Formal Concurrency

Non- atomic semantics

The most general way to describe a concurrent system is as a collection of independent atomic actions that share a collection of variables. If the actions are `A1, ..., An` then the entire system is just the ‘or’ of all these actions: `A1 [] ... [] An`. In general only some of the actions will be enabled, but for each step the system non-deterministically chooses an action to execute from all the enabled actions. Thus non-determinism encompasses concurrency. In the language of Lamport’s TLA, each action is a predicate relating a pre-state and a post-state, and `[]` is logical “or”.

This story works well for distributed systems, but for local concurrency we find it *convenient* to carry over into the concurrent world as much of the framework of sequential computing as possible. Hence threads for sequential computing. To define its sequence, each thread has a state variable called its ‘program counter’ `$pc`, and each of its actions has the form `(h.$pc = α) => c`, (this is Spec for **if** $h.\$pc = \alpha$) **then** c, and `[*]` is Spec for **else**). But threads are not fundamental.

If `c` is in thread `h`, its transition function is

```
     (\ s, o | s(h+".$pc") = α /\ o(h+".$pc) = β /\ rel')
```

Here `rel'` is `rel` with each reference to a local variable `v` changed to `h + ".v"` or `h + ".P.v"`.
If `c` is in procedure `P`, that is, `c` can execute for any thread whose program counter has reached α, its transition function is

```
     (\ s, o | (EXISTS h: Thread |
          s(h+".P.$pc") = α /\ o(h+".P.$pc) = β /\ rel'))
```

Example of labels in Spec, and translation to a pile of atomic actions.

```
                                       VAR t |
 [$\alpha_1$] DO i < n =>                  << pc=$\alpha_1$ /\ i < n => pc:=$\alpha_2$  [*] pc:=$\alpha_6$ >>
     [$\alpha_2$] sum := [$\alpha_3$] sum + x(i);  []<< pc=$\alpha_2$ => t := sum + x(i);     pc:=$\alpha_3$ >>
                                       []<< pc=$\alpha_3$ => sum := t;             pc:=$\alpha_4$ >>
     [$\alpha_4$] i := [$\alpha_5$] i + 1          []<< pc=$\alpha_4$ => t := i + 1;           pc:=$\alpha_5$ >>
                                       []<< pc=$\alpha_5$ => i := 1;               pc:=$\alpha_1$ >>
 OD [$\alpha_6$]                             []<< pc=$\alpha_6$ => ...
```

Here is a Spec program that searches for prime numbers. It is more like a spec, using an infinite number of threads, one for every odd number.

```
CONST Odds       =  {i: Nat | i // 2 = 1 /\ i > 1 }

VAR  primes      :  SET Nat := {2}
     done        :  SET Nat := {}                    % numbers checked

INVARIANT (ALL n: Nat |   n IN done /\ IsPrime(n) ==> n IN primes
                       /\ n IN primes ==> IsPrime(n))

THREAD Sieve1(n :IN Odds) =
      {i :IN Odds | i <= Sqrt(n)} <= done =>         % Wait for possible factors
          IF   (ALL p :IN primes | p <= Sqrt(n) ==> n // p # 0) =>
               << primes \/ := {n} >>
          [*]  SKIP
          FI;
          << done \/ := {n} >>                       % No more transitions

FUNC Sqrt(n: Nat) -> Int = RET { i: Nat | i*i <= n }.max
```

Handout 17 has a more practical version of this program as well, using just 10 threads.

Big atomic actions

The idea: `[α] A; [β] B   = [α] << A; B >>` if `A` denotes an atomic command that commutes with every atomic command `C` in the program (other than `B`) that is enabled at the semicolon. The transition relation for the program is

`h.$pc = α => A [] h.$pc = β => B [] `C_1` [] `C_2` [] ...`

Commutes? Easy cases are disjoint variables (and thread locals are automatically disjoint) or protected by lock. Any locking scheme (e.g., R/W) will work in which *non-commuting operations hold mutually exclusive locks*; this is the basis of rules for 'lock conflicts'.

