Advanced features of Bluespec SystemVerilog (BSV)

Mehul Tikekar

Type inference

BSV is a strongly typed language in that everything (variables, rules, functions, modules, interfaces, action blocks, etc.) has a type and all type conversions must be done explicitly. As a result, the compiler can infer the types of most things provided we supply a few types at the top level. For example:

```
interface ExampleIfc;
    interface Put#(UInt#(4)) some_subifc;
    method ActionValue#(UInt#(8)) some_method (UInt#(7) arg);
endinterface

module mkExample(ExampleIfc);
    let f1 <- mkFIFO;    // f1 is of type FIFO#(...)
    let r1 <- mkReg(0);  // r1 is of type Reg#(...)
    let r2 <- mkReg(0);  // r2 is of type Reg#(...)

    interface Put some_subifc = toPut(f1);
        // f1 is of type FIFO#(UInt#(4))

    method some_method(arg) = actionvalue
        r1 <= arg; // r1 is of type Reg#(UInt#(7))
        return r2; // r2 is of type Reg#(UInt#(8))
    endactionvalue;
endmodule
```

Thus, the `let` keyword can be used to avoid specifying any types redundantly. Incidentally, `let` can also unpack tuples: `let {a, b} = some_tuple2`. `let` is however forbidden in the top level context - it must be used only inside a module, function, etc.

Similarly, functions defined within other modules or functions need not specify argument types or return types if the compiler can infer them. That is, `function` `int f1(int x, int y)` can be written simply as `function f1(x, y)`. And what's more, the function is automatically polymorphic. That is,

```
function f(x) = 2*x;
Integer a = 15;
UInt#(4) b = 15;
$display(f(a), f(b)); // Prints 30 14
```

Function magic

Functions are first-class objects

In BSV, functions can be assigned to variables, passed as arguments to other functions, and returned by other functions. **Basically, functions can be used in any context where a variable can be used.** This lets us do things like:

```
function f1(x, y) = x + y;
let g = f1;
int h = g(1, 2); // h is 3
```

Or this:

```
function g1(f, x, y) = f(x, y);
int h = g1(f1, 1, 2); // h is 3 again
```

If a function is a variable, what is its type? In this case, `f1` is of type "a function that expects two ints and returns an int". In BSV, this is written as `function int f(int x1, int x2)`. Knowing this, the function `g1` above can be defined with explicit types as follows:

```
function int g1(function int f1(int a, int b), int x, int y)
  = f1(x, y);
```

Knowing the types of objects is going to help understand latter parts of this text better. Writing the full type every time is cumbersome, so I'll use a shorthand notation:

```
function int f(int x1, int x2) <=> (int, int) -> int
```

Currying

In BSV, `f1(x, y)` can be written instead as `f1(x)(y)`. So, we can evaluate `f1` on `x` and `y` as:

```
let val = f1(x, y);
```

or

```
let val = f1(x)(y);
```

or

```
let fx = f1(x);
let val = fx(y);
```

That is, `f1(x)` is a function (named `fx` in the above example), that expects `y` as an argument and returns `f1(x, y)`. In shorthand, the type of `fx` is `int -> int`. Writing down the types shows an interesting pattern.

Object	Type
`val = f1(x, y)`	`int`
`fx = f1(x)`	`int -> int`
`f1`	`(int, int) -> int`

The type of `f1(x)` being `int -> int` means that the type of `f1` is also `int -> (int -> int)`. **Thus, the function type `(int, int) -> int` can also be written as `int -> (int -> int)`.** This feature is called currying. BSV has two somewhat related functions in its standard library called, confusingly, curry and uncurry. We won't deal with them here.

In BSV, currying works in the reverse too. That is, for a function `f2` defined as

```
function f2(x1);
    function f3(x2);
        ...
    endfunction

    return f3;
endfunction
```

we can replace `f2(x1)(x2)` by `f2(x1, x2)` even though `f2`'s definition calls for only one argument. In shorthand, this simply means that `int -> (int -> int)` can be written as `(int, int) -> int`. Thus, the normal and curried forms are completely equivalent. i.e.

```
f(x, y)           <=> f(x)(y)
(int, int) -> int <=> int -> (int -> int)
```

The parentheses can be dropped without ambiguity to specify `f1`'s type as `int -> int -> int`.

