
Many approaches to programming language semantics keep the programming language and the semantics language separate

\[ \text{Meaning} \left[ \text{program} \right] = \text{mathetical stuff} \]

What's inside the brackets ([ ] ) is syntax, outside is semantics

Tony Hoare, inspired by Eric Hehner, suggested this distinction was artificial and unnecessary.

Extending our Predicate Language

- We are going to extend our predicate language to include our program language:

  \[
  P, Q \in \text{Pred} \quad ::= \quad \ldots \text{skip} \quad | \quad v := E \quad | \quad P; Q \quad | \quad P \triangleleft c \triangleright Q \quad | \quad c \star P
  \]

  \text{skip} is a predicate

- Program \text{skip} leaves that state unchanged

  \[ \text{skip} \triangleq x' = x \land y' = y \land z' = z \]

  (here assuming that our only variables are \( x, y \) and \( z \) !)

- We have a little problem here — we need to know which variables are in scope.

- \( c \) denotes a \textit{condition} (boolean expression)

- In effect, we shall define our programming constructs as shorthand for equivalent predicates
Alphabets

We associate the set of variables in scope with every predicate $P$:
- this set is called the alphabet of the predicate ($\alpha P$).
- if variable $x$ is in scope, the alphabet contains $x$ and $x'$
- all variables mentioned in $P$ must be in $\alpha P$.
- we denote all the un-dashed alphabet variables by $\text{in} \alpha P$
- we denote all the dashed alphabet variables by $\text{out} \alpha P$

Alphabet example

consider the assignment  

\[ x := e \]

in a program where $x$, $y$ and $z$ are in scope.
- $e$ can only mention these three variables
- We have

\[
\begin{align*}
\alpha(x := e) &= \{x, x', y, y', z, z'\} \\
\text{in} \alpha(x := e) &= \{x, y, z\} \\
\text{out} \alpha(x := e) &= \{x', y', z'\}
\end{align*}
\]

Alphabet labelling

We can label language constructs with their alphabet ($A$, say)
- $\text{skip}_A$ and $x :=_A e$
- We can infer alphabets for other language constructs
- We often omit alphabets where obvious from context
- Frequently we single out a few variables (zero or more) and just want to refer to the rest as one — we use $\nu$ for this. so $\nu' = \nu$ means $x'_1 = x_1 \land \ldots \land x'_n = x_n$ where $x_1 \ldots x_n$ are the variables in $\nu$.
- We are talking about using “alphabetised predicates” to give meanings to programs

skip as an alphabetised predicate

\[
\begin{align*}
\text{skip}_{\text{def}} & \quad \text{skip}_A \overset{\text{def}}{=} \nu' = \nu, \quad A = \{\nu, \nu'\} \\
\text{All the variables are unchanged} \\
\text{Example, if } A &= \{i, j, k\}, \text{ then} \\
\text{skip}_A &= \left( i' = i \land j' = j \land k' = k \right)
\end{align*}
\]
\( x := e \) as an alphabetised predicate

- the after-value of \( x \) takes on the value of \( e \), evaluated in the before-state.
- Example, if \( A = \{i, j, k\} \), then
  \[
  j :=_A i + k = (i' = i \land j' = i + k \land k' = k)
  \]

; as a binary predicate operator

- \( P; Q \) is the \textbf{sequential composition} of \( P \) with \( Q \).
- How is this expressed using our predicate notation?
- We phrase its behaviour as follows:
  “\( P; Q \) maps before-state \( \nu \) to after-state \( \nu' \), when there exists a mid-state \( \nu_m \), such that \( P \) maps \( \nu \) to \( \nu_m \), and \( Q \) maps \( \nu_m \) to \( \nu' \),”
- We get the following definition:
  \[
  \langle \langle \vdash-def \rangle \rangle P; Q \equiv (\exists \nu_m \bullet P[\nu_m/\nu] \land Q[\nu_m/\nu])
  \]
- The alphabets involved must satisfy the following conditions:
  \[
  \nu \mapsto \nu', \{x, \ldots, z\}^\prime = \{x', \ldots, z'\}
  \]
  \[
  \text{in}_\nu(P; Q) = \text{in}_\nu P
  \]
  \[
  \text{out}_\nu(P; Q) = \text{out}_\nu Q
  \]

\[ \]

