Handout 9: Imperative programs and the Lambda Calculus

In this handout, we discuss how imperative programming and the lambda calculus framework can work together.

In contrast to functional programming, where all computations are carried out by applying functions to arguments, imperative programming works by executing commands that perform *actions*. The most basic actions are input/output operations. The next level of actions involve creating mutable variables (storage cells or data structures), and continally modifying them. In both cases, actions involve manipulating "state", i.e., modifying some unseen dynamic state of the world that the program is operating in. But once we have the concept of actions, the lambda calculus framework is still applicable for adding the concepts of functions and other type structures.

We consider two ways of doing this:

- In Haskell, which was designed to be a "functional programming language", a type for actions is added in using a type constructor called **`IO`**. The type **`IO`** `t` represents actions that carry out some state operations and eventually return a value of type `t`.
- In Idealized Algol, a theoretical abstraction of the programming language Algol 60, we have two primitive types:
  - **`comm`**, short for commands, which represent actions that purely manipulate the state.
  - **`exp`**, short for expressions, which read the state and return values. Expressions do not involve any modification of the state.

The relationship between the two frameworks is quite close. Essentially, the Haskell type **`IO`** `t` combines both the features of **`comm`** and **`exp`**. This makes the framework slightly simpler, but it can be cumbersome if imperative computations are performed extensively. The Algol framework is more convenient if imperative computations are considered more essential. All imperative programming languages like C and Java can be regarded as variants of Algol.

Imperative programming in Haskell

1. IO type constructor. When Haskell was designed a type constructor called **IO** was defined for representing input/output operations. Later it was also used to incorporate storage variables and other actions. **IO** is a *type constructor*. Its parameter, say T, represents the type of result the IO-action returns. In other words, **IO** T is the type of T-returning actions.

2. Constants for IO. The primitive operations for **IO** type computations are as follows:

```
return :: t → IO t
>>= :: IO t → (t → IO u) → IO u
```

- The **`return`** operation takes a t-typed argument, say x. The expression **return** x represents a null action that does nothing on the state and simply return the value x. In other words, **`return`** represents a "null action".

- The **>>=** operation (normally read as "bind") represents sequencing two state actions. An expression of the form C **>>=** F carries out the actions represented by C and F in that order. However, since actions return values, the value returned by C is passed as an argument to F. This explains the type of the **>>=** operator: C is of type **IO** t. So, it return a t-typed result value, say v. F is of type $t \to$ **IO** u. So, when applied to v, $F(v)$ is a u-returning action. So, on the whole, C **>>=** F is a u-returning action, i.e., it is of type **IO** u.

Here is an example to illustrate these operations:

```
dialogue = putStrLn "Type in a string" >>=
               \() -> getStrLn >>=
                 \val -> putStrLn "you typed " + val
```

The primitive operations `putStrLn` and `getStrLn` are meant for outputting and inputting strings on an entire line. The action `dialogue` above, first outputs the string "Type in a string." This is an action of type **IO** `()`. Next it inputs a string, an action of type **IO** `String`. It calls the string obtained from the input `val`, and finally writes back this string on the output stream.

We normally type the above code with line breaks as follows:

```
dialogue = putStrLn "Type in a string" >>= \() ->
           getStrLn >>= \val ->
           putStrLn "you typed " + val
```

This seems a bit strange because we are splitting the formal parameter of a lambda abstraction and the body of the lambda abstraction into separate lines. However, this style of typing it is in fact easier to read.

1. Ouput "Type in s astring". Let the result be `()`.
2. Read a string. Let the result be `val`.
3. Output "you typed" and `val`.

So the lambda abstraction allows the English language locution "Let the result be x." The fact that the scope of a lambda abstraction extends as far to the right (and down) as possible, means that the variable x mentioned in the locution can be used in the rest of the body of the definition.