Typeclass and instance

Typeclasses for overloading functions

In BSV, parametrization and typeclasses are two mechanisms to implement overloaded functions. For example, I would like a `pop` function on `FIFO` interfaces that combines `deq` and `first` into one `ActionValue`. This can be written as a function parametrized on `FIFO`'s data type:

```
function ActionValue#(d) pop(FIFO#(d) f)
  = actionvalue
        f.deq;
        return f.first;
    endactionvalue

FIFO#(int) f <- mkFIFO;

rule ...
   let val <- pop(f);
   // instead of let val = f.first; f.deq;
```

If I want to overload it further so it works with both `FIFO` and `FIFOF` interfaces, a typeclass is needed. The idea is to capture the common behavior in the typeclass and define specific instances for different types. This can be done as follows:

```
typeclass FIFOPop#(type ifc_fifo, type t_fifo);
    function ActionValue#(t_fifo) pop(ifc_fifo#(t_fifo) f);
endtypeclass

instance FIFOPop#(FIFO, t);
    function pop(f) = actionvalue
        f.deq;
```

```
        return f.first;
    endactionvalue;
endinstance

instance FIFOPop#(FIFOF, t);
    function pop(f) = actionvalue
        f.deq;
        return f.first;
    endactionvalue;
endinstance
```

Variable number of arguments

Let's say we have a function `f(x, y, z)` where the last two arguments are optional and have a default value of 1 and 2 respectively. That is, we want a function `f1` such that:

```
f1(x)       = f(x, 1, 2)
f1(x, y)    = f(x, y, 2)
f1(x, y, z) = f(x, y, z)
```

We can use typeclasses and currying to implement this. Instead of overloading on the argument type, we will overload on the return type. This is shown for the three cases in the table below:

Normal form	Curried form	Return type of `f1(x)`
`f1(x)`	`f1(x)`	`int`
`f1(x, y)`	`f1(x)(y)`	`int -> int`
`f1(x, y, z)`	`f1(x)(y, z)`	`(int, int) -> int`

This translates to BSV as follows:

```
typeclass F1#(type d);
    function d f1(int x);
endtypeclass

instance F1#(int);
    function int f1(int x) = f(x, 1, 2);
endinstance

instance F1#(function int func(int y));
```

```
    function function int func(int y) f1(int x);
        function int f2(int y) = f(x, y, 2);
        return f2;
    endfunction
endinstance

instance F1#(function int func(int y, int z));
    function function int func(int y, int z) f1(int x);
        function int f3(int y, int z) = f(x, y, z);
        return f3;
    endfunction
endinstance
```

In all the instances, the return types (`int, int -> int, (int, int) -> int`) can be left out for the compiler to infer. Further, in the last two instances, `f1` can be defined in a normal form, rather than the curried form. This makes the code much more readable:

```
typeclass F1#(type d);
    function d f1(int x);
endtypeclass

instance F1#(int);
    function f1(x) = f1(x, 1, 2);
endinstance

instance F1#(function int func(int y));
    function f1(x, y) = f(x, y, 2);
endinstance

instance F1#(function int func(int y, int z));
    function f1(x, y, z) = f(x, y, z);
endinstance
```

Typeclasses and recursion

Typeclasses with recursive instances can be used to implement some interesting ideas.

Arbitrary number of arguments

We would like an `add` function with two or more number of arguments. We'll choose a similar strategy as for variable number of arguments. The only difference is that the base case is now a function with two arguments. Making a similar table as before:

Normal form	Curried form	Type of `add(x1, x2)`
`add(x1, x2)`	`add(x1, x2)`	`int`
`add(x1, x2, x3)`	`add(x1, x2)(x3)`	`int -> int`
`add(x1, x2, x3, x4)`	`add(x1, x2)(x3)(x4)`	`int -> int -> int`

and so on. Thus, the type of `add(x1, x2)` can be recursively defined as `d = int` and `d = int -> d`.

