Chapter 8

Forklang: A Language with Concurrency and Parallelism

Concurrency is a vital requirement for most modern, except perhaps the trivial, software systems. It is the ability of a programming model or a programming language to allow running two or more simulataneous activities or *threads of control* during the execution of a software system. Concurrency is important for:

- *Building responsive software systems.* For example, imagine that a single thread of control was available for a software system with a user interface. Such a system would perhaps run a loop that will first read response from the user, and then process it. For the duration during which a response is being processed, the software would be unable to accept user responses. This could cause user input to be lost, and at the very minimum would lead to an unsatisfactory experience typical of sluggish user interfaces.

- *Introducing parallelism in software.* Many software systems perform expensive (computationally heavy) tasks that can benefit, if the task can be decomposed into subtasks, subtasks can be performed simulatenously, and the result of the subtasks can be combined to determine the result of the original expensive task.

- *Modeling and simulation.* Another large class of software systems is designed to perform modeling and simulation of real world systems, e.g. the stock market, weather system, astronomical systems, plant

physiology, social networks, etc... These real world systems naturally involve entities that are concurrent. Although it is possible to use a single thread of control to model concurrent entities[1] such modeling introduces complications, and often does not naturally encodes the domain of simulation.

It is for those reasons that most modern programming languages provide one or more features for concurrency and parallelism.

8.1 Explicit vs. Implicit Concurrency

A programming language feature for concurrency can be *explicit* or *implicit*. Explicit features for concurrency provide programmers with greater control over concurrency in their software, concurrent tasks can be started and stopped programmatically. The programming language provides mechanisms for controlling the number and lifetime of concurrent tasks.

An example explicit feature for concurrency is `thread` in the Java programming language. A programmer can create and start a thread[2].

Implicit features for concurrency provide programmers with mechanisms to identify potential concurrency in their software. Use of these mechanisms by the programmer is then used aa a hint by the programming language implementation. The language implementation itself manages the number and lifetime of concurrent threads during the execution of the software.

An example implicit feature for concurrency is the idea of parallel arrays. Applying an operation on the array may implicitly create `n` parallel tasks, where `n` is the number of elements in the array[3].

Another example of implicit feature for concurrency are *actors*. Actors are a programming abstraction to model independently acting entities that

[1]Consider a system with n concurrent entities, one methodology to model these entities using a single thread of control is to run an entity for a time slice, and then schedule the next entity. The scheduler for an operating system typically performs such time slice based scheduling.

[2]To be precise, threads in Java are available as part of the standard library. Similar to Java, threads in C and C++ are also available as standard libraries. However, realization of these capabilities requires special treatment, at least in the runtime library so in that sense they are not exactly at par with regular libraries.

[3]It would be useful to compare this facility with the `map` higher-order functions discussed in the context of our functional language

can communicate with each other by exchanging messages. By describing a computation in terms of a collection of actors, a programmer provides hints to the programming language implementation. The language implementation then manages the number and lifetime of concurrent threads.

8.2 Explicit Concurrency Features

In this chapter, we will first discuss design, semantics, and implementation of explicit features for concurrency. At the most basic level, a language requires support for:

1. creating a collection of concurrent tasks, and
2. mutually exclusive access to memory location.

Often facilities for synchronizing between concurrent tasks are also needed, but they can be easily modeled using facilities for obtaining mutually exclusive access to memory locations.

For simplicity, we can start with an expression to create two concurrent tasks. An example usage of such expression appears below.

```
(fork
  (+ 1 2) (+ 2 1)
)
```

Here, our intent is to run two additions `(+ 1 2)` and `(+ 2 1)` as concurrent tasks.

To obtain mutually exclusive access to a memory location, we can devise a lock expression.

```
(let
  ((var (ref 0)))
  (lock var)
)
```

Here, our intent is to obtain mutually exclusive access to the memory location pointed to by `var`.

To release mutually exclusive access to a memory location, we can devise an unlock expression.

```
(let
  ((var (ref 0)))
  (let
    ((val (lock var)))
    (unlock var)
  )
)
```

Here, the intent of the newly added code is to release mutually exclusive access to the memory location pointed to by `var`.

8.3 Semantic variations of Fork

In designing a programming language feature for creating concurrent tasks, such as the fork expression, several design decisions need to be considered. Below we discuss these decisions using the fork expression as an example.

