

Handout 10: Computational Effects

The lambda calculus was invented before computers themselves were, and at any rate before any large-scale programming was undertaken. After the development of computers, the first programming languages designed were machine languages and then FORTRAN (short for “FORMula TRANslation” language) designed in mid 1950’s. Both of them took a significantly different direction from that of the lambda calculus, by basing themselves on “instructions” or “commands” (also called “statements”) rather than on function application. So, when the Lisp programming language was designed around 1960, it combined the ideas of commands found in the programming languages of the time and functional computation ideas it borrowed from the lambda calculus. The resulting mixture is called today a “lambda calculus with computational effects”. In reality, only call-by-value lambda calculi are extended with computational effects in this form. After Lisp, almost all call-by-value languages have been designed to computational effects. The list includes Scheme, ML, C and Java.

Haskell is an example language that does not have computational effects in this sense. Since it is a call-by-name language, it would be very difficult to add effects to it in the same way.

1. Computational effects

It is hard to define what a “computational effect” is in general. However, several examples of phenomena are generally accepted as computational effects. They include

- *changing* the values of variables (called *state change operations*),
- raising errors or exceptions or, more generally, making the evaluator to jump from one point in the program to another (called *control operations*), and
- making choices between sets of computations (called *nondeterministic operations*).

In general, any computational idea that is used in addition to the purely mathematical idea of function application is called a “computational effect”. In fact, the proponents of effects claim that the lambda calculus itself involves a computational effect, viz., the effect of computing results which has the possibility of nontermination.

Of all these effects, state change effect is the most basic and, perhaps, the most important.

2. State change operations

Changing the values of variables is a familiar programming idiom. In Java, one can write a statement like

```
x = x + y;
```

which means “change the value of x to the current value of x plus the current value of y ”. This is quite a different concept from that of function application. Even though we borrow the mathematical symbol “=” to mean “change this to”, the mathematical symbol itself means something quite different, viz., the equality relation. (Note that x can never be equal to $x+y$ unless y is 0.) Operations of this kind are called “commands” or “statements”; *not* “expressions.” The computer or the run-time system is said to “execute” the command in order to produce the effect described; *not* to “evaluate” it.

Since the above use of the “=” symbol conflicts with our normal use of “=”, we use a new symbol “:=” to denote the state change operation. It is called the “assignment” operator, and the statement is called an “assignment statement.”

In addition to assignment, other kinds of state change operations include changes to data structures (such as pushing values on a stack), input/output and communication operations.

3. Variables

If we admit state change operations, what are the things whose values can be changed? In normal parlance, things that change are typically called “variables”. Since lambda calculus has things called “variables,” it is somewhat immediate to say that the values of variables should be changeable.

This is an unfortunate connection. The lambda calculus use of the term “variable” (which is the same as the usage in general mathematics) is quite different from the normal English usage of the term “variable”.

- In normal English, something that is fixed for ever is said to be “constant” and something that changes during the course of some process is said to be “variable”.

- In Mathematics, on the other hand, a constant is a symbol whose meaning is fixed in advance, e.g., the numerals 0, 1, ..., or the symbol π . Variables are symbols which are used to mean different values in different contexts. However, their values are “constant,” i.e., unchanging, within each context.

The difference between the two notions of “variation” is subtle, but it is hard to miss it. In one case, variation occurs within the same context. In the other, variation occurs (if at all) in different contexts.

Algol 60 found the ambiguous use of the term “variable” unsatisfactory. It introduced the term *identifier* to mean variables in the sense of the lambda calculus. One can use identifiers to stand for or name all kinds of things: primitive values, functions, procedures, data structures, types, modules, classes etc. Algol 60 reserved the term “variable” for mutable variables in the imperative programming sense.

Algol 68 found the remaining usage of “variable” for mutable variables unsatisfactory. It introduces the term *reference* to mean mutable variables. (The concept is that a mutable variable refers to a value. Via assignment statements, we can make it refer to a different value.)

Unfortunately, both the Algol definitions have been unsuccessful in erasing the confusing usage of the term “variable”. The term continues to be used for both identifiers and references ambiguously. In many languages, this ambiguity has been turned into a feature!

