Language Engineering - A nice set of notes

Josh Felmeden

February 27, 2020

Contents

1 Introduction to Semantics **3**

2 Structural Operational Semantics **4**
2.1 Termination and looping 4
2.2 Determinism and Equivalence 5
2.3 Provably correct implementation 6
2.3.1 Arithmetic code 7
2.3.2 State changing code 7
2.3.3 Computation sequences 7
2.4 The execution function 8
2.5 Code Translation of Expressions 8
2.6 Semantic function 9
2.6.1 Correctness of translation 9

3 Denotational Semantics **10**
3.1 Conditional functions 10
3.2 Least fixpoint 11
3.2.1 The functional of a loop 12
3.3 More on partial orders 13
3.4 Fixpoints of real functions 14
3.5 Divergence and strictness 14

4 Chain-Complete Partial Orders **16**
4.1 Definitions of PO-Set (partially ordered set) 16

5 The language Exc **16**
5.1 Continuation-style denotational semantics 17

6 Axiomatic Semantics **18**

7 Lexing and Parsing **19**
7.1 Lexical Analysis 20
7.2 Regular expressions 20
7.3 Finite Automata and State Machines 21
7.4 Nondeterminism to Determinism 21
7.5 Context Free Grammars 21

8 Parsers **22**
8.1 Parser specification 22

1 Introduction to Semantics

Semantics are really complex and they actually exist in the real world as problems that can arise when the semantics are unclear. In the example of the Derek Bentley case, Bentley tells Chris (who is holding a gun, and a policeman standing in front of him to 'let him have it!'. Here, it appears that he could be talking about the gun, or to kill him. The same kind of thing can happen in computing when we are unsure of the references of certain objects.

Here are some examples learned from natural languages:

- Syntactic complexity
  - Jack built the house the malt the rat the cat killed ate lay in
- Syntactic ambiguity
  - Let him have it, Chris!
- Semantic Complexity
  - It depends on what the meaning of the word 'is' is!
- Semantic ambiguity
  - I haven't slept for ten days
- Semantic undefinedness
  - Colourless green ideas sleep furiously
- Interaction of syntax and semantics
  - Time flies like an arrow, fruit flies like a banana.

We can apply these things to computing terms, too.

- Syntactic complexity

```
x-=y = (x=x+y) - y      //switches variables x and y
```

- Syntactic ambiguity

```
if (...) if (...) ..; else ..       //dangling else
```

- Semantic Complexity

```
y = x++ + x++       //sequence points
```

- Semantic ambiguity

```
(x%2=1) ? "odd" : "even"        //unspecified in C89 if x<0
```

- Semantic undefinedness

```
while(x/x)      //division error or infinite loop
```

- Interaction of syntax and semantics

```
A * B       //lever hack
```

To put this another way:

- **Syntax**: concerned with the form of expressions and whether or not the program actually *compiles*
- **Semantics**: concerned with the meaning of expressions and what the program does when it *runs*
- **Pragmatics**: concerned with issues like design patterns, program style, industry standards, etc.

2 Structural Operational Semantics

We're going to look at doing some compilation (of the *while*) language.

2.1 Termination and looping

The execution of the statement S in state σ terminates iff there exists a finite derivation sequence from $\langle S, \sigma \rangle$. The derivation sequence looks like:

$$\langle S, \theta \rangle \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_n \text{ where } \gamma_n \text{ is terminal } \sigma' \text{ or stuck } \langle S', \sigma' \rangle$$

The while language never gets stuck, but some language might if we try to divide by zero because we don't know how to process this.

The execution of the statement S in a state θ loops iff there exists an infinite derivation sequence from $\langle S, \sigma \rangle$

$$\langle S, \sigma \rangle \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots$$

S always terminates iff its execution terminates in all states σ.

S always loops if the execution loops in all states σ.

The execution of statement S in state σ terminates successfully iff it ends with a terminal configuration.

Note while has no stuck configurations, so termination implies successful termination!

2.2 Determinism and Equivalence

The structural operation semantics is (strongly) **deterministic** iff $\langle S, \sigma\rangle \Rightarrow \gamma$ and $\langle S, \sigma\rangle \Rightarrow \gamma'$ imply that $\gamma = \gamma'$ for all $S, \sigma, \gamma, \gamma'$

It is **weakly deterministic** iff $\langle S, \sigma\rangle \Rightarrow^* \sigma'$ and $\langle S, \sigma\rangle \Rightarrow^* \sigma''$ imply that $\sigma' = \sigma''$ for all $S, \sigma, \sigma', \sigma''$. This is different from the strong determinism above because it says that for every successfully terminating branch, (it doesn't matter how we get there) we get to the same final state.

Two statements are **semantically equivalent** whenever it holds that for *all states* σ

$$\langle S_1, \sigma\rangle \Rightarrow^* \gamma \text{ iff } \langle S_2, \sigma\rangle \Rightarrow^* \gamma \text{ whenever } \gamma \text{ is terminal or stuck}$$

This means that there is an infinite derivation sequence from $\langle S_1, \sigma\rangle$ iff there is an infinite derivation from $\langle S_2, \sigma\rangle$.

