Semantics Course Notes

Herman Geuvers (based on old notes by Jozef Hooman)

September 2007

1 Functional Language

September 4, 2007

Consider the syntax of a simple functional language. To keep it simple, only functions with one parameter, no mutually recursive functions. Main aim is to get some feeling for important questions in this field.

Let *IDF* be a nonempty set of function identifiers (i.e. names of functions) with typical element f, *VAR* be a nonempty set of variables (parameters of functions) with typical element x, *CONST* be a domain of constants with typical element C. The syntax of our programming language is given in table 1, with $x \in VAR$, $C \in CONST$, and $f \in IDF$.

Table 1: Syntax of a Simple Functional Language

Boolean Expression	$B ::=$	$true \mid E_1 = E_2 \mid E_1 < E_2 \mid \neg B \mid B_1 \vee B_2$
Value Expression	$E ::=$	$C \mid x \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid$ $f(E) \mid$ **if** B **then** E **fi** $\mid$ **if** B **then** E_1 **else** E_2 **fi** $\mid$ E **where** $f(x) = E_0$

Examples:

1. $fac(3)$ **where** $fac(n) =$ **if** $n = 0$ **then** 1 **else** $n \times fac(n-1)$ **fi**
   For future use, define $E_{fac} \equiv$ **if** $n = 0$ **then** 1 **else** $n \times fac(n-1)$ **fi**.
   Then $fac(3)$ **where** $fac(n) = E_{fac}$.

2. $fib(5)$ **where** $fib(x) =$ **if** $x = 0$ **then** 0
   else if $x = 1$ **then** 1
   else if $x > 1$ **then** $fib(x-1) + fib(x-2)$
   else 0 **fi fi fi**

   Henceforth we will use
   $E_{fib} \equiv$ **if** $x = 0$ **then** 0
   else if $x = 1$ **then** 1
   else if $x > 1$ **then** $fib(x-1) + fib(x-2)$
   else 0 **fi fi fi** .

   Thus $fib(5)$ **where** $fib(x) = E_{fib}$

   Note the use of recursion.

3. $f(7)$ **where** $f(x) = 2 \times x + 9$

   And a number of strange examples:

4. $f(7)$ **where** $f(x) = f(x)$

5. $f(7)$ **where** $f(x) = f(x) + 1$

6. $f(7)$ **where** $f(x) = f(x-1) + f(x+1)$

7. $f(0)$ **where** $f(x) =$ **if** $x > 0$ **then** $x - 1$ **fi**

8. $f(7)$ **where** $f(x) = 0$ **where** $f(x) = 1$

Operational idea is to do some rewriting:
$fac(3)$ **where** $fac(n) = E_{fac} \rightarrow$
$\underline{E_{fac}[3/n]}$ **where** $fac(n) = E_{fac} \rightarrow$

if $\underline{3 = 0}$ **then** 1 **else** $3 \times fac(3-1)$ **fi** **where** $fac(n) = E_{fac} \rightarrow$
if $false$ **then** 1 **else** $3 \times fac(3-1)$ **fi** **where** $fac(n) = E_{fac} \rightarrow$
$3 \times fac(\underline{3-1})$ **where** $fac(n) = E_{fac} \rightarrow$
$3 \times fac(2)$ **where** $fac(n) = E_{fac} \rightarrow$
$3 \times E_{fac}[2/n]$ **where** $fac(n) = E_{fac} \rightarrow$
etc ...
How can the steps be justified:
Rewrite expressions with natural numbers and rewite boolean expressions.

Consider the strange examples:

1. $f(7)$ **where** $f(x) = f(x)$; leads to same expression repeatedly.
2. $f(7)$ **where** $f(x) = f(x) + 1$; leads to $f(7) + n$
3. $f(7)$ **where** $f(x) = f(x-1) + f(x+1)$; explosion of terms ...
4. $f(0)$ **where** $f(x) =$ **if** $x > 0$ **then** $x - 1$ **fi**; leads to
   if $0 > 0$ **then** $x - 1$ **fi** **where** $f(x) =$ **if** $x > 0$ **then** $x - 1$ **fi**, then
   if $false$ **then** $x - 1$ **fi** **where** $f(x) =$ **if** $x > 0$ **then** $x - 1$ **fi** and
   no further step can be made (blocking, but it is a normal form ...).

This operational semantics seems clear, but these examples already indicate that there are several cases and there are some questions. Here are the main cases and further examples.

- **non-termination**: rewriting need not always terminate
- **blocking**: rewriting may stop without leading to a value
- **non-determinism**: choices in rewriting; do they lead to same result?

2 Denotational Semantics Functional Language **September 4, 2007**

The λ-calculus is very fundamental for functional programming, but not always clearly gives the meaning of constructs (e.g. encoding of if-then-else, if-then, addition, etc is complicated). Especially fixed point constructor is not so clear; does it gives right semantics?
Note that λ-calculus is very operational; one has to rewrite (execute) to get meaning of construct.
Alternative presented here is denotational semantics, which is a general technique and also applicable to other languages (as we show later); idea is to give mathematical definition of language constructs.

Recall the syntax of a simple functional language. Basic idea is to map syntactic constructs of the language to a mathematical domain. For boolean expressions B we will define a function $Bval$ yielding the truth value of B. Similarly, $Eval(E)$ gives the value of expression E. Since these values might depend on the definitions of certain functions, we add a so-called environment ρ which gives the definitions of functions. Thus we define $Bval(B)\rho$ and $Eval(E)\rho$, for the moment in, resp., $\{T, F\}$ and D, for some data domain D.

Now consider $Eval(\textbf{if } B \textbf{ then } E \textbf{ fi})\rho$; clearly we should obtain $Eval(E)\rho$ if $Bval(B)\rho = T$. But what if $Bval(B)\rho = F$? For these cases we introduce a special value $\perp$ (undefined, no information). Note that we also have to think about the case that $Bval(B)\rho = \perp$.

Assume given a data domain D. Let $\perp$ represent undefinedness. Then functions are denoted as mappings from $D \cup \{\perp\}$ to $D \cup \{\perp\}$. Let $(V \to V)$ represent the set of functions from V to V. Declarations of functions are represented by an *environment*, which maps function identifiers to mathematical functions. Let
$Env = (IDF \to (D \cup \{\perp\} \to D \cup \{\perp\}))$
and $\rho \in Env$ a typical element.

Example:
$\rho(inc) = \{(x, y) | (x \neq \perp \to y = x + 1) \wedge (x = \perp \to \perp)\}$
or, shortly, $\rho(inc) = \lambda x.x +_3 1$, using $+_3$ to denote the extension of $+$ to $D \cup \{\perp\}$. For instance, assume $\perp +_3 1 = \perp$.
Then $\rho(inc)(3) = x +_3 1 = 4$

The *variant* of an environment ρ with respect to a function name f and a function F, denoted by $(\rho : f \mapsto F)$, is defined as follows: $(\rho : f \mapsto F)(g) = \begin{cases} F & \text{if } g \equiv f \\ \rho(g) & \text{if } g \not\equiv f \end{cases}$

Finally, note that we cannot assign a value to x. The idea is that variables from VAR should not occur free in an expression, they are only used to define functions. We say that a term is *closed* if every variable from VAR in this terms occurs *bound*, that is, occurs inside E_0 of a subterm E_1 **where** $f(x) = E_0$.
To be precise, $Eval(E)\rho \in D \cup \{\perp\}$ for any expression E which is *closed*, i.e. for which $FV(E) = \emptyset$.

2.1 Semantic function

We define a function $Eval$ which assigns to a closed expression E and an environment ρ a value in $D \cup \{\perp\}$; $Eval(E)\rho \in D \cup \{\perp\}$. To define the value of a closed boolean expression, also $Bval(B)\rho \in \{T, F, \perp\}$ is defined.
(Note that T is a mathematical representation here, different from the syntactic form.)

For the moment, we assume given some boolean connectives NOT_3 and OR_3 on $\{T, F, \bot\}$ and operations $=_3$, $<_3$, $+_3$, $-_3$, $\times_3$ in $D \cup \{\bot\}$. These will be defined later.

- $Bval(true)\rho = T$ (NOTE difference syntactic and semantic domain)
- $Bval(E_1 = E_2)\rho = (Eval(E_1)\rho =_3 Eval(E_2)\rho)$
- $Bval(E_1 < E_2)\rho = (Eval(E_1)\rho <_3 Eval(E_2)\rho)$
- $Bval(\neg B)\rho = \ NOT_3\ Bval(B)\rho$
- $Bval(B_1 \vee B_2)\rho = Bval(B_1)\rho\ OR_3\ Bval(B_2)\rho$
- $Eval(C)\rho = \text{rep}C$ (NOTE: this is the 'representation' of C in D.)
- $Eval(x)\rho = x$ (so for simplicity not in $D \cup \{\bot\}$)
- $Eval(E_1 + E_2)\rho = Eval(E_1)\rho +_3 Eval(E_2)\rho$
- $Eval(E_1 - E_2)\rho = Eval(E_1)\rho -_3 Eval(E_2)\rho$
- $Eval(E_1 \times E_2)\rho = Eval(E_1)\rho \times_3 Eval(E_2)\rho$
- $Eval(f(E))\rho = \rho(f)(Eval(E)\rho)$
- Let $ite(b, e_1, e_2) = \begin{cases} \bot & \text{if } b \equiv \bot \\ e_1 & \text{if } b \equiv T \\ e_2 & \text{if } b \equiv F \end{cases}$
  Then define
  $Eval(\textbf{if } B \textbf{ then } E \textbf{ fi})\rho = ite(Bval(B)\rho, Eval(E)\rho, \bot)$
  thus $Eval(\textbf{if } B \textbf{ then } E \textbf{ fi})\rho = \begin{cases} Eval(E)\rho & \text{if } Bval(B)\rho = T \\ \bot & \text{otherwise} \end{cases}$
- $Eval(\textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2 \textbf{ fi})\rho$:
  what if $Bval(B)\rho = \bot$, then $\bot$ seems reasonable.
  what if $Eval(E_1)\rho = \bot$ or $Eval(E_2)\rho = \bot$; then we could also give $\bot$, but this means that both parts have to be evaluated ...
  Here we are a bit more liberal (allowing parallel evaluation!!!!), although most functional langauages are implemented sequentially ...
  $Eval(\textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2 \textbf{ fi})\rho = ite(Bval(B)\rho, Eval(E_1)\rho, Eval(E_2)\rho)$
  that is, $Eval(\textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2 \textbf{ fi})\rho = \begin{cases} Eval(E_1)\rho & \text{if } Bval(B)\rho = T \\ Eval(E_2)\rho & \text{if } Bval(B)\rho = F \\ \bot & \text{if } Bval(B)\rho = \bot \end{cases}$
- $Eval(E \textbf{ where } f(x) = E_0)\rho = Eval(E)(\rho : f \mapsto FUN)$, for some function FUN.
  If E_0 does not contain f, i.e. definition is not recursive, then take for instance, a *strict* function $FUN \equiv \lambda x.Eval(E_0)\rho$.

  Example
  $Eval(inc(3) \textbf{ where } inc(x) = x + 1)\rho =$
  $Eval(inc(3))(\rho : inc \mapsto \lambda x.x +_3 1) =$
  $(\rho : inc \mapsto \lambda x.x +_3 1)(inc)\ (3) =$
  $(\lambda x.x +_3 1)\ 3 = 3 + 1 = 4$ (als $3 \neq \bot$ ....)

  Otherwise we would like to have $FUN = fix(\lambda G.\lambda x.Eval(E_0)(\rho : f \mapsto G))$. But then the question is whether the fixed point exists and how to define it.
  This will be formalized below.

2.2 Three-valued connectives and operators

First we consider the problem of dealing with $\perp$.
For negation it seems clear:

p	$NOT_3\ p$
T	F
F	T
$\perp$	$\perp$

Let's consider the truth table of OR_3 (slowly try to fill it: QUESTION)

OR_3	T	F	$\perp$
T	T	T	?
F	T	F	$\perp$?
$\perp$	?	$\perp$?	$\perp$

Main two possibilities:

$x\ OR_3\ y = \perp$ if $x = \perp$ or $y = \perp$

OR COMPLETELY: $X\ OR_3^s\ Y = \begin{cases} T & \text{if } X = T \text{ and } Y \neq \perp, \text{ or if } Y = T \text{ and } X \neq \perp \\ F & \text{if } X = F \text{ and } Y = F \\ \perp & \text{if } X = \perp \text{ or } Y = \perp \end{cases}$

This is called a *strict* interpretation: $\perp$ is one of the arguments is $\perp$. It corresponds to call-by-value: first evaluate arguments.

The *strongest* monotonic extension of the classical (two-valued) operator

OR_3	T	F	$\perp$
T	T	T	T
F	T	F	$\perp$
$\perp$	T	$\perp$	$\perp$

This version of three-valued logic was already introduced by Kleene, and is also used by Jones for VDM.
Non-strict, sometimes called "lazy"
It corresponds to parallel/simultaneous evaluation.
Note that implementations often do this sequentially and are often not even symmetric:

sequential/asymmetric interpretation: $x\ OR_3\ y = \begin{cases} T & \text{if } x = T \\ \perp & \text{if } x = \perp \\ y & \text{if } x = F \end{cases}$

Then $T\ OR_3\ \perp \neq \perp\ OR_3\ T$.

The conventional abbreviations are used, such as
$B_1 \wedge B_2 \equiv \neg(\neg B_1 \vee \neg B_2)$ and $B_1 \rightarrow B_2 \equiv \neg B_1 \vee B_2$.

Consider the last three valued interpretation. Observe that if AND_3 and $IMPLIES_3$ are defined as in Table 2 then we have

- $Bval(B_1 \wedge B_2)\rho = Bval(B_1)\rho\ AND_3\ Bval(B_2)\rho$
- $Bval(B_1 \Rightarrow B_2)\rho = Bval(B_1)\rho\ IMPLIES_3\ Bval(B_2)\rho$

Assume $=_3$, $<_3$, $+_3$, ... have a strict interpretation, that is, they yield $\perp$ if one of the arguments is $\perp$.