3. Pipe operator. Actions returning `()` are very common. To make them convenient, it is conventional to use another operator called "pipe":

```
>> :: IO t → IO u → IO u
c >> d = c >>= (\x -> d)
```

So, the expression `c >> d` simply means "do `c` and `d` in sequence, and return the value returned by `d`". The value returned by `c`, if any, is discarded. But most often `t` would be the Unit type, so no useful result is discarded in this fashion. Using the pipe operator, we can simplify the above dialogue as follows:

```
dialogue = putStrLn "Type in a string" >>
           getStrLn >>= \val ->
           putStrLn "you typed " + val
```

4. References. For every type T, Haskell allows us to create *references* (i.e., mutable variables) that can hold values of type T. Their type is denoted **Ref** T.
The constants (primitive operations) dealing with references are as follows:

```
readRef :: Ref t → IO t
writeRef :: Ref t → t → IO ()
newRef :: t → IO (Ref t)
```

- The action **readRef** v reads the reference v and returns it. This is a *state-reading action*. It does not change the state.
- The action **writeRef** v x writes the value x to the reference v and returns nothing.
- The operation **newRef** creates a new reference. The initial value of the reference will be the argument given to **newRef**.

Using these operations, here is an imperative program to calculate the factorial of a non-negative integer n:

```
factorial :: Int -> IO Int
factorial n =
   newRef 1 >>= \prod ->
   loop n
      where loop n = if n == 0 then
                       readRef prod
                     else
                       readRef prod >>= \p ->
                       writeRef prod (p * n) >>
                       loop (n-1)
```

This program calculates the factorial using a mutable reference called prod. The reference is initialized to 1, which is the identity of multiplication. Then a loop is executed, where, in each iteration, the reference prod is multiplied by an integer between $1, \ldots, n$. In the end, the value of prod is returned as the result of the computation.

5. Higher-order example. We can define higher-order operations dealing with **IO** computations in the usual manner. For example, the function

```
forEach :: (a -> IO ()) -> [a] -> IO ()
forEach p [] = return ()
forEach p (x:xs) = p x >> forEach p xs
```

defines a 'forEach' loop operator that carries out a computation p for each element of a list. It can even be defined using foldr.

```
forEach p l = foldr (>>) (return ()) (map p l)
```

Note: The expression **return** () is not executed when foldr is called! Haskell is a call-by-name language. Rather, it is executed if and when foldr forces its execution.

6. Semantics. Intuitively, the semantics of **IO** T is something akin to a function from states to states along with a result value of type T, i.e.,

$$[\![\textbf{IO}\ T]\!] = \mathit{State} \to \mathit{State} \times T$$

The input state is the initial state of all the system. The output state is the final state at the end of the computation. The value of type T is the returned result.

The primitive operations have an easy interpretation in terms of this semantics:

$$\begin{array}{rcl} [\![\textbf{return}\ x]\!] & = & \lambda s.\,(s, x) \\ [\![c \texttt{ >>= } f]\!] & = & \lambda s.\,\textbf{let}\ (s', x) = c(s)\ \textbf{in}\ f(x)(s') \end{array}$$

The **return** x computation produces the final state the same as the initial state (i.e., does not involve any state change), and gives the argument x as the return value of the compuation.

The computation c **>>=** f first runs the computation c in the initial state (s), obtaining a new state s' and a result x. Then it applies f to x, obtaining a computation, and runs it in the state s'. The final state and the return result are those produces by $f(x)(s')$.

Idealized Algol

John Reynolds[1] proposed that Algol-like languages should be viewed as *typed* lambda calculi, which share the procedure mechanism of the lambda calculus, but have base types that represent imperative computations (which are not by themselves part of the lambda calculus). His formal language is called *Idealized Algol*. It represents a very satisfying view of Algol-like programming languages, and we examine it in this section.

7. Commands and expressions. In contrast to the Haskell **`IO`** type constructor, which represents *all* stateful computations, Idealized Algol has two separate types:

- **`comm`**, for commands (or "statements", in the normal parlance), which modify the state, and
- **`exp`**, for expressions, which read the state and calculate result values.