Sequential Composition Example

- Consider program \( t := x; x := y \), with \( t \), \( x \) and \( y \) in scope.
- We calculate:
  \[
  t := x; x := y
  = \langle \vdash-def \rangle
  t' = x \land x' = x \land y' = y; x := y
  = \langle \vdash-def \rangle
  t' = x \land x' = x \land y' = y; t' = t \land x' = y \land y' = y
  = \langle \vdash-def \rangle, \text{ noting } \nu \text{ is } t, x, y
  \]
  \[
  \exists t_m, x_m, y_m \bullet
  (t' = x \land x' = x \land y' = y)[t_m, x_m, y_m/t', x', y']
  \land (t' = t \land x' = y \land y' = y)[t_m, x_m, y_m/t, x, y]
  = \langle \vdash-def \rangle, \text{ substitution, twice } \]
  \[
  \exists t_m, x_m, y_m \bullet t_m = x \land x_m = x \land y_m = y
  \land t' = t_m \land x' = y_m \land y' = y
  \]

- We continue:
  \[
  \exists t_m, x_m, y_m \bullet t_m = x \land x_m = x \land y_m = y
  \land t' = t_m \land x' = y_m \land y' = y
  = \langle \vdash-def \rangle
  \]
  \[
  t' = x \land x' = y \land y' = y
  \]\n
- We see that the net effect is like the simultaneous assignment \( t, x := x, y \),
  \( t \) now has the value of \( x \), and \( x \) has the value of \( y \).
as a ternary predicate operator

- $P \triangleleft c \triangleright Q$, behaves like $P$, if $c$ is true, otherwise it behaves like $Q$.
- Definition:
  \[
  \langle \langle \triangleright ,-def \rangle \rangle P \triangleleft c \triangleright Q = c \land P \lor \neg c \land Q
  \]
- Alphabet constraints
  \[
  \begin{align*}
  \text{in}_\alpha (P \triangleleft c \triangleright Q) &= \alpha c = \text{in}_\alpha P = \text{in}_\alpha Q \\
  \text{out}_\alpha (P \triangleleft c \triangleright Q) &= \text{out}_\alpha P = \text{out}_\alpha Q
  \end{align*}
  \]

* as a binary predicate operator

- Program $c \odot P$ checks $c$, and if true, executes $P$, and then repeats the whole process.
- We won’t define it yet, instead we give a law that describes how a while-loop can be “unrolled” once:
  \[
  \langle \langle \triangleright ,-unroll \rangle \rangle c \odot P = (P ; c \odot P) \triangleleft c \triangleright \text{skip}
  \]
- If $c$ is False, we skip, otherwise we do $P$ followed by the whole loop ($c \odot P$) once more.
- Alphabet constraints
  \[
  \begin{align*}
  \text{in}_\alpha (c \odot P) &= \alpha c = \text{in}_\alpha P \\
  \text{out}_\alpha (c \odot P) &= \text{out}_\alpha P = (\text{in}_\alpha P)'
  \end{align*}
  \]

Example

- Program: $\langle \langle \triangleright \rangle \rangle a \geq b \triangleright (t := a ; a := b ; b := t)$, with alphabet $A = \{a, a', b, b', t, t'\}$
- Meaning:
  \[
  \begin{align*}
  \text{skip}_A &\triangleleft a \geq b \triangleright t :=_A a ; a :=_A b ; b :=_A t \\
  &= \langle \langle \triangleright ,-def \rangle \rangle a \geq b \land \text{skip}_A \\
  \lor \neg (a \geq b) \land (t :=_A a ; a :=_A b ; b :=_A t) \\
  &= \langle \langle \triangleright ,-def \rangle \rangle a \geq b \land \text{skip}_A \\
  \lor \neg (a \geq b) \land (t :=_A a ; a :=_A b ; b :=_A t) \\
  \end{align*}
  \]

Loop example

- $f := 1 ; x := n ; (x > 1) \ast (f := f \ast x ; x := x - 1)$
- Meaning:
  \[
  \begin{align*}
  f' &= f \land x' = 1 \land x = x - 1 \\
  \end{align*}
  \]

There must be a better way !! (next week)
Mini-Exercise 3

Q3.1 Expand out and simplify the predicate definition of
the following program fragment

\[ f := f \cdot x; \ x := x - 1 \]

Use the Proof-Section format described in class.
(due in next Thursday, 12noon, in class)
Class 8

all the other variables …

- Consider computing the semantics of

\[
x := x + y; \quad y := x - y; \quad x := x - y
\]

in a scope with variables \(s, t, u, v, w, x, y, z\).