Table 2: Three-valued conjunction and implication

AND_3	T	F	$\perp$
T	T	F	$\perp$
F	F	F	F
$\perp$	$\perp$	F	$\perp$

$IMPLIES_3$	T	F	$\perp$
T	T	F	$\perp$
F	T	T	T
$\perp$	T	$\perp$	$\perp$

2.3 Recursive Functions

Recall

$Eval(E \textbf{ where } f(x) = E_0)\rho = Eval(E)(\rho : f \mapsto FUN)$.

Now the aim is to define $FUN : D \cup \{\perp\} \to D \cup \{\perp\}$.

The idea is that FUN represents the function f in the semantic domain.

2.3.1 Non-recursive functions

For non-recursive functions it seems clear. For an example,

$Eval(f(7) \textbf{ where } f(x) = 2x + 9)\rho = Eval(f(7))(\rho : f \mapsto \lambda x.2x +_3 9) =$
$(Eval(f)(\rho : f \mapsto \lambda x.2x +_3 9))(7) = (\lambda x.2x +_3 9)(7) = 23.$

Note that, for simplicity, we use the notation of the lambda calculus here, but we could use $\{(x, y)|y = 2x +_3 9\}$, or other notations for functions. In general,

$Eval(E \textbf{ where } f(x) = E_0)\rho = Eval(E)(\rho : f \mapsto \lambda x.Eval(E_0)\rho)$,
provided f does not occur in E_0.

2.3.2 Recursive functions

If E_0 contains f then this definition is not correct; f gets its value from ρ then, leading to arbitrary values. Consider such a recursive example.

$Eval(fac(3) \textbf{ where } fac(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times fac(n-1) \textbf{ fi})\rho =$
$Eval(fac(3))(\rho : fac \mapsto FAC)$, where FAC represents the factorial function.
Similar to the lambda calculus, observe that FAC should satisfy

$$FAC(n) = Eval(\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times fac(n-1) \textbf{ fi})(\rho : fac \mapsto FAC) = ite(n =_3 0, 1, n \times_3 FAC(n -_3 1))$$

that is,

$FAC = \lambda n.ite(n =_3 0, 1, n \times_3 FAC(n -_3 1))$

To see this as a fixed point, define

$F \equiv \lambda f.\lambda n.ite(n =_3 0, 1, n \times_3 f(n -_3 1))$.

Then FAC should be a fixed point of F that is, $FAC = F(FAC)$.

In general,

$Eval(E \textbf{ where } f(x) = E_0)\rho = Eval(E)(\rho : f \mapsto FUN)$, where
FUN is a fixed point of $F \equiv \lambda g.\lambda x.Eval(E_0)(\rho : f \mapsto g)$.

Again there are several questions: is there a fixed point, which one to choose if there are more than one, how can we construct a solution?

Example 2.1

- Note that $f(7)$ **where** $f(x) = f(x) + 1$, then FUN should be a fixed point of $F_1 = \lambda g.\lambda x.Eval(f(x) + 1)(\rho : f \mapsto g)$, i.e.,
  $F_1 = \lambda g.\lambda x.g(x) +_3 1$.
  Note that for a fixed point h we must have
  $h = F_1(h) = \lambda x.h(x) +_3 1$, thus for any y, $h(y) = h(y) +_3 1$.
  Note: assume $+_3$ is strict and normal on rest of domain.
  If $h : \mathbb{N} \to \mathbb{N}$ then not possible, but here we have $\perp$ additionally: $\perp = \perp +_3 1$, so $UNDEF \equiv \lambda n.\perp$ is (the only) fixed point. Semantics should yield this function.

- Note that $f(7)$ **where** $f(x) = f(x)$ leads to $F_2 = \lambda g.\lambda x.Eval(f(x))(\rho : f \mapsto g)$, i.e.,
  $F_2 = \lambda g.\lambda x.g(x)$.
  Note that for a fixed point h we must have
  $h = F_2(h) = \lambda x.h(x)$, thus for any y, $h(y) = h(y)$, which is valid for any function, so infinite number of solutions. Which solution to choose?.
  Intuition; also $UNDEF$ because we have no information, the values are not defined.

Basic idea is to give approximations of the solution; start with the everywhere undefined function $UNDEF \equiv \lambda n.\perp$
Then we can consider approximations of the FAC function using F, where $F \equiv \lambda f.\lambda n.ite(n =_3 0, 1, n \times_3 f(n -_3 1))$.

$Fac^0 \equiv UNDEF$, $Fac^{k+1} \equiv F(Fac^k)$.

Thus $Fac^1 \equiv F(UNDEF) \equiv$
$\lambda n.ite(n =_3 0, 1, \perp)$
Similarly, $Fac^2 \equiv F(Fac^1) \equiv$
$\lambda n.ite(n =_3 0, 1, ite(n =_3 1, 1, \perp))$
etc

Thus Fac^k represents the function computed using at most k evaluations of the body. It is the best we can do on a machine with limited memory . Note that we always have to represent and compute infinite objects by finite approximations. These partially determined objects give incomplete information and we can order the object based on the amount of information they provide.

Note also that we can order these function on what might be called the "information content" or "degree of definedness"; $Fac^0 \sqsubseteq Fac^1 \sqsubseteq Fac^2 \ldots$

2.4 Complete Partial Orders and Fixed Points

Example 2.2 A set of sets with set inclusion.
Let S be a nonempty set and define $\wp(S) = \{K | K \subseteq S\}$.
On $\wp(S)$ define order $K_1 \sqsubseteq K_2$ iff $K_1 \subseteq K_2$.

Example 2.3 Consider *strings*/words; finite sequences of symbols, ϵ is the empty sequence (0 symbols).
$s_1 \hat{} s_2$ denotes concatenation; e.g. fiets $\hat{}$ bel = fietsbel.
Several possible orderings.

1. Prefix ordering: $s_1 \sqsubseteq_1 s_2$ iff exists s such that $s_1 \hat{} s = s_2$.
   E.g., fiets $\sqsubseteq_1$ fietsbel.

2. Similarly we could define postfix, or combine it with prefix:
   $s_1 \sqsubseteq_2 s_2$ iff exists s such that $s_1 \hat{} s = s_2$ or $s \hat{} s_1 = s$.
   E.g. bel $\sqsubseteq_2$ fietsbel

3. Or substring: $s_1 \sqsubseteq_3 s_2$ iff exist s and s' such that $s \hat{} s_1 \hat{} s' = s$.
   E.g. iets $\sqsubseteq_3$ fietsbel

Example 2.4 Closed intervals on $\mathbb{R}$ as approximations (e.g. of $\sqrt{}(2)$):
$[1, 2] \sqsubseteq [1.1, 1.8] \sqsubseteq [1.18, 1.57] \sqsubseteq \ldots$
Then order is defined by
$[l_1, u_1] \sqsubseteq [l_2, u_2]$ iff $l_1 \leq l_2 \wedge u_2 \leq u_1$.
Defined precisely:
$I = \{[l, u] \mid l \in \mathbb{R}, u \in \mathbb{R}\}$, where $[l, u] = \{x \in \mathbb{R} \mid l \leq x \wedge x \leq u\}$.

Definition 2.5 (Partial Order) A *partial order* $(D, \sqsubseteq)$ is a set D with a reflexive, anti-symmetric and transitive relation $\sqsubseteq$. Thus, for all $d, d_0, d_1, d_2 \in D$,

(reflexive) $d \sqsubseteq d$

(anti-symmetric) $d_0 \sqsubseteq d_1$ and $d_1 \sqsubseteq d_0$ implies $d_0 = d_1$

(transitive) $d_0 \sqsubseteq d_1$ and $d_1 \sqsubseteq d_2$ implies $d_0 \sqsubseteq d_2$

Note: order is called *total* if also $d_1 \sqsubseteq d_2$ or $d_2 \sqsubseteq d_1$, for all d_1 and d_2

Example 2.6 See example 2.2. In general, $(\wp(S), \subseteq)$ is partially ordered, least element $\emptyset$.
Not total: $\{a\} \not\sqsubseteq \{b, c\}$ and $\{b, c\} \not\sqsubseteq \{a\}$.

Example 2.7 See example 2.3.

1. Prefix ordering: $\sqsubseteq_1$ is partial order (check), not total (see fiets and bel).

2. $\sqsubseteq_2$ is reflexive, anti-symm, but not transitive:
   fiets $\sqsubseteq_2$ fietsen and fietsen $\sqsubseteq_2$ bromfietsen, but not fiets $\sqsubseteq_2$ bromfietsen.

3. Or substring: $\sqsubseteq_3$ is partial order, again not total.

Example 2.8 See example 2.4; ordering on closed intervals is partial order, again not total.

- $(\mathbb{R}, \sqsubseteq)$, where $r_1 \sqsubseteq r_2$ iff $r_1 \leq r_2$.
  We also write $(\mathbb{R}, \leq)$.
  Partial order, in fact total order.
  Note: with $<$ it is not reflexive.

- Consider a domain D; QUESTION: what is a minimal partial order on D?
  $d_1 \sqsubseteq d_2$ iff $d_1 = d_2$
  This is needed for reflexivity.

- For some data domain D, $(D \cup \{\bot\}, \sqsubseteq_D)$ is often ordered by

  $d_1 \sqsubseteq_D d_2$ iff $d_1 = \bot$ or $d_1 = d_2$

  This is a partial order, often called the *flat* order.

- Recall idea: $Fac^0 \sqsubseteq Fac^1 \sqsubseteq Fac^2 \ldots$
  Consider functions from $D \cup \{\bot\}$ to $D \cup \{\bot\}$. Assume given partial order $(D \cup \{\bot\}, \sqsubseteq_D)$.
  Let's define the ordering on functions point-wise: $(D \cup \{\bot\} \to D \cup \{\bot\}, \sqsubseteq)$ with

  $f \sqsubseteq g$ iff $f(d) \sqsubseteq_D g(d)$ for all $d \in D \cup \{\bot\}$.

  Suppose $\sqsubseteq_D$ is the flat order defined in previous point. What is ordering then:
  $f(d) \sqsubseteq_D g(d)$ iff $f(d) = \bot$ or $f(d) = g(d)$.
  So g is the same as f except for values for which f is undefined, so g is a more defined extension of f. Equivalently, we can say that $f \sqsubseteq g$ iff $f(d_1) = d_2 \neq \bot$ implies $g(d_1) = d_2$.

- Can this be used to compare different definitions of OR_3 ?
  Define $B_\bot = \{T, F, \bot\}$ and consider the flat partial order $(B_\bot, \sqsubseteq_b)$, thus
  $X \sqsubseteq_b Y$ iff $X = \bot$ or $X = Y$.
  Note: thus only $\bot \sqsubseteq_b \bot$, $\bot \sqsubseteq_b T$, $\bot \sqsubseteq_b F$, $T \sqsubseteq_b T$, and $F \sqsubseteq_b F$.
  Consider $BFUNCT \equiv B_\bot \times B_\bot \to B_\bot$ and $(BFUNCT, \sqsubseteq)$ with, for all $f, g \in BFUNCT$,

  $f \sqsubseteq g$ iff $f(X,Y) \sqsubseteq_b g(X,Y)$, for all X, Y in $B_\bot$.

  Then this yields a partial order (see EXERCISE ...) on $BFUNCT$.
  Consider the strict, strong and asymmetric definition of OR_3 :

$$X\ OR_3^s\ Y = \begin{cases} T & \text{if } X = T \text{ and } Y \neq \bot, \text{ or if } Y = T \text{ and } X \neq \bot \\ F & \text{if } X = F \text{ and } Y = F \\ \bot & \text{if } X = \bot \text{ or } Y = \bot \end{cases}$$

$$X\ OR_3\ Y = \begin{cases} T & \text{if } X = T \text{ or } Y = T \\ F & \text{if } X = F \text{ and } Y = F \\ \bot & \text{otherwise} \end{cases}$$

$$X\ OR_3^a\ Y = \begin{cases} T & \text{if } X = T \text{ or if } X \neq \bot \text{ and } Y = T \\ F & \text{if } X = F \text{ and } Y = F \\ \bot & \text{if } X = \bot \text{ or if } X \neq \bot \text{ and } Y = \bot \end{cases}$$

  Then $OR_3^s \sqsubseteq OR_3^a \sqsubseteq OR_3$; on $\{T, F\}$ they are the same,
  OR_3^s gives $\bot$ elsewhere,
  OR_3^a gives $T\ OR_3^a\ \bot = T$ but $\bot$ elsewhere (e.g. $\bot\ OR_3^a\ T = \bot$,
  whereas $X\ OR_3\ Y$ gives $T\ OR_3\ \bot = \bot\ OR_3\ T = T$ (and $\bot$ elsewhere).

INSERT BETWEEN EXAMPLES:

Definition 2.9 (Flat Order) A partial order $(D, \sqsubseteq)$ with $\bot \in D$ is called *flat* if
$d_1 \sqsubseteq d_2$ iff $d_1 = \bot$ or $d_1 = d_2$, for all $d_1, d_2 \in D$.

Definition 2.10 (Function Order) Consider a set A and a partial order $(D, \sqsubseteq_D)$. Functions from A to D can be ordered as follows, for $f, g : A \to D$,
$f \preceq g$ iff $f(x) \sqsubseteq_D g(x)$, for all $x \in A$.

Lemma 2.11 Let A be a set and $(D, \sqsubseteq_D)$ a partial order. Then $(A \to D, \preceq)$ is a partial order, with $\preceq$ as in definition 2.10.

Definition 2.12 ((Least) Upper Bound) For a partial order $(D, \sqsubseteq)$, a set $X \subseteq D$, and element $d \in D$,

1. d is an *upper bound* of X iff $x \sqsubseteq d$, for all $x \in X$;
2. d is a *least upper bound* (lub) of X iff d is an upper bound of X and $d \sqsubseteq d_0$ for any upper bound d_0 of X.

