Capturing Functions and Catching Satellites

Andy Gill and Garrin Kimmell

Information Technology and Telecommunication Center
Department of Electrical Engineering and Computer Science
The University of Kansas
2335 Irving Hill Road
Lawrence, KS 66045
{andygill,kimmell}@ku.edu

Abstract. The 2009 ICFP programming contest problem required contestants to control virtual satellites that obey basic physical laws. The orbital physics behavior of the system was simulated via a binary provided to contestants which ran on top of a simple virtual machine. Contestants were required to implement the virtual machine along with a controller program to manipulate the satellite's behavior. In this paper, we describe the modeling of the simulation environment, with a focus on the compilation and testing infrastructure for the generated binaries for this virtual machine. This infrastructure makes novel use of an implementation of a deeply embedded Domain Specific Language (DSL) within Haskell. In particular, with use of IO-based observable sharing, it was straightforward for a function to be both an executable specification as well as a portable implementation.

1 Introduction

Organizing the ICFP contest presented a challenge for the students and faculty at the University of Kansas. As the Computer System Design Laboratory, we wanted to set a challenging controller based problem, but how do we provide an interesting simulation environment for this controller that could be used with any possible computer language, and on any possible system? The environment must be interactive (take input, generate output), and perhaps contain and encode hidden challenges and puzzles. The architecture we chose was to provide a binary that encoded the executable specification of a model behavior, and require contestants to write a small virtual machine for this binary.

This solution raised the issue of how we should write our implementation of the simulation, and how we should generate the reference binary. Rather than write an assembler or compile from a high level language, we chose to experiment with compiling a high-level *specification*, via a small custom Domain Specific Language (DSL) written for this specific problem. In particular, we used the recently developed IO-based observable sharing [6] and aggressive use of Haskell overloading to enable the sharing of a purely functional model of satellite behavior for both testing and code generation.

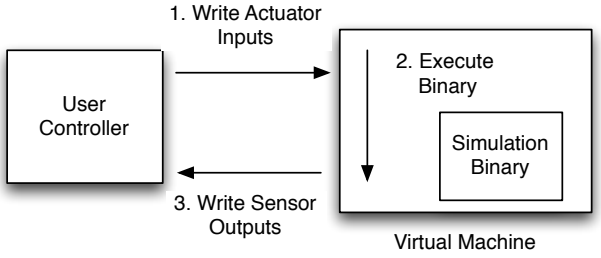

Fig. 1. Contest Architecture

We believe that IO-based observable sharing significantly reduces the cost of writing these small compilers, shrinking the gap between specifications and implementations. This paper makes the following contributions.

- We show that it is remarkably straightforward to generate cross-compiling DSLs in Haskell using observable sharing, by way of an extended real world example. This template can be copied by others when engineering DSLs.
- The example DSL system is demonstrated to be usable in two specific modes, both for straightforward (direct) execution of a Haskell program, as well as the primary purpose of compilation to a virtual machine, supporting our thesis of executable specifications can also be implementations.
- We pay special attention to the issue of expressing control flow in our DSL, and other aspects which are which traditionally hinders a specification being cross compiled. We will document our pragmatic solution, and hope that this can facilitate a future better understanding of the options available here.

2 A VM for Orbital Simulation

The 2009 ICFP contest required the integration of an Orbit Virtual Machine (OVM) with a controller program that interacts with the OVM through a port-mapped interface (depicted graphically in figure 1).

1. The user-written controller will write to the VM's input ports.
2. The VM begins execution. The VM will execute all its instructions, starting at address 0x0 and continuing on to the last address. There is no control flow inside these instructions, apart from this consecutive execution.
3. After the machine has executed the entire address space, corresponding to one simulation time step, the controller then reads the VM's output ports.
4. The user-written controller generates *new* input port values, and then repeats the process.

Nominally, each execution loop would take 1 second of virtual time, but this simulation granularity is an arbitrary design choice, and unimportant in the architecture design. The principal challenge of the contest was to write a control

program which interacts with a simulation of a physical environment through a series of actuators (supplying control inputs) and sensors (providing environment monitoring). The physical simulation was manifested as a simulation binary which was executed on a small virtual machine. The contestants were provided with an English description of the instruction set architecture of the virtual machine, which they were required to implement faithfully.