- The program only mentions \(x\) and \(y\), but we have to carry equalities around for \(s, t, u, w\) and \(z\).

- We can tidy-up using the 1-pt law, but this is tedious and error prone.

- Can we show that the above program swaps the values of \(x\) and \(y\), without mentioning the other variables?

- What we need are laws that work at the programming language level (a.k.a. “Laws of Programming”)

Laws of Imperative Programming

- When we give a law stating that programs \(p_1\) and \(p_2\) are the same:

\[
p_1 = p_2
\]

we are stating that both programs have the same behaviour, i.e. that they change the state in the same way.

- Given that our programs are predicates, we can simply use predicate calculus to prove such laws as theorems in the usual way.

- The laws typically show how various combinations of language constructs are related.

(Some) Laws of Programming

- Here \(P\), \(Q\) and \(R\) stand for arbitrary programs, \(A\) for an arbitrary alphabet, \(x\), \(y\) and \(z\) for arbitrary variables, and \(c, e\) and \(f\) for arbitrary expressions

\[
\langle \text{skip-alt} \rangle \quad \text{skip}_A = \ x :=_A x, \quad x \in A
\]
\[
\langle \text{skip-unit} \rangle \quad \text{skip} \ ; \ P = P
\]
\[
\langle \text{-skip-unit} \rangle \quad P \ ; \ \text{skip} = P
\]
\[
\langle \text{-assoc} \rangle \quad P \ ; (Q \ ; R) = (P \ ; Q) \ ; R
\]
\[
\langle \text{:=seq} \rangle \quad x := e; \quad x := f = x := f[e/x]
\]
\[
\langle \text{:=swap} \rangle \quad x := e; \quad y := f = y := f[e/x]; \quad x := e, \quad y \not\in e
\]
\[
\langle \text{<|>|true} \rangle \quad P <\text{True} > Q = P
\]
\[
\langle \text{<|>|false} \rangle \quad P <\text{False} > Q = Q
\]
\[
\langle \text{<|>|seq} \rangle \quad (P c > Q); \ R = (P ; R) c > (Q ; R)
\]
Proof of \langle skip \rangle

- **Goal**: \( \text{skip} \) = \( x := A \) \( x \)
- **Strategy**: reduce rhs to lhs
- **Proof**:
  \[
  x := A \]
  \[
  = \quad \langle := \text{-def} \rangle, \ A = \{ x, x', \nu, \nu' \} \]
  \[
  x' = x \land \nu' = \nu \]
  \[
  = \quad \langle \text{skip} \text{-def} \rangle, \ A = \{ x, x', \nu, \nu' \} \]
  \[
  \text{skip}_A \]

Proof of \langle skip; \rangle

- **Goal**: \( \text{skip} \); \( P \) = \( P \)
- **Strategy**: reduce lhs to rhs
- **Proof**:
  \[
  \text{skip} ; P
  \]
  \[
  = \quad \langle \text{skip} \text{-def} \rangle \]
  \[
  \nu' = \nu \land P
  \]
  \[
  = \quad \langle ; \text{-def} \rangle \]
  \[
  \exists v_m \bullet v_m = \nu \land P[v_m/\nu]
  \]
  \[
  = \quad \langle \exists \text{-1pt} \rangle \]
  \[
  (P[v_m/\nu])[\nu/\nu_m]
  \]
  \[
  = \quad \text{substitution} \ 'inverse' \]
  \[
  P \]

Laws of Substitution

- We have used some “laws of substitution” in our proofs.
- These too have a rigorous basis.
- First, we promote substitution to be part of our predicate language:
  \[
  \text{Pred} \ ::= \ldots | P[e_1, \ldots, e_n/v_1, \ldots, v_n]
  \]
- Next we define its effect on predicates.

