Advanced Topics in Programming Languages **Spring Semester, 2012**

Lecture 5: April 6, 2012

Lecturer: Prof. Mooly Sagiv *Scribe: Alexander Matveev and Ariel Jarovsky*

Denotational Semantics

5.1 Introduction

5.1.1 A brief overview

In the previous lecture, we learned about operational semantics which describe the behavior of programs by inductively defining transition relations to express evaluation and execution. Denotational semantics are a more abstract level of semantics. They take the meaning of a command to be a partial function on states:

- $\mathbf{A} : Aexp \rightarrow (\Sigma \rightarrow \mathbb{N})$
- $\mathbf{B} : Bexp \rightarrow (\Sigma \rightarrow \mathbb{T})$
- $\mathbf{S} : Stm \rightarrow (\Sigma \rightarrow \sigma)$

All these are defined by structural induction as described below.

Denotational semantics of Aexp

We can express the semantics as sets of using lambda calculus.
First, we present the semantics using sets:

- $\mathbf{A} : Aexp \rightarrow (\Sigma \rightarrow \mathbb{N})$
- $\mathbf{A}[\![n]\!] = \{(\sigma, n) | \sigma \in \Sigma\}$
- $\mathbf{A}[\![X]\!] = \{(\sigma, \sigma X) | \sigma \in \Sigma\}$
- $\mathbf{A}[\![a_0 + a_1]\!] = \{(\sigma, n_0 + n_1) | (\sigma, n_0) \in \mathbf{A}[\![a_0]\!], (\sigma, n_1) \in \mathbf{A}[\![a_1]\!]\}$
- $\mathbf{A}[\![a_0 - a_1]\!] = \{(\sigma, n_0 - n_1) | (\sigma, n_0) \in \mathbf{A}[\![a_0]\!], (\sigma, n_1) \in \mathbf{A}[\![a_1]\!]\}$
- $\mathbf{A}[\![a_0 * a_1]\!] = \{(\sigma, n_0 * n_1) | (\sigma, n_0) \in \mathbf{A}[\![a_0]\!], (\sigma, n_1) \in \mathbf{A}[\![a_1]\!]\}$

Lemma: $\mathbf{A}[\![a]\!]$ is a function.

Now, let's look at the semantics of *Aexp* with λ:

- $\mathbf{A} : Aexp \to (\Sigma \to \mathbb{N})$
- $\mathbf{A}[\![n]\!] = \lambda\sigma \in \Sigma.n$
- $\mathbf{A}[\![X]\!] = \lambda\sigma \in \Sigma.\sigma(X)$
- $\mathbf{A}[\![a_0 + a_1]\!] = \lambda\sigma \in \Sigma.(\mathbf{A}[\![a_0]\!]\sigma + \mathbf{A}[\![a_1]\!]\sigma)$
- $\mathbf{A}[\![a_0 - a_1]\!] = \lambda\sigma \in \Sigma.(\mathbf{A}[\![a_0]\!]\sigma - \mathbf{A}[\![a_1]\!]\sigma)$
- $\mathbf{A}[\![a_0 * a_1]\!] = \lambda\sigma \in \Sigma.(\mathbf{A}[\![a_0]\!]\sigma * \mathbf{A}[\![a_1]\!]\sigma)$

As we can see, the definitions are equivalent.

Denotational semantics of Bexp

We will just show the semantics using sets.

- $\mathbf{B} : Bexp \to (\Sigma \to \mathbb{T})$
- $\mathbf{B}[\![true]\!] = \{(\sigma, true) | \sigma \in \Sigma\}$
- $\mathbf{B}[\![false]\!] = \{(\sigma, false) | \sigma \in \Sigma\}$
- $\mathbf{B}[\![a_0 = a_1]\!] = \{(\sigma, true) | \sigma \in \Sigma \wedge \mathbf{A}[\![a_0]\!]\sigma = \mathbf{A}[\![a_1]\!]\sigma\} \cup \{(\sigma, false) | \sigma \in \Sigma \wedge \mathbf{A}[\![a_0]\!]\sigma \neq \mathbf{A}[\![a_1]\!]\sigma\}$
- $\mathbf{B}[\![a_0 \leq a_1]\!] = \{(\sigma, true) | \sigma \in \Sigma \wedge \mathbf{A}[\![a_0]\!]\sigma \leq \mathbf{A}[\![a_1]\!]\sigma\} \cup \{(\sigma, false) | \sigma \in \Sigma \wedge \mathbf{A}[\![a_0]\!]\sigma > \mathbf{A}[\![a_1]\!]\sigma\}$

- $\mathbf{B}[\![\neg b]\!] = \{(\sigma, \neg_T t) | (\sigma, t) \in \mathbf{B}[\![b]\!]\}$
- $\mathbf{B}[\![b_0 \wedge b_1]\!] = \{(\sigma, t_0 \wedge_T t_1) | (\sigma, t_0) \in \mathbf{B}[\![b_0]\!], (\sigma, t_1) \in \mathbf{B}[\![b_1]\!]\}$
- $\mathbf{B}[\![b_0 \vee b_1]\!] = \{(\sigma, t_0 \vee_T t_1) | (\sigma, t_0) \in \mathbf{B}[\![b_0]\!], (\sigma, t_1) \in \mathbf{B}[\![b_1]\!]\}$

Denotational semantics of a Statement

Denotational semantics of statements are a bit more complicated (and the main goal of denotational semantics). In the next section we present some fundamental question that arise when trying to express the semantics of a statement and then we will present the solution proposed by denotational semantics using *Domain Theory*.

5.1.2 Denotational semantics of a Statement

Running a statement s starting from a state σ yields another state σ'. Thus, we define the semantics of a statement to be:

$$\mathbf{S} : Stm \to (\Sigma \to \Sigma)$$

This notation is problematic a little bit, due to the fact that it doesn't handle not-terminating statements. To handle such statements, we added a state named $\perp$ ("bottom"), to denote a special outcome for non-terminating commands. We also used the following two conventions:

- For any set X, we write $X_\perp$ for $X \cup \perp$.
- Whenever $f \in X \to X$,, we extend f to $f \in X_\perp \to X_\perp$ such that $f(\perp) = \perp$. For a statement, we defined $\mathbf{S}[\![\cdot]\!]$ to be a function from $\Sigma_\perp$ to $\Sigma_\perp$.

Problems with the 'while' statement semantics

How can we define a loop semantics, i.e., what is $\mathbf{S}[\![$while b do s$]\!]$? We will use the following equivalence:

$$W(\sigma) = \text{ if } \mathbf{B}[\![b]\!] \text{ then } W(\mathbf{S}[\![s]\!]\sigma) \text{ else } \sigma$$

This is not well defined mathematically: W is defined in terms of itself and we don't know whether such a function W exists. Even if it does, we don't know if such a function is unique (e.g., while true do skip).

