BS better with FP

...in three acts

Mieszko Lis
Ravi Nanavati
Bluespec, Inc.

Outline

- Haskell's Adventures in the Real World
- Peddling FP under the covers
- Compiling FP into hardware

Act I

Haskell’s Adventures in the Real World

Background

- Bluespec, Inc.
  - 1yr+ VC-funded startup
  - ~20 employees, ~10 engineers
  - technology developed at MIT and Sandburst
- Chip design tool (details later)
- Code size
  - compiler: 61k lines Haskell + 109k lines C
    - (C mostly in BDD library)
  - RTS/Libs: 8k lines BS, 1.2k lines Verilog, 12.5k lines C

Benefits of Haskell

- Quick prototyping
  - optimize later when required
- Type system allows safe changes and refactoring
- Pattern matching permits concise code
- Automatic memory management
  - so good, nobody notices
- Monads clear the mind (and the sinuses)

one (Haskell-trained) intern added full SV assertions support in a summer

Business perspective

- Hiring Haskell programmers
  - the pool is *very* small
  - but smart (non-Haskell) people learn quickly
  - ramp-up cost dominated by deciphering code and articulating hidden assumptions anyway
    - but businesses need to plan for this
- Inexpensive outsourcing harder
  - training is an issue
- Scarcity of Haskell tools adds risk
  - de facto GHC dependency
  - free software license helps

Sins

Big positional data structs

- Good

```
data Maybe a = Just a | Nothing
```

- Bad

```
data Pkg = Pkg String String Foo Int Integer
               [Int] Bar ...
```

- Deadly

```
-- some thousand lines later or in another file...
frobble (Pkg _ s f _ z ys b ...) = ...
```

  - *especially* with easy-to-type variable names
- Same with functions of many arguments

Deeply nested patterns

- Obvious

```
fromBE (If e1 e2 e3) = ...
fromBE (And e1 e2) = ...
```

- Readable?

```
collEQs (IAps (ICon _ (ICPrim _ PrimBAnd)) _
   [e1, e2]) = ...
```

- Encrypted

```
vsUniv (ICon i (ICValue { iValDef = IAps (ICon _
   (ICPrim _ PrimRange)) _ [ICon _ (ICInt { iVal
   = IntLit { ilValue = lo } }), ICon _ (ICInt
   { iVal = IntLit { ilValue = hi } }), _] }))
   = ...
```

Misguided "cleverness"

- "I bet I can do it with `concatMap`, `fold`, and `scanr`…"
- Long dotted chains of list functions

```
magic = magicfold . map snd . G.toVAList .
        addMissing . foldl G.addEdge G.empty .
        map (\(a,b) -> (a,b,()))
magicfold [] = []
magicfold xs = foldl1 intersect xs
```

- Not limited to Haskell

```
while(*s++=*t++);
```

Rewrite instead of reuse

- Foo.hs, line 432...

```
fst3 (x, _, _) = x
snd3 (_, y, _) = y
thd (_, _, z) = z
```

- Bar.hs, line 1207...

```
get_1st (x, _, _) = x
get_2nd (_, y, _) = y
get_3rd (_, _, z) = z
```

- Temptation remains high
  - searching slower than rewriting

Annoyances

Creeping monadery

- "Central repository" paradigm
  - flags
  - name supply
  - symbol table
  - rename an ID across the whole program
- Foreign calls (e.g., external libraries)
  - BDD was monadic; now it's foreign and in IO
  - ...and once in IO, there is no escape
- I/O *during* long computation (warnings)
- Soon IO has crept in *everywhere!*

Laziness and debugging

- Everyone wants a gdb
  - examine/change a "universe" snapshot
  - for debugging
  - for deciphering mysterious code
  - laziness makes it hard!
- Laziness not as beneficial as expected
  - need to write out intermediate files
  - need to force thunks to limit heap leaks
  - need to attribute runtime to specific stages

Testing/counting data tags

- Pattern-matching filters for one tag; what if you want two?

```
data T a b ... = T0 | T1 a | T2 b | ...
fribble x | isT0 x || isT1 x = ...
```

  - derive `isT0, isT1` automatically

- Class *Enum* enumerates the values of *T*; what if you want to enumerate the tags?

```
let tagNames = ["foo", "bar", "quux", ...]
    name = tagNames !! tagOf x
```

  - derive `tagOf` automatically

Learning Haskell

- More realistic examples in books
  - the real world lives in IO
  - the real world is not an interpreter
- Monads considered confusing
- No “good programming style” guide
- Easier to write code than to trace code
- How useful is Haskell to one’s career?

