# CS4215—Programming Language Implementation

Martin Henz

Wednesday 28 March, 2012

# Chapter 13

# cPL: A Simple Concurrent Language

The languages covered so far are all *sequential*, meaning that the instructions of the language are to be executed in a fixed order determined by the semantics of the language. In denotational semantics, this was enforced by mechanisms such as (1) insisting on function arguments to be values, and (2) using the ouput store of the left hand side of a sequence as input store for the right hand side.

The world, however, is not sequential. Things happen concurrently, and thus, computer programs that represent or simulate the real world, need to account for this concurrency. In addition, the world of computing itself is concurrent. Separate computers are running at the same time, and need to communicate with each other. Both phenomena require computer programs to represent concurrency.

The area of concurrent computation encompasses concurrent processing at different levels of abstraction, as well as different hardware architectures. The levels of abstraction range from the lowest level where parallel and pipelined processing of machine instructions is handled, to the highest level, dealing with concurrent and communicating processes. Hardware architectures giving rise to concurrent computation range from single processor/single memory machines (where concurrency is found at the level of CPU implementation) to large-scale distributed computer networks, where concurrency is found at the highest levels, namely between the nodes of the network.

Traditionally, we distinguish between coarse-grained message-passing concurrency on one hand, and fine-grained shared-memory concurrency on the other. Today, this distinction is getting blurred with the virtualization of computing. Sometimes, it is not obvious where the actual computation happens as remote processes are represented by local proxies. In message passing concurrency, there are not many programming language issues; a treatment of this style of concurrency is beyond the scope of this module.

By focussing on shared memory concurrency, we first explore the issue of spawning concurrent activity in a shared random-access memory framework (Section 13.1). Section 13.2 discusses the consequences of threads that have access to shared variables. Sections 13.3 and 13.4 look at the task of synchronizing thread execution, and how these tasks are supported by existing concurrent programming languages. Finally, Section 13.5 explores implementation techniques for concurrent programming languages. In the tradition of this module, we use a simple concurrent programming language cPL as a running example. The language cPL is a slight syntactic extension of oPL, with minimal but sufficient support for concurrent programming.

## 13.1 Spawning Concurrent Computation

The first and most straightforward programming language issue that arises in shared-memory concurrent computation is how to initiate concurrent computation in a program. Most concurrent languages developed out of a traditional sequential language, and it is deemed most practical to stick to sequential execution as the default composition operator. In the context of the languages imPL and oPL, this means that sequential composition is retained with its conventional semantics. Following this approach, concurrent computation is introduced as "communicating sequential processes"[1].

Different languages have taken different choices here. Probably, the simplest way is a `thread ... end` construct that spawns a new thread of computation. For example, in a program

```
(f x);
thread (g y) end;
(h z)
```

the application `(f x)` is executed first. When the execution of `(f x)` terminates, a concurrent thread for the execution of `(f y)` is created. This execution proceeds concurrently on its own, while the original computation executes the subsequent application `(f x)`.

In this model, every program execution starts out with one (default) thread, and each time, program execution encounters a `thread...end` expression, a new thread is created and run concurrently. When the execution of the expression within `thread...end` terminates, the thread is discarded. This style of creating concurrent computation is followed by languages such as Ada (using a `task...end` syntax) and Oz (using `thread...end`).

Other languages take a slightly different choice. For example, the functional language Chez Scheme makes use of functions as the way to specify what program a new thread executes. The function application

```
(fork-thread (lambda () ...))
```

[1]Incidentally, this is also the name of a theoretical framework for modeling this style of concurrent computation, see [Hoa85]

Here, the program to be executed by the new thread is given by the body of the zero-argument function that is passed as argument to the built-in function `fork-thread`.

In object-oriented languages, such as Java, classes and objects are typically used to spawn concurrent computation. In Java, the behavior of a thread is described in a `run()` method of a class that extends a predefined class `Thread`. An instance of a thread class is then started by invoking a `start` method. Example:

```
class MyThread extends Thread {
   public void run() { ... }
}
...
someThread = new MyThread();
someThread.start();
```

Sticking with simplicity as a primary goal for the languages covered here, we choose the first approach for cPL. This way, we can treat object-oriented programming and concurrent programming as orthogonal issues in cPL. Thus we add the following rule to the syntax of cPL.

$$\frac{E}{\texttt{thread}\ E\ \texttt{end}}$$

In the framework of imPL/oPL, the question arises what the value of a `thread...end` expression would be. The simplest solution is to assign a particular default value as the result. Thus from the point of view of the spawning thread, the execution of `thread...end` immediately evaluates to, say, the boolean constant `true`.

## 13.2 Shared Variables and Granularity of Concurrency

Concurrent threads are executed independently from each other. The programmer does not have control over the relative speed of execution. In a language with assignment, indeterminism arises when concurrent threads have access to shared memory locations. Example:

```
let accountBalance = 20
in
   let withdraw = fun x -> if x > accountBalance then false
                           else accountBalance := accountBalance - x;
                              true
                           end
                  end
   in
      thread (withdraw 14) end;
      thread (withdraw 17) end;
      accountBalance
   end
end
```

The intention of the `withdraw` function is not to allow the value of `accountBalance` drop below 0. However, when we execute the two threads concurrently, both could reach the test `x > accountBalance` at the same time. In this case, the value of `accountBalance` is still 20, and therefore, both threads get to execute the else-part. Let us say, the first thread executes the else-part first. In this case, the value of `accountBalance` drops to 16. The second thread then would subtract 17 from 16, resulting to a final balance of -1, an unintended consequence of concurrent execution.