5.2 Domain Theory

We will use mathematics to show that the semantics of while are well defined. We will think about $W(\sigma) = \text{ if } \mathbf{B}[\![b]\!] \text{ then } W(\mathbf{S}[\![s]\!]\sigma) \text{ else } \sigma$ as a recursive equation.

An example. Think about programs as processors of streams of bits (streams of 0's and 1's terminated by \$). Which properties can we expect?
Let *isone* be a function that returns 1\$ when the input string has at least a 1, and 0\$ otherwise. Let's look at some examples:

- isone(00...00\$) = 0\$
- isone(xx..1..\$) = 1\$
- isone(0...0) =?

The last case is the interesting one and it catches the first property we want to introduce. Any program describe a function between it's input and it's output. For programs dealing with streams we do not just require that the output must be a function of the input, but that for any step in the program the (partial) output must be a function of the (partial) input. If we reason about the isone example, the only possible partial output for isone(0...0) is the empty string ϵ. This property is called *motonicity*.

5.2.1 Motonicity

Informally, motonicity requires that more information about the input is reflected in more information about the output. In order to express this precisely we are going to present some mathematical background in *Domain Theory*.

Definition (Partial Order): A relation $\sqsubseteq$ is a *partial order* if it is a refelxive, transitive and anti-symmetric relation. We write $x \sqsubseteq y$ if and only if x is in a relation with y and we say that x is "smaller" or "less than" y.
Examples:

- For 2 streams of bits, x, y, $x \sqsubseteq y$ if and only if x is a prefix of y. For example: 0001111 is a prefix of 0001111111111110001, and 0 if not a prefix of 1111.

- For programs, we use the following relation: No output is *less than* some output (other outputs are not related one to the other). Pay attention that non-terminating programs have no output. Thus, their output is smaller than the output of a terminating program.

Definition (Poset): A set equipped with a partial order is a *poset*.

Definition (Motonicity): Let D, E be posets. A function $f : D \to E$ is monotonic if and only if $\forall x, y \in D. x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.

Examples:

1. Let $D, E = \mathbb{N}$ and $\sqsubseteq = \leq$, then any linear growing function $f : \mathbb{N} \Rightarrow \mathbb{N}, f(x) = ax + b\,(a \geq 0)$ is monotonic. Let $x \leq y$, then $f(y) - f(x) = (ay + b) - (ax + b) = a(y - x) \leq 0 \Rightarrow f(x) \leq f(y)$.

2. Recall the isone example. Let's show that isone$(0^k) = \epsilon$. $0^k \sqsubseteq 0^k\$$, $0^k \sqsubseteq 0^k 1\$$. Since isone must be monotone it follows that isone$(0^k) \sqsubseteq$ isone$(0^k\$) = 0$, isone$(0^k) \sqsubseteq$ isone$(0^k 1\$) = 1$, then isone$(0^k) = \epsilon$.

.

5.2.2 Chains

Definition (Chain):

- Let $(X, \sqsubseteq)$ be a poset. A chain is a countable increasing sequence $< x_i >= \{x_i \in X | x_0 \sqsubseteq x_1 \sqsubseteq \ldots\}$
- An upper bound in a chain is an element which is "bigger" than all of the other elements in the chain.
- The *least upper bound* is the "smallest" among all of the upper bounds (we write $\bigsqcup <x_i>$, i.e. $\forall i \in \mathbb{N}.x_i \sqsubseteq \bigsqcup <x_i>$ and $\bigsqcup <x_i> \sqsubseteq y$ for any other upper bound y.

Claim 5.1 *For any upper bounded chain, there exists a unique least upper bound.*

5.2.3 Complete Partial Orders

Definition (Complete Partial Order):

- We say that a partial order P is complete if every chain in P has a least upper bound, also in P. We call such P is a *cpo* (complete partial order).
- A cpo with a least upper bound (or "bottom") element, $\bot$, is called a *pointed cpo* (or *pcpo*). $\bot$ has to be "smaller" than any other element in the order P.

Examples:

1. Any set S with the order P such that $\forall x, y \in S.x \sqsubseteq y \leftrightarrow x = y$ is a cpo. In this case, each chain contains a singe element (maybe with repetitions). Thus, for each chain, the least (upper) bound is the only element on it.
2. Adding a $\bot$ element to the last order P, which satifies $\bot \sqsubseteq x$ for every x, makes P a *pcpo*.
3. The natural numbers $\mathbb{N}$ with the order $\leq$ is not a *cpo*. This is because the chain $\{0 \leq 1 \leq 2 \leq \ldots\}$ has no upper bound.
4. The natural number and infinity (∞) with the order $\leq$ is a *pcpo*. It's bottom element is 0 and every chain has an upper bound, ∞.
5. Let S be a set, and $P(S)$ be it's power set. The order $\subseteq$ is a pcpo on $P(S)$: let C be a chain, an upper bound for C is S ($\forall A \in P(S).A \subseteq S$, thus S is an upper bound of every set in $P(S)$. The bottom element is $\emptyset \in P(S)$.

Constructing Complete Partial Order

Suppose D, E are pointed cpo's, then $D \times E$ is also a cpo. The induced order is given by:

$$(x, y) \sqsubseteq_{D\times E} (x', y') \Leftrightarrow x \sqsubseteq_D x' \wedge y \sqsubseteq_E y'$$

The "bottom" element is $\bot_{D\times E} = (\bot_D, \bot_E)$. For a pair of chains x_i, y_i we define the "join" (least upper bound) of the pair of chains as: $\bigsqcup_{D\times E}(x_i, y_i) = (\bigsqcup_D x_i, \bigsqcup_E y_i)$.

The pointwise induced relation

Suppose we have a program with variables and we don't want to bound the number of them. We define the point induced relation which maps the set of variables (S) to the set values E - a pcpo - in the following way:

$$m \sqsubseteq m' \Leftrightarrow \forall s \in S.m(s) \sqsubseteq m'(s)$$

We also define the minimal element of the relation by: $\bot_{S\to E} = \lambda s.\bot_E$. And finally, the "join" of two maps is given by the following formula: $\bigsqcup(m, m') = \lambda s.m(s) \bigsqcup_E m'(s)$.

5.2.4 Continuity

Claim 5.2 *Let $f : D \to E$ be a monotonic function, then f maps a chain of elements in D to a chain of elements in E, i.e.:*

$$x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots \Rightarrow f(x_0) \sqsubseteq f(x_1) \sqsubseteq f(x_2) \sqsubseteq \ldots$$

Continuity means that the output generated using an infinite view of the input does not contain more information than all of the output based on finit inputs.
Scott's thesis claims that the semantics of programs can be described by a continuous function.

