8. Generalizing Abstraction Functions

In this handout, we give a number of examples of specs and implementations for which simple abstraction functions (of the kind we studied in handout 6 on abstraction functions) don't exist, so that the abstraction function method doesn't work to show that the implementation satisfies the spec. We explain how to generalize the abstraction function method so that it always works.

We begin with an example in which the spec maintains state that doesn't actually affect its behavior. An optimized implementation can simulate the spec without having enough state to generate all the state of the spec. By adding *history variables* to the implementation, we can extend its state enough to define an abstraction function, without changing its behavior. An equivalent way to get the same result is to define an *abstraction relation* from the implementation to the spec.

Next we look at implementations that simulate a spec without taking exactly one step for each step of the spec. As long as the *external* behavior is the same in each step of the simulation, an abstraction function (or relation) is still enough to show correctness, even when an arbitrary number of transitions in the specification correspond to a single transition in the implementation.

Finally, we look at an example in which the spec makes a non-deterministic choice earlier than the choice is exposed in the external behavior. An implementation may make this choice later, so that there is no abstraction relation that generates the premature choice in the spec's state. By adding *prophecy variables* to the implementation, we can extend its state enough to define an abstraction function, without changing its behavior. An equivalent way to get the same result is to define a *backward simulation* from the implementation to the spec.

If we avoided extra state, too few or too many transitions, and premature choices in the spec, the simple abstraction function method would always work. You might therefore think that all these problems are not worth solving, because it sounds as though they are caused by bad choices in the way the spec is written. But this is wrong. A spec should be written to be as clear as possible to the clients, not to make it easy to prove the correctness of an implementation. The reason for these priorities is that we expect to have many more clients for the spec than implementers. The examples below should make it clear that there are good reasons to write specs that create these problems for abstraction functions. Fortunately, with all three of these extensions we can always find an abstraction function to show the correctness of any implementation that actually is correct.

A statistical database

Consider the following specification of a "statistical database" module, which maintains a collection of values and allows the size, mean, and variance of the collection to be extracted.

Recall that the mean m of a sequence db of size $n > 0$ is just the average $\frac{\sum_i db(i)}{n}$, and the

variance is $\frac{\sum_i (db(i)-m)^2}{n} = \frac{\sum_i db(i)^2}{n} - m^2$. (We make the standard assumptions of commutativity, associativity, and distributivity for the arithmetic here.)

```
MODULE StatDB [ V WITH {Zero: ()->V, "+": (V,V)->V, (V,V)->V, "-": (V,V)->V,
                        "/": (V,Int)->V} ] EXPORT Add, Size, Mean, Variance =

VAR db          :  SEQ V := {}                              % a multiset

APROC Add(v) = << db + := {v}; RET >>

APROC Size() -> Int = << RET db.size >>

APROC Mean() -> V RAISES {empty} = <<
     IF db = {} => RAISE empty [*] VAR sum := (+ : db) | RET sum/Size() FI >>

APROC Variance() -> V RAISES {empty} = <<
     IF   db = {} => RAISE empty
     [*]  VAR avg := Mean(), sum := (+ : {v :IN db | | (v - avg)**2}) |
          RET sum/Size()
     FI >>

END StatDB
```

This spec is a very natural one that directly follows the definitions of mean and variance.

The following implementation of the `StatDB` module does not retain the entire collection of values. Instead, it keeps track of the size, sum, and sum of squares of the values in the collection. Simple algebra shows that this is enough to compute the mean and variance in the manner done below.

```
MODULE StatDBImpl                                       % implements StatDB
     [ V WITH {Zero: ()->V, "+": (V,V)->V, (V,V)->V, "-": (V,V)->V,
               "/": (V,Int)->V} ] EXPORT Add, Size, Mean, Variance =

VAR  count       := 0
     sum         := V.Zero()
     sumSquare   := V.Zero()

APROC Add(v) = <<
     count + := 1; sum + := v; sumSquare + := v**2; RET >>

APROC Size() -> Int = << RET count >>

APROC Mean() -> V RAISES {empty} =
     << IF count = 0 => RAISE empty [*] RET sum/count FI >>

APROC Variance() -> V RAISES {empty} = <<
     IF   count = 0 => RAISE empty
     [*]  VAR avg := Mean() | RET sumSquare/count - avg**2
     FI >>

END StatDBImpl
```

`StatDBImpl` implements `StatDB`, in the sense of trace set inclusion. However we cannot prove this using an abstraction function, because each nontrivial state of the implementation

corresponds to many states of the specification. This happens because the specification contains more information than is needed to generate the desired external behavior. In this example, the states of the specification could be partitioned into equivalence classes based on the possible future behavior: two states are equivalent if they give rise to the same future behavior. Then any two equivalent states yield the same future behavior of the module. Each of these equivalence classes corresponds to a state of the implementation.

To get an abstraction function we must add history variables, as explained in the next section.

History variables

The problem in the `StatDB` example is that the specification states contain more information than the implementation states. A *history variable* is a variable that is added to the state of the implementation *T* in order to keep track of the extra information in the specification *S* that was left out of the implementation. Even though the implementation has been optimized *not* to retain certain information, we can put it back in to prove the implementation correct, as long as we do it in a way that does not change the behavior of the implementation. What we do is to construct a new implementation *TH* that has the *same* behavior as *T*, but a bigger state. If we can show that *TH* implements *S*, it follows that *T* implements *S*, since traces of T = traces of $TH \subseteq$ traces of *S*.

In this example, we can simply add an extra state component `db` to the implementation `StatDBImpl`, and use it to keep track of the entire collection of elements, that is, of the entire state of `StatDB`. This gives the following module:

```
MODULE StatDBImplH ... =                            % implements StatDB

VAR count        := 0                               % as before
    sum          := V.Zero()                        % as before
    sumSquare    := V.Zero()                        % as before
    db           :  SEQ V := {}                     % history: state of StatDB

APROC Add(v) = <<
    count + := 1; sum + := v; sumSquare + := v**2;
    db + := {v}; RET >>

% The remaining procedures are as before

END StatDBImplH
```

All we have done here is to record some additional information in the state. We have not changed the way existing state components are initialized or updated, or the way results of procedures are computed. So it should be clear that this module exhibits the *same* external behaviors as the implementation `StatDBImpl` given earlier. Thus, if we can prove that `StatDBImplH` implements `StatDB`, then it follows immediately that `StatDBImpl` implements `StatDB`.

However, we can prove that `StatDBImplH` implements `StatDB` using an abstraction function. The abstraction function, `AF`, simply discards all components of the state *except* `db`. The following invariant of `StatDBImplH` describes how `db` is related to the other state:


```
   count     = db.size
/\ sum       = (+ : db)
/\ sumSquare = (+ : {v :IN db | | Square(v)})
```

That is, `count`, `sum` and `sumSquare` contain the number of elements in `db`, the sum of the elements in `db`, and the sum of the squares of the elements in `db`, respectively.

With this invariant, it is easy to prove that `AF` is an abstraction function from `StatDBImplH` to `StatDB`. In this proof, it is easy to show that the abstraction function is preserved by every step, because the only variable in `StatDB`, `db`, is changed in exactly the same way in both modules. The interesting thing to show is that the `Size`, `Mean`, and `Variance` operations produce the same results in both modules. But this is easy to see because of the invariant.

In general, we can augment the state of an implementation with additional components, called *history variables* (because they keep track of additional information about the history of execution), subject to the following constraints:

1. Every initial state has at least one value for the history variables.
2. No existing step is disabled by the addition of predicates involving history variables.
3. A value assigned to an existing state component must not depend on the value of a history variable. One important case of this is that a return value must not depend on a history variable.

These constraints guarantee that the history variables simply record additional state information and do not otherwise affect the behaviors exhibited by the module. If the module augmented with history variables can be shown correct, it follows that the original module without the history variables is also correct, because they have the same traces.

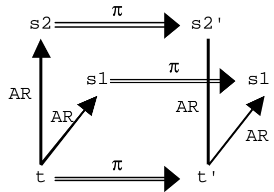
This definition is formulated in terms of the underlying state machine model. However, most people think of history variables as syntactic constructs in their own particular programming languages; in this case, the restrictions on their use must be defined in terms of the language syntax.

In the `StatDB` example, we have simply added a history variable that records the entire state of the specification. This is not necessary; sometimes there might be only a small piece of the state that is missing from the implementation. However, the brute-force strategy of using the entire specification state as a history variable will work whenever any addition of history variables will work.

Abstraction relations

If you don't like history variables, you can define an *abstraction relation* between the implementation and the spec; it's the same thing in different clothing.

An abstraction relation is a simple generalization of an abstraction function, allowing several states in *S* to correspond to the same state in *T*. An abstraction relation is a subset of $states(T) \times states(S)$ that satisfies the following two conditions:



1. If t is any initial state of T, then there is an initial state s of S such that $(t, s) \in R$.
2. If t and s are reachable states of T and S respectively, with $(t, s) \in R$, and (t, π, t') is a step of T, then there is a step of S from s to some s', having the same trace, and with $(t', s') \in R$.

The picture illustrates the idea; it is an elaboration of the picture for an abstraction function in handout 6. It shows `t` related to `s1` and `s2`, and an action π taking each of them into a state related to `t'`.

It turns out that the same theorem holds as for abstraction functions:

Theorem 1: If there is an abstraction relation from T to S, then T implements S, that is, every trace of T is a trace of S.

The reason is that for T to simulate S it isn't necessary to have a function from T states to S states; it's sufficient to have a relation. A way to think of this is that the two modules, T and S, are running in parallel. The execution is driven by module T, which executes in any arbitrary way. S follows along, producing the same externally visible behavior. The two conditions above guarantee that there is always some way for S to do this. Namely, if T begins in any initial state t, we just allow S to begin in some related initial state s, as given by (1). Then as T performs each of its transitions, we mimic the transition with a corresponding transition of S having the same externally visible behavior; (2) says we can do so. In this way, we can mimic the entire execution of T with an execution of S.