The terminology used in different languages is represented in the following table:

	lambda calculus variables	mutable variables
Algol 60	identifier	variable
Algol 68	identifier	reference
ML	variable	reference
Scheme	variable	variable?
Java	variable?	variable

The difference between identifiers and references becomes stark is when data structures like arrays are used. Consider the Java statement:

```
a[i] = pop(s);
```

On the left hand of the assignment operator, we have a *term* that denotes a mutable variable. It is often referred to as a subscripted variable. However, it is not an identifier. Rather, *a* and *i* are identifiers.

Similarly, *pop* and *s* are identifiers. They are variables in the lambda calculus sense. However, *pop* is not called a “variable” in Java. It is referred to as a “function name”. So, we have put a question mark against the Java terminology for identifiers in the table above. Suppose Java is to be extended with higher-order functions in some future version. Then it would need to allow functions like *pop* to be parameters of other functions. Would Java call them “variables” then? We don’t know. It is unfortunate that the clarity of terminology developed in the 60’s has been abandoned in later developments.

4. Mutable variables in the style of Scheme

In the programming language Scheme, all identifiers are considered mutable variables as well. Since identifiers are typically introduced as parameters to lambda procedures, this means that the values of these identifiers/variables can be modified inside the body of the procedures. Here is an example:

$$p = \lambda(x, y). (x := x + y; x * x)$$

The procedure takes two integer arguments and binds them to identifiers *x* and *y*. Then it modifies the value of *x* and returns the square of this value. Calling the procedure as *p*(3, 4) returns the value 49.

How does this work? Recall that the evaluation of a procedure application creates a frame for the parameters of the procedure. In this instance, the frame is $\{x \mapsto 3, y \mapsto 4\}$. The execution of the assignment statement $x := x + y$ has the effect of changing the frame to $\{x \mapsto 7, y \mapsto 4\}$. The expression $x * x$ is evaluated in the changed environment to yield the value 49.

Most imperative programming languages, including C and Java, work similarly. However, they typically do not allow assignments to procedure identifiers. In Scheme, no such special treatment is made to procedures. All identifiers are mutable. For example, the following example is valid in Scheme:

```
let p = λ(x, y). (x := x + y; x * x)
in p := λ(x, y). x + y; p(3, 4)
```

This expression evaluates to 7.

The change of concept from lambda calculus variables to mutable variables has a terrible cost. It invalidates almost all the theory of the lambda calculus. For example, beta reduction does not make much sense any more:

$$(\lambda(x, y). x := x + y; x * x)(3, 4) \longrightarrow_{\beta} 3 := 3 + 4; 3 * 3$$

The result of the beta reduction is not a legal term because it has 3 on the left of the assignment operator, but only “variables” are allowed there!

5. Mutable variables in the style of ML

In the programming language ML, the lambda calculus variables are untouched, but a new feature of *references* is added. Here is an example:

$$p = \lambda(x, y). \mathbf{let} \ c = \mathbf{ref} \ 0 \\ \mathbf{in} \ c := x + y; !c * !c$$

The expression `ref 0` creates a reference, i.e., mutable variable, with the initial value 0. The statement `c := x + y` assigns a new value to it. The expression `!c * !c` reads the value of `c` and squares it. The primitives `ref`, “:=” and “!” are all constants in the applied lambda calculus that is ML. These constants have the following types:

$$\begin{aligned} \mathbf{ref} &: A \rightarrow \mathbf{ref} \ A \\ := &: \mathbf{ref} \ A \times A \rightarrow \mathbf{unit} \\ ! &: \mathbf{ref} \ A \rightarrow A \end{aligned}$$

The type `ref A` denotes the type of references to `A`-typed values. So, `ref 0` has the type `ref int`. `c := x + y` has the type `unit`. (This is a dummy result type, similar to `void` in Java.) Finally, `!c` has the type `int`.

Since the lambda calculus variables are not messed with, ML continues to satisfy all the reasoning principles of the call-by-value lambda calculus, even though computational effects have been added for state change. For example, the expression `p(3, 4)` can be beta-reduced to the term:

$$\mathbf{let} \ c = \mathbf{ref} \ 0 \\ \mathbf{in} \ c := 3 + 4; !c * !c$$

Despite all the advantages of the ML-style references, we will mainly cover the Scheme-style mutable variables in this Handout because they are similar to variables in the widely used programming languages.

6. Objects

Combining higher-order procedures of the lambda calculus with assignable variables gives rise to the concept of “objects.” This was discovered by Sussman and Steele at MIT, who were trying to implement Carl Hewitt’s concept of *actors* (essentially objects with concurrency). The language they designed for the purpose is Scheme.