Note! The length of these could be different (because of the * again.)

For a deterministic structural operational semantics, we can define a semantic function as follows:

- $S_{sos}[[.]]$ Stm $\rightarrow$ (State $\hookrightarrow$ State)
- $S_{sos}[[S]]\sigma = \sigma'$ if $\langle S, \sigma\rangle \Rightarrow^* \sigma'$ and **undefined** otherwise
- Note that the semantic function is only guaranteed to return a partial function between states due to the existence of statements whose execution loops in one or more states
- $S_{sos}[[$ while true do skip $]] = \{\}$
  - If we apply this on any state, we get undefined back BUT it is not in and of itself undefined. It is simply the empty set.
  - What could we do if the semantics is not deterministic?
    - The problem is that depending on what choice we made, we might get a different answer. But the definition says that we only return one function. So therefore, we need to be able to collect them up into a list.
    - One way of doing it is: $S'_{sos}[[.]]$: Stm $\hookrightarrow$ (State $\hookrightarrow$ State) to ignore the ambiguous cases
    - Another way is to allow a set of final states: $S''_{sos}[[.]]$: Stm $\rightarrow$ (State $\rightarrow \mathcal{P}$ State). This is bad because we get a set of states.
    - $S'''_{sos}[[.]]$: Stm $\rightarrow (\mathcal{P}(\text{State}) \rightarrow \mathcal{P}(\text{State}))$ now facilitates function composition. Basically, we pass a load of states, and the function returns a list of all functions that can be reached from any of those functions.

Theorem. For all statements S of **While**, it holds that $S_{ns}[[S]] = S_{sos}[[S]]$. Basically, for all statements, then:

$$\{(\sigma, \sigma') \in \text{State}^2 \mid \langle S, \sigma\rangle \to \sigma'\} = \{(\sigma, \sigma') \in \text{State}^2 \mid \langle S, \sigma\rangle \Rightarrow^* \sigma'\}$$

This mess can be decomposed into two different facts:

$$\langle S, \sigma\rangle \Rightarrow^* \sigma' \text{ implies } \langle S, \sigma\rangle \to \sigma'$$

And

$$\langle S, \sigma\rangle \to \sigma' \text{ implies } \langle S, \sigma\rangle \Rightarrow^* \sigma'$$

Very subtle, right? This can also be decomposed further into some cool stuff but I don't think it's helpful. See lemma 2.28 in the book for the derivation sequence.

2.3 Provably correct implementation

We're now going to look at the correctness of a compiler from **While** to an abstract machine **AM**. Initially, we will consider a simple **stack** machine with a set of abstract instructions. Later on, we'll refine it to use memory addresses. Let's formalise some aspects of the abstract machine.

The configurations in the machine are going to be a triple: $\langle c, e, s\rangle$:

- c is the code to be executed $c \in \text{Code} = \text{inst}^*$
- e is the evaluation stack (of expressions) $e \in \text{Stack} = (Z \cup T)^*$
- s is the storage (for variables) $s \in \text{State} = \text{Var} \to Z$

The instructions will be:

```
inst ::= push-n | add | mult | sub
       | true | false | eq | le | and | neg
       | fetch-x | store-x | noop | branch(c,c) | loop(c,c)

c ::= empty | inst:c
```

`noop` is basically a skip. Also, we'll be passing around code in this example, but later on we'll replace the 'code' by memory addresses where the code is stored.

2.3.1 Arithmetic code

At this point, we might have the following code:

$$
\begin{array}{lcll}
\langle \text{PUSH-n}{:}c, e, s\rangle & \triangleright & \langle c, \mathcal{N}[[n]]{:}e, s\rangle & \\
\langle \text{ADD}{:}c, z_1{:}z_2{:}e, s\rangle & \triangleright & \langle c, (z_1 * z_2){:}e, s\rangle & \text{if } z_1, z_2 \in \mathbb{Z} \\
\langle \text{TRUE}{:}c, e, s\rangle & \triangleright & \langle c, \textbf{tt}{:}e, s\rangle & \\
\langle \text{EQ}{:}c, z_1{:}z_2{:}e, s\rangle & \triangleright & \langle c, (z_1 = z_2){:}e, s\rangle & \text{if } z_1, z_2 \in \mathbb{Z}
\end{array}
$$

Here, the ':' is much like the 'cons' function from Haskell, in that if we take ADD for example; ADD:c means that we have the statement ADD, and then more code following it. In the same way, with the arguments of ADD, we need two integers z_1, z_2 on the stack, represented by $z_1 : z_2 : e$.

Obviously, there are more keywords, but they follow the same format as these existing ones.