For all problems, the contestants control the satellite by providing input to a thruster actuator. The control program can observe and react to the environment using a variety of sensors. The control-program actuator inputs and sensor outputs are mapped to specific virtual machine input and output ports, respectively. This architecture allowed the contest organizers to publish small, machine independent binaries that simulated specific scenarios to be solved by contestant-supplied controllers interacting with the OVM.

Input port address	Actuator
0x2	ΔV_x
0x3	ΔV_y
0x3E80	Configuration

Table 1. Actuator input ports

Output port address	Sensor
0x0	Score
0x1	Fuel Remaining
0x2	s_x relative to earth
0x3	s_y relative to earth
0x4	Target orbit radius

Table 2. Example sensor output ports

The input address space size of 2^{14} ports was chosen based on the 14 bits available inside the instruction encodings. Table 1 gives the specific mapping used in the contest. Fundamentally, all the decisions in any user-written control logic came down to when to fire the virtual actuators, in what direction, and with what force. The supplied binaries also read a configuration port, written by the contestant before the first cycle, that specified the specific sub-problem being simulated. Each sub-problem initialized the environment with satellites in different locations.

The number of output ports was also chosen to be 2^{14}, for the same benign reason as the input size. Table 2 gives a example output mapping used in the contest, for one of the problems. The quality of a completed solution was communicated by a non-zero score on the $0x0$ port. The other ports gave virtual sensor information, like distance to earth, and specific information about the mission objective.

The virtual machine has a small instruction set, listed in Table 3. The choice of instructions are themselves unsurprising, but a notable feature is that every instruction can only write to a single destination register, r_d. The address d is the same as address of the corresponding instruction. For example, the instruction at address 0 will store its result in register r_0, the instruction at address 1 will store its result in register r_1, and so on.

A consequence of this is that there is no opportunity for two instructions to conflict and overwrite the same register. The one exception to this is the

Instruction	Semantics
Add r_1 r_2	mem[r_d] ← mem[r_1] + mem[r_2]
Sub r_1 r_2	mem[r_d] ← mem[r_1] - mem[r_2]
Mult r_1 r_2	mem[r_d] ← mem[r_1] * mem[r_2]
Div r_1 r_2	if mem[r_2] = 0.0
	then mem[r_d]← 0.0
	else mem[r_d] ← mem[r_1] / mem[r_2]
Output r_1 r_2	outport[r_1] ← mem[r_2]
Phi r_1 r_2	if status = '1'
	then mem[r_d] ←mem[r_1]
	else mem[r_d]←mem[r_2]
Noop	mem[r_d]← mem[r_d]
Cmpz *op* r_1	status← $mem[r_1]$ *op* 0.0
Sqrt r_1	mem[r_d]← $\lvert\sqrt{mem[r_1]}\rvert$
Copy r_1	mem[r_d] ← mem[r_1]
Input r_1	mem[r_d]← inport[r_1]

Table 3. Instructions in OVM

Output instruction, which writes to an machine output port. When a sequence of Output instructions are executed, each writing to the *same* PortId, then that output port will have the value written by the most recently executed Output instruction at the end of the simulation cycle. However, we sidestep this issue in our compiler by insuring that only one Output instruction ever writes to a given output PortId.

To simplify the simulator the ports and registers contained, and instructions manipulated, double precision IEEE floating point values, with the singular exception of a single 1-bit status register. This status register is used to emulate control flow in the simulation. The register, written by a comparison instruction and read by a *phi* instruction, allows the VM to copy the value of one of two registers into another, based on the value of the status register, set by a prior comparison instruction. The status register and accompanying instructions were used to initialize the initial simulator state based on the configuration port value, as well as to detect and report the successful completion of the problem objective.

The OVM provides a simple platform upon which to execute the satellite simulations, using binaries that encode a list of OVM instructions. Next, we discuss our executable model of physical behavior, before tackling the compilation of this model into OVM instructions.