An abstraction relation for `StatDB`

Recall that in the `StatDB` example we couldn't use an abstraction function to prove that the implementation satisfies the spec, because each nontrivial state of the implementation corresponds to many states of the specification. We can capture this connection with an abstraction relation. The relation that works is described in Spec[1] as:

```
TYPE T = [count: Int, sum: V, sumSquare: V]          % state of StatDBImpl
     S = [db: SEQ V]                                 % state of StatDB
```

[1] This is one of several ways to represent a relation, but it is the standard one in Spec. Earlier we described the abstraction relation as a set of pairs (t, s). In terms of `AR`, this set is `{t, s | AR(t, s) | (t, s)}` or simply `AR.set`, using one of Spec's built-in methods on predicates. Yet another way to write it is as a function `T -> SET S`. In terms of `AR`, this function is `(\ t | {s | AR(t, s)}` or simply `AR.setF`, using another built-in method. These different representations can be confusing, but different aspects of the relation are most easily described using different representations.

6.826—Principles of Computer Systems 2000

```
FUNC AR(t, s) -> Bool =
    RET    db.size = count
        /\ (+ : db)) = sum
        /\ (+ : {v :IN db | | Square(v)}) = sumSquare
```

The proof that `AR` is an abstraction relation is straightforward. We must show that the two properties in the definition of an abstraction relation are satisfied. In this proof, the abstraction relation is used to show that every response to a size, mean or variance query that can be given by `StatDBImpl` can also be given by `StatDB`. The new state of `StatDB` is uniquely determined by the code of `StatDB`. Then the abstraction relation in the prior states together with the code performed by both modules shows that the abstraction relation still holds for the new states.

An abstraction relation for `MajorityRegister`

Consider the abstraction function given for `MajorityRegister` in handout 5. We can easily write it as an abstraction relation from `MajorityRegister` to `Register`, not depending on the invariant to make it a function.

```
FUNC AR(m, d) -> Bool = VAR seqno := {p' :IN m.rng | | p'.seqno}.max |
    RET (P{d, seqno} IN m.rng)
```

For (1), suppose that t is any initial state of `MajorityRegister`. Then there is some default value d such that all copies have value d and `seqno` 0 in t. Let s be the state of `Register` with value d; then s is an initial state of `Register` and $(t, s) \in AR$, as needed.

For (2), suppose that t and s are reachable states of `MajorityRegister` and `Register`, respectively, with $(t, s) \in AR$, and (t, π, t') a step of `MajorityRegister`. Because t is a reachable state, it must satisfy the invariants given for `MajorityRegister`. We consider cases, based on π. Again, the interesting cases are the procedure bodies.

Abstraction relations vs. history variables

Notice that the *invariant* for the history variable `db` above bears an uncanny resemblance to the *abstraction relation* `AR`. This is not an accident—the same ideas are used in both proofs, only they appear in slightly different places. The following table makes the correspondence explicit.

Abstraction relation to history variable	*History variable to abstraction relation*
Given an abstraction relation `AR`, define `TH` by adding the abstract state `s` as a state variable to `T`. `AR` defines an invariant on the state of `TH`: `AR(t, s)`.	Given `TH`, `T` extended with a history variable `h`, there's an invariant `I(t, h)` relating `h` to the state of `T`, and an abstraction function `AF(t, h) -> S` such that `TH` simulates `S`.
Define `AF((t, s)) = s`	Define `AR(t, s) = (EXISTS h | I(t, h) /\ AF(t, h) = s)` That is, `t` is related to `s` if there's a value for `h` in state `t` that `AF` maps to `s`.

For each step `(t, π, t')` of `T`, and `s` such that `AR(t, s)` holds, the abstraction relation gives us `s'` such that `(t, π, t')` simulates `(s, π, s')`. Add `((t, s), p, (t', s'))` as a transition of `TH`. This maintains the invariant.	For each step `(t, π, t')` of `T`, and `h` such that the invariant `I(t, h)` holds, `TH` has a step `((t, h), π, (t', h'))` that simulates `(s, π, s')` where `s = AF(t, h)` and `s' = AF(t', h')`. So `AR(t', s')` as required.

This correspondence makes it clear that any implementation that can be proved correct using history variables can also be proved correct using an abstraction relation, and vice-versa. Some people prefer using history variables because it allows them to use an abstraction function, which may be simpler (especially in terms of notation) to work with than an abstraction relation. Others prefer using an abstraction relation because it allows them to avoid introducing extra state components and explaining how and when those components are updated. Which you use is just a matter of taste.

Taking several steps in the spec

A simple generalization of the definition of an abstraction relation (or function) allows for the possibility that a particular step of T may correspond to more or less than one step of S. This is fine, as long as the externally-visible actions are the same in both cases. Thus this distinction is only interesting when there are internal actions.

Formally, a (generalized) abstraction relation R satisfies the following two conditions:

1. If t is any initial state of T, then there is an initial state s of S such that $(t, s) \in R$.
2. If t and s are reachable states of T and S respectively, with $(t, s) \in R$, and (t, π, t') is a step of T, then there is an *execution fragment* of S from s to some s', having the same trace, and with $(t', s') \in R$.

Only the second condition has changed, and the only difference is that an execution fragment (of any number of steps, including zero) is allowed instead of just one step, as long as it has the same trace, that is, as long as it looks the same from the outside. We generalize the definition of an abstraction function in the same way. The same theorem still holds:

Theorem 2: If there is a generalized abstraction function or relation from T to S, then T implements S, that is, every trace of T is a trace of S.

From now on in the course, when we say “abstraction function” or “abstraction relation”, we will mean the generalized versions.

Some examples of the use of these generalized definitions appear in handout 7 on file systems, where there are internal transitions of implementations that have no counterpart in the corresponding specifications. We will see examples later in the course in which single steps of implementations correspond to several steps of the specifications.

Here, we give a simple example involving a large write to a memory, which is done in one step in the spec but in individual steps in the implementation. The spec is:


```
MODULE RWMem [A, D] EXPORT BigRead, BigWrite =

TYPE M            =  A -> D
VAR  memory       :  M

FUNC BigRead() -> M = RET memory

APROC BigWrite(m: M) = << memory := m; RET >>

END RWMem
```

The implementation is:

```
MODULE RWMemImpl [A, D] EXPORT BigRead, BigWrite =

TYPE M            =  A -> D
VAR  memory       :  M
     done         :  SET A := {}

FUNC BigRead() -> M = RET memory

PROC BigWrite(m) =
     << done := {} >>;
     DO << VAR a | ~(a IN done) => memory(a) := m(a); done \/ := {a} >> OD;
     RET

END RWMemImpl
```

We can prove that `RWMemImpl` implements `RWMem` using an abstraction function. The state of `RWMemImpl` includes program counter values to indicate intermediate positions in the code, as well as the values of the ordinary state components. The abstraction function cannot yield partial changes to memory; therefore, we define the function as if an entire abstract `BigWrite` occurred at the point where the *first* change occurs to the memory occurs in `RWMemImpl`. (Alternative definitions are possible; for instance, we could have chosen the *last* change.) The abstraction function is defined by:

> `RWMem.memory = RWMemImpl.memory` unless there is an active `BigWrite` and `done` is nonempty. In this case `RWMem.memory = m`, where `BigWrite(m)` is the active `BigWrite`. `RWMem`’s pc for an active `BigRead` is the same as that for `RWMemImpl`. `RWMem`'s pc for an active `BigWrite` is before the body if the pc in `RWMemImpl` is at the beginning of the body; otherwise it is after the body.

In the proof that this is an abstraction function, all the atomic steps in a `BigWrite` of `RWMemImpl` except for the step that writes to memory correspond to no steps of `RWMem`. This is typical: an implementation usually has many more transitions than a spec, because the implementation is limited to the atomic actions of the machine it runs on, but the spec has the biggest atomic actions possible because that is the simplest to understand.

In this example, it is also possible to interchange the implementation and the specification, and show that `RWMem` implements `RWMemImpl`. This can be done using an abstraction function. In the proof that this is an abstraction function, the body of a `BigWrite` in `RWMem` corresponds to the entire sequence of steps comprising the body of the `BigWrite` in `RWMemImpl`.

Exercise: Add crashes to this example. The specification should contain a component `OldStates` that keeps track of the results of partial changes that could result from a crash during the current `BigWrite`. A `Crash` during a `BigWrite` in the specification can set the memory nondeterministically to any of the states in `OldStates`. A `Crash` in the implementation simply discards any active procedure. Prove the correctness of your implementation using an abstraction function. Compare this to the specs for file system crashes in handout 7.

Premature choice

In all the examples we have done so far, whenever we have wanted to prove that one module implements another (in the sense of trace inclusion), we have been able to do this using either an abstraction function or else its slightly generalized version, an abstraction relation. Will this always work? That is, do there exist modules *T* and *S* such that the traces of *T* are all included among the traces of *S*, yet there is no abstraction function or relation from *T* to *S*? It turns out that there do—abstraction functions and relations aren't quite enough.

To illustrate the problem, we give a very simple example. It is trivial, since its only point is to illustrate the limitations of the previous proof methods.

Example: Let `NonDet` be a state machine that makes a nondeterministic choice of `2` or `3`. Then it outputs `1`, and subsequently it outputs whatever it chose.

```
MODULE NonDet EXPORT Out =

VAR i := 0

APROC Out() -> Int = <<
     IF   i = 0 => BEGIN i := 2 [] i := 3 END; RET 1
     [*]  RET i FI >>

END NonDet
```

Let `LateNonDet` be a state machine that outputs `1` and then nondeterministically chooses whether to output `2` or `3` thereafter.

```
MODULE LateNonDet EXPORT Out =

VAR i := 0

APROC Out() -> Int = <<
     IF i = 0 => i := 1 [*] i = 1 => BEGIN i := 2 [] i := 3 END [*] SKIP FI;
     RET i >>

END LateNonDet
```

Clearly `NonDet` and `LateNonDet` have the same traces: `Out() = 1; Out() = 2; ...` and `Out() = 1; Out() = 3; ...`. Can we show the implementation relationships in both directions using abstraction relations?