2.3.2 State changing code

Now, let's look at some of the state rules:

$$
\begin{array}{lcll}
\langle \text{FETCH-x}{:}c, e, s\rangle & \triangleright & \langle c, s(sx){:}e, s\rangle & \\
\langle \text{STORE-x}{:}c, z{:}e, s\rangle & \triangleright & \langle c, e, s[x \mapsto z]\rangle & \text{if } z \in \mathbb{Z} \\
\langle \text{NOOP}{:}c, e, s\rangle & \triangleright & \langle c, e, s\rangle & \\
\langle \text{BRANCH}(c_1, c_2){:}c, t{:}e, s & \triangleright & \begin{cases} \langle c_1{:}c, e, s\rangle & \text{if } t = \textbf{tt} \\ \langle c_2{:}c, e, s\rangle & \text{if } t = \textbf{ff} \end{cases} & \\
\langle \text{LOOP}(c_2, c_2){:}c, e, s\rangle & \triangleright & \langle c_1{:}\text{BRANCH}(c_2{:}\text{LOOP}(c_1, c_2), \text{NOOP}){:}c, e, s\rangle &
\end{array}
$$

2.3.3 Computation sequences

- A configuration γ can have one of two forms. It can either be **incomplete** or **terminal**.
- An incomplete configuration be either **stuck** if there is no γ' such that $\gamma \triangleright \gamma'$, OR it is **unstuck** if the opposite is true.
- A computation sequence from $\langle c, \epsilon, \sigma$ is either a **finite sequence** such that all γ is a terminal or stuck configuration, or it is **infinite**, such that $\gamma_0 = \langle c, \epsilon, \sigma\rangle$ and $\gamma_i \triangleright \gamma_{i+1}$ for all $0 \leq i$.
- **Note!** $\gamma \triangleright^k \gamma'$ means that γ' can be obtained from γ in exactly k steps of $\triangleright$.
- **Note!** $\gamma \triangleright^* \gamma'$ means that γ' can be obtained from γ in a *finite* number of steps.

Termination and looping is pretty basic and expected, so I won't cover that here.

2.4 The execution function

We can define an execution function for our abstract machine (AM) as follows:

- $\mathcal{M}[\![.]\!]$: Code $\rightarrow$ (State $\hookrightarrow$ State)
- $\mathcal{M}[\![c]\!]\sigma =$

$$\begin{cases} \sigma' & \text{if } \langle c, \epsilon, \sigma' \rangle \triangleright^* \langle \epsilon, e, \sigma' \rangle \\ (\text{Undefined} & \text{otherwise}) \end{cases}$$

2.5 Code Translation of Expressions

Now, we're looking at a function that can translate from **while** into this AM code. So, $\mathcal{CA}[\![.]\!]$: Aexp $\rightarrow$ Code.

$$\begin{aligned} \mathcal{CA}[\![n]\!] &= \text{PUSH-n} \\ \mathcal{CA}[\![x]\!] &= \text{FETCH-n} \\ \mathcal{CA}[\![a_1 + a_2]\!] &= \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \text{ADD} \\ \mathcal{CA}[\![a_1 * a_2]\!] &= \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \text{MULT} \\ \mathcal{CA}[\![a_1 - a_2]\!] &= \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \text{SUB} \end{aligned}$$

For the arithmetic ones, we need to take care because with subtraction, the order matters, hence a_2 being pushed onto the stack before a_1

Now, we can look at $\mathcal{CB}[\![.]\!]$: Bexp $\rightarrow$ Code (the binary ones).

$$\begin{aligned} \mathcal{CB}[\![\text{true}]\!] &= \text{TRUE} \\ \mathcal{CB}[\![\text{false}]\!] &= \text{FALSE} \\ \mathcal{CB}[\![a_1 = a_2]\!] &= \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \text{EQ} \\ \mathcal{CB}[\![a_1 \leq a_2]\!] &= \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \text{LE} \\ \mathcal{CB}[\![\neg b]\!] &= \mathcal{CB}[\![b]\!] : \text{NEG} \\ \mathcal{CB}[\![b_1 \wedge b_2]\!] &= \mathcal{CB}[\![b_2]\!] : \mathcal{CB}[\![b_1]\!] : \text{AND} \end{aligned}$$

We have to use the **stack** rather than work directly with the results.

Finally, we need to translate statements: $\mathcal{CS}[\![.]\!]$: Stm $\rightarrow$ Code.

$$
\begin{aligned}
\mathcal{CS}[\![x := a]\!] &= \mathcal{CA}[\![a]\!] : \text{STORE-x} \\
\mathcal{CS}[\![\text{skip}]\!] &= \text{NOOP} \\
\mathcal{CS}[\![S_1; S_2]\!] &= \mathcal{CS}[\![S_1]\!] : \mathcal{CS}[\![S_2]\!] \\
\mathcal{CS}[\![\text{if } b \text{ then } S_1 \text{ else } S_2]\!] &= \mathcal{CB}[\![b]\!] : \text{BRANCH}(\mathcal{CS}[\![S_1]\!], \mathcal{CS}[\![S_2]\!]) \\
\mathcal{CS}[\![\text{while } b \text{ do } S]\!] &= \text{LOOP}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!])
\end{aligned}
$$

2.6 Semantic function

We can now define a semantic function for **While** (by translating and executing the program on our AM), and it will be called S_{am}.

$$
\begin{aligned}
&\mathcal{S}_{am}[\![.]\!] :\rightarrow (\text{State} \hookrightarrow \text{State}) \\
&\mathcal{S}_{am}[\![.]\!] = \sigma' \text{ if } \mathcal{M}[\![\mathcal{CS}[\![]\!]\sigma = \sigma', \text{ Undefined otherwise} \\
&\mathcal{S}_{am}[\![S]\!]\sigma = (\mathcal{M}^o\mathcal{CS})[\![S]\!]\sigma \\
&\mathcal{S}_{am}[\![S]\!] = (\mathcal{M}^o\mathcal{CS})[\![S]\!]
\end{aligned}
$$