Before we address programming language features that can avoid such behavior, let us first take a closer look at concurrent access to shared memory.

When we have concurrent execution of expressions, the issue of granularity of concurrency arises. The question is at what level concurrent behavior is really, well, concurrent. Again, we have a spectrum of possibilities here. At the lowest level, we have the physical reality of computer processors. When two processors have access to the same memory location and try to simultaneously read or write from the location, the behavior depends on the design of the processor. Without specific precautions on the hardware level, simultaneous attempts to write to the same memory location would lead to indeterministic behavior. On this lowest level of abstraction, we would have no guarantee about the outcome of the simultaneous execution of

```
accountBalance := 16
```

and

```
accountBalance := 13
```

The outcome could be that `accountBalance` is assigned to 16, or 13, or some other integer that arises from electrical currents being applied to the memory circuitry that corresponds to the location of `accountBalance`.

With such a low-level memory model, concurrent access to shared memory is utterly unpredictable, and must be avoided at all cost. An extreme countermeasure would be to enforce uninterrupted execution of every thread. There would still be indeterminism with respect to the order in which threads are chosen to be executed, but once chosen and executing, a thread has exclusive access to all shared variables. While this approach would solve the problem of concurrent assignment to shared variables, it would be very restrictive, and lead to counter-intuitive behavior.

Most conventional languages choose a middle path. They allow concurrent write access to shared memory locations at a well-defined level of granularity. The idea is to define non-interruptable units of computation that are carried out by the threads. An obvious choice of granularity is the machine code. In this approach, every machine instruction is executed sequentially without interference from another thread. Concurrent execution only happens between instruction executions. This approach is called "interleaving execution", because one can think of the execution as one single computation that executes a few instructions in one thread, then stops executing in the current thread and jumps to the next runnable thread, then to the next and so on.

**Exercise 13.1** *With interleaving semantics at the level of iVML instructions, how many possible outcomes does the cPL program on page 6 have?*

**Exercise 13.2** *Interleaving access to shared memory has subtle consequences for programming language implementations. Explore Java's keyword* `volatile`*, and explain its consequences for interleaving concurrent write access.*

For cPL, we choose interleaving execution at the level of oVML machine instructions. This will allow for a simple yet realistic implementation of cPL, based on the imPL virtual machine.

## 13.3 Mutual Exclusion

With interleaving semantics, the question arises how we can protect a code section from being executed by multiple threads concurrently. In our account example on page 6, we need to make sure that the body of the `withdraw` function is executed by only one thread at a time, so that the undesired behavior described in Section 13.2 does not arise.

There are various approaches to support mutual exclusion in programming languages. The first approach is to provide built-in functions that provide the possibility of programming the mutual exclusion behavior. The classical set of functions for this purpose is the *semaphore* [Dij68], which is provided by the following two operations, which access a shared integer variable:

```
wait   = fun s -> while ! s > 0 do true end;
                  s := s - 1
         end
signal = fun s -> s := s + 1 end
```

These two operations need to be built-in in the language, since they need to execute atomically, without interruption by another thread. The syntax of imPL is used here just for illustration purposes.

In our tradition of simplicity, we choose semaphores as the language construct of cPL to achieve mutual exclusion. The operations `wait` and `signal` are provided as primitive operators in prefix notation in cPL, similar to the `empty` operator. Thus, the programmer can write `signal s` and `wait s`.

With these two functions in place, we can re-implement our `withdraw` function as follows:

```
let accountBalance = 20
    s = 1
in
   let withdraw = fun x -> wait s;
                           if x > accountBalance then false
                           else accountBalance := accountBalance - x;
                              true
                           end;
                           signal s
                  end
   in ...
   end
end
```

The execution of the first `wait s` by one thread will "close" the semaphore such that no other thread can enter the critical section. Only after the first thread executes `signal s`, the next thread gets its turn.

## 13.4 Monitors

Other languages provide more high-level support for mutual exclusion. An example is the language Java, which allows the programmer to declare (non-static) methods as *synchronized*. The execution of threads is restricted such that only one synchronized method invocation can operate on the same object at a time. If a thread $A$ is already executing a synchronized method on an object, the invocation of a synchronized method on the object by another thread $B$ leads to suspension of $B$'s execution. The thread $B$ needs to "wait" until $A$ has left the synchronized method.

The problem of mutual exclusion for our account example can be solved in Java as follows:

```
class account {
   private int accountBalance;
   public synchronized void withdraw(int x) {
      if (x > accountBalance) false;
      else accountBalance = accountBalance - x
   }
}
```

Theads that execute on the same object can be seen as organized as a queue. When a thread tries to enter a synchronized method for an object that is currently executing a synchronized method, the thread is placed into a queue of threads that are waiting for this object. Note that synchronized methods can call other methods (synchronized or not), which are also executed under mutual exclusion. Only when a thread terminates execution of the first synchronized method it started (through regular execution or through an exception), the next thread that is waiting for the object is resumed.