This translates to BSV as:

```
typeclass AddArb#(type d);
    function d add(int x1, int x2);
endtypeclass

instance AddArb#(int);
    // d = int
    function int add(int x1, int x2) = x1 + x2;
endinstance

instance AddArb#(function d1 f(int x)) provisos(AddArb#(d1));
    // d = int -> d
    function function d1 f(int x) add(int x1, int x2);
        function f2(x3) = add(x1 + x2, x3);
        return f2;
    endfunction
endinstance
```

The `AddArb#(d1)` proviso is what really allows for this recursion. Again, the instances can be simplified:

```
instance AddArb#(int);
    function add(x1, x2) = x1 + x2;
endinstance

instance AddArb#(function d1 f(int x)) provisos(AddArb#(d1));
    function add(x1, x2, x3) = add(x1 + x2, x3);
endinstance
```

As an exercise, extend this `add` function to accept one argument (`add(x) = x`) and no arguments (`add = 0`).

Adder tree

This is a more complicated example of a typeclass that implements an adder tree as a pipelined module and as a combinational function. The input is an `n`-long vector of `UInt#(nbits)` elements and in each stage of the tree the partial sums increase in bitwidth by 1. The final result is of type `UInt#(nbits + log2(n))`.

```
interface AdderTree#(numeric type n, numeric type nbits);
    // The interface to the pipelined module
    method Action put_vector(Vector#(n, UInt#(nbits)) vec);
    method ActionValue#(UInt#(TAdd#(TLog#(n), nbits))) get_result;
endinterface

typeclass Adder#(numeric type n, numeric type nbits);
    module mkAdderTree(AdderTree#(n, nbits));

    function UInt#(TAdd#(TLog#(n), nbits)) treeAdd
        (Vector#(n, UInt#(nbits)) vec);
endtypeclass

instance Adder#(2, nbits);
    // Base instance of 2-long vector
    module mkAdderTree(AdderTree#(2, nbits));
        let f <- mkFIFO;
        method put_vector(vec) = f.enq(extend(vec[0]) + extend(vec[1]));
        method get_result = pop(f);
    endmodule

    function treeAdd(vec) = extend(vec[0]) + extend(vec[1]);
endinstance

instance Adder#(n, nbits)
    provisos (Mul#(hn, 2, n), Add#(hn, a__, n),
              Add#(b__, TLog#(hn), TLog#(n)),
              Adder#(hn, nbits));

    // General case
    module mkAdderTree(AdderTree#(n, nbits));
        // two subtrees
        AdderTree#(hn, nbits) g1 <- mkAdderTree;
        AdderTree#(hn, nbits) g2 <- mkAdderTree;
        let f <- mkFIFO;

        rule getResult;
            let res1 <- g1.get_result;
            let res2 <- g2.get_result;
```

```
            f.enq(extend(res1) + extend(res2));
        endrule

        method put_vector(vec) = action
            /* Put the first and second half
            of the vector in the subtrees */
            g1.put_vector(take(vec));
            g2.put_vector(takeTail(vec));
        endaction;

        method get_result = pop(f);
    endmodule

    function treeAdd(vals) = begin
        Vector#(hn, UInt#(nbits)) vals1 = take(vals);
        Vector#(hn, UInt#(nbits)) vals2 = takeTail(vals);
        let res1 = treeAdd(vals1);
        let res2 = treeAdd(vals2);
        (extend(res1) + extend(res2));
    end;
endinstance
```

The module can be instantiated as any normal module. `TreeAdder#(16, 10) adder <- mkAdderTree`. The function can be used on any appropriate vector of `UInts`. In this example, the length of the vector is constrained by the provisos to be a power of 2. Try changing the code to remove this constraint.