Here's an example using factorials:

$$
\begin{aligned}
&\mathcal{CS}[\![\text{y:=1, while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1)]\!] \\
=\ &\mathcal{CS}[\![y := 1]\!] : \\
&\mathcal{CS}[\![\text{while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1)]\!] \\
=\ &\mathcal{CA}[\![1]\!] : \text{ store-y } : \\
&\text{loop}(\mathcal{CB}[\![y := y * x; x := x - 1]\!]) \\
=\ &\text{push-1} : \text{store-y} : \\
&\text{loop}(\mathcal{CB}[\![x = 1]\!] : \text{neg}, \\
&\qquad \mathcal{CS}[\![y := y * x]\!] : \mathcal{CS}[\![x := x - 1]\!]) \\
&\ldots \\
=\ &\text{push-1} : \text{store-y} : \\
&\text{loop}(\text{push-1} : \text{fetch-x} : \text{eq} : \text{neg}, \\
&\qquad \text{fetch-x} : \text{fetch-y} : \text{mult} : \text{store-y} : \\
&\qquad \text{push-1} : \text{fetch-x} : \text{sub} : \text{store-x})
\end{aligned}
$$

Because we have specified the functions, they are allowed to be undefined (such as when they do infinite loops).

2.6.1 Correctness of translation

We can say that for any arithmetic expression, all intermediate configurations have a non-empty stack.

Similarly, for all boolean expressions b, all intermediate configurations have a non-empty stack.

Note! We don't have to always change state because we defined the program. In fact, if we wanted to allow $\mathcal{A}[\![.]\!]$ to allow state changes, we'd have to change the definition of $\mathcal{A}$. That is to say, if we want to allow 'side-effects', we'd need to change the definition to allow it to return both an integer and a state S.

3 Denotational Semantics

A denotational semantics defines the meaning of a program by a partial function called a **state-transformer** from (initial) states to (final) states. We're gonna be operating at a level that takes functions as arguments, and gives arguments.

A denotational semantics must be **compositional**. This means we must define the meaning of expressions in terms of their *sub-expressions*.

There are two flavours of denotational semantics: **direct-style** and **continuation-style**. We'll mostly look at the first flavour.

We need two more notions to formalise the meaning of if-then-else, and while statements. For this, we'll use **conditional functions** and **fixpoint operators**.

The denotational semantics of **While** statements is specified in a similar way to the functions for arithmetic and boolean operations.

So, we define the direct-style semantic function to be:

$$S_{ds} \quad : \quad : \text{Stm} \ \rightarrow (\text{State} \ \hookrightarrow \ \text{State})$$

Here is what the semantics looks like:

$$\mathcal{S}_{ds}[\![x := a]\!]s = s[x \mapsto \mathcal{A}[\![a]\!]s]$$
$$\mathcal{S}_{ds}[\![\text{skip}]\!] = id$$
$$\mathcal{S}_{ds}[\![S_1; S_2]\!] = \mathcal{S}_{ds}[\![S_2]\!]^o\mathcal{S}_{ds}[\![S_1]\!]$$
$$\mathcal{S}_{ds}[\![\text{if } b \text{ then } S_1 \text{ else } S_2]\!] = \text{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![S_1]\!], \mathcal{S}_{ds}[\![S_2]\!])$$
$$\mathcal{S}_{ds}[\![\text{while } b \text{ do } S]\!] = \text{FIX F where} Fg = \text{cond}(\mathcal{B}[\![b]\!], g^o\mathcal{S}_{ds}[\![S]\!], id)$$

'FIX F' is the (least) fixpoint of the **functional** F, which is the functional of the loop. The g is an implicit argument of the functional F. We'll look at this a bit more, cause it's quite rough.

3.1 Conditional functions

The idea of a **conditional function** is closely related to the denotational semantics of conditionals.

We want to use one of two functions c or d to map some inputs x to some outputs y; and we have a boolean test b for, that will determine which function to apply in each case x.

$$\text{cond} : (X \to T) \times (X \hookrightarrow Y) \times (X \hookrightarrow Y) \to (X \hookrightarrow Y)$$

We give in three functions, and we get out a function that takes X to Y. We can rewrite this as:

$$\text{cond}(b, c, d)x = \begin{cases} c(x) & \text{if } b(x) = tt \\ d(x) & \text{otherwise} \end{cases}$$

Where b, c, d correspond to the functions we supply. We will be interested when X and Y are both states. So:

$$\text{cond} : (\text{State} \to T) \times (\text{State} \hookrightarrow \text{State}) \times (\text{State} \hookrightarrow \text{State}) \times (\text{State} \hookrightarrow \text{State})$$

3.2 Least fixpoint

The idea of the **least fixpoint** of a function is actually quite heavily studied in maths (particularly in lambda calculus).