Well, we can show that `NonDet` implements `LateNonDet` with an abstraction function that is just the identity. However, no abstraction relation can be used to show that `LateNonDet` implements `NonDet`. The problem is that the nondeterministic choice in `NonDet` occurs before the output of `1`, whereas the choice in `LateNonDet` occurs later, after the output of `1`. It is impossible to use an abstraction relation to simulate an early choice with a later choice. If you think of constructing an abstract execution to correspond to a concrete execution, this would mean that the abstract execution would have to make a choice before it knows what the implementation is going to choose.

You might think that this example is unrealistic, and that this kind of thing never happens in real life. The following three examples show that this is wrong; we will study implementations for all of these examples later in the course. We go into a lot of detail here because most people find these situations very unfamiliar and hard to understand.

Premature choice: Reliable messages

Here is a realistic example (somewhat simplified) that illustrates the same problem: two specs for reliable channels, which we will study in detail later, in handout 26 on reliable messages. A reliable channel accepts messages and delivers them in FIFO order, except that if there is a crash, it may lose some messages. The straightforward spec drops some queued messages during the crash.

```
MODULE ReliableMsg [M] EXPORT Put, Get, Crash =

VAR q  : SEQ M := {}

APROC Put(m)      = << q + := {m} >>
APROC Get() -> M = << VAR m := q.head | q := q.tail; RET m >>

APROC Crash()     = << VAR q' | q' <<= q => q := q' >>
% Drop any of the queued messages (<<= is non-contiguous subsequence)

END ReliableMsg
```

Most practical implementations (for instance, the Internet's TCP protocol) have cases in which it isn't known whether a message will be lost until long after the crash. This is because they ensure FIFO delivery, and get rid of retransmitted duplicates, by numbering messages sequentially and discarding any received message with an earlier sequence number than the largest one already received. If the underlying message transport is not FIFO (like the Internet) and there are two undelivered messages outstanding (which can happen after a crash), the earlier one will be lost if and only if the later one overtakes it. You don't know until the overtaking happens whether the first message will be lost. By this time the crash and subsequent recovery may be long since over.

The following spec models this situation by 'marking' the messages that are queued at the time of a crash, and optionally dropping any marked messages in `Get`.

```
MODULE LateReliableMsg [M] EXPORT Put, Get, Crash =

VAR q  : SEQ [m, mark: Bool] := {}

APROC Put(m)      = << q + := {m} >>
APROC Get() -> M =
     << DO VAR x := q.head | q := q.tail; IF x.mark => SKIP [] RET x.m FI OD >>
```

```
APROC Crash()     = << q := {x :IN q | | x{mark := true}} >>
```
% Mark all the queued messages. This is a sequence, not a set constructor, so it doesn't reorder the messages.

```
END LateReliableMsg
```

Like the simple `NonDet` example, these two specs are equivalent, but we cannot prove that `LateReliableMsg` implements `ReliableMsg` with an abstraction relation, because `ReliableMsg` makes the decision about what messages to drop sooner, in `Crash`. `LateReliableMsg` makes this decision later, in `Get`, and so do the standard implementations.

Premature choice: Consensus

For another examples, consider the *consensus* problem of getting a set of process to agree on a single value chosen from some set of allowed values; we will study this problem in detail later, in handout 18 on consensus. The spec doesn't mention the processes at all:

```
MODULE Consensus [V] EXPORT Allow, Outcome =

VAR  outcome     :  (V + Null) := nil                 % Data value to agree on

APROC Allow(v) = << outcome = nil => outcome := v [] SKIP >>
FUNC  Outcome() -> (V + Null) = RET outcome [] RET nil

END Consensus
```

This spec chooses the value to agree on as soon as the value is allowed. `Outcome` may return `nil` even after the choice is made because in a distributed implementation it's possible that not all the participants have heard what the outcome is. An implementation almost certainly saves up the allowed values and does a lot of communication among the processes to come to an agreement. The following spec has that form. It is more complicated than the first one (more state and more operations), and closer to an implementation.

```
MODULE LateConsensus [V] EXPORT Allow, Outcome =

VAR  outcome     :  (V + Null) := nil                 % Data value to agree on
     allowed     :  SET V := {}

APROC Allow(v) = << allowed \/ := {v} >>

FUNC  Outcome() -> (V + Null) = RET outcome [] RET nil

APROC Agree() = << VAR v | v IN allowed /\ outcome = nil => outcome := v >>

END LateConsensus
```

It should be clear that these two modules have the same traces: a sequence of `Allow(x)` and `Outcome() = y` actions in which every `y` is either `nil` or the same value, and that value is an argument of some preceding `Allow`. But there is no abstraction relation from `LateConsensus` to `Consensus`, because there is no way for `LateConsensus` to come up with the outcome before it does its internal `Agree` action.

Note that if `Outcome` didn't have the option to return `nil` even after `outcome # nil`, these modules would not be equivalent, because `LateConsensus` would allow the behavior

```
    Allow(1); Outcome()=nil, Allow(2), Outcome()=1
```

and `Consensus` would not.

Premature choice: Multi-word clock

Here is a third example of premature choice in a spec: reading a clock. The spec is simple:

```
MODULE Clock EXPORT Read =

VAR  t             :  Int                              % the current time

THREAD Tick() = DO << t + := 1 >> OD                   % demon thread advances t

PROC Read() -> Int = << RET t >>

END Clock
```

This is in a concurrent world, in which several threads can invoke `Read` concurrently, and `Tick` is a demon thread that is entirely internal. In that world there are three transitions associated with each invocation of `Read`: entry, body, and exit. The entry and exit transitions are external because `Read` is exported.

We may want an implementation that allows the clock to have more precision than can be carried in a single memory location that can be read and written atomically. We could easily achieve this by locking the clock representation, but then a slow process holding the lock (for instance, one that gets pre-empted) could block other processes for a long time. A clever 'wait-free' implementation of `Read` (which appears in handout 17 on formal concurrency) reads the various parts of the clock representation one at a time and puts them together deftly to come up with a result which is guaranteed to be one of the values that `t` took on during this process. The following spec abstracts this strategy; it breaks `Read` down into two atomic actions and returns some value, non-deterministically chosen, between the values of `t` at these two actions.

```
MODULE LateClock EXPORT Read =

VAR  t             :  Int                              % the current time

THREAD Tick() = DO << t := t + 1 >> OD                 % demon thread advances t

PROC Read() -> Int = VAR t1: Int |
    << t1 := t >>; << VAR t2 | t1 <= t2 /\ t2 <= t => RET t2 >>

END LateClock
```

Again both specs have the same traces: a sequence of invocations and responses from `Read`, such that for any two `Read`s that don't overlap, the earlier one returns a smaller value `tr`. In `Clock` the choice of `tr` depends on when the body of `Read` runs relative to the various `Tick`s. In `LateClock` the `VAR t2` makes the choice of `tr`, and it may choose a value of `t` some time ago. Any abstraction relation from `LateClock` to `Clock` has to preserve `t`, because a thread that does a complete `Read` exposes the value of `t`, and this can happen between any two other transitions. But `LateClock` doesn't decide its return value until its last atomic command, and when it does, it may choose an earlier value than the current `t`; no abstraction relation can explain this.

Prophecy variables

One way to cope with these examples and others like them is to use ad hoc reasoning to show that `LateX` implements `X`; we did this informally in each example above. This strategy is much easier if we make the transition from premature choice to late choice at the highest level possible, as we did in these examples. It's usually too hard to show directly that a complicated module that makes a late choice implements a spec that makes a premature choice.

But it isn't necessary to resort to ad hoc reasoning. Our trusty method of abstraction functions can also do the job. However, we have to use a different sort of auxiliary variable, one that can look into the future just as a history variable looks into the past. Just as we did with history variables, we will show that a module *TP* augmented with a *prophecy variable* has the same traces as the original module *T*. Actually, we can show that it has the same *finite* traces, which is enough to take care of safety properties. It also has the same infinite traces provided certain technical conditions are satisfied, but we won't worry about this because we are not interested in liveness. To show that the traces are the same, however, we have to work *backward* from the end of the trace instead of forward from the beginning.

A prophecy variable guesses in advance some non-deterministic choice that *T* is going to make later. The guess gives enough information to construct an abstraction function to the spec that is making a premature choice. When execution reaches the choice that *T* makes non-deterministically, *TP* makes it deterministically according to the value of the prophecy variable. *TP* has to choose enough different values for the prophecy variable to keep from ruling out any executions of *T*.

The conditions for an added variable to be a prophecy variable are closely related to the ones for a history variable, as the following table shows.

History variable	*Prophecy variable*
1. Every initial state has at least one value for the history variable.	1. Every state has at least one value for the prophecy variable.
2. No existing step is disabled by new guards involving a history variable.	2. No existing step is disabled *in the backward direction* by new guards involving a prophecy variable. More precisely, for each step (t, π, t') and state (t', p') there must be a p such that there is a step $((t, p), \pi, (t', p'))$.
3. A value assigned to an existing state component must not depend on the value of a history variable. One important case of this is that a return value must not depend on a history variable.	3. Same condition
	4. If t is an initial state of T and (t, p) is a state of TP, it must be an initial state.

If these conditions are satisfied, the state machine *TP* with the prophecy variable will have the same traces as the state machine *T* without it. You can see this intuitively by considering any finite execution of *T* and constructing a corresponding execution of *TP*, starting from the end. Condition (1) ensures that we can find a last state for *TP*. Condition (2) says that for each backward step of *T* there is a corresponding backward step of *TP*, and condition (3) says that in this step p doesn't affect what happens to t. Finally, condition (4) ensures that we end up in an initial state of *TP*.

