical).
- Instead we shall just use the unroll check.
- A challenge still remains: finding \( W \)!

\textbf{A Simple example (I)}

- Our specification: summing (natural) numbers between 1 and \( n \):
  \[ SS\text{Sum} \triangleq s, i : \begin{cases} s' = \sum_{j=1}^n j \end{cases} \]
- Our program:
  \[ PS\text{Sum} \triangleq s := 0; i := n; (i > 0) \ast (s := s + i; i := i - 1) \]
- Our goal, to prove the program satisfies our specification:
  \[ SS\text{Sum} \sqsubseteq PS\text{Sum} \]
- We assume all variables are natural numbers\(^1\).

\textbf{A convenient shorthand}

- To simplify matters we defined the following shorthand
  \[ S(m) \triangleq \sum_{i=1}^m i \]
  So \( S(m) \) is the sum of numbers from 1 to \( m \).
- Note that \( i \) above is local, being bound by the \( \sum \) operator.
- It obeys the following laws:
  \[ S(0) = 0 \]
  \[ S(n) = S(n - 1) + n, \quad n > 0 \]
- Our specification becomes:
  \[ SS\text{Sum} \triangleq s, i : [s' = S(n)] \]

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

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

\[ S_{\text{Init}} \sqsubseteq P_{\text{Init}} \]

\[ S_{\text{Loop}} \sqsubseteq P_{\text{Loop}} \]

- Here, we have

\[ P_{\text{Init}} \triangleq s := 0; \ i := n \]

\[ P_{\text{Loop}} \triangleq (i > 0) \ast (s := s + i; \ i := i - 1) \]

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

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

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

A Simple example (III — \(S_{\text{Init}} \sqsubseteq P_{\text{Init}}\))

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

\[ P_{\text{Init}} = \begin{cases} \text{"defn."} \\ s := 0; \ i := n \end{cases} \]

\[ s, i : = 0, n \]

\[ s', 0 \land i' = n \land n' = n \]

\[ \text{"by design"} \]

\[ S_{\text{Init}} \]

A Simple example (IV — \(S_{\text{Loop}} \sqsubseteq P_{\text{Loop}}\))

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

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

- But what is \(P_{\text{Loop}}\) ?

- We need to find \(W\) such that

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

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

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

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

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

- It iterates until \(i = 0\), so hence \(i' = 0\).

- It sums from starting \(i\) down to 1.

- It adds on top of the starting value of \(s\).

- It does not change \(n\).
A Simple example (VI.a — simplifying \( W \))

- It is useful to do some pre-computation:

\[
\begin{align*}
i &= 0 \land W \\
&= \text{"defn. } W \text{"} \\
&\quad i = 0 \land i' = 0 \land s' = s + S(i) \land n' = n \\
&= \text{"equality substitution"} \\
&\quad i = 0 \land i' = 0 \land s' = s + S(0) \land n' = n \\
&= \text{"arithmetic"} \\
&\quad i = 0 \land i' = 0 \land s' = s \land n' = n \\
&= \text{"skip-def"} \\
&\quad i = 0 \land \text{skip}
\end{align*}
\]

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

- We need to check our \( W \) is a fixed-point:

\[
\begin{align*}
(s := s + i ; i := i - 1 ; W) \land i > 0 & \triangleright skip \\
&= \text{"\( \langle\langle\rangle\rangle \)-def, \( \langle\langle\rangle\rangle \)-def"} \\
&\quad i > 0 \land (s, i := s + i, i - 1 ; W) \lor i = 0 \land \text{skip} \\
&= \text{"previous calculations"} \\
&\quad i > 0 \land W \lor i = 0 \land W \\
&= \text{"\( \langle\langle\rangle\rangle \)-distr"} \\
&\quad (i > 0 \lor i = 0) \land W \\
&= \text{"excluded-middle", noting } i : \mathbb{N} \\
&\quad \text{true} \land W \\
&= \text{"\( \langle\langle\rangle\rangle \)-unit"} \\
&\quad W
\end{align*}
\]

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

- More pre-computation (here ignoring \( n \) and \( n' \) for brevity):

\[
\begin{align*}
i > 0 \land (s := s + i ; i := i - 1 ; W) \\
&= \text{"defn. } W, \text{"} \\
&\quad i > 0 \land (s' = s + i \land i' = i - 1 ; W) \\
&\quad i' = 0 \land s' = s + S(i) \\
&= \text{"\( \langle\langle\rangle\rangle \)-def, substitution"} \\
&\quad i > 0 \land (\exists s_m. i_m \cdot s_m = s + i \land i_m = i - 1 ; W) \\
&\quad i' = 0 \land s' = s + S(i_m) \\
&= \text{"\( \exists\)-1pt"} \\
&\quad i > 0 \land i' = 0 \land s' = s + i + S(i - 1) \\
&= \text{"arithmetic"} \\
&\quad i > 0 \land i' = 0 \land s' = s + S(i) \\
&= \text{"defn. } W \text{"} \\
&\quad i > 0 \land W
\end{align*}
\]

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

- We need to show that \( W \), our semantics for \( PLoop \) does in fact refine the loop specification.

\[
\begin{align*}
SLoop \subseteq W \\
&= \text{"defns."} \\
&\quad [i' = 0 \land s' = s + S(i) \land n' = n \Rightarrow s, i : [s' = s + S(i)]] \\
&= \text{"\( \langle\langle\rangle\rangle \)-def"} \\
&\quad [i' = 0 \land s' = s + S(i) \land n' = n \Rightarrow s' = s + S(i) \land n' = n] \\
&= \text{"prop. law } [A \land B \Rightarrow A] \text{"} \\
&\quad \text{true}
\end{align*}
\]
A Simple example (IX — final assembly)

- We have $S_{\text{Init}} = P_{\text{Init}}$ and $S_{\text{Loop}} \sqsubseteq P_{\text{Loop}}$, so we can conclude by $\langle \sqsubseteq ; \rangle$ that $S_{\text{Init}} ; S_{\text{Loop}} \sqsubseteq P_{\text{Init}} ; P_{\text{Loop}}$.

- However, we need to show that $SSum = S_{\text{Init}} ; S_{\text{Loop}}$

\[
S_{\text{Init}}; S_{\text{Loop}}
= \text{"defns."}
\]

\[
s' = 0 \land i' = n \land n' = n; s' = s + S(i) \land n' = n
= \text{"\langle ; - def\rangle, subst."}
\]

\[
\exists s_m, i_m, n_m \cdot s_m = 0 \land i_m = n \land n_m = n
\land s' = s_m + S(i_m) \land n' = n_m
= \text{"\langle \exists-1pt \rangle"}
\]

\[
s' = 0 + S(n) \land n' = n
= \text{"arithmetic, \langle frame-def\rangle."}
\]

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