There are also a lot of heavy duty theorems, but we'll be using this one: **Definition:** For any unary operator $f : X \to X$ on some domain X with a partial order $\leq$:

- A fixpoint of f is any element $x \in X$ such that $f(x) = x$ fix(f) denotes the set of all such fixpoints.
- A least fixpoint of f that is least with respect to the $\leq$ order lfp(f) denotes the least fixpoint (which is unique if it exists).

We are interested in fixpoints of functions between state transformers, and these functions are called functionals.

We also use the notation FIX to denote the least fixpoint of a functional with a respect to $\subseteq$.

The semantics of loops can use the least fix point, and here is an example:

$$\mathcal{S}_{ds}[\![\textbf{while b do S}]\!] = \textbf{FIX F} \text{ where } Fg = \textbf{cond}\ (\mathcal{B}[\![\textbf{b}]\!], g \circ \mathcal{S}_{ds}[\![\textbf{S}]\!], \textbf{id})$$

as

$$\mathcal{S}_{ds}[\![\textbf{while b do S}]\!] = \mathcal{S}_{ds}[\![\textbf{if b then (S ; while b do S) else skip}]\!]$$

The reason we are talking about the fixpoint is going to be explained. We know that whatever the semantics of the while loop is, can be explained by the second statement above. This statement simply unfolds the loop by one step. We know that this is true.

We're trying to defined the semantics of the while loop, because the elements of the while loop is simply a conditional:

$$\begin{aligned}\mathcal{S}_{ds}[\![\textbf{while b do S}]\!] &= \mathcal{S}_{ds}[\![\textbf{if b then (S ; while b do S) else skip}]\!] \\ &= \textbf{cond}\ (\mathcal{B}[\![\textbf{b}]\!], \mathcal{S}_{ds}[\![\textbf{S ; while b do S}]\!], \mathcal{S}_{ds}[\![\textbf{skip}]\!])\end{aligned}$$

The semantics of the `if then else` statement is just the **cond** function.

Now, we've already defined program composition, and we know that loops are composed with the components of the second and first halves of the program.

$$\begin{aligned}\mathcal{S}_{ds}[\![\textbf{while b do S}]\!] &= \textbf{cond}\ (\mathcal{B}[\![\textbf{b}]\!], \mathcal{S}_{ds}[\![\textbf{S ; while b do S}]\!], \mathcal{S}_{ds}[\![\textbf{skip}]\!]) \\ &= \textbf{cond}\ (\mathcal{B}[\![\textbf{b}]\!], \mathcal{S}_{ds}[\![\textbf{S ; while b do S}]\!], \textbf{id}) \\ &\in \textbf{fix}\ (\lambda \textbf{g . cond}\ (\mathcal{B}[\![\textbf{b}]\!], \textbf{g}\ {}^{o}\mathcal{S}_{ds}[\![\textbf{S}]\!], \textbf{id}))\end{aligned}$$

The $\mathcal{S}_{ds}$ means that we have some semantics and we're trying to apply it to some other thing. This last line imagines the input (in our case the **while b do S**) as g. Now, if we can find a fixpoint of the λ function (this means that if we give it an input, we get the same value back). That is to say, if we say g is the semantics of the while loop, and we pass this into the function, we should get g back. This means that if we define the lambda function, the semantics of the while loop must be a fixpoint of the lambda function. Finally, we end up with:

$$\begin{aligned}\mathcal{S}_{ds}[\![\textbf{while b do S}]\!] &\in \textbf{fix}\ (\lambda \textbf{g . cond}\ (\mathcal{B}[\![\textbf{b}]\!], \textbf{g}\ {}^{o}\mathcal{S}_{ds}[\![\textbf{S}]\!], \textbf{id})) \\ &= \textbf{FIX}\ (\lambda \textbf{g. cond}(\mathcal{B}[\![\textbf{b}]\!], \textbf{g}\ {}^{o}\mathcal{S}_{ds}[\![\textbf{S}]\!], \textbf{id}))\end{aligned}$$

This final step is shown using math, and we'll look at this now.

Note! the semantics of the while loop can be undefined if the fixpoint is undefined.

3.2.1 The functional of a loop

The functional $F = \lambda g.\text{cond}(\mathcal{B}[\![\textbf{b}]\!], g\,{}^{o}\mathcal{S}_{ds}[\![\textbf{S}]\!], \textbf{id})$ is referred to as the functional of the loop.

The functional can be seen as a means of finding better and better approximations to the semantics of the loop.

it can be shown that FIX gives the correct semantics for all possible n. Basically, if we could do it infinite times, we'd have a perfect semantics of the particular loop in a particular state, because the loop must terminate in finite steps.

Note that $F^{n}()$ is a correct semantics for all states from which the loop ends in fewer than n iterations (but it is a pretty awful approximation).

A **direct characterisation** of F is any equivalent mathematical expression that does not contain any semantic functions.