Claim 5.3 $\bigsqcup_i \langle f(x_i)\rangle \sqsubseteq f(\bigsqcup_i \langle x_i\rangle)$.

Proof: From the definition of the upper bound it follows that $\forall x \in \langle x_i\rangle.x \sqsubseteq \bigsqcup_i \langle x_i\rangle$.
From the monotonicity of f follows that $\forall x \in \langle x_i\rangle.f(x) \sqsubseteq f(\bigsqcup_i \langle x_i\rangle)$.
Then, $f(\bigsqcup_i \langle x_i\rangle)$ is an upper bound for any element in the chain $\langle f(x_i)\rangle$, and so it is "grater" that the least upper bound, i.e. $\bigsqcup_i \langle f(x_i)\rangle \sqsubseteq f(\bigsqcup_i \langle x_i\rangle)$

■

Note that the inverse is not always true, i.e., it is not always the case that $f(\bigsqcup_i \langle x_i \rangle) \sqsubseteq \bigsqcup_i \langle f(x_i) \rangle$.
For example, let's take $D = E = \{(x, y) \,|\, x, y \in \mathbb{N} \cup \{\infty\} \wedge x \leq y\}$. Also, let $f : \mathbb{N} \to \mathbb{N}$ be $\forall x \in \mathbb{N}. f(x) = 1$ and $f(\infty) = \infty$. Now, let's look at the chain $\langle x_i \rangle = \mathbb{N}$. Then, $f(\bigsqcup \langle x_i \rangle) = f(\infty) = \infty$, but $\bigsqcup \langle f(x_i) \rangle = \bigsqcup \langle 1 \rangle = 1$.
So we got that $f(\bigsqcup \langle x_i \rangle) = \infty \neq 1 = \bigsqcup \langle f(x_i) \rangle$, i.e., there is a chain that does not fulfill the continuity condition. Thus, f is discontinuous.

Definition A function $f : D \to E$ is continuous iff for every chain of the range $\langle x_i \rangle$:

- $\bigsqcup_i \langle f(x_i) \rangle \sqsubseteq f(\bigsqcup_i \langle x_i \rangle)$.
- $f(\bigsqcup_i \langle x_i \rangle) \sqsubseteq \bigsqcup_i \langle f(x_i) \rangle$

Claim 5.4 *Every continuous function is monotonic.*

Examples of Continuos Functions

For the partial order $(\mathbb{N} \cup \{\infty\}, \leq)$:

1. The identity function $(id(\bigsqcup n_i) = \bigsqcup id(n_i))$
2. A constant function $const(n) = k$
3. If isone$(0^\infty) = \epsilon$, then isone is continuous.

Claim 5.5 *For a flat cpo A, any monotonic function $f : A_\perp \to A_\perp$, such that f is strict, is continuous.*

Proof:
Since A is a flat cpo, it follows that $\forall a_1, a_2. a_1 \sqsubseteq a_2 \Leftrightarrow a_1 = \perp \vee a_1 = a_2$.
Let $\langle x_i \rangle$ be a chain and let $\bigsqcup \langle x_i \rangle = T_i$.
A is flat, then $T_i \in \langle x_i \rangle$.
Since f is monotonic, it follows that $\forall a \in \langle x_i \rangle. f(a) \sqsubseteq f(T_i)$.
Then, $\bigsqcup \langle f(x_i) \rangle = f(T_i) = f(\bigsqcup \langle x_i \rangle)$, so f is continuous.

■

5.3 Fixed Point Theorem

Let us recall that we want to show the validity of the semantics of the while statement. We saw that the semantics of while were defined recursively. This leads us to the need of solving a recursive equation. This is equivalent to find a fixed point to the equation $F(W) = W$.

We want that the solution to be continuous. This will ensure our ability to prove correctness about composed programs using structural induction (examples can be found at the end of the chapter).

5.3.1 The Fixed Point Theorem

Theorem 5.6 *Let D be a pcpo and $F : D \to D$ be a continuous function. Let $F^k = \lambda x.F(F(\ldots(F(x))\ldots))$ be F composed on itself k times, then:*

1. *For any fixed point x of F and $k \in \mathbb{N}$: $F^k(\bot) \sqsubseteq x$.*

2. *The least of all fixed points is $\bigsqcup_k F^k(\bot)$.*

In our case F will be a state transformer function: $F : [[\Sigma_\bot \to \Sigma_\bot] \to [\Sigma_\bot \to \Sigma_\bot]]$.

Proof:

1. By induction on k:

   - Base: $F^0(\bot) = \bot \sqsubseteq x$ - follows immediately since we use a pointed cpo.
   - Induction Step: $F^{k+1}(\bot) = F(F^k(\bot)) \sqsubseteq F(x) = x$ - follows from continuity (and monotonicity) of F and from the induction assumption.

2. It suffices to show that $\bigsqcup_k F^k(\bot)$ is a fix point.

   - $F(\bigsqcup_k F^k(\bot)) = \bigsqcup_k F^{k+1}(\bot)$ - follows by the conitnuity of F.
   - $\bigsqcup_k F^{k+1}(\bot) = \bigsqcup_k F^k(\bot)$ - since we are making join over all the k's, we are there's no difference between $F^{k+1}(\bot)$ and $F^k(\bot)$.

■

Why is it sufficient to find the least fixed point?

If F is continuous on a pointed cpo, by using the theorem we know how to find the least fixed point. All other fixed points can be regarded as refinements of the least one. They contain more information, as they are more precise. In general they are also more arbitrary and make less sense for our purposes.

Back to the While Statement

We are interested in solving the recursive equation about the while statement:

$$W(\sigma) = \begin{cases} W(S[\![s]\!]\sigma) & \text{if } B[\![b]\!](\sigma) = \text{true} \\ \sigma & \text{if } B[\![b]\!](\sigma) = \text{false} \\ \perp & \text{otherwise} \end{cases}$$

This is equivalent to solving the equation $W = F(W)$ where F is defined as:

$$F = \lambda w.\lambda\sigma. = \begin{cases} w(S[\![s]\!]\sigma) & \text{if } B[\![b]\!](\sigma) = \text{true} \\ \sigma & \text{if } B[\![b]\!](\sigma) = \text{false} \\ \perp & \text{otherwise} \end{cases}$$

Several examples of how using this equation can be founded at the end of the chapter.

5.3.2 Fixed Point Theorem Requirements

The fixed point idea does not suffice for defining the semantic of while since:

- Some functions have **more than one fixed point**. For example, let's look at the function:

  $$F(w)\sigma = \begin{cases} w(\sigma) & \text{if } \sigma(x) \neq 0 \\ \sigma & \text{if } \sigma(x) = 0 \end{cases}$$

  Every function w satisfying $w(\sigma) = \sigma$ if $\sigma(x) = 0$ will be a fixed point of F.