In Reynolds's view, expressions in Algol-like languages calculate only a limited types of values, which he called *data types*. They include all the primitive types like integers, reals, characters etc. We will use the greek letter δ to range over such "data types".

8. Primitives for expressions. The fact that expressions only read the state (and do not modify them) allows us to use more convenient primitive operations than Haskell's **`IO`** types. For example, we can write $E_1 + E_2$ where E_1 and E_2 are expressions. We do not need the Haskell's cumbersome notations such as E_1 `>>= \x -> ` E_2 ` >>= \y -> return x+y`. Note that the common properties of the operations such as $E_1 + E_2 = E_2 + E_1$ still hold. This is because expressions represent read-only computations.
Moreover, in Algol-like languages, we do not have terms denoting plain values. We only have "expressions", which are always allowed to read the state. Therefore, we use the $+$ operator for expressions only. There is no other $+$ operator for values. With this understanding, we use primtives such as the following for expressions:

```
0, 1, 2, ...:: exp[int]
true, false :: exp[bool]
+, -, ... :: exp[int] → exp[int] → exp[int]
==, <, ... :: exp[int] → exp[int] → exp[bool]
&&, || :: exp[bool] → exp[bool] → exp[bool]
not :: exp[bool] → exp[bool]
```

9. Primitive operations for commands. Commands in Algol-like languages are executable actions. They do not return any values. So the primitive operations for them are the following:

```
skip :: comm
; :: comm → comm → comm
if :: exp[bool] → comm → comm → comm
```

The **`skip`** command is a null action. The "**`;`**" operator is for sequencing. These two operations are akin to Haskell's **`return`** and **`>>=`** operators. The **`if`** operator is for conditional execution.

10. Semantics. In the state-based semantics, commands are regarded as functions from states to states, and expressions are regarded as functions from states to values:

$$\begin{aligned} [\![\mathbf{comm}]\!] &= \mathit{State} \to \mathit{State} \\ [\![\mathbf{exp}[\delta]]\!] &= \mathit{State} \to [\![\delta]\!] \end{aligned}$$

[1]Reynolds, John. The essence of Algol, in *Algorithmic Languages*, North-Holland, 1981.

The primitive operations have an easy interpretation in terms of this semantics:

$$\begin{array}{rcl}
[\![\textbf{skip}]\!] & = & \lambda s.\, s \\
[\![c_1\,;c_2]\!] & = & \lambda s.\, c_2(c_1(s)) \\
[\![\textbf{if}\; b\; c_1\; c_2]\!] & = & \lambda s.\, \textbf{if}\; b(s)\; \textbf{then}\; c_1(s)\; \textbf{else}\; c_2(s) \\
[\![0]\!] & = & \lambda s.\, 0 \\
[\![e_1 + e_2]\!] & = & \lambda s.\, e_1(s) + e_2(s)
\end{array}$$

(We have shown a couple of expression primitives for illustration; the rest are similar.) Note that **skip** is the identity function when viewed as a state-to-state function ("do nothing") and ";" is function composition. The **if** operation, uses the initial state *twice*, once for evaluating the boolean expression and another time for exeucting one of the then- or else-branches.

11. Mutable variables. Mutable variables are simply called "variables" in Algol-like languages. The term "reference", as used in Haskell, was introduced by Algol 68, but by that time, the term variable was already in common use.

Just as expression results are limited to data types, variables are also limited to data types. The primitive operations on them are the following:

```
read :: var[δ] → exp[δ]
write :: var[δ] → exp[δ] → comm
local[δ] :: (var[δ] → comm) → comm
```

The **read** operation, similar to **readRef** in Haskell, reads the value of a variable. If V is a variable, then **read** V is an expression whose effect is to read the value of V and return it.

The **write** operation, similar to **writeRef** in Haskell, writes a new value into a variable. If V is a variable and E an expression then **write** V E is a command whose effect is to evaluate E and assign its value to V. The difference from Haskell's **writeRef** is that the second argument is an expression, not a value. This expression is evaluated in the current state and the resulting value is written to V.