Let's review our examples and see how to add prophecy variables, marking the additions with boxes. For `LateNonDetP` we add `pI` that guesses the choice between 2 and 3. The abstraction function is just `NonDet.i = LateNonDetP.pI`.

```
VAR i  := 0
    pI := 0

APROC Out() -> Int = <<
    IF   i = 0 => i := 1; BEGIN pI := 2 [] pI := 3 END
    [*]  i = 1 => BEGIN pI = 2 => i := 2 [] pI = 3 => i := 3 END [*] SKIP FI;
    RET i >>
```

For `LateReliableMsgP` we add a `pDead` flag to each marked message that forces `Get` to discard it. `Crash` chooses which `dead` flags to set. The abstraction function just discards the marks and the dead messages.

```
VAR q  : SEQ [m, mark: Bool, pDead: Bool] := {}

% ABSTRACTION FUNCTION ReliableMsg.q = {x :IN LateReliableMsg.q | ~ x.dead | x.m}

% INVARIANT (ALL i :IN q.dom | q(i).dead ==> q(i).mark)

APROC Get() -> M =
    << DO VAR x := q.head |
        q := q.tail; IF x.mark => SKIP [] ~ x.pDead => RET x.m FI OD >>

APROC Crash()    = << VAR pDeads: SEQ Bool | pDeads.size = q.size =>
    q := {x :IN q, pD :IN pDeads | | x{mark := true, pDead := pD}}
```

Alternatively, we can prophesy the entire state of `ReliableMsg` as we did with `db` in `StatDB`, which is a little less natural in this case:

```
VAR pQ  : SEQ M := {}

% INVARIANT {x :IN q | ~ x.mark | x.m} <<= pQ /\ pQ <<= {x :IN q | | x.m}

APROC Get() -> M =
    << DO VAR x := q.head |
        q := q.tail;
        IF x.mark /\ (pQ = {} \/ x.m # pQ.head) => SKIP
        [] pQ := pQ.tail; RET x.m
        FI OD >>

APROC Crash() =
    << VAR q' | q' <<= q => pQ := q'; q := {x :IN q | | x{mark := true}} >>
```

For `LateConsensusP` we follow the example of `NonDet` and just prophesy the outcome in `Allow`. The abstraction function is `Consensus.outcome = LateConsensusP.pOutcome`

```
VAR outcome     :  (V + Null) := nil                    % Data value to agree on
    pOutcome    :  (V + Null) := nil
    allowed     :  SET V := {}

APROC Allow(v) =
    << allowed \/ := {v}; IF pOutcome = nil => pOutcome := v [] SKIP FI >>

APROC Agree() =
    << VAR v | v IN allowed /\ outcome = nil /\ v = pOutcome => outcome := v >>
```

For `LateClockP` we choose the result at the beginning of `Read`. The second command of `Read` has to choose this value, which means it has to wait until `Tick` has advanced `t` far enough. The transition of `LateClockP` that corresponds to the body of `Clock.Read` is the `Tick` that gives `t` the pre-chosen value. This seems odd, but since all these transitions are internal, they all have empty external traces, so it is perfectly OK.

```
VAR t           :  Int                                  % the current time
    pT          :  Int

PROC Read() -> Int = VAR t1: Int |
    << t1 := t; VAR t': Int | pT := t' >>;
    << VAR t2 | t1 <= t2 /\ t2 <= t /\ t2 = pT => RET t2 >>
```

Most people find it much harder to think about prophecy variables than to think about history variables, because thinking about backward execution does not come naturally. It's easy to see that it's harmless to carry on extra information in the history variables that isn't allowed to affect the main computation. A prophecy variable, however, *is* allowed to affect the main computation, by forcing a choice that was non-deterministic to be taken in a particular way. Condition (2) ensures that in spite of this, no traces of *T* are ruled out in *TP*. It requires us to use a prophecy variable in such a way that for any possible choice that *T* could make later, there's some choice that *TP* can make for the prophecy variable's value that allows *TP* to later do what *T* does.

Here is another way of looking at this. Condition (2) requires enough different values for the prophecy variables p_i to be carried forward from the points where they are set to the points where they are used to ensure that as they are used, any set of choices that *T* could have made is possible for some execution of *TP*. So for each command that uses a p_i to make a choice, we can calculate the set of different values of the p_i that are needed to allow all the possible choices. Then we can propagate this set back through earlier commands until we get to the one that chooses p_i, and check that it makes enough different choices.

Because prophecy variables are confusing, it's important to use them only at the highest possible level. If you write a spec *SE* that makes an early choice, and implement it with a module *T*, don't try to show that *T* satisfies *SE*; that will be too confusing. Instead, write another spec *SL* that makes the choice later, and use prophecy variables to show that *SL* implements *SE*. Then show that *T* implements *SL*; this shouldn't require prophecy. We have given three examples of this strategy.

Backward simulation

Just as we could use abstraction relations instead of adding history variables, we can use a different kind of relation, satisfying different start and step conditions, instead of prophecy variables. This new sort of relation also guarantees trace inclusion. Like an ordinary abstraction relation, it allows construction of an execution of the specification, working from an execution of the implementation. Not surprisingly, however, the construction works *backwards* in the execution of the implementation instead of forwards. (Recall the inductive proof for abstraction relations.) Therefore, it is called a *backward simulation*.

The following table gives the conditions for a backward simulation using relation *R* to show that *T* implements *S*, aligning each condition with the corresponding one for an ordinary abstraction relation. To highlight the relationship between the two kinds of abstraction mappings, an ordinary abstraction relation is also called a *forward simulation*.

Forward simulation	*Backward simulation*
1. If t is any initial state of T, then there is an initial state s of S such that $(t, s) \in R$.	1. If t is any reachable state of T, then there a state s of S such that $(t, s) \in R$.
2. If t and s are reachable states of T and S respectively, with $(t, s) \in R$, and (t, π, t') is a step of T, then there is an execution fragment of S from s to some s', having the same trace, and with $(t', s') \in R$.	2. If t' and s' are states of T and S respectively, with $(t', s') \in R$, (t, π, t') is a step of T, and t is reachable, then there is an execution fragment of S from some s to s', having the same trace, and with $(t, s) \in R$.
	3. If t is an initial state of T and $(t, s) \in R$ then s is an initial state of S.

(1) applies to any reachable state t rather than any initial state, since running backwards we can start in any reachable state, while running forwards we start in an initial state. (2) requires that every backward (instead of forward) step of *T* be a simulation of a step of *S*. (3) is a new condition ensuring that a backward run of *T* ending in an initial state simulates a backward run of *S* ending in an initial state; since a forward simulation never ends, it has no analogous condition.

Theorem 3: If there exists a backward simulation from *T* to *S* then every *finite* trace of *T* is also a trace of *S*.

Proof: Start at the end of a finite execution and work backward, exactly as we did for forward simulations.

Notice that Theorem 3 only yields finite trace inclusion. That's different from the forward case, where we get infinite trace inclusion as well. Can we use backward simulations to help us prove general trace inclusion? It turns out that this doesn't always work, for technical reasons, but it works in two situations that cover all the cases you are likely to encounter:

- The infinite traces are exactly the limits of finite traces. Formally, we have the condition that for every sequence of successively extended finite traces of *S*, the limit is also a trace of *S*.

- The correspondence relation relates only finitely many states of *S* to each state of *T*.

In the `NonDet` example above, a backward simulation can be used to show that `LateNonDet` implements `NonDet`. In fact, the inverse of the relation used to show that `NonDet` implements `LateNonDet` will work. You should check that the three conditions are satisfied.

Backward simulations vs. prophecy variables

The same equivalence that holds between abstraction relations and history variables also holds between backward simulations and prophecy variables. The invariant on the prophecy variable becomes the abstraction relation for the backward simulation.

Completeness

Earlier we asked whether forward simulations always work to show trace inclusion. Now we can ask whether it is always possible to use either a forward or a backward simulation to show trace inclusion. The satisfying answer is that a *combination* of a forward and a backward simulation, one after the other, will always work, at least to show finite trace inclusion. (Technicalities again arise in the infinite case.) For proofs of this result and discussion of the technicalities, see the papers by Abadi and Lamport and by Lynch and Vondrager cited below.

History and further reading

The idea of abstraction functions has been around since the early 1970's. Tony Hoare introduced it in a classic paper (C.A.R. Hoare, Proof of correctness of data representations. *Acta Informatica* **1** (1972), pp 271-281). It was not until the early 1980's that Lamport (L. Lamport, Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems* **5**, 2 (Apr. 1983), pp 190-222) and Lam and Shankar (S. Lam and A. Shankar, Protocol verification via projections. *IEEE Transactions on Software Engineering* **SE-10**, 4 (July 1984), pp 325-342) pointed out that abstraction functions can also be used for concurrent systems.

People call abstraction functions and relations by various names. 'Refinement mapping' is popular, especially among European writers. Some people say 'abstraction mapping'.

History variables are an old idea. They were first formalized (as far as I know), in Abadi and Lamport, The existence of refinement mappings. *Theoretical Computer Science* **2**, 82 (1991), pp 253-284. The same paper introduced prophecy variables and proved the first completeness result. For more on backward and forward simulations see N. Lynch and F. Vondrager, Forward and backward simulations—Part I: Untimed systems. *Information and Computation* **121**, 2 (Sep. 1995), pp 214-233.

.

9. Atomic Semantics of Spec

This handout defines the semantics of the atomic part of the Spec language fairly carefully. It tries to be precise about all difficult points, but is sloppy about some things that seem obvious in order to keep the description short and readable. For the syntax and an informal account of the semantics, see the Spec reference manual, handout 4.

There are three reasons for giving a careful semantics of Spec:

1. To give a clear and unambiguous meaning for Spec programs.
2. To make it clear that there is no magic in Spec; its meaning can be given fairly easily and without any exotic methods.
3. To show the versatility of Spec by using it to define itself, which is quite different from the way we use it in the rest of the course.

This handout is divided into two parts. In the first half we describe semi-formally the essential ideas and most of the important details. Then in the second half we present the atomic semantics precisely, with a small amount of accompanying explanation.

Semi-formal atomic semantics of Spec[1]

Our purpose is to make it clear that there is no arm waving in the Spec notation that we have given you. A translation of this into fancy words is that we are going to study a formal semantics of the Spec language.

Now that is a formidable sounding term, and if you take a course on the semantics of programming languages (6.821—Gifford, 6.830J—Meyer) you will learn all kinds of fancy stuff about bottom and stack domains and fixed points and things like that. You are not going to see any of that here. We are going to do a very simple minded, garden-variety semantics. We are just going to explain, very carefully and clearly, how it is that every Spec construct can be understood, as a transition of a state machine. So if you understand state machines you should be able to understand all this without any trouble.

