Super Advanced BSV*

Andy Wright

October 24, 2014

*Edited for 6.175 Tutorial 6

Super Advanced BSV Scheduling!

Adding Conditions (Guards) to the Body of a Rule

- Sometimes you would like to have a path in a rule be impossible to reach.
- Arvind teaches the `when` syntax in 6.175 to introduce an action in the body of a rule that has a guard when teaching about scheduling.
  - `a when e;` has an action a with an implicit condition e
  - ... but unfortunately, `a when e;` is not part of BSV
  - Instead, lets make our own!

when Module Interface

The interface of a when module:

```
interface When;
    method Action when( Action a, Bool e );
endinterface
```

Values of type Action are statements like "reg <= 0".

when Implementation

The implementation of a when module:

```
module mkWhen( When );
    FIFO#(void) blockingFifo <- mkFIFO;

    method Action when( Action a, Bool e );
        if( !e ) blockingFifo.deq();
        a;
    endmethod
endmodule
```

The problem with this is you need one when module per when statement. Also, this may synthesize an unnecessary FIFO.

Alternate `when` Implementation

Luckily, BSV has an undocumented implementation of `when`.

```
function Action _when_( Bool e, Action a );
```

This function causes a compilation error if the condition `e` comes from an `ActionValue` method of a synthesized module.

Bluespec Schedules

What is a schedule? What information does it contain?

If you look into compiled BSV designs using Bluetcl, you can see the schedule expressed as the following information:

- Order of execution of all rules and methods.
- Urgency relation for rules and methods with conflicts.
  - An urgency relation for rules r1 and r2 says if r1 will fire, r2 will not fire.
  - If two rules are not able to fire in the same cycle due to a conflict, there is an urgency relation saying which one gets priority.

Example schedule:

- Order of Execution: `r1, r2, r3, r4`
- Urgency Relations: `(r1, {r2, r3}), (r2, r4)`
  - If r1 fires, r2 and r3 will not fire
  - If r2 fires, r4 will not fire

This schedule can be obtained for compiled modules using Bluetcl scripts.

Scheduling Annotations

These annotations can be added above rules to add to the schedule, or to assert things about the schedule.

- `(* execution_order = "..." *)`
  - Forces an execution order between rules.
- `(* descending_urgency = "..." *)`
  - Gives user control of direction of urgency relations if needed.
- `(* preempts = "..." *)`
  - Include an urgency relation to make a rule appear to preempt another.
- `(* no_implicit_conditions *)`
  - Asserts that there are no implicit conditions (guards).
  - Creates a compiler error if the assertion is invalid.
- `(* fire_when_enabled *)`
  - Asserts that `WILL_FIRE` == `CAN_FIRE`.
  - Creates a compiler error if the assertion is invalid.

Super Advanced BSV Tuples!

TupleN Type constructors

BSV has built in tuple types:

- `Tuple2#(t1, t2)`
- `Tuple3#(t1, t2, t3)`
- `Tuple4#(t1, t2, t3, t4)`
- `Tuple5#(t1, t2, t3, t4, t5)`
- `Tuple6#(t1, t2, t3, t4, t5, t6)`
- and so on... until you get to 8
- `Tuple8#(t1, t2, t3, t4, t5, t6, t7, t8)`
- There are no 9+ element tuples

Fun Fact

Tuple2 through Tuple7 existed before 2008. Tuple8 was added more recently.

TupleN Value Constructors

BSV has built in functions to construct tuple values:

- `tuple2(v1, v2)`
- `tuple3(v1, v2, v3)`
- `tuple4(v1, v2, v3, v4)`
- `tuple5(v1, v2, v3, v4, v5)`
- `tuple6(v1, v2, v3, v4, v5, v6)`
- `tuple7(v1, v2, v3, v4, v5, v6, v7)`
- `tuple8(v1, v2, v3, v4, v5, v6, v7, v8)`

TupleN Accessor functions

BSV has built in functions to access values within a tuple:

- `tpl_1(x)` – First element
- `tpl_2(x)` – Second element
- `tpl_3(x)` – and so on...
- `tpl_4(x)`
- `tpl_5(x)`
- `tpl_6(x)`
- `tpl_7(x)`
- `tpl_8(x)`

TupleN Pattern Matching

You can use tuples in pattern matching.

```
Tuple3#(Bit#(8),Bool,Bit#(2)) my_tuple = tuple3(1,True,0);
let {x, y, z} = my_tuple;
```

Tuple Quiz

```
typedef Bit#(8) Byte;
Tuple2#(Byte,Byte) x = tuple2(3, 4)
```

What is tpl_1(x)? 3

What is tpl_2(x)? 4

```
typedef Tuple2#(Byte, Byte) DoubleByte;
typedef Tuple2#(DoubleByte, DoubleByte) Word;
Word y = tuple2(tuple2(1, 2), tuple2(3, 4));
```

What is tpl_1(y)? (1, 2)

What is tpl_1(tpl_1(y))? 1

What is tpl_2(y)? 3

Why?