- *Value.* What is the value of a fork expression `(fork exp exp')`? Some languages take the value of a fork expression to be `unit` (void). In such languages concurrent tasks (children) created by a task (parent) communicate results to the parent task by writing to some previously decided upon heap location. In other languages, the value of a fork expression is a task identifier that can be used to uniquely identify forked task. When a fork expression can fork multiple children task as once, such as `(fork exp exp')`, the value of fork is a set of task identifiers. In other languages, the value of a fork expression is the value of subexpressions as computed by children tasks.

  The manner in which the value of a fork expression is communicated back to the parent task has implications on reasoning about the interaction of parent task and children. For example, if a heap location is used to communicate value then the parent might have to implement logic to periodically check that heap location.

- *Join semantics.* What is the join semantics of the fork expression? Several variations may also exist w.r.t. to this aspect. For example, when a task evaluates a fork expression `(fork exp exp')` does the evaluation of the parent task goes on? Or does it suspends until children evaluating `exp` and `exp'` complete their work, either normally or

abnormally? Alternatively, is it the case that the parent task creates a concurrent task to evaluate `exp`, and its own thread evaluates `exp'`, and the evalution in the parent task's thread waits until the child task evaluating `exp` is done?

- *Sharing semantics.* What is the heap sharing semantics of the fork expression? For example, when a fork expression `(fork exp exp')` evaluates, is the heap shared between the parent task and concurrent tasks that evaluate `exp` and `exp'`? In languages like C in which the fork construct creates a new process, no heap locations are shared between the parent process and the children, except perhaps those created using shared memory primitives. On the other hand, in languages like Java and C# in which fork like construct "thread" creates a new thread of control, parent and children may share the entire heap. Besides these extremes there are also models in which the children tasks and parent share parts of the heap.

  The sharing semantics also has implications on program design and reasoning. For example, for writing programs in which parent and children tasks collaborate very closely on shared data structures it would be more efficient to allow sharing of heap locations. On the other hand, allowing such sharing significantly increases the burden of reasoning about programs because programmers now have to reason about interleaving of concurrent tasks w.r.t. shared memory locations.

8.4 Semantic variations of Lock Expressions

Similar to the fork expression, in designing mechanisms for providing mutually exclusive access to certain resources, there are several design decisions of interest. Below we examine these design decisions using the lock and unlock expressions as example.

- *Blocking vs. non-blocking.* First question is: does evaluating `(lock exp)` blocks the execution of the current thread if the lock is not available? In some cases, it is preferable to return a boolean value representing whether lock acquisition attempt was successful or failed. However, most prevalent semantics is one in which execution of the current thread blocks if the lock is not available. In some languages,

e.g. Java, alternative primitives for "trying lock" is also available that attempts to acquire a lock and fails if that attempt is unsuccessful, instead of blocking.

- *Reentrant semantics.* What happens when a task attempts to acquire a lock that it holds already? Does the lock acquisition attempt goes through, and has no effect? Another alternative in some languages is to raise an error, treating an attempt to acquire a lock again as a logical error? Yet another alternative in some languages is to treat the second attempt to acquire as a fresh attempt, thereby implicitly releasing the previously acquired lock. Implicitly relinquishing previously acquired lock may have the effect of allowing other tasks that are waiting for the same lock to acquire the lock.

- *Memory consistency.* Modern processors consist of a number of cores, each of which is capable of running an independent thread of control. Each core may maintain its local memory known as the cache, and every core will share the same main memory. In the design of these processors, to optimize for efficiency cores do not attempt to maintain an *always consistent* memory model, i.e. it is possible for the same memory location to have different values in the cache of two or more cores of the same processor at the same time instant. Rather the processor design attempts to follow an *eventually consistent* memory model, i.e. values are eventually updated so that they are the same in the cache of all cores.

  In designing the semantics of a lock expression, the decision to incorporate memory consistency plays a big role in terms of usefulness of the lock feature as well as on the performancy efficiency of realizing this feature. If locking implies memory consistency then a concurrent task can rely on locking to safely read or update a memory location and be sure that the value read or written is the latest value at that location. That is the new value updated by the concurrent task will be visible to every other task. On the other hand, having consistency requirement would mean that the locked value ought to be evicted from the local cache of other cores.

- *Timeout.* What happens when a task attempts to acquire a lock, and that lock is held by another task, and the language provides a blocking

semantics i.e. the task that is attempting to acquire lock blocks? Will this task stay blocked forever? This situation is problematic when other task which holds the lock of interest may either be blocked or dead. In such situations a deadlock may occur. To prevent such deadlocks, some languages provide a timeout feature, i.e. an attempt to acquire a lock blocks the current task for a certain time. After that time, if the lock is still not available an error is raised and lock acquisition attempt fails.

