Functions and Types in Bluespec

Nirav Dave
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Every Object in BSV has a type

- Bit-representable types:
  - int, Bool
  - Vector#(3, Bit#(32))
  - Maybe#(FixedPoint#(15,3))
- Functions:
  - function int genVal(Integer x, Bool isPos)

Haskell syntax for types: (Integer -> int)

Inferring Types

- Unlike C, you don't have to write the type of every object in BSV
  - System can infer types
    - if (f(3'b000)) …

Bit#(3)

Bool

f's type: Bit#(3) -> Bool

- To make sure designers understand their libraries, we insist each top-level object is explicitly typed
  - Cannot put a more general type for object
  - This causes a lot of compile errors

Parameteric Polymorphism

```
function identity(x);
   return x;
endfunction
```

Most general type
forall a. a -> a

What is the type of identity?

identity(True) = True

identity(3'd3) = 3'd3

Bool -> Bool?

Bit#(3) -> Bit#(3)?

Augmenting Parametric polymorphism: Provisos

- General types work as long as we don't need use any properties of the generalized type
  - map: (a -> b) -> Vector#(n, a) -> Vector#(n, b)
  - fst: Tuple2#(a,b) -> a
- Not always good enough

```
function t add3(t a, t b, t c) provisos(Arith#(a));
 return (a + b + c);
endfunction
```

Can be any type with a notion of "+"

Setting up a typeclass

- Define typeclass

```
typeclass Arith#(type t);
   function t \+(t a, t b)
   function t \-(t a, t b)
   function t \*(t a, t b) …
endtypeclass
```

- Define instances for each type

```
instance Arith#(Bool);
  \+(x,y)= x || y
  \*(x,y)= x && y
  \-(x,y)= x ^ !y …
endinstance
```

Can also define parameterized instances

(e.g. Arith#(Bit#(n)))

Dealing with Numeric Types

Numeric type parameters

- BSV types also allows *numeric* parameters

```
Bit#(16)              // 16-bit wide bit-vector
Int#(29)              // 29-bit wide signed integers
Vector#(16,Int#(29))  // vector of 16 whose elements
                      // are of type Int#(29)
```

- These numeric types should not be confused with numeric values, even though they use the same number syntax
  - The distinction is always clear from context, i.e., type expressions and ordinary expressions are always distinct parts of the program text

Keeping Sizes Straight

```
function Vector#(n,x) vconcat
    (Vector#(k,x) a, Vector#(m,x) b)
```

- We'd like the property that k + m = n
- How do we express this notion in the type system?
  - Provisos! Add#(k,m,n)

Compiler knows basic arithmetic facts (e.g. Add#(1,1,2) and Add#(a,b,c) => Add#(b,a,c));

Examples w/ provisos

```
function Bit#(nm) packVector(Vector#(n,Bit#(m)) vs)
  provisos (Mul#(n,m,nm));

function Vector#(n1,t) cons(t x, Vector#(n,x) v)
  provisos (Add#(1,n,n1));

function Bit#(m) truncate(Bit#(n) v)
  provisos (Add#(k,m,n));
```

Since we don't use, k any k can be used as long as k + m = n. It just happens that this is unique

Some Issues with Vectors of Registers

Register Syntax

```
Reg#(int) x <- mkReg(0);

rule r(True);
  x._write(x._read() + 1);
endrule
```

- We added Syntax to make BSV easier to read
  - matches verilog syntax

Vectors sometimes cause problems with syntax

```
Vector#(3, Reg#(int)) v
            <- replicateM(mkRegU);

function int f(int x) = x + 3;

v <= map(f, v);
```

This would work if v was a Reg#(Vector#(3,int))

Useful Vector functions

```
function Vector#(n,x) readVReg(Vector#(n,Reg#(x) v);

function Action writeVReg(Vector#(n,Reg#(x)) vr,
                          Vector#(n,x      ) v);
```

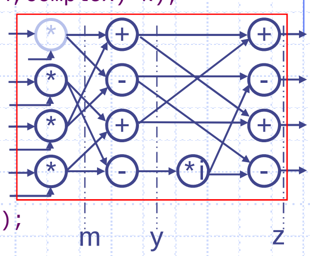
Rewrite as:

```
writeVReg(v,map(f, readVReg(v));
```

Interpreting Functions

Reminder: 4-way Butterfly

```
function Vector#(4,Complex) bfly4
           (Vector#(4,Complex) t,  Vector#(4,Complex) k);

  Vector#(4,Complex) m, y, z;

  m[0] = k[0] * t[0]; m[1] = k[1] * t[1];
  m[2] = k[2] * t[2]; m[3] = k[3] * t[3];

  y[0] = m[0] + m[2]; y[1] = m[0] – m[2];
  y[2] = m[1] + m[3]; y[3] = i*(m[1] – m[3]);

  z[0] = y[0] + y[2]; z[1] = y[1] + y[3];
  z[2] = y[0] – y[2]; z[3] = y[1] – y[3];

  return(z);
endfunction
```



There's a strong correspondence between the dataflow graph of a function and the corresponding logic

Function Arguments: What does this mean?

```
Vector#(3, Bool)  in = inV;
Vector#(3, Bool)  out;

for(Integer x =0 ; x < 3; x=x+1)
  out[x] = in3(in[x]);

outV = out;
```

```
function is3(x) = (x == 3);
outV = mapV3 (is3, inV);
```

- No direct correspondence to HW
- Inline meaning at the call site

Passing in a function as an argument

More Types

All Objects have a type

- Interfaces are just special structs

```
interface Reg#(type a)
  function a _read();
  function Action _write(a x);
endinterface
```

Methods are just functions/values

A Stateless Register

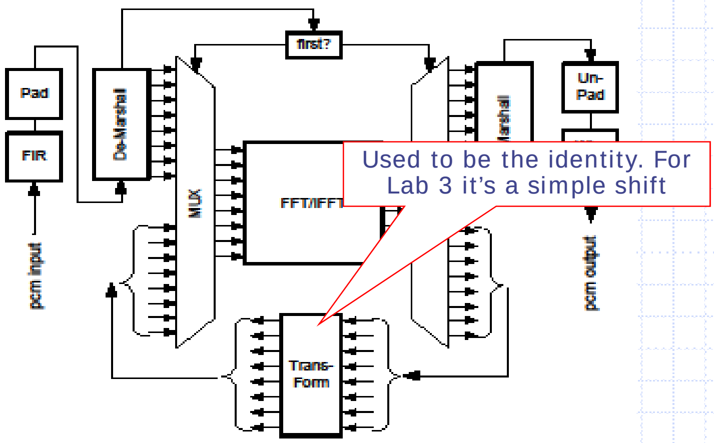
```
module mkConstantReg#(t defVal) (Reg#(t));

 method Action _write(t x);
   noAction;
 endmethod

 method t _read();
   return defVal;
 endmethod

endmodule
```

drop all written values

Just return defVal

The Lab

Lab 3


first?
Pad
FIR
De-Marshal
MUX
FFT/IFFT
Un-Pad
pcm input
pcm output
Trans-Form
Used to be the identity. For Lab 3 it's a simple shift