One reason for doing this is to make sure that we really do know what we are talking about. In general, descriptions of programming languages are not in that state of grace. If you read the Pascal manual or the C manual carefully you will come away with a number of questions about exactly what happens if I do this and this, questions which the manual will not answer adequately. Two reasonably intelligent people who have studied it carefully can come to different conclusions, argue for a long time, and not be able to decide what is the right answer by reading the manual.

[1] These semi-formal notes take the form of a dialogue between the lecturer and the class. They were originally written by Mitchell Charity for the 1992 edition of this course, and have been edited for this handout.

6.826—Principles of Computer Systems 2000

There is one class of mechanisms for saying what the computer should do that often does answer your questions precisely, and that is the instruction sets of computers (or, in more modern language, the architecture). These specs are usually written as state machines with fairly simple transitions, which are not beyond the power of the guy who is writing the manual to describe properly. A programming language, on the other hand, is not like that. It has much more power, generality, and wonderfulness, and also much more room for confusion.

Another reason for doing this is to show you that our methods can be applied to a different kind of system than the ones we usually study, that is, to a programming language, a notation for writing programs or a notation for writing specifications. We are going to learn how to write a spec for that particular class of computer systems. This is a very different application of Spec from the last one we looked at, which was file systems. For describing a programming language, Spec is not the ideal descriptive notation. If you were in the business of giving the semantics of programming languages, you wouldn't use Spec. There are many other notations, some of them better than Spec (although most are far worse). But Spec is good enough; it will do the job. And there is a lot to be said for just having one notation you can use over and over again, as opposed to picking up a new one each time. There are many pitfalls in devising a new notation.

Those are the two themes of this lecture. We are going to get down to the foundations of Spec, and we are going to see another, very different application of Spec. Certainly a programming language is very different from a file system.

For this lecture, we will only talk about the sequential or atomic semantics of Spec, not about concurrent semantics. Consider the program:

```
                     x, y = 0
thread 1:                                    thread 2:
<<  x := 3 >>                                << z := x + y >>
<<  y := 4 >>
```

In the concurrent world, it is possible to get any of the values 0, 3, or 7 for `z`. In the sequential world, which we are in today, the only possible values are 0 and 7. It is a simpler world. We will be talking later (in handout 17 on formal concurrency) about the semantics of concurrency, which is unavoidably more complicated.

In a sequential Spec program, there are three basic constructs (corresponding to sections 5, 6, and 7 of the reference manual):

Expressions
Commands
Routines

In order to describe what each of these things means, we first of all need some notion of what kind of thing the meaning of an expression or command might be. Then we have to explain in detail exactly what the meaning of each possible kind of expression is. The basic technique we use is the standard one for a situation where you have things that are made up out of smaller things: structural induction.

The idea of structural induction is this. If you have something which is made up of an `A` and a `B`, and you know the meaning of each, and have a way to put them together, you know how to get the meaning of the bigger thing.

Some ways to put things together in Spec:

```
A , B
A ; B
a + b
A [] B
```

State

What are the meanings going to be? Our basic notion is that what we are doing when writing a Spec program is describing a state machine. The central properties of a state machine are that it has states and it has transitions.

A state is a function from names to values: `State: Name -> Value`. For example:

```
VAR x: Int
    y: Int
```

If there are no other variables, the state simply consists of the mapping of the names `"x"` and `"y"` to their corresponding values. Initially, we don't know what their values are. Somehow the meaning we give to this whole construct has to express that.

Next, if we write `x := 1`, after that the value of `x` is `1`. So the meaning of this had better look something like a transition that changes the state, so that no matter what the `x` was before, it is `1` afterwards. That's what we want this assignment to mean.

Spec is much simpler than C. In particular, it does not have "references" or "pointers". When you are doing problems, if you feel the urge to call `malloc`, the correct thing to do is to make a function whose range is whatever sort of thing you want to allocate, and then choose a new element of the domain that isn't being used already. You can use the integers or any convenient sort of name for the domain, that is, to name the values. If you define a `CLASS`, Spec will do this for you automatically.

So this is the state, just these name to value mappings.

Names

Spec has a module structure, so that names have two parts, the module name and the simple name. When referring to a variable in another module, both parts must be used.

```
MODULE M                    MODULE N

VAR x                        M.x := 3
    x := 3
    ...
    M.x := 3
```


To simplify the semantics, we are going to use `M.x` as the name everywhere. In order to apply the semantics, you first must go through the program and replace every `x` declared in the current module `M` with `M.x`. You convert all references to global variables to these two part names, so that each name refers to exactly one thing. This makes things simpler to describe and understand, but uglier to read. This transformation doesn't change the meaning of the program; it could have been written with two part names in the first place.

All the global variables have these two part names. However, local variables are not prefixed by the module name:

```
PROC
    VAR i | ... i
```

This is how we tell the global state apart from the local state. Global state names have dots, local state names do not.

Question: Can modules be nested?

No. Spec is meant to be suitable for the kinds of specs and implementations that we do in this course, which are no more than moderately complex. Features not really needed to write our specs are left out to keep it simpler.

Expressions

What should the meaning of an expression be? Note that expressions do *not* affect the state.

Question: What about assignments?

Assignments are not expressions. If you have been programming in C, you have the weird idea that assignments are expressions. But not in the rest of the world. Spec in particular takes a hard line that not only are assignments not expressions, but functions are not allowed to affect the state.

What are the semantics of an expression? Well, the type for the meaning of an expression is `S -> V`. An expression is a function from state to value. It can be a partial function, since Spec does not require that all expressions be defined. But it has to be a function—we do require that expressions are deterministic. We want determinism so something like `f(x) = f(x)` always comes out true. Reasoning is just too hard if this isn't true. If a function is nondeterministic then obviously this needn't come out true. (The classic example of a nondeterministic function is a random number generator.)

So, **expressions are deterministic and do not affect state.**

There are three types of expressions:

Type	*Example*	*Meaning*
constant	`1`	`(\ s | 1)`
variable	`x`	`(\ s | s("x"))`
function invocation	`f(x)`	next sub-section

(The type of these lambda's is not quite right, as we will see later).

Note that we have to keep the Spec in which we are writing the semantics separate from the Spec of the semantics we are describing. Therefore, we had to write `s("x")` instead of just `x`, because it is the `x` of the target system we are talking about, not the `x` of the describing system.

The third type of expression is function invocation. We will only talk about functions with a single argument. If you want a function with two arguments, you can make one by combining the two arguments into a tuple or record, or by currying: defining a function of the first argument that returns a function of the second argument.

What about `x + y`? This is just shorthand for `T."+"(x, y)`, where `T` is the type of `x`. Everything that is not a constant or a variable is an invocation. This should be a familiar concept for those of you who know Scheme.

Semantics of function invocation

What are the semantics of function invocation? Given a function `T -> U`, the correct type of its meaning is `(T, S) -> U`, since the function can read the state but not modify it. Next, how are we going to attach a meaning to an invocation `f(x)`? Remember the rule of structural induction. In order to explain the meaning of a complicated thing, you are supposed to build it out of the meaning of simpler things. We know the meaning of `x` and of `f`. We need to come up with a map from states to values that is the meaning of `f(x)`. That is, we get our hands on the meaning of `f` and the meaning of `x`, and then put them together appropriately. What is the meaning of `f`? `s("f")`. So,

```
f(x) means ... s("f") ... s("x")
```

How are we going to put it together, remembering the type we want for `f(x)`?

```
f(x) means (\ s | s("f") (s("x"), s)) )
```

Now this could be complete nonsense, for instance if `s("f")` evaluates to an integer. If `s("f")` isn't a function then this doesn't typecheck. But there is no doubt about what this means if it is legal. It means invoke the function.

That takes care of expressions, because there are no other expressions besides these. Structural induction says you work your way through all the different ways to put little things together to make big things, and when you have done them all, you are finished.

Question: What about undefined functions?

Then the `(T, S) -> U` mapping is partial.

Question: Is `f(x) = f(x)` if `f(x)` is undefined?

No, it's undefined. But those are deep waters and I propose to stay out of them.

Commands

What is the type of the meaning of a command? Well, we have states and values to play with, and we have used up `S -> V` on expressions. What sort of thing is a command? It's a transition from one state to another.

Expressions: `S -> V`

Commands: `S -> S ?`

This is good for a subset of commands. But what about this one?

```
x := 1 [] x := 2
```

Is its meaning a function from states to states? No, from states to *sets* of states. It can't just be a function. It has to be a relation. Of course, there are lots of ways to code relations as functions. The way we use is:

Commands: `(S, S) -> Bool`

There is a small complication because Spec has exceptions, which are useful for writing many kinds of specifications, not to mention programs. So we have to deal with the possibility that the result of a command is not a garden-variety state, but involves an exception.

To handle this we make a slight extension and invent a thing called an *outcome*, which is very much like a state except that it has some way of coding that an exception has happened. Again, there are many ways to code that. The way we use is that an outcome has the same type as a state: it's a function from names to values. However, there are a couple of funny names that you can't actually write in the program. One of them is `$x`, and we adopt the convention that if `o("$x") = ""` (empty string), then `o` is a garden-variety state. If `o("$x") = "exception-name"`, then there is that exception in outcome `o`. Some Spec commands, in particular `";"` and `EXCEPT`, do something different if one of their pieces produces an exception.

Now we just work our way through the command constructs (with an occasional digression).

Commands — assignment

```
x := 1
```

or in general

```
variable := expression
```

What we have to come up with for the meaning is an expression of the form

```
(\ s, o | ...)
```

How do we say that `o` is related to `s`? The function returns `true`. We are encoding a relation between states and outcomes as a function from a state and outcome to a `Bool`. The function is supposed to give back `true` exactly when the relation holds.

