Explicit State

Annalise Tarhan

July 1, 2020

1 The Importance of Sequences

1.1 Compare and contrast the definition of state with the given definition of comics. Are we interested in the whole sequence or just the final result? Does the sequence exist in space or time? Is the transition between sequence elements important?

State - a sequence of values in time that contains the intermediate results of a desired computation
Comics - juxtaposed pictorial and other images in deliberate sequence, intended to convey information and/or to produce an aesthetic response in the viewer

Both definitions emphasize the importance of the entire sequence, as opposed to the final result, but the role of the sequence in both is to provide context for the final result. If everything is running smoothly, there is usually no need to examine the intermediate results of a computation. If there is a bug, though, being able to see the intermediate results could be critical for fixing it. With comics, the entire sequence must be examined for the final pane to have its desired impact.
For both computations and comics, there seem to be two actors. First, the one who engages with the entire sequence in order to either create or understand the final result. Second, the one who makes use of the final result. With comics, both actors are the same, the viewer. With computations, the two actors could both be the programmer in the case of debugging, both be the computer, in the case of normal use, where the result is used by the computer to calculate something else, or the computer could be the first and the programmer the second, if the final result is displayed for the user.
Because the intermediate results of a calculation are overwritten, state exists as a sequence in time. Comics, obviously, exist as a sequence in space.
The transition between sequence elements is crucial, since it embodies the calculation that is the function of the program. With comics, the transition could be more or less important depending on the intended impact.

2 State with Cells

2.1 Rewrite SumList so that the state is no longer encoded in arguments, but by cells.

```
local
      Total={NewCell 0}
in
      fun {SumList Xs}
            case Xs
            of nil then @Total
            [] X|Xr then
                  Total:=@Total+X
                  {SumList Xr}
            end
      end
end
```

3 Emulating State with Concurrency

3.1 Rewrite SumList to use the given MakeState container to count the number of calls.

```
local
      C0={MakeState 0}
in
      fun {SumList Xs S C}
            case Xs
            of nil then Final in
                  C=access(Final)|nil
                  {Browse Final}
                  S
            [] X|Xr then C1 Calls in
                  C=access(Calls)|assign(Calls+1)|C1
                  {SumList Xr X+S C1}
            end
      end
      {Browse {SumList [1 2 3 4 5 6] 0 C0}}
end
```

3.2 Can the container be encapsulated, i.e. added without changing the arguments of SumList? Why or why not?

No. The container is based on a list with an unbound tail, which can only be bound once. Commands work by binding the tail to a list of commands with a new unbound tail. The reference to the new tail must be passed as an argument for the container to continue to be useful.

3.3 What happens when we try to add the function SumCount?

The purpose of SumCount is to make the current number of calls observable. Trying to add it fails for the same reason the container cannot be encapsulated: observing the current value necessitates binding the tail, and if something besides the main function does that, the main function will lose the reference to the tail and be unable to proceed.

4 Implementing Ports

4.1 Implement ports in terms of cells.

```
proc {NewPort S P}
     P={NewCell S}
end
proc {Send P X}
     P:=X|@P
end
```

5 Explicit State and Security

5.1 Explicit state seems to have no role with respect to security. Is that true? Why or why not?

Stateful-declarative and open-secure are two of the three axes which define eight ways to design an ADT. It is possible to design ADTs that are secure and stateful, secure and stateless, open and stateful, and open and stateless. However, this doesn't mean state can't play a role in security. Revocable capabilities are an example of a data abstraction based on explicit state that can control the security of other ADTs.

6 Declarative Objects and Identity

6.1 Expand the given declarative objects to have an identity.

```
...
stack(name:{NewName} push:Push pop:Pop isEmpty:IsEmpty)
...
```

The only thing that needs to be added is a name field in the record that stores the stack's procedure values.

7 Revocable Capabilities

7.1 Write a version of Revocable that is a one-argument procedure and where the revoker is also a one-argument procedure.

```
proc {Revocable ?Revoker}
     C={NewCell active}
in
     Revoker=proc {$ M}
          case @C
          of revoked then raise revokedError end
          [] active then
               case M
               of revoke then C:=revoked else {M}
               end
          end
     end
end
```

8 Abstractions and Memory Management

8.1 Skipped

9 Call by Name

9.1 Explain the behavior of the swap procedure.

Call by name works by passing a function that returns the name of the cell that contains the value, not the cell or value itself. The problem is that the contents of the cell can change between the time the argument is passed and the time the cell is accessed. In this case, the arguments are functions that return i and a[i]. i is changed first, from 1 to 2, then a[i] is changed. The problem is that by the time a[i] is reassigned, i now refers to 2, not 1. So instead of switching i and a[1], it switches i and a[2].

10 Call by Need

10.1 If the previous exercise was call by need instead of call by name, would the counterintuitive behavior still occur? Can similar problems still occur with call by need by changing the definition of swap?

Call by need means the arguments are only evaluated once, the first time they are needed. With the given definition of swap, it would behave as expected. For the behavior to become counterintuitive again, the definition of swap would have to be changed such that the value of i was changed before a[i] was evaluated.

10.2 Modify the Sqr function so that it uses laziness to call A only when needed.

```
proc {Sqr A}
     B={ByNeed A}
in
     B:=@B*@B
end
```

11 Evaluating Indexed Collections

11.1 Compare the four indexed collection types: tuples, records, arrays, and dictionaries, in various usage scenarios. Evaluate their relative performance and usefulness.