- Some functions have **no fixed points**. For example, the function:

  $$F(w) = \begin{cases} w_1 & \text{if } w_1 = w_2 \\ w_2 & \text{otherwise} \end{cases}$$

  If $w_1 \neq w_2$, then clearly there will be no function w, s.t. $F(w) = w$, since $F(w_1) = w_2 \neq w_1$. Thus, F has no fixed points at all.

The solution to both problems above can be fixed by imposing another requirement on the fixed point.
Let's look at 3 scenarios in order to add requirements for the fixed point theorem.
We will use the **While b do S** statement:

1. The statement terminates.
2. The statement loops in S.
3. The statement loops in the while.

We will now examine each case.

The statement terminates:

That means, there are states $s_1..s_n$, such that:

$$B[\![b]\!]s_i = \begin{cases} t & \text{if } i < n \\ f & \text{if } i = n \end{cases}$$

Recall that the functions **B** and **S** is the semantic operation of Boolean and Statement expressions, respectively, and $\mathbf{S}_{ds}[\![S]\!]s_i = s_{i+1} for i < n$.
An example of such statement and a state that satisfy these requirements:
(While x $\geq$ 0 do x := x - 1, $s(x) \geq 0$)

Let g be any fixed point of F, where $(F, g)s = gs$ There are 2 cases:

1. $i < n$: $(g, s_i) = (F, g)s_i = if B[\![b]\!]s_i$ then $(g \circ \mathbf{S}_{ds}[\![S]\!])s_i$.
   Since we have that $i < n$, then $B[\![b]\!]s_i$ is true and the result is that:
   $if B[\![b]\!]s_i$ then $(g \circ \mathbf{S}_{ds}[\![S]\!])s_i = g(\mathbf{S}_{ds}[\![S]\!])s_i = (g, s_{i+1})$

2. $i = n$: $(g, s_n) = (F, g)s_n = if B[\![b]\!]s_n$ then $(g \circ \mathbf{S}_{ds}[\![S]\!]s_n)$.
   Since we have that $i = n$, then $B[\![b]\!]s_n$ is false. So we skip, and get:
   $if B[\![b]\!]s_n$ then $(g \circ \mathbf{S}_{ds}[\![S]\!]s_n) = s_n$

So, every fixed point g of F will suffice for $(g, s) = s$.
Conclusion: We do not need for extra requirements that will help us to choose the fixed point.

The execution loops locally:

This means that there are states $s_1, .., s_n$ such that, $B[\![b]\!]s_i = true$ for $i < n$, and:

$$\mathbf{S}_{ds}[\![S]\!]s_i = \begin{cases} s_{i+1} & \text{if } i < n \\ Undefined & \text{if } i = n \end{cases}$$

An example of such statement that satisfy these requirements is:

(while $x \leq 0$ do
  if $x = 0$ then
    while true do skip;
  else x := x + 1,
$s(x) < 0$)

Let g be any fixed point of F, where $(F,g)s = gs$. As in the previous case, we have 2 cases:

1. $i < n$: exactly like the previous.
2. $i = n$: $(g, s_n) = (F,g)s_n = if B[\![b]\!]s_n$ then $(g \circ \mathbf{S}_{ds}[\![S]\!]s_n) = g(\mathbf{S}_{ds}[\![S]\!]s_n) = Undefined$

Thus any fixed point g of F will satisfy $(g,s) = Undefined$
Conclusion: As in previous case, we do noto need for extra requirements that will help us to choose the fixed point.

The execution loops globally:

This means that there are states $s_1, ..$ such that $B[\![b]\!]s_i = true$ for all i and $\mathbf{S}_{ds}[\![S]\!]s_i = s_{i+1}$ for all n.

An example of such statement is:
(while $x = 0$ do skip, $s(x) \neq 0$)
Let g be any fixed point of F, where $(F,g)s = gs$. Like the previous cases we have that $(g, s_i) = (g, s_{i+1})$.
Thus, we get that $(g, s_0) = (g, s_i)$ for all i and we cannot determine the value of (g, s0).
This is the place where the various fixed points of F may differ.
As noted in the above example, every function F:

$$(Fg)s = \begin{cases} (g,s) & \text{if } s(x) \neq 0 \\ s & \text{if } s(x) = 0 \end{cases}$$

So any partial function g which satisfy: $(g,s) = s$ if $x = 0$ will be a fixed point of F. But we want the fixed point to be:

$$\mathbf{S}_{ds}[\![\text{ while } (x = 0) \text{ do skip }]\!]s_i = \begin{cases} s_0 & \text{if } s_0(x) = 0 \\ \text{Undefined} & \text{if } s_0(x) \neq 0 \end{cases}$$

So our preferred fixed point of F is the function:

$$(g,s) = \begin{cases} \text{Undefined} & \text{if } s(x) = 0 \\ \text{s} & \text{if } s(x) \neq 0 \end{cases}$$

The property that differ g from a different fixed point g' of F is that whenever $(g,s) = s'$ then $(g',s) = s'$ but not necessarily that if $(g',s) = s'$ then $(g,s) = s'$
Conclusion: The desired fixed point of F should be some partial function such that:

1. **(F,g) = g**
2. **if** $Fg' = g'$ **and** $g' \neq g$ **then** $(g,s) = s' \rightarrow (g',s) = s'$ **for all s and s'**.

5.4 Examples of the Fixed Point Theorem

In all the following examples we will use the function $F : [[\Sigma_\perp \to \Sigma_\perp] \to [\Sigma_\perp \to \Sigma_\perp]]$:

$$F = \lambda w.\lambda\sigma. = \begin{cases} w(S[\![s]\!]\sigma) & \text{if } B[\![b]\!](\sigma) = \text{true} \\ \sigma & \text{if } B[\![b]\!](\sigma) = \text{false} \\ \perp & \text{otherwise} \end{cases}$$

as we presented in the previous sections and we will just define the condition b and the setatement s.
Note: Whenever a function is required and $\perp$ is given instead, we refer to the function $\perp = \lambda\sigma.\perp$, i.e., the function that receives any state and allways returns $\perp$.

5.4.1 while true do skip

- $\mathbf{B}[\![b]\!] = \mathbf{B}[\![\text{true}]\!] = \lambda\sigma.\text{true}$.
- $\mathbf{S}[\![s]\!] = \mathbf{S}[\![\text{skip}]\!] = \lambda\sigma.\sigma$.