Example 2.13 Consider again example 2.2 with $(\wp(\{a,b,c\}),\subseteq)$.
For $X_1 = \{\emptyset,\{a\},\{a,b\}\}$ we have
upper bounds $\{a,b\}$, $\{a,b,c\}$,
least upper bound is $\{a,b\}$
For $X_2 = \{\{b\},\{c\}\}$ we have
upper bounds $\{b,c\}$, $\{a,b,c\}$,
least upper bound is $\{b,c\}$ NOTE: not element of X_2.
QUESTION: in general, for $(\wp(S),\subseteq)$, does every $X \subseteq \wp(S)$, thus X is a set of subsets of S, has a lub?
Yes: take $U \equiv \bigcup_{A\in X} A$. CHECK:
$A \subseteq U$, for all $A \in X$
if d is an upper bound, i.e., $A \subseteq d$, for all $A \in X$, then also $\bigcup_{A\in X} A \subseteq d$, thus $U \subseteq d$.

Example 2.14 See again example 2.3, with

1. Prefix ordering: $\sqsubseteq_1$. Note: { fiets, bel } has no upperbound (and hence no lub).
2. Substring: $\sqsubseteq_3$. E.g. { brom, fiets, bel, winkel } has upper bound "winkelvoorbromfietsbellen". Etcetera
lub? "bromfietsbelwinkel" and also "winkelbelfietsbrom" are upperbounds, but not comparable; so none of the two is a lub. No lub here!

Lemma 2.15 Consider a partial order $(D,\sqsubseteq)$ and a set $X \subseteq D$. If X has a least upper bound then the least upper bound is unique.

Proof. Suppose there are two lub's; d_1 and d_2, since both are also upper bounds, we must have: $d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_1$. By anti-symmetry we have $d_1 \equiv d_2$. □

Definition 2.16 (Chain) An *(ω-) chain* of a partial order $(D,\sqsubseteq)$ is a sequence $d_0, d_1, d_2, \ldots$ (also denoted $< d_i >_{i\geq 0}$) such that $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \ldots$

We say that d is a *(least) upper bound of a chain* $< d_i >_{i\geq 0}$ if d is a (least) upper bound of the corresponding set $\{d_i | i \geq 0\}$.

Example 2.17

- Recall $Fac^0 \sqsubseteq Fac^1 \sqsubseteq Fac^2 \ldots$.
- $(\mathbb{R},\sqsubseteq)$ with $\sqsubseteq = \leq$ has e.g. chain $0 \sqsubseteq 1/2 \sqsubseteq 3/4 \sqsubseteq 7/8 \sqsubseteq 15/16 \ldots$ with lub 1
- In $(\mathbb{N}\cup\{\bot\},\sqsubseteq)$, with
$n_1 \sqsubseteq n_2$ iff $n_1 \equiv \bot$ or ($n_1 \not\equiv \bot$, $n_2 \not\equiv \bot$ and $n_1 \leq n_2$)
the chain $\bot \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq 4 \sqsubseteq 8 \sqsubseteq 16 \ldots$ has no (least) upper bound,
but $\bot \sqsubseteq 1 \sqsubseteq 1 \sqsubseteq 1 \sqsubseteq \ldots$ has.
- For closed intervals: $[0,3] \sqsubseteq [1/2, 2\,1/2] \sqsubseteq [3/4, 2\,1/4] \sqsubseteq [7/8, 2\,1/8] \sqsubseteq \ldots$
lub: $[1,2]$ (take limits of both end points).
- For strings and prefix order:
$a \sqsubseteq ab \sqsubseteq aba \sqsubseteq abab \ldots$
(or $ping \sqsubseteq pingpong \sqsubseteq pingpongping \sqsubseteq pingpongpingpong \ldots$)
No lub if only finite strings. We need infinite strings to get lub.

- For $(\wp(\{a,b,c\}), \sqsubseteq)$, with $\sqsubseteq = \subseteq$ we can have chains
  $\emptyset \sqsubseteq \emptyset \sqsubseteq \emptyset \sqsubseteq \emptyset \sqsubseteq \ldots$ with lub $\emptyset$
  similar for other constant chains. Or;
  $\{a\} \sqsubseteq \{a\} \sqsubseteq \{a,c\} \sqsubseteq \{a,c\} \sqsubseteq \ldots$
  In general, all chains become eventually constant, and this constant yields lub.
  This can be generalized.

Lemma 2.18 Consider a partial order $(D, \sqsubseteq)$. If D is finite then every chain has a lub in D.

Definition 2.19 (Complete Partial Order) A *complete partial order* (cpo) is a partial order $(D, \sqsubseteq)$ such that

1. there is a least element $\bot \in D$ (or $\bot_D$) with respect to $\sqsubseteq$, that is, $\bot \sqsubseteq d$, for all $d \in D$;
2. every chain $d_0, d_1, d_2, \ldots$ (or $< d_i >_{i \geq 0}$) has a least upper bound (in D), denoted by $\sqcup_{i \geq 0} d_i$ (or $\sqcup^D_{i \geq 0} d_i$ if confusion is possible).

Lemma 2.20 If a partially ordered set $(D, \sqsubseteq)$ has a least element, then this least element is unique (i.e. a partial order has at most one least element).

Thus we may talk about *the* least element of D.

Example 2.21

- $(\wp(S), \subseteq)$ is a cpo with least element $\bot \equiv \emptyset$ and
  lub is obtained by taking union of all sets: $\sqcup_{i \geq 0} d_i \equiv \bigcup \{d_i | i \geq 0\}$.
  E.g. $(\wp(\{a,b,c\}), \subseteq)$ is a cpo.
- Closed intervals: $I = \{[l,u] \mid l \in \mathbb{R}, u \in \mathbb{R}\}$, where $[l,u] = \{x \in \mathbb{R} \mid l \leq x \wedge x \leq u\}$.
  Consider partial order $(I, \cap)$.
  Chain of closed intervals, for instance, $[0,3] \sqsubseteq [1/2, 21/2] \sqsubseteq [3/4, 21/4] \sqsubseteq [7/8, 21/8] \sqsubseteq \ldots$
  Take intersection: $\sqcup_{i \geq 0} d_i \equiv \bigcap \{d_i | i \geq 0\}$. Note that $\emptyset = [3,2] \in I$.
  Problem: no least element. By adding it, $(I \cup [-\infty, \infty], \cap)$, i.e. $(I \cup \mathbb{R}, \cap)$, we obtain cpo (lub still OK).
- QUESTION: $([0,1), \leq)$?
  NO: consider chain 1/2, 3/4, 7/8, etc (with limit 1) has no lub in $[0,1)$.
- Closed interval of real numbers $\mathbb{R}$, e.g.
  $([0,1], \leq)$ is a cpo. Least element is 0.
- QUESTION: $(\mathbb{N}, \leq)$?
  NO, since no least upper bound
  We could add ∞ to obtain cpo, assuming $n \leq \infty$ for all $n \in \mathbb{N}$,
- QUESTION: $(\{n \mid n < 1000\}, \leq)$?
  Least element: 0, all chains become constant; see lemma 2.18 and next lemma).

Note that by lemma 2.18 we have the following lemma.

Lemma 2.22 Consider a partial order $(D, \sqsubseteq)$ and suppose D is finite. Then $(D, \sqsubseteq)$ is a cpo iff there exists a least element in D.

Definition 2.23 (Lifting) Given a set A, the *lifted* set A, denoted $A_\perp$, is defined by $(A \cup \{\perp\}, \sqsubseteq)$ with $x \sqsubseteq y$ iff $x = \perp$ or $x = y$.

A domain $\mathcal{D} = (D, \sqsubseteq_D)$ can be lifted to $\mathcal{D}_\perp = (D \cup \{\perp\}, \sqsubseteq)$ where
$x \sqsubseteq y$ iff $x = \perp$ or $x, y \in D$ and $x \sqsubseteq_D y$.

NOTE: if $\mathcal{D}$ is a cpo then is $\perp_D$ different from $\perp$.

example: $\mathbb{N}_\perp$

A domain with order $\sqsubseteq$ is *flat* if $x \sqsubseteq y$ iff $x = \perp$ or $x = y$.

Lemma 2.24

a) $A_\perp$ is a flat cpo.

b) If $\mathcal{D}$ is a cpo then also $\mathcal{D}_\perp$ is a cpo.

Recall definition 2.10:
For a set A and a partial order $(D, \sqsubseteq_D)$, functions $f, g : A \to D$ can be ordered by
$f \preceq g$ iff $f(x) \sqsubseteq_D g(x)$, for all $x \in A$.
In exercise shown that $(A \to D, \preceq)$ this a partial order, using $A \to D$ to denote the set of all functions from A to D.

Lemma 2.25 If $(D, \sqsubseteq_D)$ is a cpo, then $(A \to D, \preceq)$ is a cpo.

Definition 2.26 (Strict & Monotonicity) Consider two cpo's $(D, \sqsubseteq_D)$ and $(R, \sqsubseteq_R)$ and a function $f : D \to R$.

- Function f is called *strict* iff $f(\perp_D) = \perp_R$.
- Function f is called *monotonic* iff $x \sqsubseteq_D y$ implies $f(x) \sqsubseteq_R f(y)$.

Strict: "no info in, no info out"
Monotonic: "more information in implies more information out".

Example 2.27 Consider the cpo's $(\wp(\{a, b, c\}), \subseteq)$ and $(\wp(\{d, e\}), \subseteq)$.
Consider the function $f_1 : \wp(\{a, b, c\} \to \wp(\{d, e\}$ which changes a's and b's to d's and c's to e's, defined by

X	$\emptyset$	$\{a\}$	$\{b\}$	$\{c\}$	$\{a, b\}$	$\{a, c\}$	$\{b, c\}$	$\{a, b, c\}$
$f_1(X)$	$\emptyset$	$\{d\}$	$\{d\}$	$\{e\}$	$\{d\}$	$\{d, e\}$	$\{d, e\}$	$\{d, e\}$

Strict and monotonic.
Consider also $f_2 : \wp(\{a, b, c\} \to \wp(\{d, e\}$ which maps all sets that contain a to $\{d\}$, others to $\{e\}$, defined by

X	$\emptyset$	$\{a\}$	$\{b\}$	$\{c\}$	$\{a, b\}$	$\{a, c\}$	$\{b, c\}$	$\{a, b, c\}$
$f_2(X)$	$\{e\}$	$\{d\}$	$\{e\}$	$\{e\}$	$\{d\}$	$\{d\}$	$\{e\}$	$\{d\}$

Not strict and not monotonic: $\{b, c\} \subseteq \{a, b, c\}$ but $f_2(\{b, c\}) \not\subseteq f_2(\{a, b, c\})$.

Example 2.28 Let $(B_\perp, \sqsubseteq)$ be the (flat) cpo obtained lifting the set of booleans $\{T, F\}$.
Consider a few examples of (non)monotonic functions $f : B_\perp \to B_\perp$ (there are 11) in table:

$f(\bot)$	$f(T)$	$f(F)$	strict	monotonic
$\bot$	$\bot$	$\bot$	Y	Y
$\bot$	F	T	Y	Y
T	T	T	N	Y
$\bot$	F	$\bot$	Y	Y
T	$\bot$	F	N	N

NOTE: write def of monotone, observe that it depends on $f(\bot)$.
Question: is strict, but not monotone possible?

Note: if strict than monotome on this (flat) domain; this holds in general.

Lemma 2.29 Consider two cpo's $(D, \sqsubseteq_D)$ and $(R, \sqsubseteq_R)$. If $(D, \sqsubseteq_D)$ is flat then every strict function $f : D \to R$ is monotonic.

Proof. Let $x \sqsubseteq_D y$. Since $(D, \sqsubseteq_D)$ is flat, there are two cases:

- $x = \bot_D$, then, by strictness, $f(x) = f(\bot_D) = \bot_R \sqsubseteq_R f(y)$.
- $x = y$, then $f(x) = f(y)$ and by reflexivity also $f(x) \sqsubseteq_R f(y)$.

□

Note that, for a monotonic function $f : D \to R$, the fact that
$< d_i >_{i\geq 0}$, i.e. $d_0 \sqsubseteq_D d_1 \sqsubseteq_D d_2 \sqsubseteq_D d_3 \ldots$ is a chain
implies that also
$< f(d_i) >_{i\geq 0}$, i.e. $f(d_0) \sqsubseteq_R f(d_1) \sqsubseteq_R f(d_2) \sqsubseteq_R f(d_3) \ldots$ is also a chain.

QUESTION: what about lubs, do monotonic functions preserve lubs, that is,

$f(\sqcup^D_{i\geq 0} d_i) = \sqcup^R_{i\geq 0} f(d_i)$??

NO: consider cpo $(\wp(\mathbb{N}), \subseteq)$.

Define $\phi : \wp(\mathbb{N}) \to \wp(\mathbb{N})$ with, for $S \subseteq \mathbb{N}$, $\phi(S) = \begin{cases} \emptyset & \text{if } S \text{ is finite} \\ \mathbb{N} & \text{otherwise} \end{cases}$

Then ϕ is monotonic: if $X_1 \subseteq X_2$ then $\phi(X_1) \subseteq \phi(X_2)$
Let $D_i = \{0, 1, \ldots, i\}$ and consider the chain $< D_i >_{i\geq 0}$, thus $\{0\} \subseteq \{0, 1\} \subseteq \{0, 1, 2\}$, etc.
This chain has least upper bound $\sqcup_{i\geq 0} D_i = \bigcup_{i\geq 0} D_i = \mathbb{N}$.
So $\phi(\sqcup_{i\geq 0} D_i) = \mathbb{N}$, but $\sqcup_{i\geq 0} \phi(D_i) = \sqcup_{i\geq 0} \emptyset = \emptyset$.

Definition 2.30 (Continuity) Consider two cpo's $(D, \sqsubseteq_D)$ and $(R, \sqsubseteq_R)$.
Function $f : D \to R$ is called *continuous* iff f is monotonic and, for any chain $< d_i >_{i\geq 0}$,
$f(\sqcup^D_{i\geq 0} d_i) = \sqcup^R_{i\geq 0} f(d_i)$.

