Bluespec — Designer's Perspective

Lennart Augustsson

augustss@sandburst.com

Sandburst Corporation

17th January 2003

Evolution of Bluespec

- -1999 TRAC work at MIT.
- 2000 Bluespec version 1, much like TRAC, but with named state components and somewhat Haskellish syntax.
- 2001- Current Bluespec, Haskell syntax, full Haskell functionality at compile time, monads for handling state.

Design choices

- Syntax
- Types
- Semantics (trickier for HW because of realizability)
- Staging

Some of these choices were already made, because the Bluespec was going to be based on TRS.

Syntax

The Bluespec syntax is based on Haskell, a choice that makes it unfamiliar to almost everyone. (To succeed these days you have to look like C, or have a truly bizarre syntax (e.g., Perl).)

> Robin Milner: People discussing concrete syntax are like people getting into a car; they become animals.

Types

Choices:

- Untyped, like the hardware (e.g., almost Verilog)
- Dynamically typed, not realizable (e.g., Scheme)
- Statically typed

Types, statically typed

- Non-polymorphic (e.g., C)
- Somewhat polymorphic (e.g., VHDL)
- Hindley-Milner polymorphism (e.g., SML)
- Hindley-Milner polymorphism+overloading (e.g., Haskell)
- Dependently typed (e.g., Cayenne)
- Subtyping
- Object typing
- Resource type systems (e.g., linear types)
- ...

Semantics, choices

The run time semantics is TRS (that was the premise of Bluespec), but what about the compile time semantics?

- imperative (C, etc.)
- object oriented (C++, Java)
- process/simulation (Verilog)
- functional

None of these is a perfect fit.

Semantics, 1

There are two semantics of Bluespec, the compile time semantics and the “run time” (i.e., hardware) semantics.

The “run time” semantics is based on TRS. Part of a TRS description specifies how to get from one state to the next, this is a pure function so a pure functional language is a good fit.

In Bluespec the state transition function is written in the subset of Bluespec that is realizable as hardware. The realizability restriction is enforced by types (except for termination issues), but types cannot guarantee a size bound on the generated hardware.

Semantics, 2

The compile time semantics of Bluespec is full Haskell (many libraries are missing, but could be implemented).

The compile time language must be able to describe both the state transition functions (as mentioned before), but also the state elements.

Describing state, i.e., items that have an identity, does not fit into a pure functional framework as neatly.

Semantics, 3

These are the same:

```
let x = 2                  let x = 2
    y = 2
in  ... x ... y ...        in  ... x ... x ...
```

These are not:

```
int x = 2;                 int x = 2;
int y = 2;
... x ... y ...            ... x ... x ...
```

Semantics, 4

To describe state Bluespec borrows from O'Haskell, an object oriented version of Haskell.

The key idea is that all descriptions of hardware is done in a monad, `Module`. Using a monad we can easily keep track of the state.

The `Module` monad is built in to the compiler and its internals are not accessible to the programmer (except for operations like `addRules`).

But, the `Module` monad is extensible, cf., `PCIModule`.

The hardware description part of Bluespec has a very object oriented flavour; but then we are really describing physical objects.

Staging, 1

All language translators involve some kind of staging: certain computations are carried out by the compiler, others at run-time.

In a hardware description language the staging issue is brought to its point, some operations can only be performed at compile time some are clearly intended to generate hardware.

Questions:

- How powerful should the compile time language be?
- Is it the same as the run-time language?
- Is the staging visible in the source?

Staging, 2

Some examples:

- C
  The preprocessor is the compile time language, it's very weak. Some uses of it (`#if`, etc) are clearly marked.

- Verilog
  Verilog has a C like preprocessor. All uses of it are marked. *But*, there are also some language constructs that are compile time, like `for` loops.

- Bluespec
  Bluespec has the same language for compile time and "run time". The language is very powerful (it's Haskell). It's generally impossible to determine the staging of an expression (although, e.g., `Integer` has to be compile time, and, e.g., rules are always run time).

Staging, 2

- Template Haskell
  Template Haskell has the same language for all stages, but there are clear annotation in the source when you switch between stages, e.g., “`$(f x) y`” means that the application “`f x`” is executed at compile time, and the last application at run time.

Staging example, tabulate

An old favourite, the factorial function.

```
fac :: Bit 3 -> Bit 32   -- not the most general type
fac 0 = 1
fac n = zeroExtend n * fac (n-1)

    ...
    i :: Reg (Bit 3)  <- ...
    o :: Reg (Bit 32) <- ...
    rules
        when c ==> o := fac i
```

Compilation if this does not terminate. The reason is that the termination condition for `fac` cannot be computed at compile time.

Example, tabulate, cont

The `tabulate` funtion to the rescue:

```
i :: Reg (Bit 3)  <- ...
o :: Reg (Bit 32) <- ...
rules
    when c ==> o := (tabulate fac) i
```

Now it does terminate, because tabulate builds a table of all the results of `fac` applied all possible arguments.

Example, tabulate, the code

The `tabulate` function is written entirely in Bluespec!

```
tabulate :: (Bounded a, Enum a, Eq a) => (a -> b) -> (a -> b)
tabulate f x =
    foldr (\ b r -> if x == b then f b else r)
          _
          (enumFromTo minBound maxBound)
```

Example, tabulate, unfolded

The `tabulate fac` application expands to

```
\ x ->      if x == 0 then fac 0
       else if x == 1 then fac 1
       else if x == 2 then fac 2
       else if x == 3 then fac 3
       else if x == 4 then fac 4
       else if x == 5 then fac 5
       else if x == 6 then fac 6
       else if x == 7 then fac 7
       else _
```

Example, tabulate, unfolded

The `tabulate fac` application expands to

```
\ x ->      if x == 0 then 1
       else if x == 1 then 1
       else if x == 2 then 2
       else if x == 3 then 6
       else if x == 4 then 24
       else if x == 5 then 120
       else if x == 6 then 720
       else if x == 7 then 5040
       else _
```

Example, tabulate, Verilog

```
case (x)
  3'h0, 3'd1: tfac_ = 32'd1;
  3'd2: tfac_ = 32'd2;
  3'd3: tfac_ = 32'd6;
  3'd4: tfac_ = 32'd24;
  3'd5: tfac_ = 32'd120;
  3'd6: tfac_ = 32'd720;
  3'd7: tfac_ = 32'h000013b0;
endcase
```

Conclusions

We got some things right in Bluespec, but there are many choices where there is no obvious answer. It would be interesting to experiment with other designs, e.g., a somewhat better type system and staging annotations.