Then, $F = \lambda w.\lambda\sigma.w(\sigma)$. We will now iterate over $F^k(\perp)$ until we reach a fixed point:

1. $F^0(\perp) = \perp$ - since 0 invocations is as no function was applied, that is, the function returns the input.
2. $F^1(\perp) = F(F^0(\perp)) = \lambda w.\lambda\sigma.w(\sigma)(\perp) = \lambda\sigma.\perp(\sigma) = \lambda\sigma.\lambda\sigma.\perp(\sigma) = \lambda\sigma.\perp$. Thus, $F(\perp) = \lambda\sigma.\perp$.
3. $F^2(\perp) = F(F(\perp)) = F(\lambda\sigma.\perp) = \lambda w.\lambda\sigma.w(\sigma)(\lambda\sigma.\perp) = \lambda\sigma.\lambda\sigma.\perp(\sigma) = \lambda\sigma.\perp$. Thus, $F^2(\perp) = \lambda\sigma.\perp$.

Hence, we have found a fixed point: $w = \lambda\sigma.\perp$. In other words, $\mathbf{S}[\![\text{while true do skip}]\!] = \lambda\sigma.\perp$. This behavior is really expected, since our program loops infinitely and does not end, thus we want the function to describe a non-terminating behavior at any state.
Now we want to show that $w = \lambda\sigma.\perp$ is the least fixed point. Since we use the flat cpo, we get that $\perp \sqsubseteq \sigma$. Hence, $\lambda\sigma.\perp$ is "smaller" than any other function (by the point inference relation) and thus, it is the least fixed point.

5.4.2 while false do skip

- $\mathbf{B}[\![b]\!] = \mathbf{B}[\![\text{false}]\!] = \lambda\sigma.\text{false}$.
- $\mathbf{S}[\![s]\!] = \mathbf{S}[\![\text{skip}]\!] = \lambda\sigma.\sigma$.

Then, $F = \lambda w.\lambda\sigma.\sigma$. We will now iterate over $F^k(\bot)$ until we reach a fixed point:

1. $F^0(\bot) = \bot$
2. $F^1(\bot) = F(F^0(\bot)) = \lambda w.\lambda\sigma.\sigma(\bot) = \lambda\sigma.\sigma$.
3. $F^2(\bot) = F(F(\bot)) = F(\lambda\sigma.\sigma) = \lambda w.\lambda\sigma.\sigma(\lambda\sigma.\sigma) = \lambda\sigma.\sigma$.

We got that $F^2(\bot) = \lambda\sigma.\sigma = F(\bot)$, thus $w = \lambda\sigma.\sigma$ is a fixed point.
Then it follows by induction that for any $k > 1$, $F^k(\bot) = F(F^{k-1}(\bot)) = F(F^{k-2}(\bot)) = F^{k-1}(\bot)$, i.e., $F^k(\bot) = F^{k-1}(\bot) = \lambda\sigma.\sigma$.
Now, by the Fixed Point Theorem, we know that the least fixed point is $\bigsqcup_k F^k(\bot) = \bigsqcup_k \lambda\sigma.\sigma = \lambda\sigma.\sigma$, as expected.

5.4.3 while x != 3 do x = x - 1

- $\mathbf{B}[\![b]\!] = \mathbf{B}[\![\text{x != 3}]\!] = \lambda\sigma.\sigma(x) \neq 3$.
- $\mathbf{S}[\![s]\!] = \mathbf{S}[\![\text{x = x - 1}]\!] = \lambda\sigma.\sigma[x \mapsto \sigma(x) - 1]$.

Then,

$$F = \lambda w.\lambda\sigma. = \begin{cases} w(\sigma[x \mapsto \sigma(x) - 1]) & \sigma(x) \neq 3 \\ \sigma & \sigma(x) = 3 \end{cases}$$

We will now iterate over $F^k(\bot)$ until we reach a fixed point:

1. $F^0(\bot) = \bot$
2. $$F^1(\bot) = F(F^0(\bot)) =_{[1]} \lambda\sigma. \begin{cases} \lambda\sigma.\bot(\sigma[x \mapsto \sigma(x) - 1]) & \sigma(x) \neq 3 \\ \sigma & \sigma(x) = 3 \end{cases} = \begin{cases} \lambda\sigma.\bot & \sigma(x) \neq 3 \\ \sigma & \sigma(x) = 3 \end{cases}$$
   In [1] we have done β-reduction and assigned $F^0(\bot) = \lambda\sigma.\bot$ to w.
3. $$F^2(\bot) = F(F(\bot)) = \begin{cases} F(\bot)(\sigma[x \mapsto \sigma(x) - 1]) & \sigma(x) \neq 3 \\ \sigma & \sigma(x) = 3 \end{cases}$$
   Let's find how this function behaves. Le σ' be a state exactly as σ (we will use it to show the "old" state, whereas σ will represent the "new" state).
   $$F(\bot)(\sigma[x \mapsto \sigma'(x) - 1]) = \lambda\sigma. \begin{cases} \sigma & \sigma(x) = 3 \\ \bot & \sigma(x) \neq 3 \end{cases} (\sigma[x \mapsto \sigma'(x) - 1]) =_{[2]}$$
   $$\lambda\sigma. \begin{cases} \sigma[x \mapsto 3] & \sigma'(x) = 4 \\ \bot & \sigma'(x) \neq 4 \end{cases}$$

Note that in [2] we applied again β-reduction. Finally, we get that:

$$F^2(\bot) = \lambda\sigma.\begin{cases} \sigma[x \mapsto 3] & \sigma(x) = \in \{3,4\} \\ \bot & \sigma(x) \notin \{3,4\} \end{cases}$$

This result agrees with our expectation that the loop terminates after one iteration only if x's original value was between 3 and 4, otherwise it does not terminate. In the later case, $\bot$ is the expresses this the best.
Then, it follows by induction on the number of iterations that,

$$F^k(\bot) = \lambda\sigma.\begin{cases} \sigma[x \mapsto 3] & 3 \le \sigma(x) \le 3+k-1 \\ \bot & otherwise \end{cases}$$

We have shown the basis of the induction, now we assume that it is true for $k-1$ and prove it for k.

$$F^k(\bot) = F(F^{k-1}(\bot)) = \lambda\sigma.\begin{cases} F^{k-1}(\bot)(\sigma[x \mapsto \sigma(x)-1]) & \sigma(x) \neq 3 \\ \sigma & \sigma(x) = 3 \end{cases} =$$

$$\lambda\sigma.\begin{cases} \sigma[x \mapsto 3] & 4 \le \sigma(x) \le 3+k-1 \\ \sigma & \sigma(x) = 3 \\ \bot & otherwise \end{cases} = \lambda\sigma.\begin{cases} \sigma[x \mapsto 3] & 3 \le \sigma(x) \le 3+k-1 \\ \bot & otherwise \end{cases}$$

Note that we have not found a fixed point yet, in contrast to what we had in examples 1 and 2. But, by the Fixed Point Theorem the least fixed point is given by $\bigsqcup_k F^k(\bot)$. Now, we will calculate it.

$$\bigsqcup_k F^k(\bot) = \bigsqcup_k \lambda\sigma.\begin{cases} \sigma[x \mapsto 3] & 3 \le \sigma(x) \le 3+k-1 \\ \bot & otherwise \end{cases} \overset{\rightarrow}{\infty} \lambda\sigma.\begin{cases} \sigma[x \mapsto 3] & \sigma(x) \ge 3 \\ \bot & otherwise \end{cases}$$

Again, the least fixed point reflects the behavior we expect from the program: the loop will terminate only if the initial value of x is grater or equal than 3.