So "continuous function is determined by finite approximations".

Example 2.31 The function ϕ defined above is not continuous.
Recall that ϕ is monotonic, so not every monotonic function is continuous.

Example 2.32 Let $f : [0, 1] \to [0, 1]$ be a continuous function from cpo $([0, 1], \leq)$ to cpo $([0, 1], \sqsubseteq)$, where $x \sqsubseteq y$ iff $x \geq y$ (so reversed order).
Consider the chain $1/2 \leq 3/4 \leq 7/8 \leq \ldots$. This chain approximates 1 (the lub of this chain), and hence $f(1)$ is determined by the lub of the image of the chain. Say f computes some complement, so leads to
$1/2 \sqsubseteq 1/4 \sqsubseteq 1/8 \sqsubseteq 1/16 \sqsubseteq \ldots$ which has lub 0, so we must have $f(1) = 0$.

Lemma 2.33 Consider two cpo's $(D, \sqsubseteq_D)$ and $(R, \sqsubseteq_R)$. If $f : D \to R$ is monotonic then $\sqcup^R_{i\geq 0} f(d_i) \sqsubseteq_R f(\sqcup^D_{i\geq 0} d_i)$, for any chain $< d_i >_{i\geq 0}$ in D.

Proof. By the lub property, $d_i \sqsubseteq_D \sqcup^D_{i\geq 0} d_i$, for all i.

By monotonicity of f, $f(d_i) \sqsubseteq_R f(\sqcup^D_{i\geq 0} d_i)$, for all i.

Hence $f(\sqcup^D_{i\geq 0} d_i)$ is an upper bound of $< f(d_i) >_{i\leq 0}$, and thus greater than the lub of this chain, i.e., $\sqcup^R_{i\geq 0} f(d_i) \sqsubseteq_R f(\sqcup^D_{i\geq 0} d_i)$. $\square$

Corollary 2.34 Consider two cpo's $(D, \sqsubseteq_D)$ and $(R, \sqsubseteq_R)$. Then $f : D \to R$ is continuous iff f is monotonic and, for any chain $< d_i >_{i\geq 0}$, $f(\sqcup^D_{i\geq 0} d_i) \sqsubseteq_R \sqcup^R_{i\geq 0} f(d_i)$.

Lemma 2.35 Consider two cpo's $(D, \sqsubseteq_D)$ and $(R, \sqsubseteq_R)$ and suppose D is finite. Then every monotonic function is continuous.

Proof. Basic idea of the proof; every chain is "finite", that is, has a constant tail.
Consider a monotonic function $f : D \to R$ and a chain $< d_i >_{i\geq 0}$ in D.
Recall that then also $< f(d_i) >_{i\geq 0}$ is a chain in R.
Since D is finite and using anti-symmetry of $\sqsubseteq$,
there exists a $k \geq 0$ such that $d_j = d_k$, for $j \geq k$.
Hence for $< f(d_i) >_{i\geq 0}$ we have that $f(d_j) = f(d_k)$, for $j \geq k$.
Thus $\sqcup^R_{i\geq 0} f(d_i) = f(d_k)$, and hence $f(\sqcup^D_{i\geq 0} d_i) = f(d_k) = \sqcup^R_{i\geq 0} f(d_i)$. $\square$

Example 2.36 Let $(V \to_m V) = \{f : V \to V \mid f \text{ is monotone } \}$ be the set of monotonic functions on cpo V.
As before ordering $\preceq$ leads to a cpo $((V \to_m V), \preceq)$
with least element $\bot_{V\to_m V} \equiv \lambda v.\bot_V$ the everywhere undefined function.
Define $F : (V \to_m V) \to (V \to_m V)$ as follows, for $f \in (V \to_m V)$,

$$F(f) = \begin{cases} \bot_{V\to_m V} & \text{if there exists a } v \not\equiv \bot_V \text{ with } f(v) = \bot_V \\ f & \text{otherwise} \end{cases}$$

Thus F maybe viewed as testing whether f is total.
F is monotonic; assume $f_1 \preceq f_2$ and show $F(f_1) \preceq F(f_2)$.

- If $F(f_2) = f_2$ then OK in all cases: $\bot_{V\to_m V} \preceq f_2$ and $f_1 \preceq f_2$.
- If $F(f_2) \equiv \bot_{V\to_m V}$ then $f_2(v) = \bot_V$ for some v, but then by $f_1 \preceq f_2$ we have $f_1(v) \sqsubseteq_V f_2(v) = \bot_V$, thus also $f_1(v) = \bot_V$, and thus $F(f_1) \equiv \bot_{V\to_m V}$

But not continuous: consider $< f_i >_{i\geq 0}$ with $f_i(v) = \begin{cases} v & \text{if } v \leq i \\ \bot_V & \text{otherwise} \end{cases}$

Then $< f_i >_{i\geq 0}$ is a chain and $\sqcup_{i\geq 0} f_i \equiv \lambda v.v$.
Thus $F(\sqcup_{i\geq 0} f_i) = F(\lambda v.v) = \lambda v.v$
BUT, for all i, $F(f_i) = \bot_{V\to_m V}$, hence $\sqcup_{i\geq 0} F(f_i) = \bot_{V\to_m V}$ (i.e. different from $F(\sqcup_{i\geq 0} f_i)$).

We use $[D \to R]$ to denote the set of all continuous functions from a cpo D to a cpo R. Thus
$[D \to R] = \{f : D \to R \mid f \text{ is continuous } \}$

Lemma 2.37 If $(D, \sqsubseteq_D)$ and $(R, \sqsubseteq_R)$ are cpo's then also $([D \to R], \preceq)$ is a cpo.

Proof. Least element $\bot_{[D\to R]} \equiv \lambda x.\bot_R$ (See earlier proof that this is indeed least element.)
NOTE that this is continuous function; monotone and
$\bot_{[D\to R]}(\sqcup^D_{i\geq 0} d^i) = \bot_R = \sqcup^R_{i\geq 0}\bot_R = \sqcup^R_{i\geq 0}\bot_{[D\to R]}(d^i)$
and the least upper bound for a chain $< f_i >_{i\geq 0}$ in $[D \to R]$ given by
$\sqcup_{i\geq 0} f^i \equiv \lambda x.\, \sqcup_{i\geq 0}\, f^i(x)$.
Note: shown before that this is the least upper bound
□

Without proof, we mention a few useful properties about constructs that preserve continuity. For instance, the choice construct; **if** b **then** f **else** g **fi** is continuous in f (consider b and g as constants).

Lemma 2.38 Consider $f, g : D \to R$, and $b : D \to \{T, F\}$.

Define $F(f)$ by $F(f)(d) = \begin{cases} f(d) & \text{if } b(d) \\ g(d) & \text{if } \neg b(d) \end{cases}$

Then F is continuous on the usual cpo of functions $(D \to R, \preceq)$.

Proof. First show that F is monotonic:
consider $g_1, g_2 : D \to R$ and suppose $g_1 \preceq g_2$. We have to show $F(g_1) \preceq F(g_2)$, that is, $F(g_1)(d) \sqsubseteq_D F(g_2)(d)$, for all d.
If $b(d)$ then this equals $g_1(d) \sqsubseteq_D g_2(d)$ which is given.
If $\neg b(d)$ then this equals $g(d) \sqsubseteq_D g(d)$ which holds by reflexivity.

For continuity, consider a chain $< g_i >_{i\geq 0}$. We have to show $F(\sqcup_{i\geq 0} g_i) \preceq \sqcup_{i\geq 0} F(g_i)$, that is, $F(\sqcup_{i\geq 0} g_i)(d) \sqsubseteq_R \sqcup_{i\geq 0} F(g_i)(d)$, for all d.

- If $b(d)$ then we obtain $F(\sqcup_{i\geq 0} g_i)(d) = (\sqcup_{i\geq 0} g_i)(d) \equiv$
  (see earlier construction of cpo on function domain)
  $(\lambda x.\, \sqcup_{i\geq 0}\, g_i(x))(d) = \sqcup_{i\geq 0} g_i(d) = \sqcup_{i\geq 0} F(g_i)(d)$.
- If $\neg b(d)$ then we obtain $F(\sqcup_{i\geq 0} g_i)(d) = (\sqcup_{i\geq 0} f)(d) = f(d) = \sqcup_{i\geq 0} f(d) = \sqcup_{i\geq 0} F(g_i)(d)$.

□

Lemma 2.39 If $f \in [D \to C]$ and $g \in [C \to R]$ then $g \circ f \in [D \to R]$, where
$(g \circ f)(x) = g(f(x))$.

Definition 2.40 (Least Fixed Point)
Consider a cpo $(D, \sqsubseteq)$ and a function $f : D \to D$.

- d is a *fixed point* of f iff $f(d) = d$.
- d is a *least fixed point* of f iff d is a fixed point and $d \sqsubseteq d'$ for any fixed point d' of f.

If the least fixed point of f exists then it is unique and often denoted by $\mu.f$ (or also by $fix(f)$).

For a function $f : D \to D$ define $f^0(x) = x$ and $f^{i+1}(x) = f(f^i(x))$.

Lemma 2.41 If, for a domain D, function $f : D \to D$ is monotonic (or continuous), then $< f^i(\bot_D) >_{i\geq 0}$ is a chain.

Proof. Using monotonicity of f and $\bot_D \sqsubseteq f(\bot_D)$ we can show by induction
$f^i(\bot_D) \sqsubseteq f^{i+1}(\bot_D)$. □

Theorem 2.42 (Fixed Point - Tarski/Knaster) Given a cpo $(D, \sqsubseteq_D)$, every continuous function $f : D \to D$ has a least fixed point $\mu.f$ given by $\mu.f = \sqcup^D_{i\geq 0} f^i(\bot_D)$.
(Or shortly $\mu.f = \sqcup f^i(\bot)$.)

Proof. Recall that, by the lemma above, $< f^i(\bot_D) >_{i\geq 0}$ is a chain, and thus $\sqcup^D_{i\geq 0} f^i(\bot_D)$ indeed exists.

1. $f(\mu.f) = f(\sqcup^D_{i\geq 0} f^i(\bot_D)) =$ { By continuity of f}
   $\sqcup^D_{i\geq 0} f(f^i(\bot_D)) = \sqcup^D_{i\geq 0} f^{i+1}(\bot_D) = \sqcup^D_{i\geq 1} f^i(\bot_D) =$
   (Note that we can add $f^0(\bot_D) = \bot_D$.)
   $\sqcup^D_{i\geq 0} f^i(\bot_D) = \mu.f$,
   thus $\mu.f$ is indeed a fixed point.

2. To show that $\mu.f$ is the least fixed point, let x be another fixed point, so $f(x) = x$.
   We have to show $\mu.f \sqsubseteq_D x$, i.e. $\sqcup f^i(\bot) \sqsubseteq_D x$.
   Thus x must be greater or equal than the least upper bound of the chain $< f^i(\bot_D) >_{i\geq 0}$.
   This follows easily if x is an upper bound of the chain. Hence we prove, by induction on i, that $f^i(\bot_D) \sqsubseteq x$, for all i.

   (Basis) $f^0(\bot_D) = \bot_D \sqsubseteq x$
   (Step) By monotonicity, $f^{i+1}(\bot_D) = f(f^i(\bot_D)) \sqsubseteq f(x) = x$.

   So x is an upper bound of $< f^i(\bot_D) >_{i\geq 0}$ and by the definition of lub we can conclude $\mu.f = \sqcup^D_{i\geq 0} f^i(\bot_D) \sqsubseteq x$.

□

2.5 Back to the Denotational Semantics

Now how do we want to use all this for our semantics?
We want to define
$Eval(E \textbf{ where } f(x) = E_0)\rho = Eval(E)(\rho : f \mapsto FUN)$
where FUN represents the definition of f, that is, we want
$FUN(x) = Eval(E_0)(\rho : f \mapsto FUN)$, thus
$FUN = \lambda x.Eval(E_0)(\rho : f \mapsto FUN)$, so FUN is a fixed point of
$F \equiv \lambda G.\lambda x.Eval(E_0)(\rho : f \mapsto G)$.
Then FUN should be a fixed point of F, $FUN = F(FUN)$. In fact we want the least fixed point:
$FUN \equiv \mu.F$

- $Eval(E \textbf{ where } f(x) = E_0)\rho = Eval(E)(\rho : f \mapsto \mu.F)$
  with $F \equiv \lambda G.\lambda x.Eval(E_0)(\rho : f \mapsto G))$, and
  where $\mu.F = \sqcup_{i\geq 0} F^i(\bot_{D_\bot \to D_\bot})$.

To justify this definition, let $D_\bot$ be the lifted set D, i.e. the flat cpo $(D \cup \{\bot\}, \sqsubseteq)$, (see lemma 2.24). From lemma 2.25 we then obtain that $(D_\bot \to D_\bot, \preceq)$ is a cpo. It can be shown that function $F : (D_\bot \to D_\bot) \to (D_\bot \to D_\bot)$, is continuous, using lemmas for the programming constructs (see e.g. lemma 2.39 and lemma 2.38). Then by theorem 2.42 the least fixed point of F indeed exists and equals the least upper bound of the iterated application of F to the least element of $(D_\bot \to D_\bot, \preceq)$, namely $\lambda d.\bot_D$.

Example 2.43 Consider $D \equiv \mathbb{N}$ and recall
$E_{fib} \equiv$ **if** $n = 0$ **then** 0
else if $n = 1$ **then** 1
else if $n > 1$ **then** $fib(n-1) + fib(n-2)$ **fi fi fi** .

