Abstract State Machines

An Introduction
(Definitions aplenty)

Why

- Mmm.. Fun! (April Fools’)
- Bridging the Gap between Specifying what an algorithm should do and the Model of that algorithm (what it does in practice)
- Describing an algorithm fully without the need to adhere to a certain implementation of a programming language, machine architecture, protocol, etc,.

No really, Why?

- To describe/simulate the algorithm in its <u>abstraction level</u>
- “The model can be recognized "by inspection" and thus justified as a faithful adequate precise representative of the system.” -- Egon BÄorger, “Logic Programming: The Evolving Algebra Approach“
- “scientifically sound warranty criteria for safety critical software systems” – same

What we need, What we'll use

- We need cement: we'll use functions, constants will be nullary functions.
- We'll begin with basic nullary functions: ***true***, ***false*** and ***undef.***
- We will also need =, <, > and other basic Boolean operators.
- These form the basic *vocabulary*, which can be extended to our needs.

- We need Types for our functions, we’ll use ***superuniverses.***
- We need to *interpret* the function names as functions.
- The *superuniverse*, and the *interpretations of function names* form a ***State.***
- A state has a vocabulary, which it interprets.
- A state allows us to describe a “snapshot” in our algorithm’s execution

We need Predicates

- We'll use relations.
- A relation is also a function → that maps elements (or a tuple of) to {true, false}
- A ***universe*** (not to be confused with a superuniverse) contains all tuples which the relation evaluates as ***true***

We need stuff

- We'll use terms:
  - A variable is a term
  - An evaluation of a function over terms – is a term.

Roadmap for assignments (for well-definedness)

- Let S be a State
- Fun(S) is the collection of functions in state S.
- An Appropriate State S for an object o, is one that maintains that Fun(S) includes the collection of function names that occur in o.

Static vs. Dynamic (basic functions)

- Right hand side, Left Hand side.
- Dynamic functions appear in function updates as subjects for updating.
- Static functions do not change during the evolution of the algebra.

We need **<var>** := <value>

- We’ll use locations to store values
  A location is a pair l = (f,x), where f in a dynamic function, and x is a tuple with arity matching to that of f
- **Important to say**: it is f(x) which we want to change, not f, nor x.

We need Assignments

- We'll use update rules to make the assignments: An update is a pair α = (l,y) where l is a location and y belongs to the superuniverse of the State in which the update is made. (y is a possible value of l)
- Consistency: we call a Set of updates (family) consistent if there is only one matching value, and one location for each update in the whole set.
- Inconsistency – Do nothing. (Or use nondeterminism)

Now we're ready

- To actually do something:
  (S is a State, R is a sequence of rules):
- Sequence:
  Updates(R,S) = Updates(R1,S)U…U Updates(Rk,S)
- Parrallism built in, all updates are done at the same time.
- Conditionals:
  (g0-gk Boolean terms):
  If g0 then R0
  elseif g1 then R1
  (…)
  elseif gk then RK
  endif

Vending Machine – 1
Buy a can of soda with exact change

- Selection: [9,9] **Product: null** Price: 0 Display: *E_InvalidCode*
- Credit : {5->0,10->0,25->0,100->0,500->0} Credit Amount: 0
- Reserve: {5->2,10->2,25->2,100->2,500->1} Reserve Amount,R: 780
- Change : {5->0,10->0,25->0,100->0,500->0} Stock Amount,S: 560
- Stock : {E_Soda->2,E_Candy->1,E_Chips->1,E_Sandwich->1} S+R: 1340

- **Input: KeyInput(key=A)**
- **Selection: [9,A]** Product: null Price: 0 *Display: E_InvalidCode*
- Credit : {5->0,10->0,25->0,100->0,500->0} Credit Amount: 0
- Reserve: {5->2,10->2,25->2,100->2,500->1} Reserve Amount,R: 780
- Change : {5->0,10->0,25->0,100->0,500->0} Stock Amount,S: 560
- Stock : {E_Soda->2,E_Candy->1,E_Chips->1,E_Sandwich->1} S+R: 1340