Another important case is mutex acquire and release operations. These operations only touch the mutex, so they commute with everything else. What about these operations on the same mutex in different threads? If both can execute, they certainly don't yield the same result in either order; that is, they don't commute. When can both operations execute? We have the following cases (writing the executing thread as an explicit argument of each operation):

`A [β]`	`C`	Possible sequence (`C` is enabled at β)?
`m.acq(h)`	`m.acq(h')`	No: `C` is blocked by `h` holding `m`
`m.acq(h)`	`m.rel(h')`	No: `C` won't be reached because `h'` doesn't hold `m`
`m.rel(h)`	`m.acq(h')`	OK
`m.rel(h)`	`m.rel(h')`	No: one thread doesn't hold `m`, hence won't do `rel`

So `m.acq` commutes with everything that's enabled at β, since neither mutex operation is enabled at β in a program that avoids havoc. But `m.rel(h)` doesn't commute with `m.acq(h')`. The reason is that the `A; C` sequence can happen, but the `C; A` sequence `m.acq(h'); m.rel(h)` cannot, because in this case `h` doesn't hold `m` and therefore can't be doing a `rel`. Hence it's not possible to flip every `C` in front of `m.rel(h)` in order to make `A; B` atomic. What does this mean? You can acquire more locks and still keep things atomic, but as soon as you release one, you no longer have atomicity. Hence two-phase locking.

A third important case of commuting operations, producer-consumer, is like the mutex case. Hence dataflow.

How do you know that you made the atomic action big enough? There should be an invariant that holds whenever you are not in an atomic action. That is, commands in the action may invalidate it, but the action must establish it before ending.

Often there's an invariant that only holds at a particular PC value. That is, it has the form pc = x → invariant. It's convenient to write it at that point in the program text, instead of writing a PC label there.

Proofs

`<< A; B >>`	(2: *S*)
`A; [β] B`	(1: *U*)

$$\begin{tikzcd}
s \arrow[r, equal, "\mathrm{SKIP}"] \arrow[d, equal, "="'] & s \arrow[r, Rightarrow, "<< A ; B >>"] \arrow[d, leftrightarrow, "\sim"'] & s' \arrow[d, equal, "="'] \\
s \arrow[r, Rightarrow, "A"] & u \arrow[r, Rightarrow, "B"] & s'
\end{tikzcd}$$

`u("$pc") = β`

Specifying Concurrency, lecture 17, 6.826 Fall 2017

We need a precise definition of "`C` is enabled at β and commutes with `A`". For any atomic command `X`, we write `u X u'` if `X` relates `u` to `u'`. The idea of 'commutes' is that `<<A; C>>`, as a relation between states, is a *subset* of the relation `<<C; A>>`. To make this precise all we need is to plug in the meaning of semicolon: `C` commutes with `A` iff

```
    (ALL u1, u2 |    (EXISTS u  | u1 A u  /\ u  C u2 /\ u("h.$pc") = β)
                 ==> (EXISTS u' | u1 C u' /\ u′ A u2) )
```

This says that any result that you could get by doing `A; C` you could also get by doing `C; A`. Note that it's OK for `C; A` to have more steps, since we want to show that `A; C; B` implements `C; << A; B >>`, not that they are equivalent. This is not just nit-picking; if `C` tries to acquire a lock that `A` holds, there is no step from `A` to `C` in the first case.

We need an abstraction relation `AR`. We write `u ~ s` for "`AR` relates `u` to `s`". `u ~ s` if

```
       u("h.$pc") ≠ β /\ s = u
    \/ u("h.$pc") = β /\ s A u.
```

Why is this an abstraction relation? It certainly relates an initial state to an initial state, and it certainly works for any transition `u -> u'` that stays away from β, that is, in which `u("h.$pc") ≠ β` and `u'("h.$pc") ≠ β`, since the abstract and concrete states are the same. What about transitions that do involve β?Then are 3 cases:

$$\begin{tikzcd}
s \arrow[r, equal, "="] \arrow[d, dash, "="'] & s' \arrow[d, "A"] \\
u \arrow[r, Rightarrow, "A"] & u'
\end{tikzcd}$$

`h.$pc` changes to β; executed `A`

$$\begin{tikzcd}
s \arrow[r, Rightarrow, "<<A;\ B>>"] \arrow[d, "A"'] & s' \arrow[d, "="] \\
u \arrow[r, Rightarrow, "B"] & u'
\end{tikzcd}$$

