Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 12 September 27, 1993

History Variables

This handout contains examples that illustrate the limitations of abstraction functions as a tool for proving the correctness of module implementations, and shows how to solve the problems using history variables.

1 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.

```
MODULE StatDB [ V WITH {
                  zero: () -> V
                  "+": (V,V) -> V,
                  "*": (V,V) -> V,
                  "-": (V,V) -> V,
                  "/": (V,INT) -> V
                } ] =

% Initialize before calling any of the routines below.
VAR db := SEQ[V]{}

PROC add (v) =
  db := db ++ v

PROC size () -> INT =
  RET db.size

PROC mean () -> V RAISES empty =
  VAR sum: V, count: INT |
  sum := zero();
  count := 0;
```

```
  do
    count < db.size =>
       sum := sum + db[count];
       count := count + 1
  od;
  count = 0 => RAISE empty
  [*] RET sum/count

PROC variance () -> V RAISES empty =
  VAR avg: V, sumSquare: V, count: INT |
  avg := mean() EXCEPT empty => RAISE empty END;
  sumSquare := zero();
  count := 0;
  do
    count < db.size =>
       sumSquare := sumSquare + square(db[count]-avg));
       count := count +1
  od;
  count = 1 => RET zero()
  [*] RET sumSquare/(count-1)

FUNC square (v) -> V =
  RET v*v

END StatDB
```

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. (A similar trick is used by hand calculators.)

```
MODULE StatDBImpl [V WITH {
                   zero: () -> V
                   "+": (V,V) -> V,
                   "*": (V,V) -> V,
                   "-": (V,V) -> V,
                   "/": (V,INT) -> V
                 } ] =

% Initialize before calling any of the routines below.
VAR count: INT := 0
    sum: V := zero()
    sumSquare: V := zero()
```

```
PROC add (v) =
  count := count + 1;
  sum := sum + v;
  sumSquare := sumSquare + square(v)

PROC size () -> INT =
  RET count

PROC mean () -> V RAISES empty =
  count = 0 => RAISE empty
  [*] RET sum/count

PROC variance () -> V RAISES empty =
  count = 0 => RAISE empty
  [*] count = 1 => RET zero()
  [*] RET (sumSquare - count*square(mean()))/(count-1)

FUNC square (v) -> V =
  RET v*v

END StatDBImpl
```

Now consider proving this implementation correct. There is no way to define an abstraction function, since for each state of the implementation there may be many corresponding states of the specification. In this case, this problem arises because the specification contains more states than necessary to specify the module. The states of the specification can be partitioned into equivalence classes based on the future behavior of the module when started in each state — two states are equivalent if they have the same future behavior — and the equivalence classes here contain more than one state. This will happen whenever the state of the specification contains information that cannot be observed by calling the operations.

2 History Variables

There are two ways to get around this problem. The first is to use a "multi-valued" abstraction mapping — i.e., an abstraction relation rather than an abstraction function. This means that we define a relation AF relating states of the implementation to states of the specification. For this example, AF would relate a state (c, s, sSq) of the implementation to any sequence db such that db has size c, the sum of elements of db is s, and the sum of the squares of the elements of db is sSq. We won't discuss the details of this approach here; if you're interested, there are some papers by Nancy Lynch and her colleagues on abstraction mappings that discuss multi-valued mappings in detail (MIT/LCS/TM-422 and MIT/LCS/TM-458).

The second solution technique is to use “history variables” in the implementation. The problem in the example above is that the abstract states contain more information than the concrete states. A multi-valued abstraction mapping gets around this by mapping each concrete state to all the possible abstract states that correspond. (If you think about an execution that leaves the implementation in state c, this is all the possible states of the specification after such an execution.) Another way to solve this problem that allows us to continue using single-valued abstraction mappings (i.e., abstraction *functions*) is to augment the implementation with additional state that keeps track of the additional information from the specification that was left out of the implementation. Even though the implementation has been highly 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 doesn't change the behavior of the implementation.

For example, suppose we add an extra state component db to the implementation of the statistical database given earlier, and use it to keep track of the entire collection of elements. This gives the following module:

```
MODULE StatDBImpl [V WITH {
                   zero: () -> V
                   "+": (V,V) -> V,
                   "*": (V,V) -> V,
                   "-": (V,V) -> V,
                   "/": (V,INT) -> V
                 } ] =

% Initialize before calling any of the routines below.
VAR count: INT := 0
    sum: V := zero()
    sumSquare: V := zero()
    db: SEQ[V] := SEQ[V]{}

PROC add (v) =
  count := count + 1;
  sum := sum + v;
  sumSquare := sumSquare + square(v);
  db := db ++ v

% The remaining procedures are as before

END StatDBImpl
```

All we have done here is record some additional information in the state. However, we have not changed the way existing state components are initialized or updated, or the way results of procedures are computed. Thus, it should be clear that this module exhibits all the possible behaviors of the implementation given earlier. Now, if we can prove this module correct, then the correctness of the implementation given earlier will follow immediately.

But proving this module correct using an abstraction function is easy. The abstraction function simply discards all components of the state *except* db. The rep invariant describes how db is related to the other state components:

```
   db.size = size
/\ the sum of the elements of db = sum
/\ the sum of the squares of the elements of db = sumSquare
```

Given this invariant and abstraction function, it is easy to show that this module implements the specification given earlier.

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:

- A value assigned to an existing state component must not depend on the value of a history variable.
- The result of an operation must not depend on a history variable.

These constraints guarantee that the history variables simply record additional state information, but 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.

In this example, I 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, throwing in the entire state from the specification as a history variable will work whenever any addition of history variables will work.

Any implementation that can be proved correct using history variables can also be proved correct using a multi-valued abstraction mapping, 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 multi-valued mappings because they avoid the need to introduce extra state components or to explain how and when those components are updated. The appropriate choice really depends on the particular application involved and on personal taste.

3 Nondeterministic Specifications

This section contains a somewhat contrived but very simple example that illustrates how a nondeterministic specification can require the use of history variables (or multi-valued mappings) to prove the correctness of an implementation.

Here is the specification:

```
MODULE Strange =

TYPE S = ENUM[a,b,c,d]

VAR s := a

PROC Current () -> S =
  s=b \/ s=c => RET d
  [] RET s

PROC B () =
  s=a => s:=b
  [*] SKIP

PROC C () =
  s=a => s:=c
  [*] SKIP

END Strange
```

The module has three possible states, a, b, and c. The procedures B and C take it from its initial state a to b and c, respectively; in other states they leave the state unchanged. The procedure *Current* either returns the value of the current state or, in states b and c, can return the value d. Thus, it is nondeterministic. The implementation below makes a particular deterministic choice in *Current*; as a result, it does not need to distinguish between states b and c.

```
MODULE StrangeImpl =

TYPE S = ENUM[a,b,c,d]
TYPE SI = ENUM[a,bc]

VAR si := a

PROC Current () -> S =
  si=a => RET a
  [*] RET d

PROC B () =
  si:=bc

PROC C () =
  si:=bc
```

```
END StrangeImpl
```

This implementation is correct, yet there is no abstraction function that can be used to prove it correct. However, we can easily define an abstraction relation or use history variables.