Arrays are rigid in the sense that their domain must be represented as consecutive integers and the size must be determined when the array is created. However, elements of the range can be changed. An example would be a representation of all the tables in a restaurant and their current occupants' orders.
Dictionaries are the most flexible option. The size is not fixed and the domain can be any simple constant, not just consecutive integers. They require more memory than any of the other data types and have slower access times. Continuing on the restaurant theme, a dictionary would be a good choice to represent menu items and descriptions.
Tuples are very restrictive, since their contents are determined at creation. Indices are consecutive integers beginning at one. Because they are so inflexible, they are the fastest option and require the least amount of memory. A tuple might be used to store a list of unchanging health and safety rules.
Records are like dictionaries in the sense that elements in their domain can be any simple constant, but the type must be the same for all elements. They are similar to tuples in that all their elements must be determined when the record is created. A record could be used to store staff names and id numbers, at least until someone is fired or hired.

12 Extensible Arrays

12.1 Modify the extensible array so it extends the array in both directions.

```
if I>High then
    High2=Low+{Max I 2*(High-Low)}
    Low2=Low-{Max I 2*(High-Low)}
    Arr2={NewArray Low2 High2 A.2}
```

13 Generalized Dictionaries

13.1 Implement a dictionary that can use any value as a key.

```
fun {NewDictionary}
   {NewCell nil}
end

fun {PutList List Key Value}
   case List
   of nil then Key#Value
   [] K#_|Rest andthen K==Key then
      Key#Value|Rest
   [] K#V|Rest then
      K#V|{PutList Rest Key Value}
   end
end
fun {Put Dictionary Key Value}
   List=@Dictionary
   Dictionary:={PutList List Key Value}
end

fun {GetList List Key Default}
   case List
   of nil then Default
   [] K#V|_ andthen K==Key then V
   [] _#_|Rest then {GetList Rest Key Default}
   end
end
fun {Get Dictionary Key Default}
   List=@Dictionary
   {GetList List Key Default}
end

fun {ListDomain List}
   case List
   of nil then nil
   [] K#_|Rest then K|{ListDomain Rest}
   end
end
fun {Domain Dictionary}
   List=@Dictionary
   {ListDomain List}
end
```

14 Loops and Invariant Assertions

14.1 Use the method of invariant assertions to show that the proof rule for while loops is correct.

Proof Rule:
If {P ∧ <expr>} <stmt> {P}
Then {P} while <expr> do <stmt> end {P∧¬ <expr>}

If {P ∧ <expr>=true} <stmt> {P}
and {P ∧ <expr>=false} skip {P}
then by proof rule for if,
{P} if <expr> then <stmt> else skip end {P}

Let $<stmt>_1$ = if <expr> then <stmt> $<stmt>_1$ else skip end
Use induction to show {P} $<stmt>_1$ {P}

Base Case for <expr>:
{P} if <expr> then <stmt> $<stmt>_1$ else skip end ...
{P} if <expr> then <stmt> if <expr> then <stmt> ... {P}
Base Case for ¬ <expr>:
{P} if <expr> then <stmt> $<stmt>_1$ else skip end ...
{P} if <expr> then <stmt> if <expr> ... else skip end {P}

Inductive Step for <expr>:
{P} ... if <expr> then <stmt> if <expr> then <stmt> ... {P}
Inductive Step for ¬ <expr>:
{P} ... if <expr> ... else skip end {P}
Therefore {P} $<stmt>_1$ {P}

This section only deals with programs that terminate normally, so <expr> must eventually be false to avoid an infinite loop.
{P ∧ <expr> = false} skip {P ∧ ¬ <expr>}
{P} $<stmt>_1$ $<stmt>_1$ ... $<stmt>_1$ {P ∧ ¬ <expr>}

Since $<stmt>_1$ is defined as if <expr> then <stmt> $<stmt>_1$ else skip end,
which is also the definition of while <expr> do <stmt> end,
this shows that {P} while <expr> do <stmt> end {P∧¬ <expr>}.

14.2 Use the method of invariant assertions to show that the proof rule for for loops is correct.

Proof Rule:
If $\forall$ i.<y>$\leq$ i $\leq$<z> : $\{P_{i-1} \wedge$ <x> = i$\}$ <stmt> $\{P_i\}$
Then $\{P_{<y>-1}\}$ for <x> in <y>..<z> do <stmt> end $\{P_{<z>}\}$

Let <stmt>$_1$ = if <y>$\leq$<z> then <stmt><stmt>$_1$ else skip end.
Use induction to show $\{P_{<y>-1}\}$ <stmt>$_1$ $\{P_{<z>}\}$ for <y> $\leq$ <x> $\leq$ <z>.

Base Case:
<x>=<y>=i: $\{P_{y-1} \wedge$ <x> = i$\}$ if <x>$\leq$<z> then <stmt> ... $\{P_i\}$ as given by the proof rule.

Inductive Step:
<y>$\leq$<x> = i <<z> $\{P_{i-1}\}$ if <x>$\leq$<z> then <stmt> ... $\{P_i\}$ also by the given.

Final Step:
<x>=i=<z>+1 $\{P_{i-1}\}$ if <x>$\leq$<z> ... else skip end $\{P_{<z>}\}$ because $P_{<z>}$=$P_{<x>-1}$=P_{i-1}.

Therefore $\{P_{<y>-1}\}$ <stmt>$_1$ $\{P_{<z>}\}$ for <y> $\leq$ <x> $\leq$ <z>.
Since <stmt>$_1$ is defined as if <y>$\leq$<z> then <stmt><stmt>$_1$ else skip end, which is also the definition of for <x> in <y>..<z> do <stmt> end, this shows that $\{P_{<y>-1}\}$ for <x> in <y>..<z> do <stmt> end $\{P_{<z>}\}$.

15 The Break Statement

15.1 Define a block construct with a break operation called using {Block proc {$ Break} <stmt> end}.

```
proc {Block Proc}
     try {Proc Break}
     catch breakPoint then skip end
end

proc {Break}
     raise breakPoint end
end
```