Consider $Eval(fib(3)$ **where** $fib(n) = E_{fib})\rho$, thus
$Eval(fib(3))(\rho : fib \mapsto \mu.F)$
with $F \equiv \lambda G.\lambda n.Eval(E_{fib})(\rho : fib \mapsto G))$, thus

$$F \equiv \lambda G.\lambda n.\begin{cases} \bot & \text{if } n = \bot \\ 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ G(n-1) + G(n-2) & \text{if } n > 1 \end{cases}$$

By theorem 2.42 we have that

$$\mu.F = \sqcup_{i \geq 0} F^i(\bot_{\mathbb{N}_\bot \to \mathbb{N}_\bot})$$

Recall that we have domain on $\mathbb{N}_\bot \to \mathbb{N}_\bot$, thus $\bot_{\mathbb{N}_\bot \to \mathbb{N}_\bot} \equiv \lambda n.\bot$.
Hence $\mu.F$ is the lub of the chain
$F^0(\lambda n.\bot) \sqsubseteq F^1(\lambda n.\bot) \sqsubseteq F^2(\lambda n.\bot) \sqsubseteq \ldots$
Note that
$F^0(\lambda n.\bot) = \lambda n.\bot$, where $n \in \mathbb{N}_\bot$

$$F^1(\lambda n.\bot) = F(\lambda n.\bot) = \begin{cases} \bot & \text{if } n = \bot \\ 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ (\lambda n.\bot)(n-1) + (\lambda n.\bot)(n-2) = \bot & \text{if } n > 1 \end{cases}$$

$$F^2(\lambda n.\bot) = F(F^1(\lambda n.\bot)) = \begin{cases} \bot & \text{if } n = \bot \\ 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ \bot & \text{if } n > 2 \end{cases}$$

NOTE: we get $(F^1(\lambda n.\bot))(n-1) + (F^1(\lambda n.\bot))(n-2)$ if $n > 1$, so for $n = 2$ we have $1 + 0$, and for $n = 3$ we have $\bot + 1 = \bot$, for $n = 4$ we have $\bot + \bot = \bot$.

$$F^3(\lambda n.\bot) = F(F^2(\lambda n.\bot)) = \begin{cases} \bot & \text{if } n = \bot \\ 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ 1+1 & \text{if } n = 3 \\ \bot & \text{if } n > 3 \end{cases}$$

etc, so we are approximating the Fibonacci function. And we can evaluate $fib(3)$ to 2.

Example 2.44 Consider $Eval(f(7)$ **where** $f(x) = f(x) + 1)\rho$, thus
$Eval(f(7))(\rho : f \mapsto \mu.F)$
with $F \equiv \lambda G.\lambda x.Eval(f(x) + 1)(\rho : f \mapsto G))$, thus
$F \equiv \lambda G.\lambda x.G(x) + 1$
Then $\mu.F$ is the least upper bound of the chain
$F^0(\lambda x.\bot) = \lambda x.\bot$
$F^1(\lambda x.\bot) = F(\lambda x.\bot) = \lambda x.\bot + 1 = \lambda x.\bot$
$F^2(\lambda x.\bot) = F(F^1(\lambda x.\bot)) = F(\lambda x.\bot) = \lambda x.\bot + 1 = \lambda x.\bot$
etc, always $\lambda x.\bot$, so lub is also $\lambda x.\bot$.
Hence $Eval(f(7)$ **where** $f(x) = f(x) + 1)\rho = \bot$.

3 Denotational Semantics of Imperative Programs **September 4, 2007**

The aim is to define $\mathcal{D}(S)\sigma_0$ as the set of states in which program S can terminate starting from a state σ_0. Thus $\mathcal{D}(S)$ is a function $\mathcal{D}(S) : STATE \to \wp(STATE)$.
The aim is to define $\mathcal{D}$ in a *compositional* way, that is, the meaning of a compound construct is defined in terms of the meaning of its components.
Recall that this does not hold for the operational semantics. Reason is that the while is defined in terms of itself

[while$_{sos}$] $\quad \langle \textbf{while } b \textbf{ do } S \textbf{ od}, \sigma_0\rangle \longrightarrow \langle \textbf{if } b \textbf{ then } (S \,;\, \textbf{while } b \textbf{ do } S \textbf{ od}) \textbf{ else skip fi}, \sigma_0\rangle$

Definition 3.1 (Denotational Semantics) The denotational semantics of a sequential program is defined by induction on the structure of statements.

a) $\mathcal{D}(x := e)\sigma_0 = \{(\sigma_0 : x \mapsto \mathcal{E}(e)\sigma_0)\}$

b) $\mathcal{D}(\textbf{skip})\sigma_0 = \{\sigma_0\}$

c) $\mathcal{D}(S_1 \,;\, S_2)\sigma_0 = \{\sigma \mid$ there exists a σ_1 such that $\sigma_1 \in \mathcal{D}(S_1)\sigma_0$ and $\sigma \in \mathcal{D}(S_2)\sigma_1\}$

To simplify definitions, we define for functions
$F_1, F_2 : STATE \to \wp(STATE)$ the function $SEQ(F_1, F_2) : STATE \to \wp(STATE)$ by
$SEQ(F_1, F_2)\sigma_0 = \{\sigma \mid$ there exists a σ_1 such that $\sigma_1 \in F_1(\sigma_0)$ and $\sigma \in F_2(\sigma_1)\}$
Then we indeed have a compositional formulation
$\mathcal{D}(S_1 \,;\, S_2) = SEQ(\mathcal{D}(S_1), \mathcal{D}(S_2))$.

Intermezzo - semantic equivalence
Let $S_1 \sim S_2$ denote that S_1 and S_2 are semantically equivalent, that is, $\mathcal{D}(S_1) = \mathcal{D}(S_2)$.
Note that $\sim$ is an *equivalence relation*, that is, for all S, S_1, S_2, S_3,

(reflexive) $S \sim S$

(symmetric) if $S_1 \sim S_2$ then $S_2 \sim S_1$,

(transitive) if $S_1 \sim S_2$ and $S_2 \sim S_3$ then $S_1 \sim S_3$.

Note that we have

- $SEQ(\mathcal{D}(S), \mathcal{D}(\textbf{skip})) = SEQ(\mathcal{D}(\textbf{skip}), \mathcal{D}(S)) = \mathcal{D}(S)$.
  Then $S \,;\, \textbf{skip} \sim \textbf{skip} \,;\, S \sim S$.
- SEQ is associative, that is,
  $SEQ(F_1, SEQ(F_2, F_3)) = SEQ(SEQ(F_1, F_2), F_3)$,
  thus $S_1 \,;\, (S_2 \,;\, S_3) \sim (S_1 \,;\, S_2) \,;\, S_3$ and we can omit the brackets.

d) $\mathcal{D}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi})\sigma_0 = \{\sigma \mid \sigma \in \mathcal{D}(S_1)\sigma_0$ if $\mathcal{B}(b)\sigma_0$ and $\sigma \in \mathcal{D}(S_2)\sigma_0$ otherwise $\}$

Define for functions $F_1, F_2 : STATE \to \wp(STATE)$ and $B : STATE \to \{T, F\}$ the function
$COND(B, F_1, F_2) : STATE \to \wp(STATE)$ by
$COND(B, F_1, F_2)\sigma_0 = \begin{cases} F_1(\sigma_0) & \text{if } B(\sigma_0) \\ F_2(\sigma_0) & \text{otherwise} \end{cases}$
Thus we have the compositional formulation
$\mathcal{D}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = COND(\mathcal{B}(b), \mathcal{D}(S_1), \mathcal{D}(S_2))$.

e) To define the semantics of **while** b **do** S **od**, observe that we should have
while b **do** S **od** $\sim$ **if** b **then** (S ; **while** b **do** S **od**) **else skip fi**, that is,
$\mathcal{D}$(**while** b **do** S **od**) =
$\mathcal{D}$(**if** b **then** (S ; **while** b **do** S **od**) **else skip fi**) =
$COND(\mathcal{B}(b),\ \mathcal{D}(S\ ;$ **while** b **do** S **od**$),\ \mathcal{D}($**skip**$))$ =
$COND(\mathcal{B}(b),\ SEQ(\mathcal{D}(S), \mathcal{D}($**while** b **do** S **od**$)),\ \mathcal{D}($**skip**$))$

Hence $\mathcal{D}$(**while** b **do** S **od**) is a fixed point of
$G : (STATE \to \wp(STATE)) \to (STATE \to \wp(STATE))$ defined by
$G(D) = COND(\mathcal{B}(b),\ SEQ(\mathcal{D}(S), D),\ \mathcal{D}($**skip**$))$,
for $D : STATE \to \wp(STATE)$.

Usual questions; is there a fixed point, and if more than one, which one to choose?

Example 3.2 Consider $\mathcal{D}$(**while** *true* **do skip od**); should be fixed point of G_0 with

$G_0(D) = COND(\mathcal{B}(true),\ SEQ(\mathcal{D}($**skip**$), D),\ \mathcal{D}($**skip**$)) = SEQ(\mathcal{D}($**skip**$), D) = D.$

Clearly

- any function D is a fixed point of G_0 with $G_0(D) = D$;
- we want $\mathcal{D}$(**while** *true* **do skip od**) $= \emptyset$, that is, the *least* fixed point in some sense.

So the aim is to use theorem 2.42 of Tarski/Knaster.
Hence we have to turn the set of functions $STATE \to \wp(STATE)$ into a cpo.

1. First observe that $(\wp(STATE), \subseteq)$ is a cpo, with least element $\emptyset$ and where the lub can be found by taking the union.
2. Then define the cpo $(STATE \to \wp(STATE), \preceq)$ with $F_1 \preceq F_2$ iff $F_1(\sigma) \subseteq F_2(\sigma)$, least element $\bot = \lambda\sigma.\emptyset$ and $\sqcup_{i\geq 0} F_i \equiv \lambda\sigma. \bigcup_{i\geq 0} F_i(\sigma)$.

As in the previous chapter, we can show that $COND$ is continuous and similarly for the other constructors.
Assuming that we can show that G is continuous, the theorem of Tarski/Knaster shows that the least fixed point exists, and we can define

$\mathcal{D}$(**while** b **do** S **od**) $= \mu.G$ where

$G : (STATE \to \wp(STATE)) \to (STATE \to \wp(STATE))$ is defined by

$G(D) = COND(\mathcal{B}(b), SEQ(\mathcal{D}(S), D), \mathcal{D}($**skip**$))$.

Note that this leads to

$\mathcal{D}$(**while** b **do** S **od**) $= \sqcup_{i\geq 0} G^i(\bot)$.

Example 3.3 Recall example 3.2. Observe that $\mathcal{D}$(**while** *true* **do skip od**) $= \sqcup_{i\geq 0} G_0^i(\bot) = \sqcup_{i\geq 0} \bot = \bot \equiv \lambda\sigma.\emptyset$.

Next consider the general case, where

$G(D) = COND(\mathcal{B}(b), SEQ(\mathcal{D}(S), D), \mathcal{D}($**skip**$))$

and $\mathcal{D}$(**while** b **do** S **od**) $= \sqcup_{i\geq 0} G^i(\bot)$. We compute $G^i(\bot)$ for a few i.

$G^0(\bot) = \bot$

$G^1(\bot)\sigma_0 = COND(\mathcal{B}(b), SEQ(\mathcal{D}(S), \bot), \mathcal{D}($**skip**$))\sigma_0$

$$= \begin{cases} \emptyset & \text{if } \mathcal{B}(b)\sigma_0 \\ \mathcal{D}(\mathbf{skip})\sigma_0 & \text{if } \mathcal{B}(\neg b)\sigma_0 \end{cases}$$
$$= \begin{cases} \emptyset & \text{if } \mathcal{B}(b)\sigma_0 \\ \{\sigma_0\} & \text{if } \mathcal{B}(\neg b)\sigma_0 \end{cases} = \{\sigma_0 \mid \mathcal{B}(\neg b)\sigma_0\}$$

So this gives the final states that can be reached without executing S.

To compute $G^2(\bot)\sigma_0$ first consider
$SEQ(\mathcal{D}(S), G^1(\bot))\sigma_0 =$
$\{\sigma \mid \exists \sigma_1 : \sigma_1 \in \mathcal{D}(S)\sigma_0 \wedge \sigma \in G^1(\bot)\sigma_1\} = \{\sigma \mid \exists \sigma_1 : \sigma_1 \in \mathcal{D}(S)\sigma_0 \wedge \mathcal{B}(\neg b)\sigma_1\}$
So
$G^2(\bot)\sigma_0 = COND(\mathcal{B}(b), SEQ(\mathcal{D}(S), G^1(\bot)), \mathcal{D}(\mathbf{skip}))\sigma_0$

$$= \begin{cases} \{\sigma \mid \exists \sigma_1 : \sigma_1 \in \mathcal{D}(S)\sigma_0 \wedge \sigma = \sigma_1 \wedge \mathcal{B}(\neg b)\sigma_1\} & \text{if } \mathcal{B}(b)\sigma_0 \\ \{\sigma_0\} & \text{if } \mathcal{B}(\neg b)\sigma_0 \end{cases}$$
$$= \{\sigma \mid (\mathcal{B}(\neg b)\sigma_0 \wedge \sigma = \sigma_0) \vee (\exists \sigma_1 : \sigma_1 \in \mathcal{D}(S)\sigma_0 \wedge \sigma = \sigma_1 \wedge \mathcal{B}(\neg b)\sigma_1 \wedge \mathcal{B}(b)\sigma_0)\}$$
$$= \{\sigma \mid (\mathcal{B}(\neg b)\sigma_0 \wedge \sigma = \sigma_0) \vee (\sigma \in \mathcal{D}(S)\sigma_0 \wedge \mathcal{B}(b)\sigma_0 \wedge \mathcal{B}(\neg b)\sigma)\}$$

In general,

$$\mathcal{D}(\mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od})\sigma_0 = \{\sigma \mid \text{there exist } k \in \mathbb{N},\ \sigma_1, \ldots, \sigma_k \text{ such that } \sigma = \sigma_k,\ \mathcal{B}(\neg b)\sigma_k \text{ and for all } i \in \mathbb{N}, 0 \leq i < k\text{: } \mathcal{B}(b)\sigma_i \text{ and } \sigma_{i+1} \in \mathcal{D}(S)\sigma_i\}$$

Next we prove the equivalence of the operational and the denotational semantics:
$\mathcal{O}(S)\sigma_0 = \mathcal{D}(S)\sigma_0$
for any statement S and any state σ_0.