Vending Machine - 2

- **Input: KeyInput(key=1)**
- **Selection: [A,1] Product: E_Soda** Price: 60 Display: *E_InsertCoins*
- Credit : {5->0,10->0,25->0,100->0,500->0} Credit Amount: 0
- Reserve: {5->2,10->2,25->2,100->2,500->1} Reserve Amount,R: 780
- Change : {5->0,10->-1,25->-2,100->0,500->0} Stock Amount,S: 560
- Stock : {E_Soda->2,E_Candy->1,E_Chips->1,E_Sandwich->1} S+R: 1340

- **Input: CoinInput(coin=25)**
- Selection: [A,1] Product: E_Soda Price: 60 Display: *E_InsertCoins*
- **Credit : {5->0,10->0,25->1,100->0,500->0} Credit Amount: 25**
- Reserve: {5->2,10->2,25->2,100->2,500->1} Reserve Amount,R: 780
- Change : {5->0,10->-1,25->-1,100->0,500->0} Stock Amount,S: 560
- Stock : {E_Soda->2,E_Candy->1,E_Chips->1,E_Sandwich->1} S+R: 1340

Vending Machine - 3

- **Input: CoinInput(coin=25)**
- Selection: [A,1] Product: E_Soda Price: 60 Display: *E_InsertCoins*
- **Credit : {5->0,10->0,25->2,100->0,500->0} Credit Amount: 50**
- Reserve: {5->2,10->2,25->2,100->2,500->1} Reserve Amount,R: 780
- Change : {5->0,10->-1,25->0,100->0,500->0} Stock Amount,S: 560
- Stock : {E_Soda->2,E_Candy->1,E_Chips->1,E_Sandwich->1} S+R: 1340

- **Input: CoinInput(coin=10)**
- Selection: [A,1] Product: E_Soda Price: 60 Display: *E_InsertCoins*
- Credit : {5->0,10->0,25->0,100->0,500->0} Credit Amount: 0
- Reserve: {5->2,10->3,25->4,100->2,500->1} Reserve Amount,R: 840
- Change : {5->0,10->-1,25->-2,100->0,500->0} Stock Amount,S: 500
- Stock : {E_Soda->1,E_Candy->1,E_Chips->1,E_Sandwich->1} S+R: 1340

We Want More

- We want something new.
- In the factory design pattern, you call a method and receive a new instance of a certain class
- We’ll use Reserve – an unlimited collection of new objects, each time we want something new, we’ll import from the Reserve, and then use.

Import from Reserve Example

- import v
- Parent(v):=CurrentNode
- endimport
- import v
- Parent(v):=CurrentNode
- Endimport
- <u>2 separate nodes with the same parent</u>
- To throw away unneeded objects, set U(x):= false (x is no longer in universe U)

So What do we run?

- We run a program – which is a Rule (or a sequence of rules) without free variables:
- (If we leave out the import v line in the last example, then v in
  “parent(v) := CurrentNode” is free)
- A run is an ordered set of States, each of them is the result of the former State, being fired upon with the corresponding Rule.

We need a window

- To see what's out there.
- External Functions act as user input or as a non-deterministic function (maybe an Oracle)
- We must require that whatever value the function outputs is consistent along the run.
- UserInput(Sensor) = 8, all along the run
- Can also add describe UserInput (Sentor, at t= t0) = x0

We don’t know what we need

- But we’ll choose something
- Nondeterminism can be achieved with choosing some element from a certain Universe
- Choose v in Nodes
  Some Rules with v
  End choose
- Each run v can be a different node from the graph.
- Can’t choose from Reserve, because in Reserve the element is <u>different, and also generic.</u>

We want to say less, do more

- If we want to do some work in parallel, using only one Rule:
- Use variables defined in the following manner:
  Var C ranges over Nodes
- if Color(C) = green and x is a child of C then
  Color(x) := red
  endif
- All child nodes of green colored nodes will be colored red at the same time, in a single step
- Conflicts? Discrepancies? Either fix or use nondeterminism to decide which will be executed.

We want Help

- Why should we do all the work?
- Use distributed Calculations
- We’ll have modules (threads)
- We’ll have agents (processors)
- We’ll have the function SELF to act out as thread id.
- We’ll associate modules to agents to tell who’s doing what
- We’ll all start from a shared initial state
- Each agent acting out a module has its own vocabulary (private memory), a local State
- We can import more agents (fork)
- If an agent spawned n other agents, he can coordinate between them to do team work – he sees all of their vocabularies.

Questions?