The **local** operation is similar to Haskell's **newRef**, but there are several important differences. First of all, the type itself looks strange. That is because it is simulating the effect of **IO** type constructor which does not exist in Idealized Algol. But its use is similar. To create a new integer variable x and use it in a command C, we write:

```
local[int] \x -> C
```

This is similar to writing the corresonding Haskell code:

```
newRef 0 >>= \x -> C
```

So we can think of **local** operator as subsuming the **>>=** operator that is required in Haskell. In general, any type of the form (T -> **comm**) -> **comm** has the effect of **IO** T.

12. Example. Using the above operations, we can write an imperative program to compute the factorial as follows:

```
factorial :: exp[int] -> var[int] -> comm
factorial n result =
   write result 1;
   loop n
      where loop n = if n == 0 then
                        skip
                      else
                        write result (read result * n);
                        loop (n-1)
```

The differences from the Haskell version of the program are that:

- the first argument is an expression rather than a value,
- we pass in a variable through which the result of factorial will be returned, and
- the overall type of the factorial program is **comm**.

Functions whose result type is **comm** are called *procedures*.

13. Syntactic sugar for variables. In practice, we use syntactic sugar for the variable operations, whereby **read** operations are omitted (the compiler can fill them in) and the **write** operations are written using the assignment operator symbol "`:=`". Using these notations, the factorial program is written as follows:

```
factorial :: exp[int] -> var[int] -> comm
factorial n result =
    result := 1;
    loop n
        where loop n = if n == 0 then
                           skip
                         else
                           result := result * n;
                           loop (n-1)
```

14. Block expressions. Normally, we would want to treat factorial as a "function" so that we embed it in expressions such as `factorial n < 100`. The above treatment only gives us the ability to write `factorial` as a procedure, which has to be called in a command. To get around this limitation, Reynolds proposed a concept called "block expression". (The term "block" here refers to a command block, i.e., a possibly compound command.) This is expressed as a primitive operation of the following type:

blockexp[δ] :: (**var**[δ] $\rightarrow$ **comm**) $\rightarrow$ **exp**[δ]

Note that the type is similar to the **local** operator. The difference is that the final result is an expression rather than a command. A block expression **blockexp** P works by first creating a local variable `result` and passing it to P. The procedure P is expected to assign a value to `result`, but other than that it cannot modify any non-local variables. At the end of the execution of P, the value of `result` is produced as the value of the expression.

Using this operator, the factorial program can be written as follows:

```
factorial :: exp[int] -> exp[int]
factorial n =
  blockexp \result ->
    result := 1;
    loop n
        where loop n = if n == 0 then
                           skip
                         else
                           result := result * n;
                           loop (n-1)
```

Programs of this kind were called "function procedures" in Algol 60. Internally, they are procedures executing commands. But from the outside they appear as functions. In case a function procedure modified a non-local variable, it was called *side effect*. Side effects were considered a poor programming practice, because they break the normal reasoning about expressions. For example, algebraic laws such as $E_1 + E_2 = E_2 + E_1$ would not hold if expressions had side effects. However, Algol 60 was unable to prohibit them. Reynolds produced a type system called *Syntactic control of interference*, which was able to prohibit side effects in expressions. (We will not study this system in this notes.)

Algol as it really was

The original Algol 60 did not use the functional programming syntax we used. But it was quite close. In this section, I show the (almost) original syntax and type system of Algol 60 so you can relate to it.

In the first place, Algol did not have the full higher-order type system. It only had functions that had the result types **comm** and **exp**[δ]. Functions with the result type **comm** were called "procedures". Those with result type **exp**[δ] were officially called "δ procedures", but in informal parlance they got to be called "function procedures".