3 Executable Model of Orbital Behaviors

The OVM simulation steps enumerated at the beginning of section 2 suggests a simple interface to a possible purely functional model of the OVM behavior. There is a function that takes the input and contents of registers, and returns

output, along with the new contents of registers. This function *is* the specification of a specific instance of the Orbit Simulator. In Haskell, we can write

```
-- First attempt at specification (TOO LOW LEVEL)
step :: (...) => Input d -> Output d

data Input d  = Input (InputPorts d) (Registers d)
data Output d = Output (Registers d) (OutputPorts d)

type PortId        = Int          -- value between 0 and 2^14-1
type PortMap d     = Map PortId d -- map from PortId to 'd'
type InputPorts d  = PortMap d
type Registers  d  = PortMap d
type OutputPorts d = PortMap d
```

We abstract over the contents of the registers, but can assume for now that they will be instantiated to `Double`. `Map` provides a finite map from `Int`, or address, to value, simulating memory.

The main challenge with this model is that it is to low level. It simulates the behavior of the *machine*, not the problem being simulated, and requires the simulation writer to handle low-level details, such as data layout and representation of simulation object, that are typically handled by a compiler.

With the general shape of the OVM in mind, we can consider what artifacts we might want to model.

Scalars such as fuel remaining, time, and distance to target.
Bodies in our simulated universe. These bodies will contain several vectors, like x, y coordinates and velocity, as well as scalar attributes such as weight.

With this in mind, we define a `State` datatype that can explicitly handle the data shapes of both types of artifacts, and the datatype of the contents of the state.

```
type Registers d = PortMap d
data Body d = Body { position :: !(Vec2 d)
                   , mass     :: !d
                   , velocity :: !(Vec2 d)
                   }
data Vec2 d = Vec2 !d !d
data State d = State [(Int,Body d)]         -- n bodies
                     (Registers d)          -- registers
```

This `State` is a higher level representation of what state we want to keep between steps in our simulator. In this way, we have an arbitrary number of named bodies, and a number of registers, all abstracted over the actual scalar data type, expected to be `Double`. By capturing a body abstractly, the DSL allowed the behavior of the body to be represented independently of the actual mapping of the body data to OVM registers and ports.

The final types used for `Input` and `Output` are shown below. The `Input` type includes an extra field for a clock input, allowing us to enforce problem cycle limits. Again, `d` is expected to be `Double`.

```
data Input d = Input (PortMap d)        -- 2^14 input ports
                     d                  -- clock
                     (State d)
data Output d = Output (PortMap d)      -- 2^14 output ports
                       (State d)
```

The `step` function takes `Input`, and returns `Output`. In order to help project into `Input`, and construct `Output`, we defined some helper functions.

```
-- readers
bodyRead  :: Int -> Input d -> Body d
ioRead    :: PortId -> Input d -> d
clockRead :: Input d -> d

-- writers
bodyWrite :: Int -> Body d -> Output d
ioWrite   :: PortId -> d -> Output d
regWrite  :: PortId -> d -> Output d
```

The projection functions, such as `ioRead`, extract the contents of a register from the `Input` value. The writers all build an `Output` value. The `Output` type is an instance of the `Monoid` class, allowing fragments of type `Output` to be accumulated into a final `Output`. This accumulation relies there being only one (or zero) writes to a particular output port per simulation cycle. An example use of these functions is shown in the definition of `step` in figure 2.

One useful property resulting from this form of `step` is that it is trivial to call from within GHC or GHCi. It can be tested and executed in isolation. All the "effects" of this function are captured inside the the `Output`; this definition is classic functional programming!

In this `step` example, `SpecialOps` is a Haskell class that that encapsulates everything we want our type parameter 'd' to do. `SpecialOps` is the hook we will use for implementing control flow, in section 5.