It is proved by two lemmas, using a few auxiliary lemmas.

Lemma 3.4 For any statement S and any state σ_0, $\mathcal{O}(S)\sigma_0 \subseteq \mathcal{D}(S)\sigma_0$.

Proof. Suppose $\sigma \in \mathcal{O}(S)\sigma_0$, thus $< S, \sigma_0 > \longrightarrow^* \sigma$, i.e., there exists a $k \in \mathbb{N}$ such that $< S, \sigma_0 > \longrightarrow^k \sigma$. We prove by induction on k that this implies $\sigma \in \mathcal{D}(S)\sigma_0$.
If $k = 0$ then it holds trivially (since the two configurations are not the same).
Next assume the property holds for $k \leq k_0$ and show that it holds for $k_0 + 1$.
We proceed by cases on how the first step of $< S, \sigma_0 > \longrightarrow^{k_0+1} \sigma$ was obtained.

- By [assign$_{sos}$]; then $k_0 = 0$, we have $\langle x := e, \sigma_0\rangle \longrightarrow (\sigma_0 : x \mapsto \mathcal{E}(e)\sigma_0)$ and $(\sigma_0 : x \mapsto \mathcal{E}(e)\sigma_0) \in \mathcal{D}(x := e)\sigma_0$.
- By [skip$_{sos}$]; similarly ($k_0 = 0$).
- By [seq$^1_{sos}$] or [seq$^2_{sos}$]. In both cases we have, for some k_0, $\langle S_1\ ;\ S_2, \sigma_0\rangle \longrightarrow^{k_0+1} \sigma$.
  By lemma **??**, there exist σ_1, k_1, and k_2 such that $\langle S_1, \sigma_0\rangle \longrightarrow^{k_1} \sigma_1$ and $\langle S_2, \sigma_1\rangle \longrightarrow^{k_2} \sigma$ with $k_0 + 1 = k_1 + k_2$. Since $k_1 \leq k_0$ and $k_2 \leq k_0$, we obtain by the induction hypothesis $\sigma_1 \in \mathcal{D}(S_1)\sigma_0$ and $\sigma \in \mathcal{D}(S_2)\sigma_1$. Hence $\sigma \in \mathcal{D}(S_1\ ;\ S_2)\sigma_0$.
- By [if$^T_{sos}$] ; then $\mathcal{B}(b)\sigma_0$ and we have
  $\langle \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}, \sigma_0\rangle \longrightarrow \langle S_1, \sigma_0\rangle \longrightarrow^{k_0} \sigma$.
  The induction hypothesis leads to $\sigma \in \mathcal{D}(S_1)\sigma_0$ and thus $\sigma \in \mathcal{D}(\mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi})\sigma_0$.
- By [if$^F_{sos}$] ; analogous.

- By [while$_{sos}$] ; then we have
  $\langle$**while** b **do** S **od**$, \sigma_0\rangle \longrightarrow \langle$**if** b **then** $(S$; **while** b **do** S **od**$)$ **else skip fi**$, \sigma_0\rangle \longrightarrow^{k_0} \sigma$.
  The induction hypothesis leads to
  $\sigma \in \mathcal{D}($**if** b **then** $(S$; **while** b **do** S **od**$)$ **else skip fi**$)\sigma_0$.
  Since **if** b **then** $(S$; **while** b **do** S **od**$)$ **else skip fi** is semantically equivalent to **while** b **do** S **od** (see lemma **??**), we obtain $\sigma \in \mathcal{D}($**while** b **do** S **od**$)\sigma_0$.

□

For the other direction, first two lemmas.

Lemma 3.5 $\mathcal{O}(S_1\,;\,S_2) = SEQ(\mathcal{O}(S_1), \mathcal{O}(S_2))$.

Lemma 3.6 $\mathcal{O}($**if** b **then** S_1 **else** S_2 **fi**$) = COND(\mathcal{B}(b), \mathcal{O}(S_1), \mathcal{O}(S_2)))$.

Proof. We have to show
$\mathcal{O}($**if** b **then** S_1 **else** S_2 **fi**$) = \begin{cases} \mathcal{O}(S_1)\sigma_0 & \text{if } \mathcal{B}(b)\sigma_0 \\ \mathcal{O}(S_2)\sigma_0 & \text{otherwise} \end{cases}$
Suppose $\mathcal{B}(b)\sigma_0$ then
$\mathcal{B}(b)\sigma_0$ and $\langle$**if** b **then** S_1 **else** S_2 **fi**$\rangle, \sigma_0 \longrightarrow^* \sigma$ iff
$\mathcal{B}(b)\sigma_0$ and $\langle S_1, \sigma_0\rangle \longrightarrow^* \sigma$ iff
$\mathcal{B}(b)\sigma_0$ and $\sigma \in \mathcal{O}(S_1)\sigma_0$.
Similarly for $\mathcal{B}(\neg b)\sigma_0$. □

Lemma 3.7 For any statement S and any state σ_0, $\mathcal{D}(S)\sigma_0 \subseteq \mathcal{O}(S)\sigma_0$.

Proof. We prove by induction on the structure of S that, for all σ_0, $\mathcal{D}(S)\sigma_0 \subseteq \mathcal{O}(S)\sigma_0$.
(That is, if $\sigma \in \mathcal{D}(S)\sigma_0$ then $\langle S, \sigma_0\rangle \longrightarrow^* \sigma$.)

- $S \equiv x := e$. Then $\sigma \in \mathcal{D}(x := e)\sigma_0$ implies $\sigma = (\sigma_0 : x \mapsto \mathcal{E}(e)\sigma_0)$.
  By [assign$_{sos}$] we have $\langle x := e, \sigma_0\rangle \longrightarrow (\sigma_0 : x \mapsto \mathcal{E}(e)\sigma_0) = \sigma$, so $\sigma \in \mathcal{O}(x := e)\sigma_0$.

- $S \equiv$ **skip**; similarly.

- $S \equiv S_1\,;\,S_2$. By the induction hypothesis and lemma 3.4 we obtain
  $\mathcal{O}(S_1) = \mathcal{D}(S_1)$ and $\mathcal{O}(S_2) = \mathcal{D}(S_2)$.
  Using lemma 3.5, $\mathcal{O}(S_1\,;\,S_2) = SEQ(\mathcal{O}(S_1), \mathcal{O}(S_2)) = SEQ(\mathcal{D}(S_1), \mathcal{D}(S_2)) = \mathcal{D}(S_1\,;\,S_2)$.

- $S \equiv$ **if** b **then** S_1 **else** S_2 **fi**. By the induction hypothesis and lemma 3.4 we obtain $\mathcal{O}(S_1) = \mathcal{D}(S_1)$ and $\mathcal{O}(S_2) = \mathcal{D}(S_2)$. Using lemma 3.6, $\mathcal{O}($**if** b **then** S_1 **else** S_2 **fi**$) = COND(\mathcal{B}(b), \mathcal{O}(S_1), \mathcal{O}(S_2))) = COND(\mathcal{B}(b), \mathcal{D}(S_1), \mathcal{D}(S_2))) = \mathcal{D}($**if** b **then** S_1 **else** S_2 **fi**$)$.

- S is of the form **while** b **do** S **od**. (NOTE the double use of S ..)
  Let $W \equiv$ **while** b **do** S **od**. The aim is to show $\mathcal{D}(W)\sigma_0 \subseteq \mathcal{O}(W)\sigma_0$, which can also be written as $\mathcal{D}(W) \preceq \mathcal{O}(W)$.
  Recall that $\mathcal{D}(W) = \mu.F$ for some function F.
  The idea is to use Exercise 1 of week 6 which gives that $F(X) \preceq X$ implies $\mu.F \preceq X$.
  Hence we have to prove $F(\mathcal{O}(W)) \preceq \mathcal{O}(W)$, where
  $F(D) = COND(\mathcal{B}(b),\ SEQ(\mathcal{D}(S), D),\ \mathcal{D}($**skip**$))$.

  $F(\mathcal{O}(W)) =$
  $COND(\mathcal{B}(b),\ SEQ(\mathcal{D}(S), \mathcal{O}(W)),\ \mathcal{D}($**skip**$)) \preceq$
  { by the induction hypothesis $\mathcal{D}(S) \preceq \mathcal{O}(S)$, using monotonicity of SEQ and $COND$ }
  $COND(\mathcal{B}(b),\ SEQ(\mathcal{O}(S), \mathcal{O}(W)),\ \mathcal{D}($**skip**$)) =$

{ it is easy to see that $\mathcal{D}(\textbf{skip}) = \mathcal{O}(\textbf{skip})$}
$COND(\mathcal{B}(b),\ SEQ(\mathcal{O}(S), \mathcal{O}(W)),\ \mathcal{O}(\textbf{skip})) =$
{ by lemma 3.5 }
$COND(\mathcal{B}(b),\ \mathcal{O}(S\,;\,W),\ \mathcal{O}(\textbf{skip})) =$
{ by lemma 3.6 }
$\mathcal{O}(\textbf{if } b \textbf{ then } (S\,;\,W) \textbf{ else skip fi}) = \mathcal{O}(W)$,
where the last equality follows from lemma **??**.

Since $\mathcal{D}(W)$ is the least fixed point we have by Exercise 1 of week 6, that $\mathcal{D}(W) \preceq \mathcal{O}(W)$, which by definition of the order implies $\mathcal{D}(\textbf{while } b \textbf{ do } S \textbf{ od})\sigma_0 \subseteq \mathcal{O}(\textbf{while } b \textbf{ do } S \textbf{ od})\sigma_0$.

□

Clearly, lemmas 3.4 and 3.7 imply

Theorem 3.8 For any statement S and any state σ_0, $\mathcal{O}(S)\sigma_0 = \mathcal{D}(S)\sigma_0$.

3.1 Extensions

Idea is mainly to show variations, and indicate that it is often far from trivial to define a denotational semantics.

3.1.1 Non-determinism

QUESTION: how to define it for S_1 **or** S_2?

$\mathcal{D}(S_1 \textbf{ or } S_2)\sigma_0 = \mathcal{D}(S_1)\sigma_0 \cup \mathcal{D}(S_2)\sigma_0$

Since we have operationally that $\langle S_1 \textbf{ or } S_2, \sigma_0\rangle \longrightarrow \sigma$ iff $\langle S_1, \sigma_0\rangle \longrightarrow \sigma$ or $\langle S_2, \sigma_0\rangle \longrightarrow \sigma$.

3.1.2 Parallelism

Can we find PAR such that

$\mathcal{D}(S_1 \| S_2) = PAR(\mathcal{D}(S_1), \mathcal{D}(S_2))$?

Problem: note that $x := 2 \sim (x := 1\,;\, x := x + 1)$, i.e., $\mathcal{D}(x := 2) = \mathcal{D}(x := 1\,;\, x := x + 1)$. Thus $\mathcal{D}(x := 2 \| S) = PAR(\mathcal{D}(x := 2), \mathcal{D}(S)) = PAR(\mathcal{D}(x := 1\,;\, x := x + 1), \mathcal{D}(S)) = \mathcal{D}((x := 1\,;\, x := x + 1) \| S)$, for any S.
QUESTION: is this OK?
NO, take $S \equiv x := 0$ then $\mathcal{D}(x := 2 \| x := 0)\sigma_0 = \{(\sigma_0 : x \mapsto 0), (\sigma_0 : x \mapsto 2)\}$ but $\mathcal{D}((x := 1\,;\, x := x + 1) \| x := 0)\sigma_0 = \{(\sigma_0 : x \mapsto 0), (\sigma_0 : x \mapsto 1), (\sigma_0 : x \mapsto 2)\}$.
Problem is that semantics identifies too many programs; the internal states are no longer visible ...
In this example $x := 2$ and $x := 1\,;\, x := x + 1$ should get a different semantics, because they might behave differently in a parallel contect.
To obtain a denotational semantics these internal states have to be made visible; the semantics should record these internal states in some way.
This is very much a research topic and I will not do this here.

Note that for other communication mechanisms (no shared variables) it is sometimes easier, e.g., for synchronous communication by channels.

3.1.3 Timing

There might be other reasons for identifying less programs.
E.g., when considering real-time properties of programs and one wants to reason about their execution time ..
Add a special variable *now* which denotes the current time; states then also give the value of this variable.
If $\sigma \in \mathcal{D}_{rt}(S)\sigma_0$ then there is a computation of S starting in state σ_0, so with starting time $\sigma_0(now)$ which terminates at time $\sigma(now)$.
The value of *now* is some time value, say in $\mathbb{R}$.

Using Parameters
Assume given a parameter T_a such that each assignment takes T_a time units, then we have the following semantics.

$$\mathcal{D}_{rt}(x := e)\sigma_0 = \{(\sigma_0 : x \mapsto \mathcal{E}(e)\sigma_0, now \mapsto \sigma_0(now) + T_a)\}$$

QUESTION: what about semantic equivalence compared to $\mathcal{D}$?
NOTE: less programs get the same semantics.
$x := x + 1\ ;\ x := x + 1$ and $x := x + 2$ are no longer equivalent.
But $x := x + 1\ ;\ x := x + 1$ is semantically equivalent to $x := x + 2 \times 7\ ;\ x := x - 2 \times 6$.
Maybe not so realistic; it might depend on e.

Using Fixed Evaluation Time of Expressions
So we could define $\mathcal{T}(e)\sigma_0 \in \mathbb{R}$; the time it takes to evaluate expression e in σ_0 and assign it to a variable. Then

$$\mathcal{D}_{rt}(x := e)\sigma_0 = \{(\sigma_0 : x \mapsto \mathcal{E}(e)\sigma_0, now \mapsto \sigma_0(now) + \mathcal{T}(e)\sigma_0)\}$$

Using Interval of Evaluation Times of Expressions
But maybe not so deterministic; suppose $\mathcal{T}(e)\sigma_0$ yields a set of possible values, e.g. an interval (evaluation of $x \times y + 7$ takes between 3 and 7 micro seconds ...). Then, e.g., $\mathcal{T}(e)\sigma_0 \subseteq \mathbb{R}$ and

$$\begin{aligned}\mathcal{D}_{rt}(x := e)\sigma_0 = \{\sigma \mid\ & \text{there exists a } t \in \mathcal{T}(e)\sigma_0 \text{ such that} \\ & \sigma = (\sigma_0 : x \mapsto \mathcal{E}(e)\sigma_0, now \mapsto \sigma_0(now) + t)\}\end{aligned}$$

We could keep the original definition of sequential composition and if-then-else, assuming no overhead.

