Bluespec-2: Types

Arvind

Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology
Example: Shifter

Goal: implement: \( y = \text{shift} \ (x, s) \)

where \( y \) is \( x \) shifted by \( s \) positions. Suppose \( s \) is a 3-bit value.

Strategy:

- Shift by \( s = \)
  
  \begin{align*}
  &\text{shift by} & 4 \ (=2^2) & \text{if } s[2] \text{ is set,} \\
  &\text{and by} & 2 \ (=2^1) & \text{if } s[1] \text{ is set,} \\
  &\text{and by} & 1 \ (=2^0) & \text{if } s[0] \text{ is set}
  \end{align*}

- A shift by \( 2^j \) is trivial: it’s just a “lane change” made purely with wires
Cascaded Combinational Shifter

A family of functions

```c
typedef struct {
  int x;
  int s;
} Pair;

function Pair step_j (Pair sx);
{
  return ((sx.s[j]==0) ? sx :
            Pair{ s: sx.s, x: sh_k(sx.x)});
}
endfunction

function int shifter (int s, int x);
{
  Pair sx0, sx1, sx2;
  sx0 = step_0(Pair{ s: s, x: x });
  sx1 = step_1(sx0);
  sx2 = step_2(sx1);
  return (sx2.x);
}
endfunction
```

where $k=2^j$
Asynchronous pipeline with FIFOs (regs with interlocks)

```
rule stage_0 (True);
    Pair sx0 = fifo0.first(); fifo0.deq(); fifo1.enq(step_0(sx0));
endrule

rule stage_1 (True);
    Pair sx1 = fifo1.first(); fifo1.deq(); fifo2.enq(step_1(sx1));
endrule

rule stage_2 (True);
    Pair sx2 = fifo2.first(); fifo2.deq(); fifo3.enq(step_2(sx2));
endrule
```
Required simultaneity

If it is *necessary* for several actions to happen together, (i.e., indivisibly, atomically)

Put them in the same rule!
Synchronous pipeline (with registers)

```
rule sync-shifter (True);
  sx1 <= step_0(sx0);
  sx2 <= step_1(sx1);
  sx3 <= step_2(sx2);
endrule
```

sx1, sx2 and sx3 are registers defined outside of the rules

```
Reg#(Pair) sx_i <- mkRegU();
```

Will it start properly?
Will it leave some values in the pipe?
Discussion

In the synchronous pipeline, we compose actions in parallel
- All stages move data simultaneously, in lockstep (atomic!)

In the asynchronous pipeline, we compose rules in parallel
- Stages can move independently (each stage can move when its input fifo has data and its output fifo has room)
- If we had used parallel action composition instead, all stages would have to move in lockstep, and could only move when all stages were able to move

Your design goals will suggest which kind of composition is appropriate in each situation
Expressions vs. Functions

- A function is just an abstraction of a combinational expression
- Arguments are inputs to the circuit
- The result is the output of the circuit

```
function int discr (int a, int b, int c);
    return b*b - 4*a*c;
endfunction
```

expression

```
const 4
```

```
a c
```

```
x
```

```
discr
```

```
x
```

```
-
```

```
b
```

expression
Function Application

- Instantiates combinational hardware of the function body
- Connects the body to argument expressions

```plaintext
function int
discr (int a, int b, int c);
    return b*b – 4*a*c;
endfunction
```

No runtime allocation of stack frames or passing of arguments; only meaningful for static elaboration.
Types and type-checking

BSV is strongly-typed

- Every variable and expression has a type
- The Bluespec compiler performs strong type checking to guarantee that values are used only in places that make sense, according to their type

This catches a huge class of design errors and typos at compile time, i.e., before simulation
What is a Type?

A type describes a set of values

Types are orthogonal (independent) of entities that may carry values (such as wires, registers, …)
  - No inherent connection with storage, or updating

This is true even of complex types
  - E.g., `struct { int ..., Bool ...}`
  - This just represents a set of pairs of values, where the first member of each pair is an int value, and the second member of each pair is a Bool value
SV notation for types

Some types just have a name

int, Bool, Action, ...

More complex types can have parameters which are themselves types

FIFO#(Bool) // fifo containing Booleans
Tuple2#(int,Bool) // pair of int and Boolean
FIFO#(Tuple2#(int,Bool)) // fifo of pairs of int and Boolean

Type names begin with uppercase letter

- Exceptions: ‘int’ and ‘bit’, for compatibility with Verilog

bit[15:0] is the same as Bit#(16)
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
Common scalar types

- **Bool**
  - Booleans

- **Bit#(n)**
  - Bit vectors, with a width n bits

- **Int#(n)**
  - Signed integers of n bits

- **UInt#(n)**
  - Unsigned integers of n bits

- **Integer**
  - Unbound integers; has meaning only during static elaboration
Some Composite Types

- **Enumerations**
  - Sets of symbolic names

- **Structs**
  - Records with fields

- **Tagged Unions**
  - Unions, made “type-safe” with tags
Types of variables

Every variable has a *data type*:

```作物
bit[3:0] vec;  // or   Bit#(4) vec;
vec = 4'b1010;
Bool cond = True;
typedef struct  
     {Bool b; bit[31:0] v;} Val;
Val x = Val {b: True, v: 17};
```

BSV will enforce proper usage of values according to their types

- You can't apply “+” to a struct
- You can’t assign a boolean value to a variable declared as a struct type
"let" and type-inference

- Normally, every variable is introduced in a declaration (with its type)
- The "let" notation introduces a variable with an assignment, with the compiler inferring its correct type

```python
let vec = 4'b1010; // bit[3:0] vec = …
let cond = True;  // Bool cond = …;
```

- This is typically used only for very "local" temporary values, where the type is obvious from context
Type synonyms with typedef

typedef is used to define a new, more readable synonym for an existing type

```c
typedef existingType NewType;
typedef int Addr;
typedef bit [63:0] Data;
typedef bit [15:0] Halfword;
typedef Bool Flag;
```

Reminder: type names begin with uppercase letter!