Defining Single Substitution

\[
\begin{align*}
  k[e/x] & \equiv k \\
  v[e/x] & \equiv v, \ v \neq x \\
  x[e/x] & \equiv e \\
  (e_1 + e_2)[e/x] & \equiv e_1[e/x] + e_2[e/x] \\
  (\neg P)[e/x] & \equiv \neg P[e/x] \\
  (P \land Q)[e/x] & \equiv P[e/x] \land Q[e/x] \\
  (\forall x \bullet P)[e/x] & \equiv \forall x \bullet P \\
  (\forall v \bullet P)[e/x] & \equiv \forall v \bullet P[e/x], \ v \neq x, \ v \not\in e \\
  (\forall v \bullet P)[e/x] & \equiv \forall w \bullet (P[w/v])[e/x], \ v \neq x, \ v \not\in e, w \not\in P, e, x
  \end{align*}
\]

The constructs not mentioned above follow the same pattern.
We shall now look at the last four lines in more detail.
Substitution and Quantifiers (I)

\((\forall x \cdot P)[e/x] \equiv \forall x \cdot P\)

The simplest case: \(x\) is simply not free in \(\forall x \cdot P\), so nothing changes.

Substitution and Quantifiers (II)

\((\forall v \cdot P)[e/x] \equiv \forall v \cdot P[e/x], \; v \neq x, \; v \notin e\)

- We are not substituting for \(v\), and \(v\) does occur in \(e\), so there is no possibility of name capture.
- We simply recurse to the body predicate \(P\).

Substitution and Quantifiers (III)

\((\forall v \cdot P)[e/x] \equiv \forall w \cdot (P[w/v])[e/x], \; v \neq x, \; v \epsilon e, \; w \notin P, \; e, \; x\)

- The tricky case, where \(v\) occurs in \(e\).
- To avoid name capture of \(v\) in \(e\):
  - Pick a fresh variable \(w\) — i.e. one not currently in use.
  - \(\alpha\)-rename the bound and binding occurrences of \(v\) to \(w\).
  - Then recurse into the now \(\alpha\)-renamed body \(P\).

Laws of single substitution

- We can identify a number of useful laws
  - subst-inv: \(P[x/y][y/x] = P, \; x \notin P\)
  - subst-comp: \(P[e/x][f/x] = P[e[f/x]/x]\)
  - subst-swap: \(P[e/x][f/y] = P[f/y][e/x], \; x \notin f, \; y \notin e\)

- We shall not prove these laws at this point
  - to do so requires induction,
  - over the grammar (?!) of our predicate language.
### Substitution Law examples

- **subst-inv**: $P[x/y][y/x] = P, \ x \not\in P$
- $(x + y)[z/x][x/z] = (z + y)[x/z] = x + y$
- The side-condition matters:
  - $(x + y)[y/x][x/y] = (y + y)[x/y] = x + x$
  - $(x + y)[x/y][y/x] = (x + x)[y/x] = y + y$
- **subst-comp**: $P[e/x][f/x] = P[e[f/x]/x]$
  - $x[x + y/x][z/x] = (x + y)[z/x] = z + y$
  - $x[(x + y)[z/x]/x] = x[z + y/x] = z + y$

### Simultaneous Substitution

- Simultaneous substitution does several replacements at once:
  - $(x + y)[y^2, k + x/x, y] = y^2 + k + x$
- In general, it is not the same as doing each replacement one at a time:
  - $(x + y)[y^2/x][k + x/y] = (y^2 + y)[k + x/y] = (k + x)^2 + k + x$
- When the first substitution does not introduce the “target” of the second, then we can do them one at a time.

### Laws of Simultaneous Substitution

- Order of substitutions does not matter:
  - **subst-comm**: $P[e, f/x, y] = P[f, e/y, x]$
- We can merge in later substitutions if they don’t act on earlier replacements:
  - **subst-seq**: $P[e/x][f/y] = P[e, f/x, y], \ y \not\in e$
- The above law generalises to many substitutions:
  - $P[e_1, \ldots, e_n/x_1, \ldots, x_n][f/y] = P[e_1, \ldots, e_n, f/x_1, \ldots, x_n, y]$
  - (provided $y \not\in e_1, \ldots, e_n$)

### Proof of **⟨⟨; -assoc⟩⟩**

- **Goal**: $P; (Q; R) = (P; Q); R$
- **Strategy**: reduce both lhs and rhs to same predicate
- **Proof**: hold on tight!
Proof «;−assoc» (lhs)

\[ P : (Q ; R) = (\exists \nu_m \cdot Q[\nu_m/\nu] \land R[\nu_m, \nu]) \]

Proof «;−assoc» (rhs)

\[ (P ; Q) ; R = (\exists \omega, \omega' \cdot \omega Q \cup \omega R) \]

Proof using Laws (example)