To execute `step`, we need a way of generated an initial state. To do this, we also used a function, from the initial input port, to list of bodies. The actual code from our first scenario was.

```
initState :: (SpecialOps a) => PortMap a -> [Body a]
initState actuators = [earth,sat1 actuators]

-- earth starts at the origin , and does not have initial velocity.
earth :: (Floating a) => Body a
earth = Body (Vec2 0 0) massOfEarth (Vec2 0 0)

sat1 :: (SpecialOps d) => PortMap d -> Body d
sat1 actuators = ...
```

```
step :: (SpecialOps d) => Input d -> Output d
step input = output
  where
        earth           = bodyRead 0 input
        sat1vGrav       = attract earth sat1
        sat1'           = newBody 1 sat1 sat1vGrav
        sat1'vGrav      = attract earth sat1'
        avGrav          = avg sat1'vGrav sat1vGrav
        velocity2       = newVelocity 1 (sat1vThrust ^+^ avGrav) sat1
        sat1''          = sat1' { velocity = velocity2 }
        score           = ...

        output = mconcat
                 [ bodyWrite 1 sat1''
                 , bodyWrite 0 earth
                 , ioWrite 0 score
                 ]
```

Fig. 2. Example Orbit step program

The initializer and stepper from our design were paired together inside a data structure, `OrbitProgram`.

```
data OrbitProgram = OrbitProgram
   { -- return the initial bodies
     initializer :: forall d . (SpecialOps d) => PortMap d -> [Body d]
     -- our stepping simulator
   , stepper     :: forall d . (SpecialOps d) => Input d -> Output d
   }
```

This closes our behavior specifications for the OVM inside a single datatype, which can be initialized using `initializer`, and stepped using `stepper`. Both of these functions must work for *any* `d` that admits the class `SpecialOps`.

4 Our Deep Embedding DSL and OVM compiler

The `Input -> Output` step function provides a simple way to specify satellite behavior. If we had could capture and compile a step function into the instructions of our VM, all inside a single Haskell program, we would have four distinct use-case scenarios. Figure 3 illustrates these use-cases.

1. We already have our original path from an `OrbitProgram` to execution of a preexisting solution.
2. We could also compile to our VM instructions, simulate them using a VM simulator, and also execute our preexisting solution.
3. We could use the compiler to generate VM instructions, and instead of executing them, we could serialize them into a binary format.

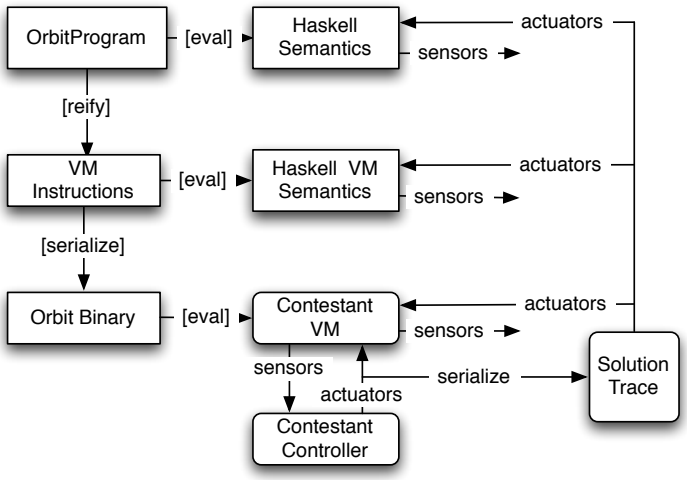

Fig. 3. Use Cases of reify based compiler

4. Again within the same framework, we could read an existing serialized list of VM commands, from 3, and execute them, much like our second scenario above.

What does this simulation architecture buy us? Testing via redundancy. Assuming our specification is correct, we can compare the behavior of 1 and 2, giving confidence in the compiler and VM simulator. Comparing 2 and 4 gives us confidence in our serialization. VM instruction streams can also be tested inside other VMs, with other implementations of solutions.

This architecture motivates our central question: How can we compile an `OrbitProgram` to our VM? We do this by providing an overloaded *observable value*, which captures the shape of the computations inside our specification.