So when does the relation hold for `x := exp`? Well, perhaps when `o(x) = exp`? (`ME` is the meaning function for expressions.)

```
o("x") = ME(e)(s)
```

. Well, the valid transition

```
x=0              x=1
         ->
y=0              y=0
```

would certainly be allowed. But what others would be allowed? What about:

```
x=0              x=1
         ->
y=0              y=94
```

It would also be allowed, so this can't be quite right. Half right, but missing something important. You have to say that you don't mess around with the rest of the state. The way you do that is to say that the outcome is equal to the state except at the variable.

```
o = s{"x" -> ME(e)(s)}
```

This is just a Spec function constructor, of the form `f{arg -> value}`.

Aside—an alternate encoding for commands

As we said before, there are many ways to code the command relation. Another possibility is:

Commands: `S -> SET S`

This encoding seems to make the meanings of commands clumsier to write, though it is entirely equivalent to the one we have chosen.

There is a third approach, which has a lot of advantages: write predicates on the state values. If `x` and `y` are the state variables in the pre-state, and `x'` and `y'` the state variables in the post-state, then

```
(x' = 1 /\ y' = y)
```

is another way of writing

```
o = s{"x" -> 1}
```

In fact, this approach is another way of writing programs. I could write everything just as predicates. Of course, I could also write everything as this `o = s{...}` sort of cruft, but that would look pretty awful. It is more conceivable that we would want to write things as predicates, because it doesn't look so bad.

Sometimes it's actually nice to do this. Say you want to write the predicate that says you can have any value at all for `x`. The spec

```
VAR z | x := z
```

is just

```
(y' = y)
```

(in the simple world where the only state variables are `x` and `y`). This is much simpler that the previous, rather inscrutable, piece of program. So sometimes this predicate way of doing things can be a lot nicer, but in general it seems to be not as satisfactory, mainly because the `y'=y` stuff clutters things up an awful lot.

That was just an aside, but sometimes it's convenient to describe the things that can go on in a specification using predicates rather than functions from state pairs to `Bool`.

Commands — routine invocation `p(x)`

What are the semantics of routine invocation? Well, it has to do something with `s`. What about the argument? There are many ways to deal with the argument. The way we do it is to use another pseudo-variable `$a` to pass the argument and get back the result.

The meaning of `p(e)` is going to be

```
LAMBDA (s, o) = RET         (s                              Take the state,
                              {"$a" -> ME(e)(s)}            append the argument,
                    ME(p)(s)                                get the routine
                                                  , o)      and invoke it
```

or, writing the whole thing on one line in the normal way,

```
LAMBDA (s, o) = RET ME(p)(s)(s{"$a" -> ME(e)(s)}, o)
```

What does this say? This invocation relates a state to an outcome if, when you take that state, and modify its `$a` component to be equal to the value of the argument, the meaning of the routine relates that state to the outcome. Another way of writing this, which isn't so nested and might be clearer, would be to introduce an intermediate state `s'`:

```
VAR s' = s{"$a" -> ME(e)(s)} | ME(p)(s)(s', o)
```

These two are exactly the same thing. The invocation relates `s` to `o` iff the routine relates `s'` to `o`, where `s'` is just `s` with the argument passing component modified. `$a` is just a way of communicating the argument value to the routine.

Question: Why use `ME(p)(s)` rather than `MR`?

`MR` is the meaning function for routines, that is, it turns the syntax of a routine declaration into a function on states and arguments that is the meaning of that syntax. We would use `MR` if we were looking at a `FUNC`. But `p` is just a variable (of course it had better be bound to a routine value, or this won't typecheck).

Aside—an alternate encoding for invocation

Here is a different way of communicating the argument value to the function; you can skip this section if you like. We could take the view that the routine definition

```
PROC P(i: Int) = ...
```

is defining a whole flock of different commands, one for every possible argument value. Then we need to pick out the right one based on the argument value we have. If we coded it this way (and it is merely a coding thing) we would get:

```
ME(p)(s)(ME(e)(s)) (s,o)
```

This says, first get `ME(p)`, the meaning of `p`. This is not a transition but a function from argument values to transitions, because the idea is that for every possible argument value, we are going to get a different meaning for the routine, namely what that routine does when given that particular argument value. So we pass it the argument value `ME(e)(s)`, and invoke the resulting transition.

These two alternatives are based on different choices about how to code the meaning of routines. If you code the meaning of a routine simply as a transition, then Spec picks up the argument value out of the magic `$a` variable. But there is nothing mystical going on here. Setting `$a` corresponds exactly to what we would do if we were designing a calling sequence. We would say "I am going to pass the argument in register 1". Here, register 1 is `$a`.

The second approach is a little bit more mystical. We are taking more advantage of the wonderful abstract power and generality that we have. If someone writes a factorial function, we will treat it as an infinite supply of different functions; one computes the factorial of 1, another the factorial of 2, another the factorial of 3, and so forth. In `ME(p)(s)(ME(e)(s))(s, o)`, `ME(p)(s)` is the infinite supply, `ME(e)(s)` is the argument that picks out a particular function, to which we finally pass `(s, o)`.

However, there are lots of other ways to do this. One of the things which makes the semantics game hard is that there are many choices you can make. They don't really make that much difference, but they can create a lot of confusion, both because a bad choice can leave you in a briar patch of notation, and because you can get confused about what choice was made.

So, while this

```
RET ME(p) (S) (S("$a" -> ME(e) (s)),o)
```

and this

```
VAR s' := s{"$a" -> ME(e)(s)} | RET ME(p)(s) (s',o)
```

are two ways of writing exactly the same thing, this

```
ME(p)(s)(ME(e)(s)) (s,o)
```

is different, and only makes sense with a different choice about what the meaning of a function is. The latter is more elegant, but we use the former because it is less confusing.

Stepping back from these technical details, what the meaning function is doing is taking an expression and producing its meaning. The expression is a piece of syntax, and there are a lot of possible ways of coding the syntax. Which exact way we chose isn't that important.

Commands — `SKIP`

```
LAMBDA (s, o) = RET s = o
```

In other words, the outcome after `SKIP` is the same as the pre-state. Later on, in the formal half of the handout, we give a table for the commands which takes advantage of the fact that there is a lot of boilerplate—the `LAMBDA (s, o) = RET` stuff is always the same, and so is the treatment of exceptions. So the table just shows, for each syntactic form, what goes after the `RET`.

Commands — *`HAVOC`*

```
LAMBDA (s, o) = true
```

In other words, after `HAVOC` you can have any outcome. Actually this isn't quite good enough, since we want to be able to have any *sequence* of outcomes. We deal with this by introducing another magic state component `$havoc` with a `Bool` value. Once `$havoc` is true, any transition can happen, including one that leaves it true and therefore allows havoc to continue. We express this by adding to the command boilerplate the disjunct `s("$havoc")`, so that if `$havoc` is true in `s`, any command relates `s` to any `o`.

Now for the compound commands.

Commands — `c1 [] c2`

```
MC(c1)          MC(c2)

     (s, o)          (s, o)

            \/
```

or on one line,

```
MC(c1)(s, o) \/ MC(c2)(s, o)
```

Non-deterministic choice is the 'or' of the relations.

Commands — `c1 [*] c2`

It is clear we should begin with

```
MC(c1)(s, o) \/ …
```

But what next? One possibility is

```
                ~ MC(c1)(s, o) ==> ...
```

This is in the right direction, but not correct. Else means that if there is no *possible* outcome of `c1`, then you get to try `c2`. So there are two possible ways for an else to relate a state to an outcome. One is for `c1` to relate the state to the outcome, the other is that there is no possible way to make progress with `c1` in the state, and `c2` to relates the state to the outcome.

The correct encoding is

```
MC(cl)(s,o) \/ (ALL o' | ~ MC(cl) (s, o')) /\ MC(c2)(s,o) )
```

Commands — `c1 ; c2`

Although the meaning of semicolon may seem intuitively obvious, it is more complex than one might first suspect—more complicated than or, for instance. We interpreted the command `c1 [] c2` as `MC(cl) \/ MC(c2)`. Because semicolon is a sequential composition, it requires that our semantics move through an intermediate state.

If these were functions (if we could describe the commands as functions) then we could simply describe a sequential composition as `(F2 (Fl s))`. However, because Spec is not a functional language, we need to compose relations, in other words, to establish an intermediate state as a precursor to the final output state. As a first attempt, we might try:

```
(LAMBDA (s, o) -> Bool = RET
     (EXISTS o' | MC(cl)(s, o') /\ MC(c2)(o', o)))
```

In words, this says that you can get from `s` to `o` via `c1 ; c2` if there exists an intermediate state `o'` such that `c1` takes you from `s` to `o'` and `c2` takes you from `o'` to `o`. This is indeed the composition of the relations. But is this always the meaning of `";"`? In particular, what if `c1` produces an exception? When `c1` produces an exception, we should *not* execute `c2`. Our first try does not capture that possibility. To correct for this, we need to verify that `o'` is a normal state. If it is an exceptional state, then it is the result of the composition and we ignore `c2`.

```
     (EXISTS o' | MC(cl)(s, o') /\ (   ~IsX(o') /\ MC(c2)
                                    \/  IsX(o') /\ o' = o))
```

Commands — `c1 EXCEPT xs => c2`

Now, what if we have a handler for the exception? If we assume (for simplicity) that all exceptions are handled, we would simply implement the complement of the semicolon case. If we get an exception, then do `c2`. If there is no exception, do *not* do `c2`. We also need to include an additional check to insure that the exception considered is an element of the exception set—that is to say, that it is a handled exception.

```
     (EXISTS o' | MC(cl)(s, o') /\
                (   ((~IsX(o') \/ ~o'("$x") IN xs) /\ o' = o)
                 \/    IsX(o') /\  o'("$x") IN xs) /\ MC(c2)(o'{"$x" -> ""}, o)
     )
```

So, with this semantics for handling exceptions, the meaning of:

```
      (c1 EXCEPT xs => c2); c3)
```

is

if normal	do `c1`, no `c2`, do `c3`
if exception, handled	do `c1`, do `c2`, do `c3`
if exception and not handled	do `c1`, no `c2`, no `c3`

Commands — `VAR id: T | c0`

The idea is "there exists a value for `id` such that `c0` succeeds". This intuition suggests something like

```
     (EXISTS v |  v IN T /\ MC(c0)(s{"id" -> v}, o))
```

However, if we look carefully, we see that `id` is left defined in the output state `o`. (Why is this bad?) To correct this omission we need to introduce an intermediate state `o'` from which we may arrive at the final output state `o` where `id` is undefined.

```
      (EXISTS v, o' | v IN T /\ MC(c0)(s("id" -> v}, o') /\ o = o'(id -> })
```

Routines

In Spec, routines include functions, atomic procedures, and procedures. For simplicity, we focus on atomic procedures. How do we think about `APROCs`?