`h.$pc` starts at β; executed `B`

$$\begin{tikzcd}
\mathbf{u1} & s \arrow[r, Rightarrow, "C"] \arrow[d, "A"'] & s' \arrow[d, "A"] & \mathbf{u'} \\
 & u \arrow[r, Rightarrow, "C"] & u' & \mathbf{u2}
\end{tikzcd}$$

`h.$pc` starts at β; executed `C`

Examples of concurrency

Mutexes

Here is a spec of a simple `Mutex` module, which can be used to ensure mutually exclusive execution of critical sections. The state of a mutex is the thread that holds the mutex, or `nil` if it's free.

```
CLASS Mutex EXPORT acq, rel =
VAR  m           :  (Thread + Null) := nil
% Each mutex is either nil or the thread holding the mutex.
% The variable SELF is defined to be the thread currently making a step.
APROC acq() = << m = nil  => m := SELF; RET >>
APROC rel() = << m = SELF => m := nil ; RET [*] HAVOC >>
END Mutex
```

If a thread invokes `acq` when `m ≠ nil`, then the body fails. This means that there's no possible step for that thread, and the thread is blocked, waiting at this point until the guard becomes true. If many threads are blocked at this point, then when `m` is set to `nil`, one is scheduled first and it sets `m` to itself atomically; the other threads are still blocked.

The spec says that if a thread that doesn't hold `m` does `m.rel`, the result is `HAVOC`. As usual, this means that the code is free to do anything when this happens. As we shall see in the `SpinLock` code below, one possible 'anything' is to free the mutex anyway.

Here is a simple use of a mutex `m` to make the `Increment` procedure atomic:

```
PROC Increment() = VAR t: Int |
     m.acq; t := Register.Read(); t := t + 1; Register.Write(t); m.rel
```

This keeps concurrent calls of `Increment` from interfering with each other. If there are other write accesses to the register, they must also use the mutex to avoid interfering with threads executing `Increment`.

Spin locks

A simple way to code a mutex is to use a *spin lock*. Here is incorrect code:

```
CLASS BadSpinLock EXPORT acq, rel =

TYPE FH          =  ENUM[free, held]
VAR  fh          := free

PROC acq() =
     DO << fh = held => SKIP >> OD;                    % wait for fh = free
     << fh := held >>                                  % and acquire it
PROC rel() = << fh := free >>

END BadSpinLock
```

This is wrong because two concurrent invocations of `acq` could both find `fh = free` and subsequently both set `fh := held` and return. Here is correct code.

```
PROC acq() = VAR t: FH |
     DO << t := fh; fh := held >>; IF t = free => RET [*] SKIP FI OD

PROC rel() = << fh := free >>
```

The `SpinLock` code differs from the `Mutex` spec in another important way. It "forgets" which thread owns the mutex. The following `ForgetfulMutex` module is useful in understanding the `SpinLock` code—in `ForgetfulMutex`, the threads get forgotten, but the atomicity is the same as in `Mutex`.

```
CLASS ForgetfulMutex =

PROC acq() = << fh = free => fh := held >>
PROC rel() = << fh := free >>
END ForgetfulMutex
```

Of course this code is not practical in general unless each thread has its own processor; it is used, however, in the kernels of most operating systems for computers with several processors.

The strategy for proofs

We need to show that each step of the code simulates a sequence of steps of the spec. An external step must simulate a sequence that contains exactly one instance of the same external step and no other external steps; it can also contain any number of internal steps. An internal step must simulate a sequence that contains only internal steps.

Theorem: If there exists an abstraction function or relation from *U* to *S* then *U* implements *S*; that is, every trace of *U* is a trace of *S*.

The formal method suggests the following strategy for doing hard concurrency proofs.

1. Start with a spec, which has an abstract state.
2. Choose a concrete state for the code.
3. Choose an abstraction function, perhaps with history variables, or an abstraction relation.
4. Write code, identifying the critical actions that change the abstract state.
5. While (checking the simulation fails) do

    Add an invariant, checking that all actions of the code preserve it, or

    Change the abstraction function (step 3), the code (step 4), the invariant (step 5), or more than one, or

    Change the spec (step 1).

This approach always works. The first four steps require creativity; step 5 is quite mechanical except when you find an error. It is somewhat laborious, but experience shows that if you are doing hard concurrency and you omit any of these steps, your program will have bugs. Be warned.