As an example of the sort of operations we want to capture, consider the function for computing distance between two points.

```
distance :: (Num d) => (d,d) -> (d,d) -> d
distance (x0,y0) (x1,y1) = sqrt (xd * xd + yd * yd)
   where
        xd = x1 - x0
        yd = y1 - y0
```

The majority of the computations inside our `stepper` are simple floating point arithmetic like this. Specializing `distance` over `Double` *is* the function for computing distance. We want to have a second overloading, that allow the observation of the *structure* or internals of `distance`, and we do that using a deeply embedded DSL.

A deeply embedded DSL is where the structure of proposed computation can be observed. To enable this, we used a slightly customized early version of Kansas Lava, The principal type inside this version of Kansas Lava was a `Signal`, which adds phantom type around `Wire`, the internal type of a circuit.

```
newtype Signal a = Signal Wire
newtype Wire = Wire (Entity Wire)
```

`Entity` is a node in our "circuit" graph, which can represent gate level circuits, as well are more complex blocks.

```
data Entity s
  = Entity String [s] -- an entity
  | Var String        -- an input pad
  | Lit Double        -- a constant
```

Using this simple structure, we can express abstract syntax trees, where the nodes are named `Entity`, and the leaves are `Var` or `Lit`.

In the `distance` example, we capture this by overloading arithmetic over `Signal`. For example, `+` for `Signal Double` is shown below. Similar definitions are used for other arithmetic operations.

```
(+) :: Signal Double -> Signal Double -> Signal Double
(Signal w1) + (Signal w2) = Signal (Wire (Entity "+" [w1,w2]))
```

`sqrt` is overloaded as:

```
sqrt :: Signal Double -> Signal Double
sqrt (Signal w) = Signal (Entity "sqrt" [w])
```

If `xd` is bound to `Var "xd"`, and `yd` is bound to `Var "yd"`, the expression `sqrt (xd * xd + yd * yd)` gets interpreted as

```
Signal (Wire (Entity "sqrt"
         [ Wire (Entity "+"
            [ Wire (Entity "*" [ Wire (Var "xs"), Wire (Var "ys") ])
            , Wire (Entity "*" [ Wire (Var "xs"), Wire (Var "ys") ])
            ]])
```

As can be seen, inside the `Signal` is an untyped abstract syntax tree. The trick with capturing functions is apply them to `Signal`s that contain `Var` inside them. In this way, by providing a deep embedding for *all* the functions used inside our OVM executable model, we can capture the computation involved in the `OrbitProgram stepper` function.

Compilation simply requires a linearization of the AST into a sequence of VM instructions. Each `Entity` maps directly to a single machine instruction, where the result of the instruction being stored in a temporary VM register. Likewise, each `Var` maps to a particular reserved register (for elements of a body) or ports (for io reads and writes).

Using a deep embedding is a powerful design pattern. There are, however, three shortcomings of a straightforward deep embedding.

1. The embedding is extracted as a *tree*, even if it represents an underlying direct acyclic graph. Host-language sharing of embedded values is lost in the interpretation function. This problem can be overcome using common sub-expression elimination.
2. Furthermore, if the the underlying graph contains loops, then the extracted tree is infinite, and the structure can not be given a finite interpretation.
3. Finally, the underlying structure must be constructive. An interpretation function which discriminates on DSL nodes cannot be used on values which are not known statically. That is, deconstructing operations like conditions, case statements, or pattern matching can not in general be used. In our example above, every operator mapped directly onto onto a constructor in our abstract syntax tree.

Despite these issues, the technique of deep embedding has been successfully used for many DSLs. In our compiler, we address the first issue using our IO-based observable sharing library which retains sub-expression sharing. The second issue, value recursion, does not occur in our given target application The final issue, constructiveness, we address using a special combinator, which we discuss in the next section.

5 Handling Conditionals in the OVM DSL

In Haskell, conditionals are expressed using a `if-then-else` syntax. What we want is some way of observing such conditionals as abstract syntax. We do so by making `if` a function, called `iF`. The `iF` function is used within the DSL as shown below.

```
where
     ...
     var = iF (pred)
              (... evaluate if pred == True ...)
              (... evaluate if pred == False ...)
```

This formulation is usable, if idiomatic, and almost scheme like. From an abstract syntax point of view, there are three requirements of this construct.