In addition to `synchronized` methods, Java supports a wait/notify mechanism through which the threads can caordinate their activities. The two built-in functions `wait()`, and `notify()` operate as follows:

- There are two ways for a thread to get into the queue, either by calling a synchronized method while another thread is executing a synchronized method on the object, or by calling `wait()`.
- When a synchronized method call returns, or when a method calls `wait()`, another thread gets access to the object.
- If a thread was put in the queue by a call to `wait()`, it must be "unfrozen" by a call to `notify()` or `notifyAll()` before it can be scheduled for execution again.
- The function `notifyAll()` unfreezes all threads that wait for the object, whereas `notify()` picks a random waiting thread and unfreezes it.

The combination of synchronized methods and wait/notify is called *monitor*, and was pioneered by Per Brinch Hansen in the context of the language Concurrent Pascal [BH75].

## 13.5 Implementation of Concurrent Constructs

For cPL, we are choosing interleaving execution of threads at the level of virtual machine instructions. Thus, we are using the virtual machine for imPL/oPL as starting point.

Threads are running independently, each with their own set of registers, which include program counter, operand stack, environment and runtime stack. When a thread terminates its execution, these registers need to be reliquished so that the memory they occupied can be reused. In order to accomplish this, we translate `thread...end` expressions as follows.

$$\frac{E \hookrightarrow s}{\texttt{thread}\ E\ \texttt{end} \hookrightarrow \texttt{STARTTHREAD}\ |s+2|.s.\texttt{ENDTHREAD}}$$

Interleaving is implemented on a sequential processor by switching execution from thread to thread, also called "time-slicing". In practice, this is done by keeping a queue of threads in the machine, each with its own registers. The machine picks a thread from the queue, and executes a certain number of instructions in that thread. Then it suspends the execution of the thread, and starts execution of the next thread in the queue. Each time the machine moves from one thread to another, the registers of the old thread are saved, and the registers of the new thread are installed. This process is called *context switching.*

Thus the execution of `STARTTHREAD` $n$ creates a new thread in the machine, sets the program counter of the new thread to the address after the instruction, sets the environment of the new thread to the current environment, and initializes the operand and runtime stacks of the new thread to be empty stacks. According to the convention on return values of `thread` expressions, the value `true` is pushed on the operand stack of the old thread. The current program counter is then incremented by $n$, which makes the old thread jump to the code after the new thread.

Note that in this implementation, exception handling and threads are interacting as follows. Exceptions raised in a thread do not have any effect outside the thread. When the execution of a `THROW` instruction reaches the bottom of the runtime stack, the executing thread is terminated. In this respect, oPL follows common practice among languages with threads and exception handling. An interesting alternative would be to copy the current runtime stack to the new thread upon thread creation, and to record the parent thread. In this case, an exception in a thread could interrupt its parent thread and be handled by the parent's try expression.

The execution of the `ENDTHREAD` instruction simply deallocates the executing thread object, along with its registers.

The semaphore primitives `signal` and `wait` are supported by primitive operations that are translated to specific machine instructions.

$$\frac{}{\texttt{signal}\ v \hookrightarrow \texttt{SIGNAL}\ v} \qquad \frac{}{\texttt{wait}\ v \hookrightarrow \texttt{WAIT}\ v}$$

The execution of `SIGNAL` simply increments its semaphore variable.

$$\frac{s(pc) = \texttt{SIGNAL}\ x}{(os, pc, e, rs, h) \rightrightarrows_s (\mathit{deref}(e, x, h) + 1.os, pc + 1, e, rs, \mathit{update}(e, x, \mathit{deref}(e, x, h) + 1))}$$

Note that the heap is shared between different threads, and that other threads are not represented in the rule.

The execution of `WAIT` checks whether the current value of the semaphore variable is positive, and then decrements it.

$$\frac{s(pc) = \texttt{WAIT}\ x}{(os, pc, e, rs, h) \rightrightarrows_s (\mathit{deref}(e, x, h) - 1.os, pc + 1, e, rs, \mathit{update}(e, x, \mathit{deref}(e, x, h) - 1))} \text{if } \mathit{deref}(e, x, h) > 0$$

If the current value is not positive, the `WAIT` instruction leaves the program counter unchanged.

$$\frac{s(pc) = \texttt{WAIT}\ x}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc, e, rs, h)} \text{if } \mathit{deref}(e, x, h) \leq 0$$

That means the executing thread keeps checking the semaphore variable. This behavior is called *busy waiting*. Busy waiting is a common albeit inefficient implementation technique for synchronization primitives.

# Bibliography

[BH75] Per Brinch Hansen. The Progamming Language Concurrent Pascal. *IEEE Transactions of Software Engineering*, 1(2):199–207, 1975.

[Dij68] Edsger Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*. Academic Press, London, 1968.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, 1985.