Living with Kind
Improving the Usability of Numeric Types in Bluespec SystemVerilog

Joe Stoy

Work by Ravi Nanavati *and* Lennart Augustsson

DCC 2010

March, 2010

What is Bluespec SystemVerilog?

- Bluespec SystemVerilog (BSV) is a high-level, statically-typed hardware design language
  - We have some users you might recognize
- Primarily implemented in Haskell
  - Over 120,000 LoC of Haskell code
  - Some C++, Tcl for simulator and debugging environment
- Language itself heavily influenced by Haskell
  - Started with a Haskell-like syntax (replaced with a Verilog-like imperative "skin")
  - BSV also has a Haskell-style type system
  - Type system includes Haskell typeclasses with functional dependencies and overlapping instances

Why does BSV need numeric types?

- Hardware is pervasively parameterized by a wide variety of numbers:
  - register width
  - memory sizes
  - number of read / write ports of a register file
  - number of connections on a bus
- Even lower-level hardware-design languages support some numeric parameterization
  - often with weak or nonexistent checking, leading to size-mismatch bugs

Numeric types in BSV

- Add kind `#` inhabited by 0, 1, 2, ...
- `primitive type Bit :: # -> *` (for bit vectors)
- User-defined types with numeric parameters:
  - `UInt :: # -> *`
  - `Vector :: # -> * -> *`
- Now we can express the types of some primitive operations:
  - `bitwiseOr :: Bit n -> Bit n -> Bit n`
  - `reduceOr  :: Bit n -> Bit 1`

Capturing numeric relations

- What is the type of bit concatenation?
  `bitConcat :: Bit n -> Bit k -> Bit (n+k)` ?

- BSV uses typeclasses to express these relationships

```
class Add a b c | a b -> c, a c -> b,
                  b c -> a
where { } -- a + b == c

bitConcat :: (Add n k m) =>
             Bit n -> Bit k -> Bit m
```

- Other relations: `Max`, `Mul`, `Div`, `Log`

Functions using numeric relations

```
last      :: (Add 1 m n) =>
              Vector n a -> a

vecConcat :: (Mul m n mn) =>
              Vector m (Vector n a) ->
              Vector mn a

rotateBy  :: (Log n ln) =>
              Vector n a -> UInt ln ->
              Vector n a
```

Numeric type functions

- Typeclasses for numeric relations are not enough
  - Cannot express relationships where typeclass contexts are forbidden (e.g. type synonyms)
  - Cannot capture computations about numeric types internal to a function being checked
- Numeric type functions solve these problems
- `TAdd`, `TMul`, `TMax`, `TDiv`, `TLog` are the function versions of the existing relations
- `TExp` and `TSub` are useful inverse functions

`Bits` typeclass

```
class Bits a n | a -> n where
  pack :: a -> Bit n
  unpack :: Bit n -> a
```

- Canonical way to move between an abstract type and a bit-level representation
- Used pervasively in BSV code (e.g. storing abstract types in registers, FIFOs, etc.)
- `SizeOf` pseudo-function goes from a type in `Bits` to its size

`Bits` instances

```
instance Bits (Bit n) n where
  pack x = x
  unpack x = x

instance (Bits a sa, Add sa 1 sa1) =>
         Bits (Maybe a) sa1 where ...
instance (Bits a sa, Mul sa n san) =>
         Bits (Vector a n) san where ...
instance (Bits a sa, Bits b sb, Max sa sb sz,
          Add sz 1 sz1) =>
          Bits (Either a b) sz1 where ...
```

Problems with numeric types

- Structural unification makes mistakes
  - `(TAdd depth 1) == (TAdd 1 depth)`
    does NOT imply `depth == 1`
- Limited numeric reasoning
  - `(Add a b (TAdd a b))` is a trivial instance
  - Cannot make other "obvious" inferences:
    `(Max a b c) => (Add a _ c)`
    `(Add 8 _ y) => (Add 4 _ y)`
  - More complex reasoning:
    `(Add 7 _ (TMul (TDiv n 4) 8)` is equivalent to
    `(Add 1 _ n)`
- Weak reasoning means the users often second-guess the compiler