● Goal:

\[ (f := 1; x := n; f := f \cdot x; x := x - 1) \]

\[ = \]

\[ (f := n; x := n - 1) \]

● Strategy: reduce lhs to rhs

● Proof:

\[ f := 1; x := n; f := f \cdot x; x := x - 1 \]

\[ = \]

\[ "\cdot \text{swap}, f \notin n" \]

\[ f := 1; f := f \cdot n; x := n; x := x - 1 \]

\[ = \]

\[ "\cdot \text{seq}" \]

\[ f := 1 \cdot n; x := n - 1 \]

\[ = \]

\[ "\text{arithmetic}" \]

\[ f := n; x := n - 1 \]
Subtleties with Substitution (I)

Consider the following proof of \(\langle \langle \text{:=}-\text{seq} \rangle \rangle\) reducing LHS to RHS:

\[
\begin{align*}
\text{x} &\overset{\text{:=}}{=} e; \text{x} \overset{\text{:=}}{=} f \\
&= \text{" \langle \langle \text{:=}-\text{def} \rangle \rangle \" } A = \{x, x', \nu, \nu'\} \\
x' &\overset{\text{:=}}{=} e \land \nu' = \nu; \text{x}' = f \land \nu' = \nu \\
&= \text{" \langle \langle \text{:=}-\text{def} \rangle \rangle \" } A = \{x, x', \nu, \nu'\} \\
\exists x_m, \nu_m \bullet (x' = e \land \nu' = \nu)[x_m, \nu_m/x', \nu'] \\
&\land (x' = f \land \nu' = \nu)[x_m, \nu_m/x, \nu] \\
&= \text{" substitution, noting that e has no dashed vars \" } \\
\exists x_m, \nu_m \bullet x_m = e \land \nu_m = \nu \land x' = f[x_m, \nu_m/x, \nu] \land \nu' = \nu_m \\
&= \text{" \langle \langle \exists-1\text{pt} \rangle \rangle \" } x_m = e, \nu_m \not\in e, \nu \\
x' &\overset{\text{:=}}{=} f[e, \nu/x, \nu] \land \nu' = \nu \\
&= \text{" ignore [\nu/\nu], \langle \langle \text{:=}-\text{def} \rangle \rangle \" } \\
x &\overset{\text{:=}}{=} f[e/x]
\]

Subtleties with Substitution (II)

- We might have been tempted to apply \(\langle \langle \text{:=}-\text{def} \rangle \rangle\) first, “do” the substitution, and then use \(\langle \langle \text{:=}-\text{def} \rangle \rangle\):

\[
\begin{align*}
\text{x} &\overset{\text{:=}}{=} e; \text{x} \overset{\text{:=}}{=} f \\
&= \text{" \langle \langle \text{:=}-\text{def} \rangle \rangle \" } \\
\exists x_m, \nu_m \bullet (x' = e)[x_m, \nu_m/x', \nu'] \\
&\land (x' = f)[x_m, \nu_m/x, \nu] \\
&= \text{" doing substitution into assignment \" } \\
\exists x_m, \nu_m \bullet x \overset{\text{:=}}{=} e \\
&\land x_m = f[x_m, \nu_m/x, \nu] \\
&= \text{" \langle \langle \text{:=}-\text{def} \rangle \rangle, twice \" } \\
\exists x_m, \nu_m \bullet x' = e \land \nu' = \nu \\
&\land x_m' = f[x_m, \nu_m/x, \nu] \land \nu' = \nu
\]

- What is \(x_m'\)? We seem to have a problem!

The Problem

- Sequential composition is designed to work with predicates using \(x, \nu\) for before-variables, and \(x'\) and \(\nu'\) for after variables.
- In \(x := e\), the variable \(x\) stands for the program variable, and not its initial value.
- We cannot do substitutions safely until we have expanded its definition.
- In fact, we cannot determine what its free variables are until it has been expanded.

- What is \(x_m'\)? We seem to have a problem!
Non-substitutable Predicates

- Some of our new predicate (programming) constructs are non-substitutable (n.s.).
- Substitution can only be applied to these once their definitions have been expanded.
- We have assignment as one example, but there are others:
  
  \[
  \text{skip} \quad P; \quad Q \quad c \times P
  \]

- Of the new constructs so far, only conditional is substitutable:

  \[
  (P \land c \Rightarrow Q)[e/x] = P[e/x] \land c[e/x] \Rightarrow Q[e/x]
  \]

Simultaneous Assignment

- We introduce a further extension to predicate syntax, simultaneous assignment:

  \[
  \text{Pred} ::= \ldots
  \| \quad x_1, \ldots, x_n := e_1, \ldots, e_n
  \]

- We introduce shorthands: \( \vec{x} \) and \( \vec{e} \) for \( x_1, \ldots, x_n \) and \( e_1, \ldots, e_n \) resp.