4 Axiomatic Semantics

September 4, 2007

In an *axiomatic semantics* we characterize the meaning of a program in terms of a *specification* (a formula). We give a set of rules and axioms to be able to derive certain properties of programs.

Here we are interested in partial correctness properties of programs, expressing that if a program terminates then certain relations hold between initial and final state.

Example 4.1 How could be specify and verify such properties by means of the operational semantics?
Consider $W \equiv$ **while** $x > 1$ **do** S **od** with $S \equiv y := y \times x\ ;\ x := x - 1$ and suppose
$\langle y := 1\ ;\ W, \sigma_0\rangle \longrightarrow^* \sigma$.
Then σ represents the final state of a terminating computation.
QUESTION: What can we show?
$\sigma(y) = \sigma_0(x)!$.
HOW: idea, prove something for the while induction on length of the derivation.
We prove first:

Lemma 4.2 If $\langle W, \sigma_0\rangle \longrightarrow^k \sigma$ then $\sigma(y) = \sigma_0(y) \times \sigma_0(x)!$.

Then from
$\langle y := 1\ ;\ W, \sigma_0\rangle \longrightarrow \langle W, (\sigma_0 : y \mapsto 1)\rangle \longrightarrow^* \sigma$
we obtain by lemma 4.2 that $\sigma(y) = (\sigma_0 : y \mapsto 1)(y) \times (\sigma_0 : y \mapsto 1)(x)!$.

This is not so easy, very operational, with explicit induction. (Similarly for denotational semantics.)

Therefore often a programming language is characterized by an *axiomatic semantics*, i.e., a *set of axioms and rules that allow us to derive properties of programs.* So it expresses which specifications are satisfied by which programs.
Here we consider *partial correctness* properties of programs, i.e. properties that hold if the program terminates and use pre- and postcondition style specifications.

We describe partial correctness by a *Hoare triple*, a formula of the form
$\{p\}\ S\ \{q\}$ where

- p is an assertion called the *precondition*,
- S is a program, and
- q an assertion called the *postcondition.*

Informally, $\{p\}\ S\ \{q\}$ expresses that

> if p holds in the initial state of S, i.e., for the values of the variables at the start of the execution of S, then q holds for any final state of S, that is, if a computation of S terminates then q holds for the values of the variables at termination.

The assertions p and q are expressed in a first-order assertion language.

4.1 Assertion Language

The syntax of the assertion language is given in table 3, with $x \in VAR$, and $c \in CONST$.
Henceforth, we use the standard abbreviations, such as $true \equiv 0 = 0$, $p_1 \rightarrow p_2 \equiv \neg p_1 \vee p_2$, $p_1 \wedge p_2 \equiv \neg(\neg p_1 \vee \neg p_2)$, and $\forall x : p \equiv \neg\exists x : \neg p,$, etc. As usual, names of quantified variables are irrelevant. For instance, $\exists x : x < 5$ is considered to be the same formula as $\exists y : y < 5$.

Table 3: Syntax of the Assertion Language

Value Expression	$exp ::=$	$c \mid x \mid exp_1 + exp_2 \mid exp_1 - exp_2 \mid exp_1 \times exp_2$
Assertion	$p ::=$	$exp_1 = exp_2 \mid exp_1 < exp_2 \mid exp \in \mathbb{N} \mid$ $\neg p \mid p_1 \vee p_2 \mid \exists x : p$

4.2 Proof System for Sequential Programs

We present an axiomatic semantics in terms of a set of axioms and rules, also called a *proof system*, PS_{seq}. In this proof system we can derive formulae of the form $\{p\}\ S\ \{q\}$ by means of *axioms*, which allow the derivation of Hoare triples without any assumption, and *rules* of the form

$$\frac{\cdots,\ \{p_i\}\ S_i\ \{q_i\},\ \cdots,\ p_j \rightarrow q_k,\ \cdots}{\{p\}\ S\ \{q\}}$$

by which the formula below the line can be derived if we have derived all the formulae above the line. Observe that we allow two types of formulae above the line: Hoare triples, for which we give a proof system in the remainder of this section, and (implications between) assertions, for which we formulate an assumption in the next section.

Axiom 4.3 (Skip) $\{p\}$ **skip** $\{p\}$

Axiom 4.4 (Assignment) $\{q[e/x]\}\ x := e\ \{q\}$

NB. We do NOT have $\{p\}\ x := e\ \{p[e/x]\}$ try on $x := 1$

We used $q[e/x]$ to denote the substitution of e for each free occurrence of x in assertion q. To define this formally, first define $exp_0[exp/x]$ to denote the expression that is obtained by substituting exp for each occurrence of variable x in expression exp_0.

Definition 4.5 (Substitution in Value Expressions) Define $exp_0[exp/x]$ by

- $c[exp/x] \equiv c$
- $y[exp/x] \equiv \begin{cases} exp & \text{if } y \equiv x \\ y & \text{if } y \not\equiv x \end{cases}$
- $(exp_1 + exp_2)[exp/x] \equiv exp_1[exp/x] + exp_2[exp/x]$
- $(exp_1 - exp_2)[exp/x] \equiv exp_1[exp/x] - exp_2[exp/x]$
- $(exp_1 \times exp_2)[exp/x] \equiv exp_1[exp/x] \times exp_2[exp/x]$

Next we define $p[exp/x]$, denoting the substitution of each free occurrence of variable x by expression exp.

Definition 4.6 (Substitution in Assertions) Define $p[exp/x]$ as follows.

- $(exp_1 = exp_2)[exp/x] \equiv exp_1[exp/x] = exp_2[exp/x]$
- $(exp_1 < exp_2)[exp/x] \equiv exp_1[exp/x] < exp_2[exp/x]$
- $(exp_0 \in \mathbb{N})[exp/x] \equiv exp_0[exp/x] \in \mathbb{N}$
- $(\neg p)[exp/x] \equiv \neg(p[exp/x])$
- $(p_1 \vee p_2)[exp/x] \equiv (p_1[exp/x]) \vee (p_2[exp/x])$
- $(\exists y : p)[exp/x] \equiv \exists y' : (p[y'/y][exp/x])$ where $y' \notin var(p) \cup var(exp) \cup \{x\}$

In the last definition there is a subtle point, to avoid that variables in *exp* are bound by a quantification in p. For instance, we used y' to avoid that $(\exists y : x^2 + z = y)[y/x]$ leads to $\exists y : y^2 + z = y$. Now we obtain $(\exists y' : y^2 + z = y')$.

We should be able to change the precondition. Therefore we introduce a general rule which can be applied to any statement. With this rule the precondition of an already derived Hoare triple can be strengthened and the postcondition can be weakened.

Rule 4.7 (Consequence) $$\frac{p \to p_0,\ \{p_0\}\ S\ \{q_0\},\ q_0 \to q}{\{p\}\ S\ \{q\}}$$

Assuming that we can derive the valid implication $(x = 2 \wedge y = 3) \to (x + y = 5 \wedge y = 3)$, the consequence rule leads to $\{x = 2 \wedge y = 3\}\ x := x + y\ \{x = 5 \wedge y = 3\}$.

Rule 4.8 (Sequential Composition) $$\frac{\{p\}\ S_1\ \{r\},\ \ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$

Example 4.9 By the assignment axiom and the consequence rule we can derive
$\{x = 2\}\ y := 3\ \{x = 2 \wedge y = 3\}$, and $\{x = 2 \wedge y = 3\}\ x := x + y\ \{x = 5 \wedge y = 3\}$. Hence, the sequential composition rule leads to $\{x = 2\}\ y := 3\ ;\ x := x + y\ \{x = 5 \wedge y = 3\}$.

Rule 4.10 (if) $$\frac{\{p \wedge b\}\ S_1\ \{q\},\quad \{p \wedge \neg b\}\ S_2\ \{q\}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{q\}}$$

We can now derive
$\{x = v\}$ **if** $x = 0$ **then** $x := x + 1$ **else** $x := 1$ **fi** $\{x = 1\}$
by deriving the following $\{x = v \wedge x = 0\}x := x+1\{x = 1\}$ and $\{x = v \wedge x \neq 0\}x := 1\{x = 1\}$.

Rule 4.11 (While) $$\frac{\{p \wedge b\}\ S\ \{p\}}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}\ \{p \wedge \neg b\}}$$

Example 4.12 Consider again $W \equiv$ **while** $x > 1$ **do** $y := y \times x\ ;\ x := x - 1$ **od**. The aim is to show $\{x = v \wedge v \in \mathbb{N}\}\ y := 1\ ;\ W\ \{y = v!\}$.
By the sequential composition rule, we are done if we show
$\{x = v \wedge v \in \mathbb{N}\}\ y := 1\ \{x = v \wedge v \in \mathbb{N} \wedge y = 1\}$, which follows easily from the assignment axiom, and
$\{x = v \wedge v \in \mathbb{N} \wedge y = 1\}\ W\ \{y = v!\}$.
To prove this last formula by the while rule, define invariant $I \equiv (y \times x! = v! \wedge v \in \mathbb{N})$.
By the while rule (and consequence), we are now done if we prove
(1) $\{I \wedge x > 1\}\ y := y \times x\ ;\ x := x - 1\ \{I\}$
This follows by the sequential composition rule from the following two Hoare triples.

- $\{y \times x! = v! \wedge x > 1\}\ y := y \times x\ \{y \times (x - 1)! = v!\}$

- $\{y \times (x - 1)! = v!\}\ x := x - 1\ \{y \times x! = v!\}$

These two are derivable using the assignment rule and the consequence rule.

Let PS_{seq} be the proof system above, consisting of the axioms 4.3 and 4.4, and the rules 4.7, 4.8, 4.10, and 4.11.

4.2.1 Questions about soundness and completeness

Suppose we make small changes to the rules above; e.g. for assignment:

$$\{true\}\ x := e\ \{x = e\}$$

Allows us to derive $\{true\}\ x := x + 1\ \{x = x + 1\}$, so $\{true\}\ x := x + 1\ \{false\}$, This is not what we want

Or for if-rule

$$\frac{\{p \wedge b\}\ S_1\ \{q_1\},\quad \{p \wedge \neg b\}\ S_2\ \{q_2\}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{q_1 \vee q_2\}}$$

This rule is correct (derivable from the existing if rule, using consequence).

To answer such questions we give an interpretation of Hoare triples using the denotational semantics formulated earlier. So we give a relation between the axiomatic semantics and the denotational semantics in the next section.

4.3 Interpretation of Hoare Triples

We give a meaning to Hoare triples independent of the proof system.
The basic idea is to say that a triple $\{p\}\ S\ \{q\}$ is *valid* iff
if p holds in the initial state σ_0 and $\sigma \in \mathcal{D}(S)\sigma_0$ then q holds in σ.

Next we formalize what it means for an assertion to hold in a certain state.

Observe that an expression in the assertion language of table 3 is also an expression in the programming language of table **??**. Therefore the function $\mathcal{E}$ for expressions in the programming language is also used to obtain the value of expressions from the assertion language in a certain state. Next we define inductively when an assertion p holds in a state σ, denoted by $\sigma \models p$.
For instance, we want $(\sigma : x \mapsto 4) \models x + 3 < 8$, etc.

- $\sigma \models exp_1 = exp_2$ iff $\mathcal{E}(exp_1)\sigma = \mathcal{E}(exp_2)\sigma$
- $\sigma \models exp_1 < exp_2$ iff $\mathcal{E}(exp_1)\sigma < \mathcal{E}(exp_2)\sigma$
- $\sigma \models exp \in \mathbb{N}$ iff $\mathcal{E}(exp)\sigma \in \mathbb{N}$
- $\sigma \models \neg p$ iff not $\sigma \models p$
- $\sigma \models p_1 \vee p_2$ iff $\sigma \models p_1$ or $\sigma \models p_2$
- $\sigma \models \exists x : p$ iff there exists a $\mu \in D$ such that $(\sigma : x \mapsto \mu) \models p$

Example 4.13 $(\sigma : x \mapsto 7, y \mapsto 21) \models x \times y = 21$, since $\sigma(x) \times \sigma(y) = 21$ and
$(\sigma : y \mapsto 21) \models \exists x : x \times y = 21$, since we have the one above.

NOTE: a boolean expression b in the programming language is also an assertion and, e.g.,
$\mathcal{B}(x + y = 4)\sigma$ iff $\sigma(x) + \sigma(y) = 4$ iff $\sigma \models x + y = 4$.
In general, for all σ and b, we have $\mathcal{B}(b)\sigma$ iff $\sigma \models b$.

Definition 4.14 (Validity Assertions) An assertion p is *valid*, denoted by $\models p$, iff
$\sigma \models p$, for all $\sigma \in STATE$.

Example 4.15 $\models x + y = y + x$ and $\models x + x = 2 \times x$ (assuming first order arithmetic ...)

Next we define when a correctness formula $\{p\}\ S\ \{q\}$ is valid.

Definition 4.16 (Validity of a Correctness Formula) For a program S and assertions p and q, a correctness formula $\{p\}\ S\ \{q\}$ is *valid*, denoted by $\models \{p\}\ S\ \{q\}$, iff
for all $\sigma \in \mathcal{D}(S)\sigma_0$: if $\sigma_0 \models p$ then $\sigma \models q$.