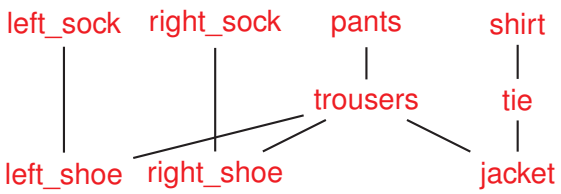
Note! the functions have the following types:

- FIX : $((\textbf{State} \hookrightarrow \textbf{State}) \to (\textbf{State} \hookrightarrow \textbf{State})) \to (\textbf{State} \hookrightarrow \textbf{State})$
- F : $(\textbf{State} \hookrightarrow \textbf{State}) \to (\textbf{State} \hookrightarrow \textbf{State})$
- g : $\textbf{State} \hookrightarrow \textbf{State}$

3.3 More on partial orders

- A (**weak**) partial order is:
  - Reflexive ($a \leq a \forall a$)
  - Transitive ($a \leq b$ and $b \leq c$ implies $a \leq c$ for all a, b, c)
  - Antisymmetric ($a \leq b$ and $b \leq a$ implies $a = b$ for all a, b)
- A **strong** partial order is:
  - Irreflexive $a \leq a$ for no a.
  - Transitive
  - Antisymmetric (because assume $a \leq b$ and $b \leq a$, this contradicts the rules of transitivity, so $a = b$)
- A **total** partial order (or **chain**) is:
  - Connex ($a \leq b$ or $b \leq a$ for all a, b)
  - Reflexive (because setting $a = b$ gives $a \leq a$ or $b \leq a$ for all a.)
  - Transitive
  - Antisymmetric

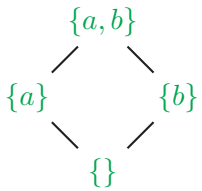
As an example, we can use getting dressed. When we get dressed, we know that we put our *left sock* strictly before our *left shoe*. This is a strict order because we couldn't do it the other way around or at the same time. However, it doesn't matter which way around which sock we put on. Similarly, our boxers go on before our trousers, and it is also better to put on trousers before your shoes.



So, this is an example of a **weak** partial order because it is **reflexive**. It's not a chain because it's not

a straight line. It's not a reflexive

Here's another example:



Here, this is another weak partial order (it actually represents to $\subseteq$ relation). Finally, here's a chain.

This is the $\leq$ relation, and it is a **total** order, shown with the implicit transitive and reflexive links omitted.

3.4 Fixpoints of real functions

Let's look at some real functions, and their respective fixpoints:

square	$= \lambda x.x * x$	fix(square) $= \{0, 1\} = 0$
double	$= \lambda x.x + 2$	fix(double) $= \{0\} = 0$
increment	$= \lambda x.x + 1$	fix(increment) $= \{\} =$ undefined
identity	$= \lambda x.1$	fix(identity) $= \mathbb{R} =$ undefined
one	$= \lambda x.1$	fix(one) $= \{1\} = 1$
magnitude	$= \lambda x.$if $x \leq 0$ then x else $-x$	fix(magnitude) $= \mathbb{R}^+ = 0$

The lowest fixed point for each function are on the right.

3.5 Divergence and strictness

A **divergent** definition means a definition with *non-termination* or *undefined* cases. Haskell deals with this by assigning expressions a **bottom** value ($\perp$). Every data type in Haskell has a $\perp$ (one for

each type).

A Haskell function is said to be **strict** if the function is divergent whenever it is applied to a divergent argument. For example, $*2$ is as strict as $(*2)\bot = \bot * 2 = \bot$. Therefore, $\bot$ is technically the least fixpoint of any strict Haskell function, but that is a little stupid so we ignore it a little bit.

To compute the least fixpoints of functions, we can use the following definition:

- `fix f = f (fix f)`

For example:

- `fix (*2)= (*2)(fix (*2))= (*2)(*2)(fix (*2))= ... = (bot)`
- `fix (const 2)= (const 2)(fix (const2))= 2`

We can use fixpoints to redefine recursive functions (such as factorials). Look at this example:

- `fac = fix (\f n -> if n==0 then 1 else n*f (n-1))`

Similarly (by making the anonymous functions explicit):

- `fac = fix fg`
- `fg f = g where g n = (if n==0 then 1 else n*f (n-1))`

We can verify this by:

- `fac 0 = (fix fg)0 = fg (fix fg)0 = 1`
- `fac 1 = fg (fix fg)1 = 1 * (fix fg)0 = 1`
- `fac 2 = fg (fix fg)2 = 2 * (fix fg)1 = 2`
- `fac 3 = fg (fix fg)3 = 3 * (fix fg)2 = 6`

Any fixpoint `f` of `fg` must clearly compute factorials as this would imply that:

- `f = fg f`
- `f n = fg f n`
- `f n = (if n == 0 then 1 else n * f (n-1))`

Now, we can see `fg` as a function that takes an approximation to the factorial function and returns a better approximation (sound familiar??) For example, we could take `id` as an initial approximation. It will get at least two outputs right ($1! = 1, 2! = 2$). But, `(fg id)` is a better approximation as now it gets three right. Continuing on like this gets $n + 2$ values correct (shown by the Kleene fixpoint theorem which we don't really need to know).

4 Chain-Complete Partial Orders

Sets can have upper and lower bounds depending on where they come in the order of chains. For example, those at the top of the set are the upper bound of every element since they are unable to have an upper bound themselves, while the ones at the bottom are the lower bound, since they have no lower bound themselves.