An object encapsulates some store for assignable variables, and provides one or more procedures (“methods”) that act on this state. We show how to do this with an example object that implements a counter. It has a single method which, when called, increments the internal count and returns the current value.

$$\mathit{counter} = \\ \mathbf{let} \ c = 0 \\ \mathbf{in} \ \lambda(). (c := c + 1; c)$$

The definition of `counter` creates a local variable `c` and returns a 0-argument procedure. Recall that `λ` procedures evaluate to closures, which have an environment captured in them. In the normal lambda calculus, the environment only serves to store the values of variables for eventual look-up. However, once we have admitted assignments to these variables, the values in the environment can also be changed. Accordingly, the `λ` procedure of our counter changes the value of `c` by incrementing it and returns the new value.

Suppose `F0` is a frame that holds all the global variables. The definition of `counter` creates a new frame `F1 = {c ↦ 0}` for the `let` binding, and returns the closure

$$\mathit{cls}(\lambda(). (c := c + 1; c), F_0 \leftarrow F_1)$$

Successive calls of the closure have the following effect:

	<u>result</u>	<u>frame F₁</u>
<code>counter()</code>	1	{c ↦ 1}
<code>coutner()</code>	2	{c ↦ 2}
<code>coutner()</code>	3	{c ↦ 3}

Notice that each call to *counter* changes the frame F_1 so that it has a new value for the next call of *counter*.

As you can see, adding assignment to the applied lambda calculus has a fundamental effect on the meaning of the language. In standard lambda calculus, a call to a procedure with the same arguments *always* gives the same result. In the extended calculus with assignments, it is not so. The idea that a procedure may change some internal (or even external) values in addition to giving its normal result, is called a *side effect*. It is really best to think of side effecting procedures as methods of an object. Our intuitive notion of objects automatically involves having internal state. So, methods are almost always expected to have side effects.

We have seen how to implement objects with a *single* method. How about multiple methods? There are a variety of ways to implement multiple methods. We could construct:

- *lists* of methods, since our language already has lists,
- *tuples* of methods, since we have already used used tuples for multiple arguments of functions, or
- *procedures* that take a method name as an argument and return the corresponding method, called *dispatch procedures* in Scheme.

In this notes, we will add a new feature called records to create objects with multiple methods.

7. Records

A *record* (also called a *structure*, or *struct*) is essentially a tuple of values, but it uses mnemonic field names for the components of the tuple. Here is an example:

$$p = \mathbf{struct} \{x = 2.0; y = 3.2\}$$

This is a record with two fields, named x and y , for representing a two-dimensional point. To access the fields of a record, the dot notation is used: $p.x$ and $p.y$ access the two fields of p . In addition, we also need a way to add or update the fields of a record. We use an operation “**with**” to do such updates. For instance, the term

$$p \mathbf{with} \{z = 1.5\}$$

updates p with an additional field z . This is now a three-dimensional point. (The original p is still unchanged.)

To support records, we extend the syntax of applied lambda calculus as follows:

$$M ::= \dots \mid \mathbf{struct} \{x_1 = M_1; \dots; x_n = M_n\} \mid M.x \mid M \mathbf{with} \{x_1 = M_1; \dots; x_n = M_n\}$$

The evaluation rule (similar to beta-reduction) for field selection is a straightforward one:

$$(\mathbf{struct}\{x_1 = M_1; \dots; x_n = M_n\}).x_i \longrightarrow M_i$$

The evaluation rule for field update is a bit more cumbersome. We express it with the vector notation to convey the idea:

$$\mathbf{struct}\{\vec{x} = \vec{M}; \vec{y} = \vec{N}\} \mathbf{with} \{\vec{y} = \vec{K}; \vec{z} = \vec{L}\} \longrightarrow \mathbf{struct}\{\vec{x} = \vec{M}; \vec{y} = \vec{K}; \vec{z} = \vec{L}\}$$

where \vec{x} and \vec{y} and \vec{z} are mutually disjoint collections of variables

When a record is updated with fields, some of the fields might already be present in the record (represented by \vec{y} in the rule above), and some of the fields might be new (represented by \vec{z}). The new records will have all these fields as specified by the update, and the other remaining fields from the original record.

