1 Introduction

In previous lectures we described functions as semantic objects that map arguments to values and we used programs to describe their syntax. In this lecture we will see three ways of writing programs:

- imperative programs,
- circuits,
- functional programs.

We have seen imperative and functional programs before, but now we will talk more about their semantics.

2 Imperative programs

To put it simply, an imperative program is a sequence of instructions. Formally, it is a function from states to states, where a state is a value of program variables. As an example, consider the program below which computes the factorial of $n$.

```plaintext
procedure Fact(n)
  x := 1
  y := 1
  while x \neq n do
    x := x + 1
    y := y \times y
  end while
  z := y
  \{ z = n! \}
end procedure
```

The annotation $\{ z = n! \}$ is an assertion that the value of $z$ at that point is the factorial of $n$.

Semantics of imperative program can be described in three ways:

1. **Operational semantics**: consists of rules that simulate how the program is executed. Rules have the form

   $$< A, s > \rightarrow s'$$

   1
which says that after executing program instructions $A$ from some initial state $s$ we will end up in state $s'$.

2. **Axiomatic semantics**: instructions are annotated by invariants. A statement

$$\{p\} A \{q\}$$

means that if assertion $p$ is true before $A$ is executed, then assertion $q$ will hold after the execution.

3. **Denotational semantics**: semantics of instructions is given as a function from states to states after an instruction is executed:

$$\llbracket A \rrbracket = \lambda s. \text{“expression on } s \text{ to compute } s'\rrbracket.$$

Operational and axiomatic semantics were described in the previous lectures. The rules for denotational semantics will be given in Section 5.

## 3 Circuits

Circuits are another way of writing programs. A circuit is a graphical description of a complex function that is built from simpler functions. The figure below show three circuits that define simple arithmetic functions.

These are examples of *combinatorial circuits*, where the output at time $t$ depends only on the input at time $t$. A circuit whose output at time $t$ may depend on the inputs at times $0, 1, \ldots, t-1$, at $t$ is called **sequential**. Sequential circuits have a “memory” that stores the current value of variables; the memory allows this type of circuits to model programs that do more advanced computations.

The figure below shows a sequential circuits that computes the factorial of input $n$. Variables $x, y$ serve as the memory of the program and their initial value is 1. Function $+1$ adds 1 to its input and function $\ast$ multiplies its arguments. The meaning of function $=$ is explained in the figure. Output $z_2$ shows the final result and $z_1$ becomes true when the result is ready.
Let us now see how the states of the memory changes over time. Let us write $x(t)$ and $y(t)$ for the value of these variables at time $t \geq 0$. The initial conditions are

$$x(0) = 1 \quad y(0) = 1.$$ 

At every time step the value of $x$ is incremented by one and $y$ becomes the multiplication of the last values of $y$ and $x$. Thus, the state at time $t + 1$ depends on the state at time $t$ by the following equations:

$$x(t + 1) = x(t) + 1 \quad y(t + 1) = y(t) \ast x(t).$$

The following table shows the values of $x$, $y$ for time steps $t \in \{0 \ldots 5\}$:

<table>
<thead>
<tr>
<th>time $t$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>$y$</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>24</td>
<td>120</td>
<td></td>
</tr>
</tbody>
</table>

Sequential circuits – and programs in general – are equivalent to reactive modules. For instance the factorial circuit is the following reactive module

```plaintext
module Factorial
state $x, y \in \mathbb{N}$
input $n \in \mathbb{N}$
output $z_1 : \{false, true\}, z_2 \in \mathbb{N}$
initially $x = 1, y = 1$
update
$$x < n \rightarrow x := x + 1; y := y \ast x$$
$$x = n \rightarrow z_1 := true; z_2 := y$$
```

### 4 Functional programs

Functional programs is yet another method of defining programs. Let us look at a functional program $f$ that computes the factorial:

$$f = \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n \ast f(n - 1)$$
To compute complex functions, like the factorial, a program needs to compute new values from the old ones. This can be done by one of the following means:

1. memory, like in sequential circuits,
2. loop statements, like in imperative programs,
3. recursion, like in functional programs.

In functional programs the intermediate computations are not stored in memory or in variables, but are hidden in recursion steps. For instances consider the recursions for $f(5)$

\[
\begin{align*}
\text{intermediate result} & \quad 5 \ast 4 \ast f(3) = 5 \ast 4 \ast 3 \ast f(2) = \ldots = 5 \ast 4 \ast 3 \ast 2 \ast 1 \ast 1 = 120.
\end{align*}
\]

In circuits and imperative programs memory and variables change over time. In functional programs the values change over recursion depth.