We know that the body of an `APROC` describes transitions from its input state to its output state. Given this transition, how do we handle the results? We introduce a pseudo name `$a` to which a procedure's argument value is bound, and the caller also collects the value from `$a` after the procedure body's transition. Refer to the definition of `MR` below for a more complete discussion.

In reality, Spec is more complex because it attempts to make `RET` more convenient by allowing it to occur anywhere in a routine. To accommodate this, the meaning of `RET e` is to set `$a` to the value of `e` and then raise the special exception `$RET`, which is handled as part of the invocation.

Formal atomic semantics of Spec

In the rest of the handout, we describe the meaning of atomic Spec commands in complete detail, except that we do not give precise meanings for the various expression forms other than lambda expressions; for the most part these are drawn from mathematics, and their meanings should be clear. We also omit the detailed semantics of modules, which is complicated and uninteresting.

Overview

The semantics of Spec are defined in three stages: expressions, atomic commands, and non-atomic commands (treated in handout 17 on formal concurrency). For the first two there is no concurrency: expressions and atomic commands are atomic. This makes it possible to give their meanings quite simply:

Expressions as *functions* from states to results, that is, values or exceptions.

Atomic commands as *relations* between states and outcomes: a command relates an initial state to every possible outcome of executing the command in the initial state.

An outcome maps names (treated as strings) to values. It also maps three special strings that are not program names (we call them pseudo-names):

`$a`, which is used to pass argument and result values in an invocation;

`$x`, which records an exceptional outcome;
`$havoc`, which is true if any sequence of later outcomes is possible.

A state is a normal outcome, that is, an outcome which is not exceptional; it has `$x=noX`. The looping outcome of a command is encoded as the exception `$loop`; since this is not an identifier, you can't write it in a handler.

The state is divided into a *global* state that maps variables of the form `m.id` (for which `id` is declared at the top level in module `m`) and a *local* state that maps variables of the form `id` (those whose scope is a `VAR` command or a routine). Routines share only the global state; the ones defined by `LAMBDA` also have an initial local state, while the ones declared in a `routineDecl` start with an empty local state. We leave as an exercise for the reader the explicit construction of the global state from the collection of modules that makes up the program.

We give the meaning of a Spec program using Spec itself, by defining functions `ME`, `MC`, and `MR` that return the meaning of an expression, command, and routine. However, we use only the functional part of Spec. Spec is not ideally suited for this job, but it is serviceable and by using it we avoid introducing a new notation. Also, it is instructive to see how the task is writing this particular kind of specification can be handled in Spec.

You might wonder how this specification is related to an implementation of Spec, that is, to a compiler or interpreter. It does look a lot like an interpreter. As with other specifications written in Spec, however, this one is not a practical implementation because it uses existential quantifiers and other forms of non-determinism too freely. Most of these quantifiers are just there for clarity and could be replaced by explicit computations of the needed values without much difficulty. Unfortunately, the quantifier in the definition of `VAR` does not have this property; it actually requires a search of all the values of the specified type. Since you have already seen that we don't know how to give a practical implementation of Spec, it shouldn't be surprising that this handout doesn't contain one.

Note that before applying these rules to a Spec program, you must apply the syntactic rewriting rules for constructs like `VAR id := e` and `CLASS` that are given in the reference manual. You must also replace all global names with their fully qualified forms, which include the defining module, or `Global` for names declared globally (see section 8 of the reference manual).

Terminology

We begin by giving the types and special values used to represent the Spec program whose meaning is being defined. We use two methods of functions, `+` (overlay) and `restrict`, that are defined in section 9 of the reference manual.


```
TYPE V          =  (Routine + ...)                     % Value
     Routine    =  aTr                                 % defined as the last type below

     Id         =  String                              % Identifer
                   SUCHTHAT (\ id | (EXISTS c: Char, s1: String, s2: String |
                         id = {c} + s1 + s2 /\ c IN letter + digit
                      /\ s1.set <= letter\/digit\/{'_'} /\ s2.set <= {'''} ))
     Name       =  String
                   SUCHTHAT (\ name | name IN    ids \/ globals
                                                \/ {"$a", "$x", "$havoc"})
     X          =  String                              % eXception
                   SUCHTHAT (\ x | x IN ids \/ {noX, retX, loopX, typeX})
     XS         =  SET X                               % eXception Set

     O          =  Name -> V WITH {isX:=OIsX}          % Outcome
     S          =  O SUCHTHAT (\ o | ~ o.isX)          % State
     ATr        =  (S, O) -> Bool                      % Atomic Transition

CONST
     letter     := "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".set
     digit      := "0123456789".set
     ids        := {id | true}
     globals    := {id1, id2 | id1 + "." + id2}
     noX        := ""
     retX       := "$ret"
     loopX      := "$loop"
     typeX      := "$type error"
     trueV      :  V                                   % the value true

FUNC OIsX(o) -> Bool = RET o("$x") # noX               % o.isX
```

To write the meaning functions we need types for the representations of the main non-terminals of the language: `id, name, exceptionSet, type, exp, cmd, routineDecl, module`, and `program`. Rather than giving the detailed representation of these types or a complete set of operations for making and analyzing their values, we write `C« c1 [] c2 »` for a command composed from subcommands `c1` and `c2` with `[]`, and so forth for the rest of the command forms. Similarly we write `E« e1 + e2 »` and `R« FUNC Succ(x: INT)->INT = RET x+1 »` for the indicated expression and function, and so forth for the rest of the expression and routine forms. This notation makes the specification much more readable. `Id`, `Name`, and `XS` are declared above.

```
TYPE T          =  SET V                               % Type
     E          =  [...]                               % Expression
     C          =  [...]                               % Command
     R          =  [id, ...]                           % RoutineDecl
     Mod        =  [id, tops: SET TopLevel]            % Module
     TopLevel   =  (R + ...)                           % module toplevel decl
     Prog       =  [ms: SET Mod, ts: SET TopLevel]     % Program
```

The meaning of an `id` or `var` is just the string, of an `exceptionSet` the set of strings that are the exceptions in the set, of a `type` the set of values of the type. For the other constructs there are meaning functions defined below: `ME` for expressions and `MC` and `MR` for atomic commands and routines. The meaning functions for `module`, `toplevel`, and `program` are left as exercises.

Expressions

An expression maps a state to a value or exception. Evaluating an expression does not change the state. Thus the meaning of expressions is given by a partial function `ME` with type `E->S->(V + X)`; that is, given an expression, `ME` returns a function from states `S` to results (values `V` or exceptions `X`). `ME` is defined informally for all of the expression forms in section 5 of the reference manual. The possible expression forms are literal, variable, and invocation. We give formal definitions only for invocations and `LAMBDA` literals; they are written in terms of the meaning of commands, so we postpone them to the next section

Type checking

For type checking to work we need to ensure that the value of an expression always has the type of the expression. We do this by structural induction, considering each kind of expression. The type checking of return values ensures that the result of an invocation will have its declared type. Literals are trivial, and the only other expression form is a variable. A variable declared with `VAR` is initialized to a value of its type. A formal parameter of a routine is initialized to an actual by an invocation, and the type checking of arguments (see `MR` below) ensures that this is a value of the variable's type. The value of a variable can only be changed by assignment.

An assignment `var := e` requires that the value of `e` have the type of `var`. If the type of `e` is not equal to the type of `var` because it involves a union or a `SUCHTHAT`, this check can't be done statically. To take account of this and to ensure that the meaning of expressions is independent of the static type checking, we assume that in the context `var := e` the expression `e` is replaced by `e AS t`, where `t` is the declared type of `var`. The meaning of `e AS t` in state `s` is `ME(e)(s)` if that is in `t` (the set of values of type `t`), and the exception `typeX` otherwise; this exception can't be handled because it is not named by an identifier and is therefore a fatal error.

We do not give a practical implementation of the type check itself, that is, the check that a value actually is a member of the set of values of a given type. Such an implementation would require too many details about how values are represented. Note that what many people mean by "type checking" is a proof that every expression in a program always has a result of the correct type. This kind of completely static type checking is not possible for Spec; the presence of unions and `SUCHTHAT` makes it undecidable. Sections 4 and 5 of the reference manual define what it means for one type to fit another and for a type to be suitable. These definitions are a sketch of how to implement as much static type checking as Spec easily admits.

Atomic commands

An atomic command relates a state to an outcome; in other words, it is defined by an `ATr` (atomic transition) relation. Thus the meaning of commands is given by a function `MC` with type `C->ATr`, where `ATr = (S, O) -> Bool`. We can define the `ATr` relation for each command by a predicate: a command relates state `s` to outcome `o` iff the predicate on `s` and `o` is true. We give the predicates in the table on the next page and explain them in the text that follows the table; the predicates apply provided there are no exceptions.