5.4.4 A nested loop example

```
Z = 0;
while (X > 0) do {
  Y = X;
  while (Y > 0) do {
    Z = Z + Y;
    Y = Y - 1;
  }
  X = X - 1;
}
```

This is a deep example which should finally provide the reader the understanding of the Fixed Point Theorem and it's applications. This example also makes use of the power of Denotational semantics: the fact that they are completely **compositional**. In order to make this example easier we assume that $\sigma : Vars \to \Sigma \cup \{\bot\}$. In order to understand and calculate the result of this program we will examine each of it's sub-parts and then we will join the whole results into the mathematical object which describes the program output (depending on the given input).

Let's start by the inner loop, let's call it $Loop_{in}$.

```
Z = 0;
while (Y > 0) do {
  Z = Z + Y;
  Y = Y − 1;
}
```

As in the previous examples we start by defining the function F as described previously. Define

- $\mathbf{B}[\![b]\!] = \mathbf{B}[\![y > 0]\!] = \lambda\sigma.\sigma(y) > 0$.
- $\mathbf{S}[\![s]\!] = \mathbf{S}[\![\text{z = z + y; y = y - 1}]\!] = \lambda\sigma.\sigma[(\sigma(z) + \sigma(y))/z, (\sigma(y) - 1)/y]$.

We define F as follows:

$$F = \lambda w.\lambda\sigma.\begin{cases} w(\sigma[(\sigma(z) + \sigma(y))/z, (\sigma(y) - 1)/y] & \text{if } \sigma(y) > 0 \\ \sigma & \text{if } \sigma(y) = 0 \end{cases}$$

Start by $F_0(\bot) = \bot$. Now,

$$F(F_0(\bot)) = \lambda\sigma.\begin{cases} w(\bot[(\sigma(z) + \sigma(y))/z, (\sigma(y) - 1)/y] & \text{if } \sigma(y) > 0 \\ \sigma & \text{if } \sigma(y) = 0 \end{cases}$$

and then:

$$F(F_0(\bot)) = \lambda\sigma.\begin{cases} \bot & \text{if } \sigma(y) > 0 \\ \sigma & \text{if } \sigma(y) = 0 \end{cases}$$

Now let's calculate the next iteration:

$$F_2(\bot) = F(F(\bot)) =$$

$$\lambda\sigma.\begin{cases} \lambda\sigma.\begin{cases} \bot & \text{if } \sigma(y) > 0 \\ \sigma & \text{if } \sigma(y) = 0 \end{cases} (\sigma[(\sigma(z) + \sigma(y))/z, (\sigma(y) - 1)/y]) & \text{if } \sigma(y) > 0 \\ \sigma & \text{if } \sigma(y) = 0 \end{cases}$$

$$F_2(\perp) = \lambda\sigma. \begin{cases} \perp & \text{if } \sigma(y) > 1 \\ \sigma(0/y, \sigma(z) + 1/z) & \text{if } \sigma(y) = 1 \\ \sigma & \text{if } \sigma(y) = 0 \end{cases}$$

We can appreciate from the last iteration that after one iteration of the function (we should remember that the first invocation of F is just testing the condition of the loop without entering it) we get that the loop terminates only if the value of y is less than 2, this agrees with the intuition since the loop decrements 1 from y each iteration and stops when y = 0. Now, let's prove by induction on the number of invocations of F (that is the number of iterations of the loop -minus 1) that F terminates if the initial value of y is less than k and in this case the final value of y is 0 and the final value of z is the initial value of z plus the sum of the numbers from 1 to $\sigma(y)$, or as a function:

$$F_k(\perp) = \lambda\sigma. \begin{cases} \sigma[0/y, (\sigma(z) + 0.5\sigma(y)(\sigma(y) + 1))/z & \text{if } \sigma(y) < k \\ \perp & \text{if } \sigma(y) \geq k \end{cases}$$

The basis of the induction was proven when we calculated the first 2 invocations of F. Now we assume that this is valid for k-1 and prove for k:

$$F_{k-1}(\perp) = \lambda\sigma. \begin{cases} \sigma[0/y, (\sigma(z) + 0.5\sigma(y)(\sigma(y) + 1))/z] & \text{if } \sigma(y) < k - 1 \\ \perp & \text{if } \sigma(y) \geq k - 1 \end{cases}$$

$$F_k(\perp) = F_k(F_{k-1}(\perp)) = \lambda\sigma. \begin{cases} F_{k-1}(\perp)(\sigma[(\sigma(z) + \sigma(y))/z, (\sigma(y) - 1)/y]) & \text{if } \sigma(y) > 0 \\ \sigma & \text{if } \sigma(y) = 0 \end{cases}$$

then:

$$F_k(\perp) = \lambda\sigma. \begin{cases} \begin{cases} \sigma[0/y, (\sigma(z) + \sigma(y) + 0.5(\sigma(y) - 1)(\sigma(y))/z] & \text{if } \sigma(y) < k \\ \perp & \text{otherwise} \end{cases} & \text{if } \sigma(y) > 0 \\ \sigma & \text{if } \sigma(y) = 0 \end{cases}$$

then:

$$F_k(\perp) = \lambda\sigma. \begin{cases} \begin{cases} \sigma[0/y, (\sigma(z) + \sigma(y)(\sigma(y) + 1))/z] & \text{if } \sigma(y) < k \\ \perp & \text{otherwise} \end{cases} & \text{if } \sigma(y) > 0 \\ \sigma & \text{if } \sigma(y) = 0 \end{cases}$$

then:

$$F_k(\perp) = \lambda\sigma. = \begin{cases} \sigma[0/y, (\sigma(z) + \sigma(y)(\sigma(y) + 1))/z] & \text{if } \sigma(y) < k \\ \perp & \text{if } \sigma(y) \geq k \end{cases}$$

Explanations: [1] Note that since we assign $\sigma(y) - 1$ to y, and $\sigma(z) + \sigma(y)$ to z the expression $\sigma(z) + 0.5\sigma(y)(\sigma(y) + 1)$ turns into $\sigma(z) + \sigma(y) + 0.5(\sigma(y) - 1)\sigma(y)$. Note that $\sigma(y) + 0.5(\sigma(y) -$

$1)\sigma(y) = \sigma(y)(1 + 0.5\sigma(y) - 0.5) = 0.5\sigma(y)(\sigma(y) + 1)$. [3] Note that if we take $\sigma(y) = 0$ then $\sigma(z) + \sigma(y)(\sigma(y) + 1) = \sigma(z)$, so in this case doing the assignment to σ will remain σ as is, since we assign to y and z the values they had previously.