- Its meaning is that the expressions \( e_1 \) through \( e_n \) are evaluated, and then all the \( x_i \) are updated simultaneously:

  \[
  \langle \langle \text{sim-:=def} \rangle \rangle \quad \vec{x} := \vec{e} \quad \Leftrightarrow \quad x_1' = e_1 \land \ldots \land x_n' = e_n \land \nu' = \nu,
  \]

  \[
  A = \{ x_1, \ldots, x_n, x_1', \ldots, x_n', \nu, \nu' \}
  \]

Swapping “trick” revisited

- Re-consider computing the semantics of

  \[
  x := x + y; \quad y := x - y; \quad x := x - y
  \]

  in a scope with variables \( s, t, u, v, w, x, y, z \)

- We’d like to use the laws of programming, so we can ignore \( s, t, u, v, w \) and \( z \).

- We can’t use \( \langle \langle \text{:=swap} \rangle \rangle \), because the side-condition does not hold.

- We can’t use \( \langle \langle \text{:=seq} \rangle \rangle \), because we don’t have two assignments to the same variable one after the other.

- In fact it’s not clear what laws would work: after the first two assignments we have \( x' = x + y \land y' = x \land \nu' = \nu \) whilst at the end we get \( x' = y \land y' = x \land \nu' = \nu \).

Laws of Simultaneous Assignment

- A single assignment to \( y \) can always be merged with a preceding simultaneous assignment to \( \vec{x} \), provided \( y \notin \vec{x} \):

  \[
  \langle \langle \text{sim-:=merge} \rangle \rangle \quad \vec{x} := \vec{e}; \quad y := f \quad = \quad \vec{x}, y := \vec{e}, \{ f[e/x] \}
  \]

- Proof, reducing lhs to rhs:

  \[
  \vec{x} := \vec{e}; \quad y := f
  \]

  \[
  = \quad \langle \langle \text{sim-:=def} \rangle \rangle, \langle \langle \text{:=def} \rangle \rangle
  \]

  \[
  \vec{x}' = \vec{e} \land y' = y \land \nu' = \nu; \quad \vec{x}' = \vec{x} \land y' = f \land \nu' = \nu
  \]

  \[
  = \quad \langle \langle \text{:=def} \rangle \rangle, \text{and substitute } \]

  \[
  \exists x_m, y_m, \nu_m. \quad x_m = \vec{e} \land y_m = y \land \nu_m = \nu
  \]

  \[
  \land \vec{x}' = x_m \land y' = f[x_m, y_m, \nu_m, \vec{x}, y, \nu] \land \nu' = \nu_m
  \]

  \[
  = \quad \langle \langle \text{:=1pt} \rangle \rangle
  \]

  \[
  \vec{x}' = \vec{e} \land y' = f[\vec{e}, y, \nu/\vec{x}, y, \nu] \land \nu' = \nu
  \]

  \[
  = \quad \langle \langle \text{sim-:=def} \rangle \rangle, \text{ignoring } [y, \nu/\nu', \nu] \]

  \[
  \vec{x}, y := \vec{e}, f[\vec{e}/\vec{x}]
  \]
Laws of Simultaneous Assignment (II)

- If \( y \in \vec{x} \), the sequencing law has to be slightly different
  \[
  \langle \text{sim-}:=\text{-seq} \rangle \quad \vec{x}, y := \vec{\epsilon}, f; \ y := g = \vec{x}, y := \vec{\epsilon}, g|\vec{\epsilon}, f/\vec{x}, y
  \]