Here is a sample procedure to swap the values of two variables:

```
procedure swap(int x, int y);
{
  int t;
  t := x; x := y; y := t
}
```

The same procedure in our notaion is as follows:

```
swap :: var[int] -> var[int] -> comm
swap x y = {local[int] \t ->
             t := read x; x := read y; y := read t
            }
```

The prominent differences are:

- Algol 60 did not declare the type of parameters as being that of variables. Only the data type was indicated. This caused confusion because both variable parameters and expression parameters were declared the same way.
- The **local** operator was indicated by just the data type declaration for the variable name `t`. It was implicitly understood that it was local variable declaration.

Algol's failure to declare the full type information for the parameters had bad consequences in the years following its definition. Recall that the **read** operation is omitted in the normal use of the language. So, if one finds a procedure call like:

```
swap(a, b)
```

where `a` and `b` are local variables, it can also be interpreted to mean:

```
swap(read a, read b)
```

In one case, the *variables* `a` and `b` are passed to `swap`. In the other, the *values* of the variables `a` and `b` are passed. Both the procedure calls look the same. This difference in the interpretation came to be seen as a "parameter passing mode". Passing the variables was called "call by reference" and passing the values of the variables was called "call by value". In reality, the problem was that Algol's type system did not declare the full type information of `swap`. Had it been fully declared, there would have been no ambiguity at all, and there would have been no such thing called the "parameter passing mode".

Tricky issues with Algol

Note that terms in Algol are in general state-dependent entities. They can have different values or different effects in different states. This can give rise to anomalous behaviour.

15. Call by name. The mode of parameter passing in Algol 60 is termed "call by name," the same as in lambda calculus. That means that the *terms* denoting the arguments are substituted for formal parameters in the body of the procedure. When used in conjunction with imperative programs, the call-by-name parameter passing gives rise to a surprising amount of interference effects.

16. Interference between parameters. Consider using the `swap` procedure with arrays. Let `a` be an array of integers, whose component variables are written as `a[0]`, `a[1]`, ..., and consider a procedure call

```
swap i a[i]
```

Recall that `a[i]` should be really thought of as `a[`**`read`**` i]` because `i` is a variable used in place of an expression and therefore involves the implicit coercion **`read`**. As per the Algol 60 copy rule, the procedure call

```
swap i (a[read i])
```

unfolds to the following command:

```
{int t;
 t := read i; i := read a[read i]; a[read i] := read t }
```

Consider executing this command from an initial state where `i` $= 0$, `a[0]` $= 1$, `a[1]` $= 2, \ldots$.
We might expect that the effect of the procedure call `swap i (a[`**`read`**` i])` should be to swap the variables `i` and `a[`**`read`**` i]`, *i.e.*, `i` should become 1, a[0] should become 0 and all other elements of the array should remain unchanged.
However, what happens is quite different. The first assignment sets `t` to 0. The second assignment sets `i` to 1. Since, `i` is now 1, the third assignment has the effect of `a[1] := t`. So, `a[1]` changes to 0, and `a[0]` remains unchanged!
The problem here is that the two parameters `i` and `a[`**`read`**` i]` *interfere*, i.e., changing one of them affects the meaning of the other term. On the other hand, when we define the procedure `swap x y`, we tend to assume that `x` and `y` are independent, i.e., changing one of them does not the affect the meaning of the other. Thus, there is a mismatch of expectations.

17. Call by value. In order to solve the problem of unwanted interference, one design choice made in the successor languages of Algol 60 was to eschew call-by-name parameter passing and use call-by-value instead. Call-by-value means that only expressions are allowed as parameters of procedures (no variables) and, moreover, the *values* of these expressions are passed as arguments to the procedures. The C programming language, in particular, made this choice, and Java was defined to follow the same conventions.

In a call-by-value imperative language, if `a[i]` is used as a procedure argument, the *value* of the array element at position `i` is passed as the argument. In that case, modifying the variable `i` or even the array `a` inside the procedure will have no effect on the value of the argument.

By making this choice, C and other C-like languages have considerably simplified the meaning of procedures. However, in the process, they have also moved away from the β-equivalence semantics of procedure calls. It is not possible to substitute the procedure parameters by the argument expressions. Some people, including John Reynolds, believed that this was unbearable loss.