Here are the details of how to handle exceptions and how to actually define the `MC` function. You might want to look at the predicates first, since the meat of the semantics is there.


```
Command                 Predicate

SKIP                    o = s
HAVOC                   true

RET e                   o = s{"$x" -> retX, $a -> ME(e)(s)}
RET                     o = s{"$x" -> retX}

RAISE id                o = s{"$x" -> "id"}

e1(e2)                  ( EXISTS r: Routine |
                            r = ME(e1)(s) /\ r(s{"$a" -> ME(e2)(s)}, o) )

var := e        [1]     o = s{var -> ME(e)(s)}
var := e1(e2)   [1]     MC(C« e1(e2); var := $a »)(s, o)

e  =>  c0               ME(e)(s) = trueV /\ MC(c0)(s, o)
c1 []  c2               MC(c1)(s, o) \/     MC(c2)(s, o)
c1 [*] c2               MC(c1)(s, o) \/ (   MC(c2)(s, o)
                                         /\ ~(EXISTS o' | MC(c1)(s, o')) )

c1  ;  c2                                   MC(c1)(s, o)  /\   o .isX
                        \/ ( EXISTS o' |    MC(c1)(s, o') /\ ~ o'.isX
                                         /\ MC(c2)(o',o ) )

c1 EXCEPT xs => c2                          MC(c1)(s, o)  /\ ~ o ("$x") IN xs
                        \/ ( EXISTS o' |    MC(c1)(s, o') /\   o'("$x") IN xs
                                         /\ MC(c2)(o'{"$x" -> noX}, o) )

VAR id: T | c0          ( EXISTS v, o' |   v IN T
                                        /\ MC(c0)(s {id -> v}, o')
                                        /\ o =    o'{id ->  } )

VAR id: T := e | c0     MC(C«VAR id: T | id = e => c0»)(s, o)

<< c0 >>                MC(c0)(s, o)
IF c0 FI                MC(c0)(s, o)
BEGIN c0 END            MC(c0)(s, o)
```

`DO c0 OD` is the fixed point of the equation `c = c0; c [*] SKIP`

[1] The first case for `assignment` applies only if the right side is not an `invocation` of an `APROC`. Because an invocation of an `APROC` can have side effects, it needs different treatment.

Table 1: The predicates that define `MC(command)(s, o)` when there are no exceptions raised by expressions at the top level in `command`, and `$havoc` is false.

The table of predicates has been simplified by omitting the boilerplate needed to take account of `$havoc` and of the possibility that an expression is undefined or yields an exception. If a command containing expressions `e1` and `e2` has predicate `P` in the table, the full predicate for the command is

```
    s("$havoc")                                  % anything if $havoc
\/     ME(e1)!s /\ ME(e2)!s                      % no outcome if undefined
    /\ (   ME(e1)(s) IS V /\ ME(e2)(s) IS V /\ P
        \/ ME(e1)(s) IS X /\ o = s{ "$x" -> ME(e1)(s) }
        \/ ME(e2)(s) IS X /\ o = s{ "$x" -> ME(e2)(s) } )
```

If the command contains only one expression `e1`, drop the terms containing `e2`. If it contains no expressions, the full predicate is just the predicate in the table.

Once we have the full predicates, it is simple to give the definition of the function `MC`. It has the form

```
FUNC MC(c) -> ATr =
     IF
     ...
     []   VAR var, e | c = «var := e» =>
          RET (\ o, s | full predicate for this case )
     ...
     []   VAR c1, c2 | c = «c1  ; c2» =>
          RET (\ o, s | full predicate for this case )
     ...
     FI
```

First we do the simple commands, which don't have subcommands. All of these that don't involve an invocation of an `APROC` are deterministic; in other words, the relation is a function. Furthermore, they are all total unless they involve an invocation that is partial.

A `RET` produces the exception `retX` and leaves the returned value in `$a`.

A `RAISE` yields an exceptional outcome which records the exception `id` in `$x`.

An `invocation` relates `s` to `o` iff the routine which is the value of `e1` (produced by `ME(e1)(s)`) does so after `s` is modified to bind `"$a"` to the actual argument; thus `$a` is used to communicate the value of the actual to the routine.

An `assignment` leaves the state unchanged except for the variable denoted by the left side, which gets the value denoted by the right side. Recall that assignment to a component of a function, sequence, or record variable is shorthand for assignment of a suitable constructor to the entire variable, as described in the reference manual. If the right side is an `invocation` of a procedure, the value assigned is the value of `$a` in the outcome of the invocation; thus `$a` also communicates the result of the invocation back to the invoker.

Now for the compound commands; their meaning is defined in terms of the meaning of their subcommands.

A guarded command `e => c` has the same meaning as `c` except that `e` must be true.

A choice relates `s` to `o` if either part does.

An else `c1 [*] c2` relates `s` to `o` if `c1` does or if `c1` has no outcome and `c2` does.

A sequential composition `c1 ; c2` relates `s` to `o` if there is a suitable intermediate state, or if `o` is an exceptional outcome of `c1`.

`c1 EXCEPT xs=>c2` is the same as `c1` for a normal outcome or an exceptional outcome not in the exception set `xs`. For an exceptional outcome `o'` in `xs`, `c2` must relate `o'` as a normal state to `o`. This is the dual of the meaning of `c1 ; c2` if `xs` includes all exceptions.

`VAR id: t | c` relates `s` to `o` if there is a value `v` of type `t` such that `c` relates (`s` with `id` bound to `v`) to an `o'` which is the same as `o` except that `id` is undefined in `o`. It is this existential quantifier that makes the specification useless as an interpreter for Spec.

`<< ... >>`, `IF ... FI` or `BEGIN ... END` brackets don't affect `MC`.

The meaning of `DO c OD` can't be given so easily. It is the fixed point of the sequence of longer and longer repetitions of `c`.[2] It is possible for `DO c OD` to loop indefinitely; in this case it relates `s` to `s` with `"$x"->loopX`. This is not the same as relating `s` to no outcome, as `false => SKIP` does.

The multiple occurrences of `declInit` and `var` in `VAR declInit*` and `(varList):=exp` are left as boring exercises, along with routines that have several formals.

Routines

Now for the meaning of a routine. We define a meaning function `MR` for a `routineDecl` that relates the meaning of the routine to the meaning of the routine's body; since the body is a command, we can get its meaning from `MC`. The idea is that the meaning of the routine should be a relation of states to outcomes just like the meaning of a command. In this relation, the pseudo-name `$a` holds the argument in the initial state and the result in the outcome. For technical reasons, however, we define `MR` to yield not an `ATr`, but an `S->ATr`; a local state (`static` below) must be supplied to get the transition relation for the routine. For a `LAMBDA` this local state is the current state of its containing command. For a routine declared at top level in a module this state is empty.

The `MR` function works in the obvious way:

1. Check that the argument value in `$a` has the type of the formal.
2. Remove local names from the state, since a routine shares only global state with its invoker.
3. Bind the value to the formal.

[2] For the details of this construction see G. Nelson, A generalization of Dijkstra's calculus, *ACM Trans. Programming Languages and Systems* **11**, 4, Oct. 1989, pp 517-562.

4. Find out using MC how the routine body relates the resulting state to an outcome.
5. Make the invoker's outcome from the invoker's local state and the routine's final global state.
6. Deal with the various exceptions in that outcome.

   A `retX` outcome results in a normal outcome for the invocation if the result has the result type of the routine, and a `typeX` outcome otherwise.

   A normal outcome is converted to `typeX`, a type error, since the routine didn't supply a result of the correct type.

   An exception raised in the body is passed on.

```
FUNC MR(r) -> (S->ATr) = VAR id1, id2, t1, t2, xs, c0 |
        r = R« APROC id1(id2: t1)->t2 RAISES xs = << c0 >> »
     \/ r = R« FUNC  id1(id2: t1)->t2 RAISES xs =    c0    » =>
     RET (\ static: S | (\ s, o |
          s("$a") IN t1                               % if argument typechecks
       /\ ( EXISTS g: S, s', o' |
              g = s.restrict(globals)                 % g is the current globals
           /\ s' = (static + g){id2 -> s("$a")}       % s' is initial state for c0
           /\ MC(c0)(s', o' )                         % apply c0
           /\ o = (s + o'.restrict(globals))          % restore old locals from s
                {"$x" ->                              % adjust $x in the outcome
                  (    o'("$x") = retX =>
                         (    o'("$a") IN t2 => noX   % retX means normal outcome
                          [*] typeX )                 % if result typechecks;
                   [*] o'("$x") = noX => typeX        % normal outcome means typeX;
                   [*] o'("$x)                        % pass on exceptions
                  )
                }
     \/ ~ s("$a") IN t1 /\ o = s{"$x" -> typeX}       % argument doesn't typecheck
     ) )                                              % end of the two lambdas
```

We leave the meaning of a routine with no result as an exercise.

Invocation and LAMBDA expressions

We have already given in MC the meaning of invocations in commands, so we can use MC to deal with invocations in expressions. Here is the fragment of the definition of ME that deals with an E that is an invocation `e1(e2)` of a function. It is written in terms of the meaning `MC(C«e1(e2)»)` of the invocation as a command, which is defined above. The meaning of the command is an atomic transition `aTr`, a predicate on an initial state and an outcome of the routine. In the outcome the value of the pseudo-name `$a` is the value returned by the function. The definition given here discards any side-effects of the function; in fact, in a legal Spec program there can be no side-effects, since functions are not allowed to assign to non-local variables or call procedures.


```
FUNC ME(e) -> (S -> (V + X)) =
     IF
     ...
     [] VAR e1, e2 | e = E« e1(e2) » =>
     % if E is an invocation its meaning is this function from states to values
          VAR aTr := MC(C« e1(e2) ») |
            RET ( LAMBDA (s) -> V =
               % the command must have a unique outcome, that is, aTr must be a
               % function at s. See Relation in section 9 of the reference manual
               VAR o := aTr.func(s) | RET (~o.isX => o("$a") [*] o("$x")) )
     ...
     FI
```

The result of the expression is the value of `$a` in the outcome if it is normal, the value of `$x` if it is exceptional. If the invocation has no outcome or more than one outcome, `ME(e)(s)` is undefined.

The fragment of ME for LAMBDA uses MR to get the meaning of a FUNC with the same signature and body. As we explained earlier, this meaning is a function from a state to a transition function, and it is the value of `ME((LAMBDA ...))`. The value of `(LAMBDA ...)`, like the value of any expression, is the result of evaluating `ME((LAMBDA ...))` on the current state. This yields a transition function as we expect, and that function captures the local state of the LAMBDA expression; this is standard static scoping. .

```
     IF
     ...
     [] VAR signature, c0 | e = E« (LAMBDA signature = c0) » =>
          RET MR(R« FUNC id1 signature = c0 »)
     ...
     FI
```