In summary, we prove:

$$F_k(\perp) = \lambda\sigma. \begin{cases} \sigma[0/y, (\sigma(z) + 0.5\sigma(y)(\sigma(y) + 1))/z] & \text{if } \sigma(y) < k \\ \perp & \text{if } \sigma(y) \geq k \end{cases}$$

if we take k to (positive) infinity we get that: $F_k(\perp) = \lambda\sigma.\sigma[0/y, (\sigma(z)+0.5\sigma(y)(\sigma(y)+1))/z]$. Thus, by the Fixed Point Theorem we get that the mathematical function of the inner loop is:
$\mathbf{S}[\![Loop_{in}]\!] = \lambda\sigma.\sigma[0/y, (\sigma(z) + 0.5\sigma(y)(\sigma(y) + 1))/z]$.
Now we want to analyze the outer loop

Z = 0;
while $(X > 0)$ do {
  $Y = X$;
  $Loop_{in}$
  $X = X - 1$;
}

Since the Denotational Semantics are compositional, we can write the inner loop as $Loop_{in}$. Moreover, when analyzing the semantic meaning of the outer loop we can look at the inner one as a simple command making use of the function we computed above. We define $h(x) = 0.5x(x + 1)$. Now let's denote by s the body of the loop, then: $\mathbf{S}[\![s]\!]\sigma = \mathbf{S}[\![x = x - 1;]\!](\mathbf{S}[\![Loop_{in};]\!](\mathbf{S}[\![y = x;]\!]\sigma)) = \mathbf{S}[\![x = x - 1;]\!](\mathbf{S}[\![Loop_{in};]\!]\sigma[\sigma(x)/y]) = \mathbf{S}[\![x = x - 1;]\!]\sigma[0/y, \sigma(z) + h(\sigma(x))/z] = \sigma[(\sigma(x) - 1)/x, 0/y, (\sigma(z) + h(\sigma(x)))/z]$
Also, denote by $b = x > 0$, then $\mathbf{B}[\![x > 0]\!] = \lambda\sigma.\sigma(x) > 0$. So, we define F us follows:

$$F = \lambda w.\lambda\sigma. \begin{cases} w(\sigma[(\sigma(x) - 1)/x, 0/y, (\sigma(z) + h(\sigma(x)))/z]) & \text{if } \sigma(x) > 0 \\ \sigma & \text{if } \sigma(x) = 0 \end{cases}$$

As in the previous examples, we start with $F_0(\perp) = \perp$, and now we calculate the first invocation of F:

$$F(F(\perp)) = \lambda\sigma. \begin{cases} w(\perp[(\sigma(x) - 1)/x, 0/y, (\sigma(z) + h(\sigma(x)))/z]) & \text{if } \sigma(x) > 0 \\ \sigma & \text{if } \sigma(x) = 0 \end{cases}$$

then:

$$F(F(\perp)) = \lambda\sigma. \begin{cases} \perp & \text{if } \sigma(x) > 0 \\ \sigma & \text{if } \sigma(x) = 0 \end{cases}$$

This is the expected value of entering the loop 0 times, that is the condition must be fulfilled before doing any iteration of the loop.

$$F_2(\bot)) = \lambda\sigma.\begin{cases} \lambda\sigma. = \begin{cases} \bot & \text{if } \sigma(x) > 0 \\ \sigma & \text{if } \sigma(x) = 0 \end{cases} & \\ \sigma & \end{cases} \quad \begin{array}{ll} (\sigma[(\sigma(x)-1)/x, 0/y, (\sigma(z)+h(\sigma(x)))/z]) & \text{if } \sigma(x) > 0 \\ & \text{if } \sigma(x) = 0 \end{array}$$

then:

$$F_2(\bot)) = \lambda\sigma.\begin{cases} \sigma & \text{if } \sigma(x) = 0 \\ \sigma(0/x, 0/y, (\sigma(z)+1)/z) & \text{if } \sigma(x) = 1 \\ \bot & \text{if } \sigma(x) > 1 \end{cases}$$

To give the reader a bit more intuition before going through the general case, k, we will calculate the next iteration:

$$F_3(\bot)) = \lambda\sigma.\begin{cases} \sigma & \text{if } \sigma(x) = 0 \\ \sigma(0/x, 0/y, (\sigma(z)+1)/z) & \text{if } \sigma(x) = 1 \\ \sigma(0/x, 0/y, (\sigma(z)+1+3)/z) & \text{if } \sigma(x) = 2 \\ \bot & \text{if } \sigma(x) > 1 \end{cases}$$

We left the reader the proof of this last function. Now, the general case is:

$$F_k(\bot) = \lambda\sigma.\begin{cases} \sigma[0/x, 0/y, (\sigma(z)+\sigma(x)(\sigma(x)+1)(\sigma(x)+2)/6)z & \text{if } \sigma(x) < k \\ \bot & \text{if } \sigma(x) \geq k \end{cases}$$

We will show this by induction on k, the number of iterations in the loop: First of all, we define $u(x) = x(x+1)(x+2)/6$. Now,

$$F_k(\bot) = F(F_{k-1}(\bot)) = \lambda\sigma.\begin{cases} F_{k-1}(\bot)\sigma[(\sigma(x)-1)/x, 0/y, (\sigma(z)+h(\sigma(x)))/z] & \text{if } \sigma(x) > k \\ \sigma & \text{if } \sigma(x) = k \end{cases}$$

then:

$$F_k(\bot) = \lambda\sigma.\begin{cases} \begin{cases} \sigma[0/x, 0/y, (\sigma(z)+h(\sigma(x))+u(\sigma(x)-1))/z] & \text{if } \sigma(x) < k-1 \\ \bot & \text{otherwise} \end{cases} & \text{if } \sigma(x) > k \\ \sigma & \text{if } \sigma(x) = k \end{cases}$$

then:

$$F_k(\bot) = \lambda\sigma.\begin{cases} \sigma[0/x, 0/y, (\sigma(z)+u(\sigma(x)))/z] & \text{if } \sigma(x) < k \\ \bot & \text{otherwise} \end{cases}$$