1. The predicate must be observable. The obvious solution is to use Signal Bool, with associated operations. However, given the principle of overloading our specification both `Double` and `Signal Double`, we choose a slightly different representation for booleans, based on currying.
2. The branches must in general be polymorphic.
3. The branches must be the same type.

To satisfy these requirements, we define a new type of Bool, called a `SpecialBool`.

```
newtype SpecialBool d = SpecialBool (d -> d -> d)
```

`SpecialBool` is a Church-encoding of booleans as a function with takes two arguments, the true and false cases. The type parameter `d` is the institution used for the specification, `Double` or `Signal Double`. The intuition is "if we know how to switch on a single `Double`-shaped value, and our condition branches can be composed of a serialization of such values, then we can construct the operational semantics of the conditional using many smaller, primitive conditionals, over `d`." This is hardware like thinking, about using small muxes to give large muxes. It also assumes that both sides may be evaluated, without changing the semantics of the result. Haskell is a pure language, and our specification is without loops, so this will simply cause extra evaluation.

Standard boolean operation can be encoded using Church encodings. For example, the or combinator can be constructed using

```
(.||.) :: SpecialBool d -> SpecialBool d -> SpecialBool d
(SpecialBool a) .||. (SpecialBool b) = SpecialBool $ \ t f -> a t (b t f)
```

We can now construct our `iF` implementation, first overloaded, then for instance `Double`, then finally for `Signal Double`. We start with a serialization class, and a class to capture the primitive boolean constructions.

```
class Serialize e d where
    serialize :: e -> [d]
    unserialize :: [d] -> (e,[d])

class (Floating d, Serialize d d) => SpecialOps d where
  (.==.) :: d -> d -> SpecialBool d
  (.<.) :: d -> d -> SpecialBool d
  constantDouble :: Double -> d
```

The `Serialize` class provides the ability to turn objects of type `e` into a list of type `d`, and back again. The `SpecialOps` class captures the operations that the `d` type argument to our specification admits, in this case two styles of comparison. Other comparisons can be constructed out of these two primitives, and the standard boolean combinators.

```
iF :: (SpecialOps d, Serialize e d)
   => SpecialBool d -> e -> e -> e
iF (SpecialBool f) = \ e1 e2 ->
   unserialize' [ f d1 d2
                | (d1,d2) <- zip (serialize e1)
                                 (serialize e2)
                ]
  where unserialize' ds = case unserialize ds of
                            (e,[]) -> e
```

Literally, we implement `if` by serializing both the `then` and `else` branch into a list of basic `d` values, performing the condition that is encoded inside `f`, then unserializing the values back again. The code for the instance for `Double` is

```
instance Serialize Double Double where
  serialize e = [e]
  unserialize (d:ds) = (d,ds)

instance SpecialOps Double where
  a .==. b = SpecialBool $ \ c d -> if a == b then c else d
  a .<. b  = SpecialBool $ \ c d -> if a < b then c else d
  constantDouble = id
```

We also provided instances of serializing `Body` (not given), and `SpecialOps` for `Signal Double`.

```
instance SpecialOps (Signal Double) where
  x .==. y = SpecialBool  $ \ (Signal a) (Signal b) ->
               Signal $ Wire $ Entity (Name ".==.") [a,b]
```

With instances for all our functions at the `Double` and `Signal Double` type, and with overloading our key operators using `SpecialOps`, a stepper function with control flow can be both interpreted and be captured into an AST.

6 Controlled Partial Evaluation

The class `SpecialOps` was named for special operations, originally collection of operations that the DSL might need beyond simple the arithmetic. As such, it offered a good place allow simplifications opportunities to the virtual machine. The original virtual machine included basic trigonometry operations, like sin and cos, complicating the implementation requirements for the contestants. In all our generated binaries for the contest the use of trigonometry was restricted to only the initialization stage, and was never used in the stepper. Specifically, placing the initial position of a satellite used sin and cos to compute the placement all satellites at regular distances from each other. One team took advantage of this pattern, by performing a hard-reversing operation, and spinning outwards in a different direction to all the satellites positioned at different orbits. The pattern only occurred at the start of the simulation, and it did not occur to us that someone might reverse to take advantage of our computationally based positioning.