4.1 Definitions of PO-Set (partially ordered set)

A PO-Set (D, $\sqsubseteq$) is called a *chain-complete* partially ordered set (ccpo) whenever the least upper bound $\sqcup y$ exists for all chains $Y \subseteq D$.

- To be a CCPO, each chain must have an upper bound

Furthermore, a PO-Set (D, $\sqsubseteq$) is called a *complete lattice* whenever the least upper bound $\sqcup Y$ exists for all subsets $Y \subseteq D$.

- To be a lattice, each chain must have a **

Note that every CCPO (D, $\sqsubseteq$) has a (necessarily unique) element denoted $\bot = \sqcup \emptyset$. This means that it is given by the lease upper bound of the empty chain. First observe $\emptyset$ is a chain, since we know that $\emptyset \subseteq D$ by the basic set properties and $d \sqsubseteq e$ vacuously holds for all $d, e \in \emptyset$ (of which there are none). So, the least upper bound $\bot$ **

The question is: is our relation a chain-complete partial order (from the slides)? The answer, simply, is no.

- This is because we have a whole bunch of items at the end. There is no least element of the ordering, so it cannot be a CCPO (this is an important part of the CCPO).

What we can do is the **lifted** relation, which we obtain by adding a least element $\bot$. But, does this now make it a CCPO? Yeah, it does. Nice. HOWEVER, it does not make it a **complete lattice**.

5 The language Exc

Exc is an extension of the **while** language whose statements are defined like as follows. Exc stands for extension:

$$S ::= x := a \mid \text{skip} \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S \mid \text{begin } S_1 \text{ handle } e : S_2 \text{ end} \mid \text{raise } e$$

The idea is that whenever a *raise* exception instruction is encountered, then the execution of the current (encapsulating blocks of) code (such as S_1) is aborted and control is passed to the (most recently defined) handler (such as S_2) for the exception (e).

Consider the following **Exc** statement.

```
begin
    while true do
        if x <= 0
        then raise exit
        else x := x-1
    hand exit:
        y:=7
end
```

If this program is run when x is negative then it will terminate after setting y to 7 (leaving x unchanged).

The meaning of an exception needs to capture the result of executing the relevant handler and following by any remaining code after the definition of that handler.

But this requires a more complicated semantic definition which allows, for example, for the fact that (differently from the *while* language) we don't necessarily have to continue running S_3 in the following program after running either S_1 or S_2 (if either of them raises an exception) **

5.1 Continuation-style denotational semantics

We can define the set **Cont** of continuations as the set of all partial functions between states so that **Cont** = State $\hookrightarrow$ State

Intuitively, a continuation is simply a state transformer that describes the input-output behaviour of a (part of) program.

This concept allows us to define the behaviour o a statement by the effect that it has (c') on a continuation defining the behaviour of the code following that statement (c).

Recall the **direct style denotational semantics** (3). Continuation style denotational semantics are as follows:

$$
\begin{aligned}
&S'_{cs} : \textbf{Stm} \rightarrow (\textbf{Cont} \rightarrow \textbf{Cont}) \\
&S'_{cs}[\![x := a]\!]cs &&= cs[x \mapsto \mathcal{A}[\![a]\!]s] \\
&S'_{cs}[\![\textbf{skip}]\!] &&= \textbf{id} \\
&S'_{cs}[\![S_1; S_2]]\!] &&= S'_{cs}[\![S_1]\!]^o S'_{cs}[\![S_2]\!]
\end{aligned}
$$

6 Axiomatic Semantics

We are usually more interested in some properties of a program. For example, given the program `while not(x=1)do (y:=y*x; x:=x-1)`. An important property might be that if `y` is initially 1 and `x` is a strictly positive integer `n` then `y` will become `n!`.

Typically, we are interested in one of two common classes of properties:

- **Partial correctness properties** – This states that *if* a program terminates, then a certain relationship will hold between the initial and final variable values.
- **Total correctness properties** – This states that a program will terminate and a certain relationship will hold between the initial and final variable values.

The axiomatic semantics allows us to formalise the interesting properties of a program using **preconditions** (which are true in the state prior to executing the program) and **postconditions** (which are true in any state right afterwards).

Such properties are denoted by *assertions* in the form of triples known as *Hoare triples*. They are written:

Precondition Program Postcondition

The axiomatic semantics is a calculus for deriving correct triples. There are two types: *partial* and *total* correctness.

Given the program `while not(x=1)do (y:=y*x; x:=x-1)`, we can state whether the following assertions are intuitively true or false:

When we write postconditions, it is important to make it very good.

We have both **program** variables and **logical** variables, the logical ones enable us to 'remember' the value of `x` that we had before the start of the program.

We write the *partial* correctness assertions with curly brackets {}, and *total* correctness assertions with square brackets [].

An axiomatic semantics consists of a set of *axioms* and a set of *rules* that provide a method for finding all true assertions. An axiom is a rule with no premises (which means that the conclusion can be assumed at any time). The individual axioms and rules are specified by **schemata** which contain meta variables.