Problems with numeric types

```
module mkDivide(PipeFragment#(Tuple2#(FixedPoint#(ni,nf),
                                      FixedPoint#(di,df)),
                                               FixedPoint#(qi,qf)))
  provisos(       Add#(di,df,d_sz),
                  Add#(di,df,TAdd#(di,df)),
                  Add#(2,TSub#(d_sz,2),d_sz),
                  Bits#(FixedPoint#(2,TSub#(TAdd#(di,df),2)), TAdd#(di,df)),
                  Add#(4, _1, TAdd#(32, TSub#(TAdd#(di,df),1))),
                  Add#(1, TSub#(TAdd#(di,df),1), TAdd#(1,TSub#(TAdd#(di,df),1))),
                  Add#(1, TAdd#(1, TSub#(TAdd#(di,df),2)), TAdd#(2,TSub#(TAdd#(di,df),2))),
                  Add#(_2, 11, TAdd#(3,TSub#(TAdd#(di,df),2))),
                  Add#(3, TSub#(TAdd#(di,df),2), TAdd#(3,TSub#(TAdd#(di,df),2))),
                  Add#(1, TAdd#(2,TSub#(TAdd#(di,df),2)), TAdd#(3,TSub#(TAdd#(di,df),2))),
                  Add#(TAdd#(2, TSub#(TAdd#(di,df),2)), 11, TAdd#(4,TAdd#(TSub#(TAdd#(di,df),2),9))),
                  Add#(ni, nf, TAdd#(ni,nf)),
                  Add#(qi, qf, TAdd#(qi,qf)),
                  Add#(_3, qf, 24),
                  Add#(14, qf, TAdd#(14,qf)),
                  Add#(4, _4, TAdd#(32, nf)),
                  Add#(TSub#(TAdd#(di,df),2), 9, TAdd#(TSub#(TAdd#(di,df),2),9)),
                  Arith#(FixedPoint#(4,TAdd#(TSub#(TAdd#(di,df),1),8))),
                  Add#(ni,3,TAdd#(ni,3)),
                  Add#(nf,16,TAdd#(nf,16)),
                  Bitwise#(FixedPoint#(TAdd#(ni,3), TAdd#(nf,16))),
                  Add#(TAdd#(ni,nf), 19, TAdd#(TAdd#(ni,3), TAdd#(nf,16))),
                  Add#(1, _5, ni),
                  Add#(1, _6, TAdd#(ni,3)),
                  Add#(TAdd#(ni,3), TAdd#(nf,16), TAdd#(TAdd#(ni,3), TAdd#(nf,16))),
                  Add#(4, _7, TAdd#(32, TAdd#(nf,16))),
                  Log#(TSub#(d_sz,2),sh_bits),
                  Bits#(FixedPoint#(1,TSub#(TAdd#(di,df),1)), TAdd#(2,TSub#(TAdd#(di,df),2))),
                  Add#(2, TSub#(TAdd#(di,df),1), TAdd#(2,TSub#(TAdd#(di,df),1))),
                  Add#(_8, 11, TAdd#(2,TSub#(TAdd#(di,df),1))),
                  Add#(1, TAdd#(1,TSub#(TAdd#(di,df),1)), TAdd#(2,TSub#(TAdd#(di,df),1))),
                  Add#(TAdd#(1,TSub#(TAdd#(di,df),1)), 11, TAdd#(4,TAdd#(TSub#(TAdd#(di,df),1),8))),
                  Add#(TSub#(TAdd#(di,df),1), 8, TAdd#(TSub#(TAdd#(di,df),1),8))   );
```

`NumEq` typeclass

```
class NumEq a b | a -> b, b -> a where { }
```

- Replace structural unification with introduction of numeric equality constraints
- Discharge those constraints with `NumEq` instances

```
instance (Add a b c) => NumEq (TAdd a b) c
instance (Mul a b c) => NumEq (TMul a b) c
instance (Bits a sa) => NumEq (SizeOf a) sa
...
-- base case: special compiler instance for
-- eliminating type variables
```

What does `NumEq` fix?

- Does not make the mistakes of structural unification
  - Enables some of the “obvious” numeric instances (e.g. `Add a b (TAdd a b)`)
- General framework for handling non-syntactic equalities
- Can give more direct message about unequal numeric types in some cases
- Downside:
  - Must avoid compile-time looping
  - Downstream linting becomes more complex

Improving numeric reasoning

- New built-in instances can improve reasoning:

```
Add <n'> x <exp> =>
Add <n> (TAdd x c) (TMul <m> <exp>)
-- where n' = n `divC` m; c = m * n' – n

Add <n'> x <exp> =>
Add <n> (TQuot x m) (TDiv <exp> <m>)
-- where n' = m * (n – 1) + 1
```

- Simplify greater-than and less-than relationships involving constants and numeric type functions