Against this background on design decisions, we now begin discussion of *Forklang*, a language with explicit features for concurrency and mutually exclusive locks. The Forklang language builds upon the Reflang language discussed in the previous section, which integrated a model of heap as well as explicit features for memory allocation, dereference, assignment, and deallocation. We use a language with references as a basis for Forklang to be able to illustrate behavior of shared memory concurrent programs and their challenges.

8.5 New Expressions for Concurrency

The grammar for newly added expressions is shown in figure 8.1.

`Expression`	::=	`...`	*Expressions*
	|	`(fork Expression Expression)`	*Fork Expression*
	|	`(lock Expression)`	*Lock Expression*
	|	`(unlock Expression)`	*Unlock Expression*

Figure 8.1: New expressions for explicit concurrency

For simplicity we limit the fork expression to being able to create two concurrent tasks, but being able to create an arbitrary number of concurrent tasks doesn't pose any significant new challenges.

As is usual, to realize these new expressions new AST nodes are needed to store their kind and subexpressions. We do not define them here, but the interpreter corresponding to this chapter implements these new AST nodes as classes `ForkExp`, `LockExp`, and `UnlockExp`.

8.6 Semantics of Fork Expression

We have earlier discussed several semantic variations of the fork expression. In the rest of this chapter, we will assume a variation in which the value of a fork expression is a pair containing the value of two subexpressions, the join semantics is such that parent task suspends execution waiting for children to complete, and execution of the parent resumes when both children evaluate to a value.

```
$ (fork 342 342)
(342 342)
```

The sharing semantics that we adopt is one in which parents and two children tasks all share the same heap. Therefore, writes to memory location by one would be seen by the other.

```
$ (let ((var (ref 0))) (fork (set! var 342) (deref var)))
(342 342)
```

To run the subexpressions of a fork expression, we create a data structure that builds on the thread primitive in the defining language[4].

```
class EvalThread extends Thread {
  Env env;
  Exp exp;
  Evaluator evaluator;
  private volatile Value value;

  protected EvalThread(Env env, Exp exp, Evaluator evaluator){
    this.env = env;
    this.exp = exp;
    this.evaluator = evaluator;
  }

  public void run(){
    value = (Value) exp.accept(evaluator, env);
  }
  ...
```

[4]This realization strategy is very similar to the realization strategy for the `Thread` class in the defining language Java that is defined in terms of the OS thread primitive, e.g. `PThreads` on Linux systems.

```
}
```

Since the underlying data structure for threads does not allow passing parameters when a thread is created, the `EvalThread` class is designed to hold the starting environment of the concurrent task, expression, and the evaluator that is to be used. When this `EvalThread` is started, it simply evaluates the expression `exp` in the current environment. Note that since the heap is stored as part of the evaluator, this concurrent task may share the heap with other tasks.

```
public Value value(){
  try {
    this.join();
  } catch (InterruptedException e) {
    return new Value.DynamicError(e.getMessage());
  }
  return value;
}
```

To find out the result of the subexpression evaluation, the `EvalThread` class also provides a value method, which attempts to `join` with the thread. The attempt to join may cause the calling thread to block until evaluation of the subexpression is done, or an exception is thrown.

Using this data structure, we can easily implement the semantics of the fork expression in the evaluator as follows.

```
class Evaluator implements Visitor<Value> {
  ...
  Value visit(ForkExp e, Env env) {
        Exp fst = e.fst_exp();
        Exp snd = e.snd_exp();
        EvalThread fst_thread = new EvalThread(env, fst, this);
        EvalThread snd_thread = new EvalThread(env, snd, this);
        fst_thread.start();
        snd_thread.start();
        Value fst_val = fst_thread.value();
        Value snd_val = snd_thread.value();
    return new Value.PairVal(fst_val, snd_val);
  }
  ...
```

```
}
```

This implementation creates and starts two threads to evaluate both subexpressions of the fork expression (constructor call and start calls), finds out the resulting value of both expressions and returns these value as a pair. From the implementation it is intuitive to see that the fork expression can be generalized to run two or more expression concurrently.