Word is actually Tuple3#(Tuple2#(Byte, Byte), Byte, Byte)

Weird Type Definitions

```
typedef Tuple2#(t1,Tuple2#(t2,t3)) Tuple3#(t1,t2,t3)
typedef Tuple2#(t1,Tuple2#(t2,Tuple2#(t3,t4)))
          Tuple4#(t1,t2,t3,t4)
...
typedef Tuple2#(t1,Tuple2#(t2,Tuple2#(t3,Tuple2#(t4,
          Tuple2#(t5,Tuple3#(t6,Tuple2#(t7,t8)))))))
          Tuple8#(t1,t2,t3,t4,t5,t6,t7,t8)
```

This may not be exactly how they are implemented in BSV, but this is how they behave.

Weird Pattern Matching

Pattern matching can get weird:

```
Tuple3#(Bit#(8),Bool,Bit#(2)) my_tuple = tuple3(1,True,0);
let {x, y} = my_tuple;
// x == 1
// y == tuple2(True, 0)
// tpl_1(tuple2(x,y)) == x
// tpl_2(tuple2(x,y)) != y
```

There are some benefits to this though...

TupleN Polymorphism

Using a Typeclass

Lets say you want an increment function to add one to each entry in a Tuple.

```
typeclass CanIncrement#(type t);
    function t increment(t x);
endtypeclass
```

You could create an instance for each size of tuple, but that would take a lot of work.

- Instead, you will have instances of this typeclass for `Tuple2#(t1,t2)` and for `t`.

TupleN Polymorphism

Instances

Here is your instance of CanIncrement for tuples:

```
instance CanIncrement#(Tuple2#(t1,t2))
      provisos(Arith#(t1), CanIncrement#(t2));
  function Tuple2#(t1,t2) increment(Tuple2#(t1,t2) t);
    let {x, y} = t;
    return tuple2(x+1, increment(y));
  endfunction
endinstance
```

And here is your instance of CanIncrement for non-tuples:

```
instance CanIncrement#(t) provisos(Arith#(t));
  function t increment(t x);
    return x + 1;
  endfunction
endinstance
```

With these, you can increment all types of tuples!

Super Advanced BSV Functions!

Curried Functions

Assume a function f(x,y) where the type of f is

```
f   :: (Integer, Integer) -> Integer
```

The curried form of this function is fc where the type of fc is

```
fc  :: Integer -> (Integer -> Integer)
```

fc(x) produces a function fcx where the type is

```
fcx :: Integer -> Integer
```

Using the curried function fc(x)(y) is the same as f(x,y).

Curried Functions in BSV

All functions in BSV are curried.

```
function Integer add(Integer x, Integer y);
    return x + y;
endfunction
let add1 = add(1);
add1(5) -> 6
add(1,5) -> 6
add(1)(5) -> 6
```

Defining a Function from its Lookup Table

```
function Bit#(1) generic_op(Bit#(4) table, Bit#(1) a, Bit#(1) b);
    let index = {a, b};
    return table[index];
endfunction

let and_f = generic_op(4'b1000);
let or_f = generic_op(4'b1110);
let xor_f = generic_op(4'b0110);
```

`and_f`, `or_f`, and `xor_f` are all functions that take in two `Bit#(1)` inputs and output a `Bit#(1)`.

This can be used as an implementation of `unpack` to convert `Bit#(4)` to a function.

Converting a Function to Bits

```
function Bit#(4) op_to_bits(function Bit#(1) f(Bit#(1) a,
    Bit#(1) b));
     return {f(1,1), f(1,0), f(0,1), f(0,0)};
endfunction

let and_f_bits = op_to_bits(and_f);
let or_f_bits = op_to_bits(or_f);
let xor_f_bits = op_to_bits(xor_f);
```

Now we can create an instance of a typeclass for a function.

Custom Instance of Bits

```
typedef struct {
    function Bit#(1) f(Bit#(1) a, Bit#(1) b);
} BinaryBitOp;

instance Bits#(BinaryBitOp, 4);
    function Bit#(4) pack(BinaryBitOp op);
        return op_top_bits(op.f);
    endfunction
    function BinaryBitOp unpack(Bit#(4) x);
        BinaryBitOp op;
        op.f = generic_op(x);
        return op;
    endfunction
endinstance

// This is now valid!
Reg#(BinaryBitOp) op <- mkReg(BinaryBitOp{f: or_f});
```

Custom Instance of FShow

```
instance FShow#(BinaryBitOp);
    function Fmt fshow( BinaryBitOp op );
        return $format( "(table: %b)", pack(op) );
    endfunction
endinstance

Reg#(BinaryBitOp) op <- mkReg(BinaryBitOp{f: or_f});
// This is now valid!
$display("op = ", fshow(op));
```