Problems with numeric types

```
module mkDivide(PipeFragment#(Tuple2#(FixedPoint#(ni,nf),
                                      FixedPoint#(di,df)),
                                              FixedPoint#(qi,qf)))
  provisos(        Add#(di,df,d_sz),
                   Add#(di,df,TAdd#(di,df)),
                   Add#(2,TSub#(d_sz,2),d_sz),
                   Bits#(FixedPoint#(2,TSub#(TAdd#(di,df),2)), TAdd#(di,df)),
                   Add#(4, _1, TAdd#(32, TSub#(TAdd#(di,df),1))),
                   Add#(1, TSub#(TAdd#(di,df),1), TAdd#(1,TSub#(TAdd#(di,df),1))),
                   Add#(1, TAdd#(1, TSub#(TAdd#(di,df),2)), TAdd#(2,TSub#(TAdd#(di,df),2))),
                   Add#(_2, 11, TAdd#(3,TSub#(TAdd#(di,df),2))),
                   Add#(3, TSub#(TAdd#(di,df),2), TAdd#(3,TSub#(TAdd#(di,df),2))),
                   Add#(1, TAdd#(2,TSub#(TAdd#(di,df),2)), TAdd#(3,TSub#(TAdd#(di,df),2))),
                   Add#(TAdd#(2, TSub#(TAdd#(di,df),2)), 11, TAdd#(4,TAdd#(TSub#(TAdd#(di,df),2),9))),
                   Add#(ni, nf, TAdd#(ni,nf)),
                   Add#(qi, qf, TAdd#(qi,qf)),
                   Add#(_3, qf, 24),
                   Add#(14, qf, TAdd#(14,qf)),
                   Add#(4, _4, TAdd#(32, nf)),
                   Add#(TSub#(TAdd#(di,df),2), 9, TAdd#(TSub#(TAdd#(di,df),2),9)),
                   Arith#(FixedPoint#(4,TAdd#(TSub#(TAdd#(di,df),1),8))),
                   Add#(ni,3,TAdd#(ni,3)),
                   Add#(nf,16,TAdd#(nf,16)),
                   Bitwise#(FixedPoint#(TAdd#(ni,3), TAdd#(nf,16))),
                   Add#(TAdd#(ni,nf), 19, TAdd#(TAdd#(ni,3), TAdd#(nf,16))),
                   Add#(1, _5, ni),
                   Add#(1, _6, TAdd#(ni,3)),
                   Add#(TAdd#(ni,3), TAdd#(nf,16), TAdd#(TAdd#(ni,3), TAdd#(nf,16))),
                   Add#(4, _7, TAdd#(32, TAdd#(nf,16))),
                   Log#(TSub#(d_sz,2),sh_bits),
                   Bits#(FixedPoint#(1,TSub#(TAdd#(di,df),1)), TAdd#(2,TSub#(TAdd#(di,df),2))),
                   Add#(2, TSub#(TAdd#(di,df),1), TAdd#(2,TSub#(TAdd#(di,df),1))),
                   Add#(_8, 11, TAdd#(2,TSub#(TAdd#(di,df),1))),
                   Add#(1, TAdd#(1,TSub#(TAdd#(di,df),1)), TAdd#(2,TSub#(TAdd#(di,df),1))),
                   Add#(TAdd#(1,TSub#(TAdd#(di,df),1)), 11, TAdd#(4,TAdd#(TSub#(TAdd#(di,df),1),8))),
                   Add#(TSub#(TAdd#(di,df),1), 8, TAdd#(TSub#(TAdd#(di,df),1),8))   );
```

Problems with numeric types

```
module mkDivide( Divide#(ni, nf, di, df, qi, qf))

  provisos (        Add#(3, df, xi),
                    Add#(a__, qf, 21),
                    Add#(b__, qi, 2),
                    Add#(1, c__, xi),
                    Add#(ni, nf, TAdd#(di, d__)),
                    Add#(ni, df, TAdd#(qi, e__)),
                    Add#(1, h__, TAdd#(qi, nf)),
                    Add#(j__, TAdd#(di, df), TAdd#(TAdd#(qi, e__), nf)),
                    Add#(df, TAdd#(di, d__), TAdd#(TAdd#(qi, e__), nf))
                    );
```

Problems with numeric types

```
module mkDivide( Divide#(ni, nf, di, df, qi, qf))

   provisos (Add#(3, df, xi),

             Add#(a__, qf, 21),
             Add#(b__, qi, 2),

             Add#(1, c__, xi),

             Add#(ni, nf, TAdd#(di, d__)),
             Add#(ni, df, TAdd#(qi, e__)),
             Add#(1, h__, TAdd#(qi, nf)),

             Add#(d__, TAdd#(di, df), TAdd#(TAdd#(qi, e__), nf)),
             Add#(df, TAdd#(di, d__), TAdd#(TAdd#(qi, e__), nf))

             );
```

Open questions – non-instance reasoning

- Given `(Add a b c)` and `(Add c d e)` it is possible to infer `(Add a (TAdd b d) e)`
- Similarly, `(Add 8 x n)` implies `(Add 4 (TAdd x 4) n)`
- How can we capture this robustly and without duplication of reasoning?
  - Substitute away intermediate variables like `c`?
  - Build a graph of numeric relationships?
  - New "`AtMost`" typeclass?
  - Systematic treatment of aliasing?

Open questions – new numeric relations

- Users want new numeric relations like `Min`, `Quot` and `Rem` (and `TMin`, `TQuot` and `TRem`)

- These new relations introduce new identities:

```
TAdd (TMin a b) (TMax a b) == TAdd a b
TMin a (TMin b c) == TMin (TMin a b) c
TAdd (TQuot a b) 1 == TDiv (TAdd a 1) b
```

- One option:

```
type TMin a b = TSub (TAdd a b) (TMax a b)
class Min a b c | a b -> c where { }
instance Min a b (TMin a b) where { }
```

Open questions – post-typechecker complexity

- What should be in our post-typechecking linting?
  - Equality witnesses? (Yes)
  - Transitive equality? (Probably – see System FC)
  - Commutative operations? (Possibly)
  - Other algebraic reasoning? (We hope not)

Conclusions

- BSV's numeric type system is powerful and useful
- Many of its problems can be addressed with a numeric equality typeclass
- Other issues can be addressed with numeric reasoning instances, and perhaps an `AtMost` typeclass
- There are some open engineering challenges in putting this all together