The idea now is that objects are modelled as records whose fields are methods.

8. Example The following is a constructor procedure that generates bank account objects with methods for deposit, withdrawal and reading balance:

```
newaccount =
  λ(initial).
    let balance = initial
    in struct {
      deposit =
        λ(amount). balance := balance + amount;
      wdraw =
        λ(amount). if balance > amount then
                      balance := balance - amount
                    else print(`insufficient funds`);
      getBalance =
        λ(). balance
    }
```

When the `newaccount` procedure is called with some initial balance, it allocates a local variable `balance` and returns a record containing the three procedures `deposit`, `wdraw` and `getBalance`. All these procedures access the local variable `balance` and modify it when necessary. So, they should be thought of as the methods of an object with hidden private state containing a balance.

Here is a sample usage of the `newaccount` procedure:¹

```
let Tom = newaccount(100);
    Mary = newaccount(50);
    Jerry = Tom
in
  (Tom.wdraw(50);
   Mary.deposit(100);
   Jerry.wdraw(50);
   Tom.getBalance()
  )
```

In this block, two account objects are created and given the names `Tom` and `Mary`. In addition, the `Tom` account is given the additional name `Jerry`. Each of the account objects will have an environment frame containing a balance variable:

$$F_T = \{\text{balance} \mapsto 100\}$$

$$F_M = \{\text{balance} \mapsto 50\}$$

`Tom`'s account methods as well as `Jerry`'s account methods refer to the frame F_T , whereas `Mary`'s account methods refer to the frame F_M . Now, each of the method calls has the following effect:

- `Tom.wdraw(50)` changes the balance value in F_T to 50.
- `Mary.deposit(100)` changes the balance value in F_M to 150.
- `Jerry.wdraw(50)` changes the balance value in F_T to 0.
- `Tom.getBalance()` reads the balance variable in F_T and obtains the value 0.

As you can see, `Tom` and `Jerry` essentially refer to the *same* account object. Multiple variables referring to the same mutable objects is called *aliasing*. It is generally considered a bad programming practice because it leads to unexpected behaviour. In the above example, just looking at the code in the `let` body, we might expect that `Tom`'s balance should remain 50. However, the aliasing of `Jerry` to `Tom` belies the expectation.

9. Classes

Classes, as in Java, are essentially constructors for objects. However, Java classes also have “static” variables and/or methods. Such static elements are the members of the class rather than of the objects generated by the class. So, we model classes as records in general. The records generally have a “new” procedure for creating instances. In addition, they can also have fields corresponding to the static members of the class.

We illustrate this by showing the equivalent of the following Java class definition in our language:

```
class PrintingClock {
  public static void start(int interval, boolean beep) {
    ActionListener ticker = new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        Date now = new Date();
        System.out.println(now);
        if (beep) Toolkit.beep();
      }
    }
    Timer t = new Timer(interval, ticker);
    t.start();
  }
}
```

¹We use a new form of syntactic sugar in this example: `let $x_1 = M_1; x_2 = M_2; \dots; x_n = M_n$ in N` is equivalent to

```
let  $x_1 = M_1$ 
in let  $x_2 = M_2$ 
  in ... let  $x_n = M_n$ 
    in  $N$ 
```

The class `PrintingClock` is unlike a normal class in that it is not used to generate instance objects. It is more like a module that simply gathers certain procedures together. In this case, there is a single procedure `start`. So, in our Scheme-like applied lambda calculus, we model `PrintingClock` as a record containing a single field `start`. This field is a procedure with two arguments: `interval` and `beep`.

The `start` procedure creates two objects: `ticker` and `t`. The `ticker` object appears to be an instance of something called `ActionListener`. However, `ActionListener` is not a class in Java. It is an interface, i.e., a type. Since our calculus is untyped, we ignore type issues. So `ticker` is then simply a record containing a procedure called `actionPerformed`. On the other hand, the object `t` is an instance of the `Timer` class. It is obtained by invoking the `Timer`'s `new` operation.

Notice all this in the following term in our calculus:

```
PrintingClock = struct {
  start =  $\lambda$ (interval, beep).
    let ticker = struct {
      actionPerformed =  $\lambda$ (event).
        let now = Date.new()
        in (System.out.println(now);
           if beep then Toolkit.beep()
           else skip)
    };
  t = Timer.new(interval, ticker)
  in t.start()
}
```