Example 4.17 $\models \{x = 3 \wedge y = 4\}\ x := x + y\ \{x = 7 \wedge y = 3\}$, since
$\sigma_0 \models x = 3 \wedge y = 4$ implies $\sigma_0(x) = 3$ and $\sigma_0(y) = 4$, and thus
$\sigma \in \mathcal{D}(x := x + y)\sigma_0$ implies $\sigma = (\sigma_0 : x \mapsto \sigma_0(x) + \sigma_0(y)) = (\sigma_0 : x \mapsto 3 + 4) = (\sigma_0 : x \mapsto 7)$.

Recall that $\mathcal{D}(S)$ represents only the terminating computations of S, and hence validity of $\{p\}\ S\ \{q\}$ does not require termination of S and it does not specify anything about nonterminating computations of S, i.e. only expresses partial correctness.
Note, for instance, that $\models \{true\}$ **while** $true$ **do skip od** $\{false\}$.

4.4 Soundness and Relative Completeness

We write $\vdash \{p\}\ S\ \{q\}$ if formula $\{p\}\ S\ \{q\}$ can be derived in proof system PS_{seq}. For such a proof system there are two basic questions that have to be considered:

- Is the proof system *sound*, that is, is every formula that can be derived indeed valid? Formally, the soundness of the proof system is expressed as follows:
  if $\vdash \{p\}\ S\ \{q\}$ then $\models \{p\}\ S\ \{q\}$.
- Is the proof system *complete*, i.e., is it possible to derive every valid formula? Formally, if $\models \{p\}\ S\ \{q\}$ then $\vdash \{p\}\ S\ \{q\}$.

4.5 Soundness

Theorem 4.18 Proof system PS_{seq} is sound, that is, $\vdash \{p\}\ S\ \{q\}$ implies $\models \{p\}\ S\ \{q\}$.

Proof. Soundness is proved by induction on the derivation. We have to show that every axiom is valid and that all rules preserve validity. We only give a few cases here and leave the other cases to exercise 4.32.

- We have to show that the assignment axiom is sound, i.e., $\models \{q[e/x]\}\ x := e\ \{q\}$.
  Consider $\sigma \in \mathcal{D}(x := e)\sigma_0$ and suppose $\sigma_0 \models q[e/x]$.
  Then $\sigma = (\sigma_0 : x \mapsto \mathcal{E}(e)\sigma_0)$. It remains to prove that $\sigma_0 \models q[e/x]$ implies
  $(\sigma_0 : x \mapsto \mathcal{E}(e)\sigma_0) \models q$ which is done by lemma 4.21 below.

  The next lemmas can be proved by induction on the structure of expressions and assertions.

  Lemma 4.19 (Substitution Expressions)
  $\mathcal{E}(exp_0[exp/x])\sigma = \mathcal{E}(exp_0)(\sigma : x \mapsto \mathcal{E}(exp)\sigma)$

  Example 4.20 Suppose $\sigma(y) = 4$, then
  $\mathcal{E}(x + 1[y + 3/x])\sigma = \mathcal{E}(x + 1)(\sigma : x \mapsto \mathcal{E}(y + 3)\sigma) = \mathcal{E}(x + 1)(\sigma : x \mapsto 7) = 7 + 1 = 8$.

  Lemma 4.21 (Substitution Assertions)
  $\sigma \models p[exp/x]$ iff $(\sigma : x \mapsto \mathcal{E}(exp)\sigma) \models p$

Example 4.22 Suppose $\sigma(y) = 4$, then
$\sigma \models (x+1=8)[y+3/x]$ iff $(\sigma : x \mapsto \mathcal{E}(y+3)\sigma) \models x+1=8$ iff $(\sigma : x \mapsto 7) \models x+1=8$ which clearly holds.
This is OK since $\sigma \models (x+1=8)[y+3/x]$ iff $\sigma \models y+3+1=8$ which also holds since $\sigma(y) = 4$.

- Assume $\models \{p \wedge b\}\ S\ \{p\}$.
  We have to show $\models \{p\}$ **while** b **do** S **od** $\{p \wedge \neg b\}$.
  Consider $\sigma \in \mathcal{D}(\textbf{while } b \textbf{ do } S \textbf{ od})\sigma_0$ and suppose $\sigma_0 \models p$.
  By the alternative formulation of the semantics, then there exist $k \in \mathbb{N}$, $\sigma_1, \ldots, \sigma_k$ such that $\sigma = \sigma_k$, $\mathcal{B}(\neg b)\sigma_k$ and for all $i \in \mathbb{N}, 0 \leq i < k$: $\mathcal{B}(b)\sigma_i$ and $\sigma_{i+1} \in \mathcal{D}(S)\sigma_i$.

  We prove by induction on i that $\sigma_i \models p$, for $i \leq k$.

  (Basic Step) For $i = 0$ we have by our assumption $\sigma_0 \models p$.

  Induction Step) Let $i+1 \leq k$ and show $\sigma_{i+1} \models p$. By the induction hypothesis, $\sigma_i \models p$. Since $i < k$, we have $\mathcal{B}(b)\sigma_i$, thus $\sigma_i \models p \wedge b$. Since $\sigma_{i+1} \in \mathcal{D}(S)\sigma_i$, we obtain by $\models \{p \wedge b\}\ S\ \{p\}$ that $\sigma_{i+1} \models p$.

  Thus we have, for $i = k$, $\sigma_k \models p$. Since $\mathcal{B}(\neg b)\sigma_k$, this leads to $\sigma_k \models p \wedge \neg b$.

□

The proof system PS_{seq} is sound.

Definition 4.23 *We can also say that* a (new) rule is sound. *Suppose we have the following rule.*

$$\frac{\cdots,\ \{p_i\}\ S_i\ \{q_i\},\ \cdots,\ p_j \rightarrow q_k,\ \cdots}{\{p\}\ S\ \{q\}}$$

This rule is sound *if*
$(\cdots \wedge \models \{p_i\}\ S_i\ \{q_i\} \wedge \cdots \wedge \models p_j \rightarrow q_k \wedge \cdots) \Rightarrow \models \{p\}\ S\ \{q\}$.
That is, if all the premises are valid *(in the semantics), then the conclusion must be* valid.

NB The soundness proof of 4.18 consists of just proving all rules and axioms of PS_{seq} to be sound in this sense.

We can prove soundness of a new rule in two ways.

- Use the denotational semantics (like in the proof of 4.18).
- Show that a new rule is a *derived* rule in PS_{seq}, that is, it can be derived from the rules in PS_{seq}. Given that PS_{seq} is sound, then also the new rule is sound.

As an example, consider the following rule.

Rule 4.24 (While-Invariant) $\dfrac{\{p\}\ S\ \{p\}}{\{p\}\ \textbf{while } b \textbf{ do } S \textbf{ od}\ \{p\}}$

Seems a good rule, can be proved to be sound by deriving it from the other rules in PS_{seq}.
Suppose we have $\vdash \{p\}\ S\ \{p\}$.
Since $p \wedge b \rightarrow p$, the consequence rule leads to
$\vdash \{p \wedge b\}\ S\ \{p\}$.

By the while rule, rule 4.11, we obtain
$\vdash \{p\}$ **while** b **do** S **od** $\{p \wedge \neg b\}$.
Hence by the consequence rule,
$\vdash \{p\}$ **while** b **do** S **od** $\{p\}$.

NB. we did not use semantics here, this method does not always work ...

4.6 Relative Completeness

For the proof of completeness, observe that in the hypothesis of the consequence rule we have implications between assertions.

Example 4.25 Can we derive $\{\forall v : v \times y = v\}\ x := y + 1\ \{x = 2 \wedge y = 1\}$?
By the assignment axion: $\vdash \{y = 1\}\ x := y + 1\ \{x = 2 \wedge y = 1\}$.
Do we have $\vdash \forall v : v \times y = v \rightarrow y = 1$?

Hence to derive a Hoare triple with this rule we have to derive assertions.
Since we want to concentrate on the axiomatization of programming language constructs, we do not give a proof system for assertions.
Furthermore, by Gödel's incompleteness result, a sound and complete proof system need not exist for the assertion language.

Basic idea is that in first-order arithmetic, thus quantification, $+$ and $\times$ on $\mathbb{N}$ we cannot find a proof system which is complete, thus in which every valid formula can be derived.
Gödel showed this by constructing, using the so-called Gödel numbering, a formula which expresses "this formula is not derivable".
If this formula is derivable then it is not true and we have an unsound proof system;
if it is not derivable then it is true, and the proof system is not complete.

Hence, one usually proves *relative* completeness, that is, the proof that $\models \{p\}\ S\ \{q\}$ implies $\vdash \{p\}\ S\ \{q\}$ is relative to a theory (in the formal sense of logic) for assertions. Therefore, we add an infinite list of axioms to our proof system, namely all valid assertions.

(Relative Completeness Assumption) We consider as axioms all formulas p for which $\models p$.

Let PS^{+}_{seq} be the proof system consisting of PS_{seq} and all valid assertions as axioms.
To prove completeness of PS^{+}_{seq}, and thus relative completeness of PS_{seq}, we introduce the notion of a strongest postcondition.

Definition 4.26 (Strongest Postcondition) An assertion q is a *strongest postcondition* of a precondition p and a statement S if

a) $\models \{p\}\ S\ \{q\}$, and

b) for all r, if $\models \{p\}\ S\ \{r\}$ then $\models q \rightarrow r$.

First we show that for every p and statement S there exists an assertion q which is a strongest postcondition of p and S.

Lemma 4.27 For every p and statement S there exists an assertion which is a strongest postcondition of p and S.

Proof. The proof is complicated, especially for the while-construct. We don't do it. □

Theorem 4.28 Proof system PS^{+}_{seq} is complete, that is if $\models \{p\}\ S\ \{q\}$ then $\vdash \{p\}\ S\ \{q\}$.

Proof. Not done. □

4.7 Exercises

Exercise 4.29 Let $var(exp)$ denote the set of variables occurring in expression exp. For an assertion p, $var(p)$ is the set of variables that occur free, i.e. not bound by any quantification, in p. These definitions can easily be made formal, e.g. $var(\exists x : p) = var(p) - \{x\}$. Prove the following properties.

a) If $x \equiv y$ then $(\exists y : p)[exp/x]$ iff $\exists y : p$.

b) If $x \not\equiv y$ and $y \notin var(exp)$ then $(\exists y : p)[exp/x]$ iff $\exists y : (p[exp/x])$.

c) If $x \notin var(p)$ then $p[exp/x]$ iff p.

Exercise 4.30 Prove the following two properties.

a) If $\sigma_1(x) = \sigma_2(x)$, for all $x \in var(exp)$, then $\mathcal{E}(exp)\sigma_1 = \mathcal{E}(exp)\sigma_2$.

b) If $\sigma_1(x) = \sigma_2(x)$, for all $x \in var(p)$, then $\sigma_1 \models p$ iff $\sigma_2 \models p$.

Exercise 4.31 Prove the lemmas 4.19 and 4.21.

Exercise 4.32 Complete the proof of theorem 4.18, showing that proof system PS_{seq} is sound.

Exercise 4.33 We consider an alternative for the assignment axiom.
$\{p\}\ x := e\ \{\exists v : p[v/x] \wedge (x = e[v/x])\}$
where v is fresh, i.e., not occurring in p and $x := e$.
Prove that this formula is valid in two ways:

a) using the interpretation of Hoare triples (definition 4.16),

b) by showing that the new axiom can be derived from proof system PS_{seq} (which is sound).

Exercise 4.34 For each of the following rules, either prove soundness or give an example to show that the rule is not sound.

a) $$\frac{\{p\}\ S\ \{q\},\ \ q \wedge b \rightarrow p,\ \ q \wedge \neg b \rightarrow r}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}\ \{r\}}$$

b) $$\frac{\{p\}\ S\ \{q\},\ \ q \wedge b \rightarrow p,\ \ q \wedge \neg b \rightarrow r}{\{q\}\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}\ \{r\}}$$

Exercise 4.35 Let **repeat** S **until** $b \equiv S$; **while** $\neg b$ **do** S **od**. Prove soundness of the following rule.

Rule 4.36 (Repeat) $$\frac{\{p\}\ S\ \{q\},\quad q \wedge \neg b \rightarrow p}{\{p\}\ \ \textbf{repeat}\ S\ \textbf{until}\ b\ \ \{q \wedge b\}}$$

Exercise 4.37 Prove that the conjunction rule is sound.

Rule 4.38 (Conjunction) $$\frac{\{p_1\}\ S\ \{q_1\},\quad \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

Exercise 4.39 Prove soundness of the following invariance axiom.

Axiom 4.40 (Invariance) $\{p\}\ S\ \{p\}$

provided S does not contain any of the variables in $var(p)$.

Exercise 4.41 Let $x :=?$ denote a *random assignment* which assigns a random value to x.

a) Extend the denotational semantics to random assignments.

b) Extend the proof system with an axiom for random assignments.

c) Prove soundness and relative completeness of the extended proof system.

Exercise 4.42 Define, for an assertion p, $[\![p]\!] = \{\sigma \in \mathit{STATE} \mid \sigma \models p\}$. Prove

a) $[\![\neg p]\!] = \mathit{STATE} - [\![p]\!]$

b) $[\![p_1 \vee p_2]\!] = [\![p_1]\!] \cup [\![p_2]\!]$

c) $[\![p_1 \wedge p_2]\!] = [\![p_1]\!] \cap [\![p_2]\!]$

d) $\models p_1 \to p_2$ iff $[\![p_1]\!] \subseteq [\![p_2]\!]$

e) $\models p_1 \leftrightarrow p_2$ iff $[\![p_1]\!] = [\![p_2]\!]$

Define, for a set of states $\Sigma \subseteq \mathit{STATE}$,

$$\mathcal{D}^\dagger(S)\Sigma = \{\sigma \mid \text{ there exists a } \sigma_0 \in \Sigma \text{ such that } (\sigma_0, \sigma) \in \mathcal{D}(S)\ \}.$$

Prove

f) $\models \{p\}\ S\ \{q\}$ iff $\mathcal{D}^\dagger(S)[\![p]\!] \subseteq [\![q]\!]$