So we used partial evaluation, to perform trigonometry at Haskell runtime, as we are generating our VM code. We do this by using an `evaluate` function.

```
evaluate :: (SpecialOps d, Functor t) => t Double -> t d
evaluate e = fmap constantDouble e
```

That is, given a structure that is defined over `Double`s, we promote the contents of the structure to use something overloaded with `SpecialOps`. In this way, we had a mechanism that allowed actual `Double` values to be used in our specification execution, but construct `constant Signal Double`, which become constant values on our virtual machine.

7 Compiling to the OVM

The `OrbitProgram` DSL provides a mechanism for constructing a deep embedding of a simulation scenario. Having constructed this deep embedding, we then convert it into a stream of instructions which can then be executed by the OVM. This process consists of reifying the deep embedding into an explicit graph of the program data structure, linearizing the graph into a flat stream of instructions, and then assigning instruction argument node identifiers to OVM registers.

The layout of `Body` state is a side-effect of the translation from an `OrbitProgram` to OVM instructions. First, we invoke the `OrbitProgram` `initializer` function on a `PortMap`, with every port initialized to contain a `Var` signal with a name corresponding to the `PortId`. This yields a list of Orbit `Body`s, where each field contains a `Signal`.

Next, we invoke the `stepper` function of the `OrbitProgram` with an initial `Input`. The `Input` has an collection of ports populated with `Var`s. Similarly, the `Body`s within the state have their fields populated with `Var`s. Applying the `stepper` to the `Input` yields an `Output`, which includes a `State` with a collection of resulting `Body`s. It is necessary to merge the body output from the `initializer` with that of the `stepper`, while also noting the `Var` associated with each element of a body. We do that with a simple list comprehension and `mux`.

```
inStateMuxed :: [Signal Double]
inStateMuxed = [ iF (clk .==. 0) a b
               | (a,b) <- zip initState inState
               ]
```

Having done this, we now know how the state of a body should be updated at the end of each cycle, furthermore, the `Output` from the invocation of the `stepper` function provides us with the values that should be written to each output port.

We then use an observable sharing library to `reify` this structure into an explicit Graph. Reification takes a `Signal Double` and converts it into a data structure that associates a unique identifier with each `Entity` value in the `Signal`, and replaces all child nodes of an entity with the associated node identifier.

```
type Graph = Map Int (Entity Int))
reify :: Signal a -> IO Graph
```

The compiler defines a lifting of `reify` over an entire `Output`, yielding one graph for all of the `Signal`s in the `Output`. We invoke the lifted reify function on an Output that contains the `PortMap` from the `stepper` `Output` and the list of `Body`s resulting from the merging of the bodies from the `initializer` and `stepper` outputs.

The resulting graph contains `Var Entity` values corresponding to locations where `Body` state should be stored. The compiler updates the graph, using the mapping from the `Var` to the associated `Signal` calculated by `mergeBody`. Any

`Entity` in the graph that refers to a node identifier, where that node identifier maps to a `Var`, is replaced with the node identifier associated with the `Var`.

After `Var` nodes have been eliminated, the reified graph is converted into a linear sequence by performing a post-order traversal of the graph, starting at the roots defined by the output of the `stepper` function. The resulting sequence of Entities will now be in the order that they will be executed in.

Recall that the instructions in the OVM write to a register address that is the same as the instruction's address. We scan the linearized graph of entities, associating each node identifier with its positional index in the sequence. Then, we use that association to update each `Entity` to refer to the position of the sub-term nodes, rather than the node identifiers. At this point, the registers associated with every instruction will now be fixed.

The final step in the compilation process is to convert each `Entity` into an associated OVM instruction, which is also represented as a Haskell data type. This instruction selection process is largely a translation, based on the name of each entity, into the entity's corresponding OVM instruction.

After the `OrbitProgram` is converted into a list of OVM instructions (represented as Haskell values), we can serialize the list into the binary format defined in the contest problem specification. The serialized version of the instruction stream is the binary that was supplied to participants during the course of the contest.