8.7 Semantics of Lock-related Expression

For lock and unlock expression, also recall that we had discussed several semantic variations. For this chapter we adopt the blocking semantics for locks, i.e. an attempt to acquire a lock blocks until the lock is available. Our locks are also reentrant, i.e. the locking succeeds if current task already holds the lock as in the example below.

```
(let
  ((var (ref 0)))
  (let
    ((val (lock var)))
    (lock var)
  )
)
```

To realize lock-related expressions, we also rely upon the facilities provided by the defining language. We implement the model by extending reference values with locking capabilities.

```
class RefVal extends ReentrantLock implements Value {
  ...
}
```

Here, we have the `RefVal` extend standard utility for reentrant locks. The effect of doing so is that each reference value can potentially be locked as we originally intended.

With this semantic change in place, we can implement lock and unlock expressions as follows.

```
Value visit (LockExp e, Env env) {
  Exp value_exp = e.value_exp();
  Object result = value_exp.accept(this, env);
```

```
  if (!( result  instanceof Value.RefVal))
    return new Value.DynamicError("Locking non-ref val");
  Value.RefVal loc = (Value.RefVal) result ;
  loc.lock();
  return loc;
}
```

Here, the lock expression's evaluation first evaluates the subexpression, checks whether the result is a reference value, and then uses the underlying reentrant lock's facilities to acquire a lock on that location. The call `loc.lock()` may block if the underlying lock is not available.

```
Value visit (UnlockExp e, Env env) {
    Exp value_exp = e.value_exp();
    Object result = value_exp.accept(this, env);
  if (!( result  instanceof Value.RefVal))
    return new Value.DynamicError("Unlocking non-ref val");
    Value.RefVal loc = (Value.RefVal) result ;
    try{
      loc.unlock();
    } catch( IllegalMonitorStateException ex){
      return new Value.DynamicError("Lock held by another thread");
    }
  return loc;
}
```

Our realization of the unlock expression also utilizes the underlying facilities of the reentrant lock to release a previously acquired lock.

8.8 Data races in Forklang programs

Data races are one of the notorious problems of concurrent programs which are difficult to detect and reproduce.

A data race happens when two or more threads that share a memory location, access it without synchronization and one of these accesses writes to the memory location. Data race can result in different results for the same program. Accesses to shared memory locations can be synchronized by requiring acquiring a lock before accessing the shared location and release the lock after the access is done.

To illustrate, consider the program below

```
(let
  ((x (ref 0)))
  (fork
    (set! x (+ 1 (deref x))) (set! x (+ 1 (deref x)))
  )
)
```

in which two threads share a memory location `x` and both these threads write into `x` by increasing its value. According to the definition above, such a program, has a data race on location `x` and depending on scheduling of the threads the value of the location `x` can be either 1 or 2. That is, if thread one evaluates `(deref x)` and then the scheduler switches to thread two to evaluate its `(deref x)` then both threads have the value of `x` as 0. Then each thread increases the value 0 of `x` by 1, resulting in value 1 for `x`. However, in a different scheduling first thread can run and finishes, setting `x` to 1, and then the second thread can run setting value of `x` to 2. As it can be seen, without proper synchronization among accesses to `x` in these two threads, the program can evaluate to values 1 or 2.

8.9 Deadlocks in Forklang programs

Deadlocks are another key problem of concurrent programs. To illustrate deadlocks, consider the following Forklang program.

```
(let
  ((x (ref 0)))
  (fork
    (lock x) (lock x)
  )
)
```

In this program, two children tasks of a fork expression share a lock `x`. In evaluation of this program, one of threads locks `x` without unlocking it later. This in turn means the other thread can never lock `x` and thus cannot finish its evaluation.

Deadlocks may not necessarily arise due to a single lock. To illustrate deadlocks using more than one resources, consider the following Forklang program.

```
(let
  ((x (ref 0)) (y (ref 0)))
  (fork
    (let ((val (lock x))) (lock y))
    (let ((val (lock y))) (lock x))
  )
)
```

In this program, two children tasks of a fork expression acquire lock in a different order. First task acquires lock on `x` and then on `y`, whereas the second task does the reverse. As a result, neither of these tasks may succeed because each will be waiting for the other to relinquish lock, which will not happen.

Exercise

8.9.1. *[Join]* Our concurrency model includes facilities for creating a collection of tasks and mutually exclusive access to memory locations. These facilities can be used to model synchronization between two tasks, typically known as join. For example, consider *join*, an expression that allows a current task to wait for another task specified by the join expression. This can be easily modeled by assigning a lock to each task. Then, a task `t` waiting for another task `t'` to finish can be modeled as trying to acquire a lock `l'`, at the join point in task `t`, that is acquired at the beginning of the tasks `t'` and released at the end of the task `t'`. Write an example program in Forklang that illustrates this pattern for two and three tasks created using the fork expression.