- Proof, reducing lhs to rhs:
  \[
  \begin{align*}
  \vec{x}, y := \vec{\epsilon}, f; \ y := g \\
  &= \langle \text{sim-}:=\text{-def} \rangle, \langle :=\text{-def} \rangle \\
  \vec{x}^f = \vec{\epsilon} \land y' = f \land \nu' = \nu; \ \vec{x}^f = \vec{x} \land y' = g \land \nu' = \nu \\
  &= \langle :=\text{-def} \rangle, \text{and substitute } \nonumber \\
  \exists x_m, y_m, \nu_m \bullet \vec{x}^f = \vec{\epsilon} \land y_m = f \land \nu_m = \nu \\
  \land \vec{x}' = \vec{x}_m \land y' = g[\vec{x}_m, y_m, \nu_m/\vec{x}, y, \nu] \land \nu' = \nu_m \\
  &= \langle \exists\text{-lpt} \rangle \\
  \vec{x}^f = \vec{\epsilon} \land y' = g|\vec{\epsilon}, f/\vec{x}, y, \nu \land \nu' = \nu \\
  &= \langle \text{sim-}:=\text{-def}, \text{ignoring } \nu/\nu \rangle \\
  \vec{x}, y := \vec{\epsilon}, g|\vec{\epsilon}, f/\vec{x}, y
  \end{align*}
  \]

Swapping Trick Proof

- Goal:
  \[
  x := x + y; \ y := x - y; \ x := x - y = x, y := y, x
  \]
- Proof, reducing lhs to rhs:
  \[
  \begin{align*}
  x := x + y; \ y := x - y; \ x := x - y \\
  &= \langle \text{sim-}:=\text{-seq} \rangle \\
  x, y := x + y, (x - y)[x + y/x]; \ x := x - y \\
  &= \langle :=\text{-seq} \rangle \\
  x, y := (x - y)[x + y, (x + y) - y/x, y], (x + y) - y \\
  &= \langle :=\text{-def} \rangle, \text{and substitute } \nonumber \\
  x, y := x + y - ((x + y) - y), (x + y) - y \\
  &= \langle \text{arith} \rangle \\
  x, y := y, x
  \end{align*}
  \]

Messing with notation

- Is simultaneous assignment a programming language construct?
- Depends on the language:
  - in languages like C, Java, it is not allowed
  - in Handel-C it is allowed, as it targets hardware and so we have real parallelism
- It does not matter!
  - It is a predicate with a sensible meaning
  - It is convenient for certain proofs
  - It can describe outcomes concisely that are not possible using only single assignments, e.g. \( x, y := y, x \).
- Not all predicate language extensions have to be “code”.

Programs as Predicates

- If programs are predicates, then we can join them up using predicate notation.
  - e.g \( \text{prog}_1 \land \text{prog}_2 \)
  - e.g \( \text{prog}_1 \lor \text{prog}_2 \)
  - e.g \( \text{prog}_1 \Rightarrow \text{prog}_2 \)
- We can also mix them with non-program predicates
  - e.g \( \text{prog} \land \text{pred} \)
  - e.g \( \text{prog} \lor \text{pred} \)
  - e.g \( \text{prog} \Rightarrow \text{pred} \)
- Do these make sense? If so, how?
- Are any of these useful?
Consider the following:

\( (x := 2) \land (x := 3) \)
\( (x := 2) \land (y := 3) \)
\( (x := 2) \land x = 2 \)
\( (x := 2) \land y = 3 \)

What behaviour do these describe?

Not unexpectedly, we get \textbf{false}
we cannot assign 2 and 3 to \( x \) (at the same time)

We get a predicate stating that the assignment occurred, in
a starting state where \( x \) had value 2.

It is the same as \textbf{Skip} \land x = 2
Examining \((x := 2) \land y = 3\)

- \[(x := 2) \land y = 3 = " \text{\texttt{:=def}} \]

- \(x' = 2 \land v' = v \land y = 3\)

- We get a predicate stating that the assignment occurred, in a starting state where \(y\) had value 3.

Programs and Conjunction—Comment

- Conjoining two programs \((\text{prog}_1 \land \text{prog}_2)\) easily leads to contradiction
- Predicate \(\text{prog} \land \bar{x} = \bar{e}\) describes a run of \(\text{prog}\) that started in a state where variables \(\bar{x}\) had values \(\bar{e}\).
- Remember, \(x := e\) means \(x\) is changed, and that all other \textit{variables are left unchanged}.