Act II

Peddling FP
under the covers

Tool and market

- For designing chips (ASICs, FPGAs, ...)
  - currently low-level with Verilog or VHDL
  - chip complexity rising (millions of gates)
- For chip designers, verification engineers, system architects
  - ASICs have huge NREs ($500K–$1M)
  - mistakes (respins) cost another NRE
  - tools run into millions of $$$ per team, form a significant fraction of a company's budget (e.g., ~10%)
  - tools tend to run on UNIX (Solaris, Linux)

Bluespec Classic: a Haskell-based HDL

bluespec

```
package Shift(shift) where
import List

sstep :: Bit m -> Bit n -> Nat -> Bit n
sstep s x i when s[i:i] == 1 = x << (1 << i)
sstep s x i = x

shift :: Bit m -> Bit n -> Bit n
shift s x = foldl (sstep s) x
                  (map fromInteger
                       (upto 0 ((valueOf m) - 1)))
```

Selling BS Classic

- Unfamiliar syntax a significant barrier
  - even in marketing slides
  - even ()s in function calls are different!
- Many fronts in adoption war
  - *new* hardware design methodology
  - *new* unfamiliar syntax
  - *new* type system
  - *new* purely functional thinking
  - *new* FP concepts (map, fold, monads)

Adapt an existing HDL

- Map matching concepts
  - expressions, bit vectors, functions, modules
- Extend where straightforward
  - higher-order functions, first-class objects, polymorphism
- Standardize where possible
  - tagged unions, pattern matching (SV 3.1a)
- Desugar where required
  - imperative assignments, loops

Bluespec SystemVerilog: FP with Verilog Syntax

bluespec

```
function Bit#(n) sstep(Bit#(m) s, Bit#(n) x, Nat i);
  if(s[i] == 1)
    return(x << (1 << i));
  else
    return x;
endfunction

function Bit#(n) shift(Bit#(m) s, Bit#(n) x);
  return(foldl((sstep(s)),
               x,
               (map(fromInteger,
                    upto(0, valueof(m) - 1)))));
endfunction
```

Bluespec SystemVerilog: Imperative circuit construction

```
function Bit#(n) shift(Bit#(m) s, Bit#(n) x);
   Integer max = valueof(m);
   Bit#(n) xA [max+1];
   xA[0] = x;
   for (Integer j = 0; j < max; j = j + 1)
      if (s[fromInteger(j)] == 1)
         xA[j+1] = xA[j] << (1 << fromInteger(j));
      else
         xA[j+1] = xA[j];
   return xA[max];
endfunction
```

Teaching BSV

- Limited training time (2-4 days typical)
- Audience: hardware designers
  - little or no FP background
  - wires and registers, not abstractions
  - conservative (remember cost of mistakes?)
- Format: lectures interspersed with labs
- Need to communicate basics
  - or else evaluation project might be hard
- Want to show full range of features
  - or else benefits not perceived and no sale