The theoretical foundation of functional programs is the $\lambda$-calculus. An expression in $\lambda$-calculus is called a $\lambda$-term. Let $x$ be some variable and $c$ some constant, then a $\lambda$-term can be built according to the following rules

\[
\begin{align*}
& \frac{x}{\lambda x. M} \quad \frac{M}{N} \quad \frac{M \circ N}{M} \quad \frac{c}{M}.
\end{align*}
\]

where $\circ$ is any primitive function like addition or multiplication.

The two rules on the right are not part of the original $\lambda$-calculus, but we added them for convenience. The first and fourth rule say that any variable or any constant is a $\lambda$-term. For example constant 3 and variable $x$ are $\lambda$-terms. Variables, unlike constants, do not have a meaning on their own – an expression “$x$” can mean anything. We give a meaning to a variable by binding it to a function argument. For instance $\lambda x. x$ is a function that maps any argument to itself and function $\lambda x. x + 3$ maps the argument $x$ to the argument $x + 3$. The second rule, called the $\lambda$-abstraction, says that $\lambda$-term $M$ can be turned into a function.

The third rule is called application and states that $\lambda$-term $N$ can be given as an argument to a $\lambda$-term $M$. For example, let $M$ be $\lambda x. x + 3$ and $N$ be 5, then the following expression means that we wish to compute $x + 3$ for argument 5:

\[(\lambda x. x + 3)(5).
\]

To evaluate this function we replace every occurrence of $x$ in the function by 5; this operation is called $\beta$-reduction. The $\beta$–reduction for this expression gives:

\[(\lambda x. x + 3)(5) \xrightarrow{\beta} x + 5.
\]

By looking at rules for $\lambda$-calculus we see that it does not allow recursive definitions. Our factorial program $f$ can be written in some functional programming language, but cannot be expressed in $\lambda$-calculus. In $\lambda$-calculus we were interested in a function that computes the factorial, but without the recursive definition. To obtain such function we use the fixed point combinator $Y$. This combinator is defined as:

\[Y = \lambda y. (\lambda x. y(xx))(\lambda x. y(xx)).\]
To see how the operator works let us now apply some \( \lambda \)-term \( g \) to \( Y \):

\[
Yg = (\lambda y. (\lambda x. y(xx))(\lambda x. y(xx)))g
\]

\[
\xrightarrow{\beta} (\lambda x. g(xx))(\lambda x. g(xx)) \quad \text{substitute } y
\]

\[
\xrightarrow{\beta} g((\lambda x. g(xx))(\lambda x. g(xx))) \quad \text{substitute } x
\]

\[= g(Yg).\]

The value of \( Yg \) is the infinite number of compositions of \( g \):

\[Yg = g(g(...)).\]

Under certain conditions \( Yg \) returns a unique function in \( \lambda \)-calculus. Let us use the fixed-point combinator to convert our functional program \( f \) to a function in \( \lambda \)-calculus:

\[\text{fact} = \varepsilon f. (f = \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \ast f(n - 1)),\]

where \( \varepsilon f \) is a shorthand for \( Y\lambda f \). As an exercise the interested reader may check what is the value of \( \text{fact} \ 5 \) by applying the fixed point combinator.

5 Denotational semantics

After seeing the fixed-point operator, we can define the denotational semantics of imperative program. This semantics describes the meaning of a program by functions from states before an operations to states after the operations. The functions are:

Assignment:

\[[x := e] = \lambda s. s'\quad \text{where } s' \text{ results from } s \text{ by mapping } x \text{ to } [e](s)\]

Concatenation:

\[[A; B] = \lambda s. [B][A](s)\]

Branch:

\[[\text{if } c \text{ then } A \text{ else } B] = \lambda s. \text{if } [e](s) = \text{true then } [A](s) \text{ else } [B](s)\]

Loop:

\[[\text{while } c \text{ do } A] = \lambda s. \text{if } [e](s) = \text{false then } s \text{ else } [\text{while } c \text{ do } A][A](s))\]

\[= \varepsilon f. (f = \text{if } [e](s) = \text{false then } s \text{ else } [\text{while } c \text{ do } A][A](s)))\]

We can notice that the loop function uses the fixed-point combinator. This makes sens, because the state resulting after executing the loop

\[\text{while } c \text{ do } A\]

from state \( s \) is the effect of applying operation \( A \) as long as condition \( c \) is satisfied. But this can be see as infinite composition of the function

\[\text{if } [e](s) = \text{false then } s \text{ else } [\text{while } c \text{ do } A][A](s)\]

which is the argument of the fixed-point combinator.
6 Summary

We can summarize the formalisms that we have seen in the discrete part of the course:

1. rules, automate and derivations as a way to define languages,

2. reactive modules as a way of modeling state-transition systems,

3. programs which compute functions; their meaning was described by operational, axiomatic and denotational semantics.

These formalism are a mathematical way of describing discrete dynamic systems.