8.9.2. *[Fork many]* Extend the fork expression in the language of this chapter to be able to create two or more concurrent tasks, instead of exactly two concurrent tasks.

8.9.3. *[Protected access]* In the Forklang language, it is possible for a task to access a memory location without acquiring a lock first. This makes it possible for a task to introduce errors in the execution of other task even if that other task properly acquires and releases locks. Modify the semamtics of heap read and write operations in Forklang such that they enforce locking discipline.

8.9.4. *[Data races]* Following problems relate to data races in Forklang programs.

1. Write a Forklang function that creates a location in the heap, stores an empty list at that location, and assigns the name "buffer". Following this, the program forks two concurrent tasks: first task replaces the value stored at the location "buffer" with a list containing '342', second task reads the list value stored at the location "buffer". The value of this program is the value of these two concurrent tasks. Can this programs produce different values on different runs?
2. Write two Forklang programs that have exactly one data race. Each program must share only one location among its threads and must use deref, set! and call expressions and must evaluate to two different values for different schedules.
3. Modify your programs in the first part by adding sufficient synchronization using lock and unlock expressions, such that these programs no longer have data races, i.e. they are data race free.
4. Modify your programs in the first part by adding sufficient synchronization using synchronized expressions, such that these programs no longer have data races.

8.9.5. *[Deadlock]* Following problems are about deadlocks in Forklang programs.

1. Write a Forklang function that creates three references and assigns them the name 'a', 'b', and 'c'. After creating these three references, the program forks three concurrent tasks: first task acquires lock on 'a' and then lock on 'b', second task acquires lock on 'b' and then lock on 'c', and the third task acquires lock on 'c' and then lock on 'a'. Will this program always terminate?
2. Write two Forklang programs that have a deadlock. Each program must share two locks among its threads and must use call, lock or unlock expressions.
3. Modify your programs in the previous part such that these programs do not have deadlocks anymore.

8.9.6. *[Synchronized blocks]* Forklang supports locking and unlocking of locks to synchronize accesses (read and write) to memory.

Languages like Java provide higher level synchronization mechanisms such as synchronized methods or blocks. A synchronized method of an object locks the object at the beginning of the execution of the method and unlocks it before returning from the method. Similarly, a synchronized block locks the object upon entering the block and unlocks it when exiting the block.

Extend Forklang with a synchronized expression.

Syntax of a synchronized expression follows the following grammar:

```
syncexp: '(' 'synchronized' exp exp ')'  ;
```

Semantically, a synchronized expression `(synchronized exp1 exp2)` treats `exp1` as a lock. Similar to the lock expression, it evaluates `exp1` to a location and acquires the lock. Then the synchronization expression evaluates `exp2` while the lock is acquired. And finally, the lock is released when the evaluation of `exp2` is done.

The following interaction log illustrates the semantics of synchronized expression:

```
$ (let ((x (ref 0))) (synchronized x (set! x (+ 1 (deref x)))) (
    synchronized x (set! x (+ 1 (deref x)))) )
2
```

Whereas the same program without synchronization, see below, could evaluate to 1 or 2.

```
$ (let ((x (ref 0))) (set! x (+ 1 (deref x))) (set! x (+ 1 (deref
    x))))
1
```

8.9.7. em [Heap-passing Interpreter] Our realization of the Forklang language maintained a shared heap. Those program expressions that assess heap directly manipulated this shared heap. This design had the advantage that it avoided having to pass around the heap object (similar to environments). Main disadvantage of this design is that it is difficult to implement variations of heap semantics, where parts of the program may potentially use a variation of the original heap.

Modify the Forklang interpreter so that an expression can explicitly control the heap that is available during evaluation of its subexpressions.

8.9.8. *[Fork Processes]* Our realization of the Forklang language creates two threads. In some programming languages, fork creates a separate process, i.e. the forked task and the original task do not share memory. In this exercise, you will gradually modify the Forklang interpreter to implement processes.

- *[Heap clone]* Extend the heap abstraction with a cloning functionality. Clone of a heap has all the locations that are present in the original heap. For each valid location, i.e. locations that contain a stored value, the value stored in the heap clone is the same as the value stored in the original heap. Is it essential to clone each value stored in the original heap?
- *[Process]* Modify the semantics of fork such that the two new concurrent subtasks utilize heap that is a clone of the original heap.