We're defining the behaviour of a program by saying what it does to the properties of the program. For example:

{P} skip {P}

Here, **{P}** is the same before and afterwards, showing that the properties have not changed.

In the assignment example, we write:

P(a) x:=a P(x)

Note how we use **P(a)** **before** and not after. I don't really understand why so go and look at this.

Basically, we say that if a statement **P(a)** (that might involve the program variable **x**) is true after assigning **x** to **a**, then the statement **P(a)** (where all occurrences of **x** are replaced by **a**) must have been true before.

P is any expression in the assertion language, while **x** is any program variable, and **a** is any arithmetic expression in **While**.

To prove an assertion about a composition of two program statements, it suffices to find an intermediate property (sometimes called a **midcondition**). This is a post-condition of the former and a pre-condition of the latter.

We can't show the premises directly using the assignment axiom. We need all of the later consequence rules to link them together in a real proof.

Proving an assertion in the axiomatic semantics amounts to just chaining the rules together in the form of a proof tree. We normally label each rule with the schema that it comes from, and sometimes we draw dotted lines above the leaves (that are instances of ASS axioms or SKIP axioms.)

Each application of the consequence rule will be numbered with a number we can use to identify the associated entailment proofs.

7 Lexing and Parsing

Lexer specifications:

- Regular Expressions
- Automata
- The fslex lexer generator tool

Parser specifications

- Grammars
- The *happy* parser generator tool

Abstract syntax is pretty boring. We have stuff like:

```
Prim "+" (CstI 7) (Prim "*" (CstI 9) (CstI10))
```

But, programmers want to be able to write source code: which is in text!

The aim of the compiler front end is to decide whether a program respects the language rules. It can perform *lexical analysis* to decide whether the individual tokens are well formed, as well as *syntactical analysis* to decide whether the composition of tokens is well formed. It also has a *type checker* to verify that the program compiles with some of the language semantics.

7.1 Lexical Analysis

Lexical relates to the words of the vocabulary of a language. For example, in the sentence: "My mother **cooooookes** dinner not", the lexical analyser would split the input program as seen as a stream of characters and turn it into a sequence of **tokens**. Tokens are the words of the programming language, such as keywords, numbers, tokens, comments, parentheses and so on.

Lexical analysis transforms a character stream to a token sequence. The tokens can be:

- (fixed) vocabulary words such as keywords (like `let`), built-in operators (*, :), and special symbols ([,]).
- Identifiers and Number literals are **classes** of tokens which are formed compositionally according to certain rules

For example, we allow integers in decimal format like 234, 0, 8 but not 08 or abc. Additionally, in Haskell, we allow variable names like `x, x3y, aB` but not `Ab, 3d`.

A decimal integer is either 0 or a sequence of digits 0-9 that does not start with a 0. A Haskell variable name starts with a lowercase letter followed by a sequence of lower/upper case letters or digits that does not form a reserved keyword.

7.2 Regular expressions

A regular expression describes a set of *strings*.

R.E. r	Meaning	Language L(r)
a	Symbol a	{ "a" }
ϵ	empty sequence	{ "" }
r_1, r_2	r_1 followed by r_2	$\{ s_1 s_2 \mid s_1 \in L(r_1), s_2 \in L(r_2) \}$
$r*$	zero or more r	$s_1 \cdots s_n \mid s_i \in L(r), n \geq 0$
$r_1 \mid r_2$	r_1 or else r_2	$L(r_1) \cup L(r_2)$

We can use regex to define integers in decimal format. For example: $(1|2|3|\ldots|9)(0|1|2|\ldots|9)|0$.

A *lexer* converts a character stream to a token stream (as previously discussed) while a *lexer specification* is a description of tokens. A *lexer generator* takes as input a lexer specification and generates a lexer.

7.3 Finite Automata and State Machines

A finite automaton is a graph of states and labelled transitions. A finite automata (FA) accepts a string s if there is a path from start to an accept state such that the labels make up s. The ϵ transition does not contribute to the string.

> For every regular expression r there is a finite automaton that recognizes exactly the strings described by r.
> The opposite of this statement is also true.

7.4 Nondeterminism to Determinism

A deterministic FA has no ϵ transitions, and distinct labels on all transitions from a state. A DFA is easy to implement with a 2D table. It decides in *linear time* whether it accepts a string s.

For every NFA there is a corresponding DFA.

7.5 Context Free Grammars

Context free grammars are pretty powerful. We have a set of rules with respect to some derivation.

Note: there are a number of derivations sometimes that end up in the correct result. It is possible to do things in a slightly different order and still get the same result. If this is the case, then the grammar is *ambiguous*. Ambiguous is not good.

Parsing is the inverse of derivation. That is, determine whether a string can be derived and if so, reconstruct the derivation steps. There are a few ways to do this:

- Hand-written top-down parsers
- Generated bottom up parsers
  - Write parser specification
  - Use tool to generate parser

A *parser* converts a token stream into an abstract syntax tree. The *parser specification* describes the well-formed streams. The *parser generator* takes as input a parser specification and generates a parser.

8 Parsers

8.1 Parser specification

Here are a list of the tokens that the parser will recognise:

```
let
in
end
num
var
'='
'+'
'-'
'*'
'('
')'
```