8 Related Work

Deep embedding is a well understood part of the folklore in both in the functional language and theorem proving communities. There have been many instances of so called deeply embedded DSLs in Haskell before, including various flavors of Lava [1, 7] and Elliott's family of graphical description languages [5, 4].

Direct observable sharing [3] is implemented by generating indirection constructors that admit equality. Provided that the deeply embedded DSL's API used this indirect constructor, which is typically straightforward, then sharing can be observed. The IO-based observable sharing [6], as used in this paper, restricts the types that can be observed by using a partial *type function* [2] to map from observed type to graph representation, and this graph can only be observed using `IO`. In this way, IO-based observable sharing is strictly less powerful than direct observable sharing, though we would argue that these restrictions provide reasonable engineering safeguards consistent with functional programming practices.

There is also rich history of executable specifications, with Haskell itself being an example. The challenge remains that an executable specification is invariably not an optimized implementation, though perhaps a reasonable implementation. We need ways of applying aggressive and powerful optimizations, regaining efficiency. Our use of deep embedding to do this is a special instance of powerful idiom of multi-staged programming [9].

9 Conclusion and Observations

This paper has given a short overview of our language for expressing orbital dynamics, which can be compiled or executed as a model. As such, it is a small case study in deeply embedded DSLs, which with careful construction can also be directly executed. We needed to be creative with the compilation of conditionals, and we are not completely happy with our solution. If we were to revisit our design, we would use Template Haskell [8], or some similar system to circumvent this issue. An open question is the design of a light-weight template system that would allow this single issue to be resolved neatly, which again returns us to multi-staging.

Our OVM DSL compiler was a first application of Kansas Lava framework. Indeed, we actually moved our Kansas Lava version control repository to a secured server in the weeks leading up to the contest, so that we would not give anything away as we enhanced Kansas Lava to support the OVM compiler. We expect that Kansas Lava will continue to be the platform for CSDL to continue exploring expressive and compilable DSLs.

As a system, our dual embedding facility to both interpret and compile was useful in practice. We could link directly against our executable model, test our strategies and scorings, and then have confidence that the generated code would perform in a robust way during contest deployment. Indeed, there were no issues with the quality of generated code.

- The generated code ran fast enough for simple interpreters to be useable, and the design allowed for some teams to choose to pre-compile the OVM instructions our compiler generated.
- Furthermore, there were no (known) correctness issues with the OVM code generated by our compiler. We attribute this to the fact that we could compare output from executable model and compiled output as we were constructing the compiler, the direct nature of the path from abstract syntax tree to VM instructions, and the finally because we were actually compiling our executable specifications.

We wrote an optimizer for our OVM compiler that did common subexpression elimination, constant folding and a few other simple optimizations, but did not deploy it. This was partly because we had not build up sufficient trust in the optimizer, and partly because the different in performance was less that a factor of two. Teams were free, of course, to write there own OVM instruction optimizer from the given instruction semantics.

In conclusion, we would encourage others to look at dual embedded systems as a low-cost way of constructing a compiler for custom virtual machines. Factoring out our Kansas Lava work, the dual model/compilation system only took 2 engineering weeks to complete, work reliably, and allowed us to experiment with several OVM designs quickly.

Acknowledgments

We would like to thank all the members of the ICFP 2009 Programming Contest organizing team, especially Kevin Matlage, who wrote the prototype control system that used our model, and Wesley Peck, who wrote the graphical simulator that we linked to our executable model.

References

1. Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.
2. Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253, New York, NY, USA, 2005. ACM.
3. Koen Claessen and David Sands. Observable sharing for functional circuit description. In P. S. Thiagarajan and Roland H. C. Yap, editors, *Advances in Computing Science - ASIAN'99*, volume 1742 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 1999.
4. Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.
5. Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003. Updated version of paper by the same name that appeared in SAIG '00 proceedings.
6. Andy Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, Sep 2009.
7. Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing Kansas Lava. In *Proceedings of the Symposium on Implementation and Application of Functional Languages*, Sep 2009.
8. Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.
9. Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. 2004.