Note that [1] follows from the induction assumption on $k = k-1$. We only have to prove [2]. We will prove that $h(x) + u(x-1) = u(x)$, and [2] follows from that when assigning x the

value of $\sigma(x)$. $h(x)+u(x-1) = x(x+1)/2+(x-1)x(x+1)/6 = x(x+1)/2*(1+(x-1)/3) = x(x+1)/2*(x-1+3)/3 = x(x+1)(x+2)/6$. And this proves [2]. Now, by the Fixed Point Theorem, the function that represents the outer loop is:

$$\mathbf{S}[\![Loop_{out}]\!] = F_k = \lambda\sigma. \begin{cases} \sigma[0/x, 0/y, (\sigma(z) + u(\sigma(x)))/z] & \text{if } \sigma(x) < k \\ \perp & \text{otherwise} \end{cases}$$

then:

$$F_k = \lambda\sigma.\sigma = \begin{cases} [0/x, 0/y, (\sigma(z) + \sigma(x)(\sigma(x)+1)(\sigma(x)+2)/6)/z] & \text{if } \sigma(x) < k \\ \perp & \text{otherwise} \end{cases}$$

Now we've just been left with the following program: $Z = 0$;
$Loop_{out}$ In order to analyze this small program we just have to apply the concatenating rule: $\mathbf{S}[\![z = 0; Loop_{out};]\!] = \mathbf{S}[\![Loop_{out}]\!] \odot \mathbf{S}[\![z = 0]\!] = \lambda\sigma.\sigma[0/x, 0/y, (\sigma(z) + \sigma(x)(\sigma(x)+1)(\sigma(x)+2)/6)/z] \odot \lambda\sigma.\sigma[0/z] = \lambda\sigma.\sigma[0/x, 0/y, (\sigma(x)(\sigma(x)+1)(\sigma(x)+2)/6)/z]$.
Thus, the function that represents the whole program is: $\lambda\sigma.\sigma[0/x, 0/y, (\sigma(x)(\sigma(x)+1)(\sigma(x)+2)/6)/z]$.

5.4.5 Factorial Example

The factorial program: $S_{ds}[\![y := 1; while(x = 1)do(y := y * x; x := x - 1)]\!]$. Our function F (for the while) will be:

$$(F, g)s = = \begin{cases} g(S_{ds}[\![(y := y * x; x := x - 1)]\!]s) & \text{if } B[\![(x = 1)]\!]s = t \\ s & \text{if } B[\![(x = 1)]\!]s = f \end{cases}$$

Lets calculate $F_n(\perp)$:
$F_0(\perp)s = \perp$ (since F_0 is the identity function).
Since the factorial must start with 1 we get:

$$F_1(\perp)s = F(\perp)s = = \begin{cases} \perp & \text{if } (s, x) \neq 1 \\ s & \text{if } (s, x) = 1 \end{cases}$$

Now, the second iteration is:

$$F_2(\perp)s = F(F(\perp)s) = = \begin{cases} \perp & \text{if } s(x) \neq 1, 2 \\ s[(s(y) * 2)/y, 1/x] & \text{if } s(x) = 2 \\ s[(s(y))/y, 1/x] & \text{if } s(x) = 1 \end{cases}$$

Thus, we can conclude that:

$$F_n(\perp)s = = \begin{cases} \perp & \text{if } s(x) \neq 1, 2 \\ s[(s(y) * n!)/y, 1/x] & \text{if } 1 \leq s(x) \leq n \\ \perp & \text{otherwise} \end{cases}$$

So, the fixed point is:

$$F(\bot) = \lambda w.\lambda s. = \begin{cases} \bot & \text{if } s(x) \neq 1,2 \\ s[(s(y) * s(x)!)/y, 1/x] & \text{if } 1 \leq s(x) \\ \bot & \text{otherwise} \end{cases}$$

5.4.6 Advanced Topics in Denotational Semantics

In this section, we present a couple of theorems (without proofs) that talk about advanced concepts in denotational semantics.

Equivalence Of Semantics

It can be easily shown that there is an equivalence between denotational semantics, natural semantics and structural operational semantics (small step semantics) as claimed by the next theorem:

Theorem 5.7 $\forall \sigma, \sigma' \in \Sigma : \sigma' = S[\![s]\!]\sigma \Leftrightarrow < s, \sigma > \rightarrow \sigma' \Leftrightarrow < s, \sigma > \Rightarrow * \sigma'$

Complete Lattices

Definition Let $(D, \sqsubseteq)$ be a partial order.
D is a *complete lattice* if every subset has both greatest lower bounds and lest lower bounds.

The following theorem shows that for a complete lattice, there is another way to find the least fixed point of a function.

Theorem 5.8 (Knaster-Tarski Theorem) *Let $f : L \rightarrow L$ be a monotonic function on a complete lattice L. Then a least fixed point $lfp(f)$ exists, and $lfp(f) = \sqcap\{x \in L : f(x) \sqsubseteq x\}$*

This is a practical way to find the lfp since it might take us an infine number of applications of f to reach the lfp.

Or in a more formal way:
Let $f : L \rightarrow L$ be a monotonic function where $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a complete lattice and define:

1. $Fix(f) = \{l : l \in L, f(l) = l\}$
2. $Red(f) = \{l : l \in L, f(l) \sqsubseteq l\}$
3. $Ext(f) = \{l : l \in L, l \sqsubseteq f(l)\}$

Theorem 5.9 (Tarski 1955) *If f is monotone then:*

- $lfp(f) = \sqcap Fix(f) = \sqcap Red(f) \sqsubseteq Fix(f)$
- $gfp(f) = \sqcup Fix(f) = \sqcup Ext(f) \sqsubseteq Fix(f)$

5.4.7 Summary

1. Denotational definitions are not necessarily better than operational semantics, and they usually require more mathematical work.
2. The mathematics may be done once and for all
3. The mathematics may pay off:
4. Some of its techniques are being transferred to operational semantics.
5. It is trivial to prove that: If $B[\![b_1]\!] = B[\![b_2]\!]$ *and* $C[\![c_1]\!] = C[\![\ c_2\ ]\!]$ then $C[\![while\ b_1\ do\ c_1\ ]\!] = C[\![while\ b_2\ do\ c_2\ ]\!]$ (compare with operational semantics).
6. Denotational semantics provides a way to declare the meaning of programs in an abstract way. It can handle *side-effects*, *loops*, *recursion*, *gotos*, *non-determinism*, but now low-level concurrency.
7. Fixed point theory provides a declarative way to specify computations, which gives many usages.

Bibliography