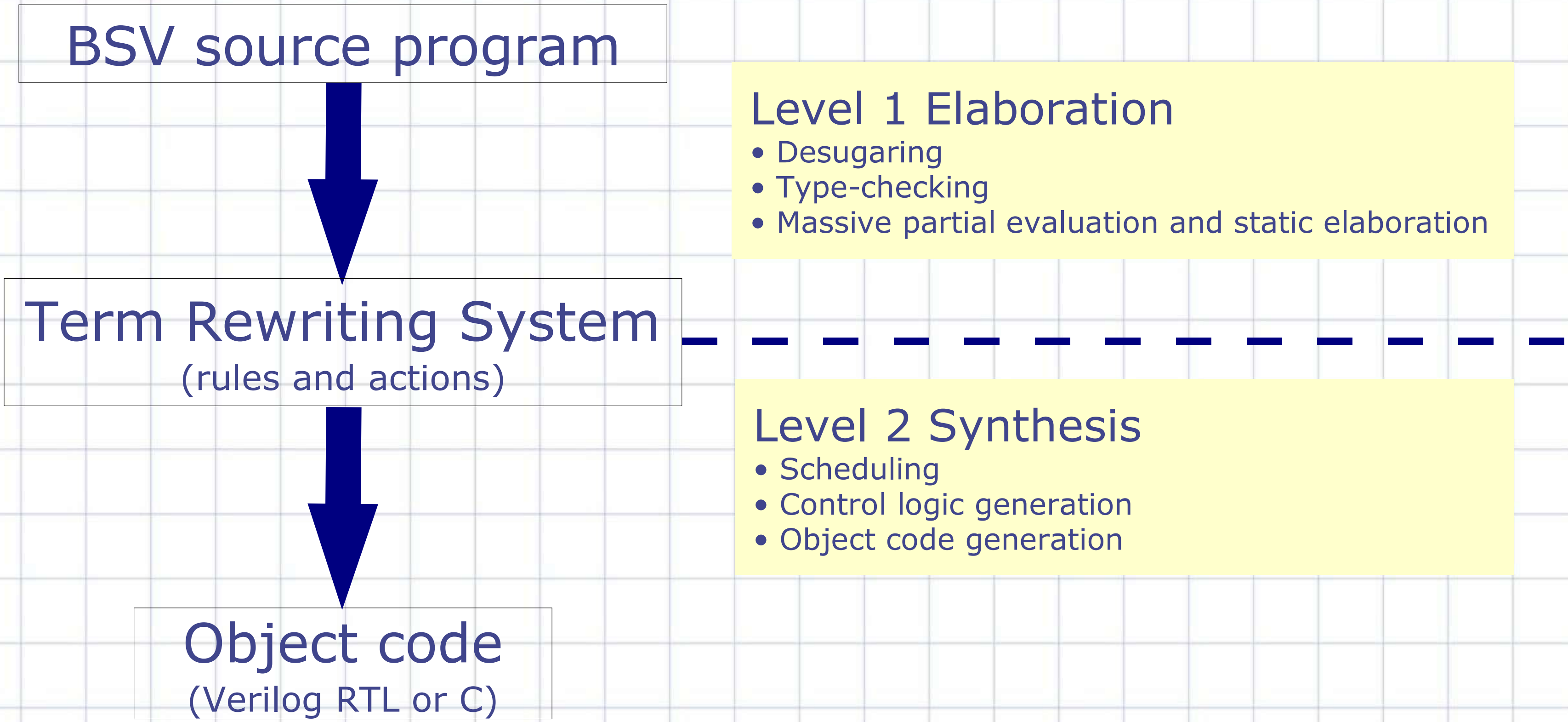
Teaching conclusions

- Functional features are advanced
  - Get in the way of communicating basics
- Strict typing seen as restrictive
  - bit vs. Bool
  - bit-width constraints
  - structures vs. bit representations
- Standards less relevant when teaching
  - damn the torpedoes and teach the sugar
- Key challenge: build intuition about generated hardware

Act III

Compiling FP
into hardware

Implementing BSV: two-level compilation



- For historical reasons, the level one evaluator is lazy
- Is this still a good idea as the language becomes more imperative?

Laziness is hard work

- Performance is a challenge
  - graph reduction required to avoid duplicating work
- Non-strict primitives (if, and, or) require careful handling
  - symmetric short-circuiting
  - undetermined values must be propagated correctly
- Error messages can be confusing
  - "Compile-time expression did not evaluate"

Being lazy pays off

- Consider: `let z = x + y`
- Is this:
  - a static constant?
  - a fixed incrementer?
  - a full adder?
- A lazy evaluator does not care!
  - evaluates what it can
  - defers (or suspends) what it can't
- User benefit: can move freely between static and dynamic code

Conclusions

- Using FP not at all tragic :)
  - makes a small team powerful and agile
  - power can easily be abused
  - does not cure common engineering ills
- Teaching FP *quickly* is a challenge
  - especially new thinking on multiple fronts
  - most professionals averse to change
- FP techniques apply in new contexts
  - good for your mental toolbox

The End