Programs and Disjunction

- Consider the following:
  - \((x := 2) \lor (x := 3)\)
  - \((x := 2) \lor (y := 3)\)
  - \((x := 2) \lor x = 2\)

Examining \((x := 2) \lor (x := 3)\)

- \[(x := 2) \lor (x := 3) = " \text{\texttt{:=def}}, \text{\texttt{twice}} \]

- \(x' = 2 \lor x' = 3 \land v' = v\)

- Variable \(x\) ends up having either value 2 or 3, and all other variables are unchanged.
- The choice between 2 or 3 is arbitrary — nothing here states how that choice is made.
Examining \((x := 2) \lor (y := 3)\)

- \((x := 2) \lor (y := 3)\)
  = "\(\def \times\), twice "
  \(x' = 2 \land y' = y \land \nu' = \nu \lor x' = x \land y' = 3 \land \nu' = \nu\)
  = "\(\land \lor \)-distr "
  \((x' = 2 \land y' = y \lor x' = x \land y' = 3) \land \nu' = \nu\)

- Either \(x\) ends up having value 2, or \(y\) ends up equal to 3, and all other variables are unchanged.
- The choice between changing \(x\) or \(y\) is arbitrary — nothing here states how that choice is made.

Examining \((x := 2) \lor x = 2\)

- \((x := 2) \lor x = 2\)
  = "\(\def \times\), twice "
  \(x' = 2 \land \nu' = \nu \lor x = 2\)

- Either \(x\) becomes 2, or we started with \(x\) equal to 2 and then \textit{anything} could have happened!
  - if \(x\) is not equal to 2 to start, then it equals 2 at the end and no other variable changes
  - if \(x\) equals 2 at the beginning, then the final values of \textit{all} variables are arbitrary.

Programs and Disjunction—Comment

- Disjoining two programs \((prog_1 \lor prog_2)\) denotes an arbitrary choice between the two behaviours.
- Predicate
  \(prog \lor \bar{x} = \bar{e}\)

  tells us that we get:
  - the behaviour of \(prog\), if initial values of \(\bar{x}\) are \textit{not all} equal to \(\bar{e}\);
  - arbitrary behaviour, if they are all equal.

Programs and Negation

- How do we “not” assign something?
  \(\neg (x := e)\)

- Let’s calculate:
  \(\neg (x := e)\)
  = "\(\def \times\) "
  \(\neg (x' = e \land \nu' = \nu)\)
  = "\(\def \times\lor \def \times\lor \) "
  \(x' \neq e \lor \nu' \neq \nu\)

- So, \(\neg (x := e)\) is any situation where
  - either \(x\) does not end up with value \(e\)
  - or some other variable gets changed
Consider the following:

\[(x := 2) \Rightarrow (x := 3)\]
\[(x := 2) \Rightarrow (y := 3)\]
\[(x := 2) \Rightarrow x' = 2\]
\[(x := 2) \Rightarrow x' \in \{1, \ldots, 10\}\]

This can only be true if the assignment \(x := 2\) does not happen.

This is always true: if we assign 2 to \(x\), then the final value of \(x\) is 2.
Examining $(x := 2) \Rightarrow x' \in \{1, \ldots, 10\}$

- $(x := 2) \Rightarrow x' \in \{1, \ldots, 10\)$
  - "\texttt{\langle \langle :=-\text{def} \rangle \rangle}"
  - $x' = 2 \land \nu' = \nu \Rightarrow x' \in \{1, \ldots, 10\}$
  - "\texttt{\langle \Rightarrow-\text{def}, \text{deMorgan} \rangle}"
  - $x' \neq 2 \lor \nu' \neq \nu \lor x' \in \{1, \ldots, 10\}$
  - "\texttt{\langle \text{excluded-middle} \rangle}"
  - \texttt{true} \lor \nu' \neq \nu \lor x' \in \{1, \ldots, 10\}
  - "\texttt{\langle \lor-\text{zero} \rangle}"
  - \texttt{true}

- This is always true: if we assign 2 to $x$, then the final value of $x$ is between 1 and 10.

Programs and Implication—Comment

- Predicate $\text{prog}_1 \Rightarrow \text{prog}_2$ is true if
  - the before-after relationship described by $\text{prog}_1$ does not hold, or ...
  - $\text{prog}_1$ holds, and the behaviour described by $\text{prog}_2$ somehow includes the behaviour of $\text{prog}_1$

- Predicate $\text{prog} \Rightarrow \text{pred}$ is true if
  - the before-after relationship described by $\text{prog}$ does not hold, or ...
  - $\text{prog}$ holds, and the situation described by $\text{pred}$ is covered by the behaviour of $\text{prog}$