Type synonyms do not introduce new types. For example, `Bool` and `Flag` can be intermixed without affecting the meaning of a program.
Enumeration

typedef enum {Red; Green; Blue} Color;
    Red = 00, Green = 01, Blue = 10

typedef enum {Waiting; Running; Done} State;
    Waiting = 00, Running = 01, Done = 10

typedef enum {R0; R1; R2; R3} RName;
    R0 = 00, R1 = 01, R2 = 10, R3 = 11

Enumerations define new, distinct types:

- Even though, of course, they are represented as bit vectors
Type safety

Type checking guarantees that bit-vectors are consistently interpreted.

If a Color and a State are different types, a Color cannot accidentally be used as a State:

```
Reg#(Color) c <- mkRegU();
Reg#(State) s <- mkRegU();
...
\_\_\_c\_\_
\_s\_<-\_c\_;
```
typedef Bool FP_Sign;
ytypedef Bit#(2) FP_RS;

typedef struct {
    FP_Sign sign; // sign bit
    Bit#(ee) exp; // exponent
    Bit#(ss) sfd; // significand
    FP_RS rs;     // round and sticky bit
} FP_I#(type ee, type ss);

// exponent and significand sizes are
// *numeric* type parameters
Bit interpretation of structs

<table>
<thead>
<tr>
<th>sign</th>
<th>sfd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ee</td>
</tr>
<tr>
<td>exp</td>
<td>ss</td>
</tr>
<tr>
<td>2</td>
<td>rs</td>
</tr>
</tbody>
</table>
typedef union tagged {
    struct {RName dst; RName src1; RName src2;}  Add;
    struct {RName cond; RName addr;}              Bz;
    struct {RName dst; RName addr;}             Load;
    struct {RName dst; Immediate imm;}        AddImm;
    ...}
} Instr;
The Maybe type

The Maybe type can be regarded as a value together with a “valid” bit

typedef union tagged {
    void Invalid;
    t Valid;
} Maybe#(type t);

Example: a function that looks up a name in a telephone directory can have a return type Maybe#(TelNum)

- If the name is not present in the directory it returns tagged Invalid
- If the name is present with number x, it returns tagged Valid x
The Maybe type

The `isValid(m)` function
- returns `True` if `m` is tagged `Valid x`
- returns `False` if `m` is tagged `Invalid`

The `fromMaybe(y,m)` function
- returns `x` if `m` is tagged `Valid x`
- returns `y` if `m` is tagged `Invalid`
When defining new types, by attaching a “deriving” clause to the type definition, we let the compiler automatically create the “natural” definition of certain operations on the type.

```c
typedef struct { ... } Foo
deriving (Eq);
```

Eq generates the “==” and “!=” operations on the type via bit comparison.
Deriving Bits

typedef struct { ... } Foo
  deriving (Bits);

- Automatically generates the “pack” and “unpack” operations on the type (simple concatenation of bit representations of components)
- This is necessary, for example, if the type is going to be stored in a register, fifo, or other element that demands that the content type be in the Bits typeclass
- It is possible to customize the pack/unpack operations to any specific desired representation
Pattern-matching

Pattern-matching is a more readable way to:
- test data for particular structure and content
- extract data from a data structure, by binding “pattern variables” (.variable) to components

\[
\text{case (m) matches } \\
\text{tagged Invalid : return 0; } \\
\text{tagged Valid .x : return x; } \\
\text{endcase}
\]

\[
\text{if (m matches (Valid .x) && (x > 10)) ...}
\]

The && is a conjunction, and allows pattern-variables to come into scope from left to right
**Example: CPU Instructions Operands**

```
typedef union tagged {
    bit [4:0] Register;
    bit [21:0] Literal;
    struct {
        bit [4:0] regAddr; bit [4:0] regIndex;
    } Indexed;
} InstrOperand;
```

```
case (operand) matches
    tagged Register .r : x = rf[r];
    tagged Literal .n : x = n;
    tagged Indexed {regAddr: .ra, regIndex: .ri } :
        begin Iaddress a = rf[ra]+rf[ri];
            x = mem.get(a);
        end
endcase
```
Other types in BSV

- **String**
  - Character strings

- **Action**
  - What rules/interface methods do

- **Rule**
  - Behavior inside modules

- **Interface**
  - External view of module behavior

Useful during static elaboration
Instantiating interfaces and modules

The SV idiom is:

- Instantiate an interface
- Instantiate a module, binding the interface
  - Note: the module instance name is generally not used, except in debuggers and in hierarchical names

BSV also allows a shorthand:

\[
\text{FIFO\#(DataT) inbound1 <- mkSizedFIFO(fifo\_depth);}
\]

We will only use the shorthand
Module Syntax

Module declaration

```module mkGCD (I_GCD#(t));
...
endmodule```

Module instantiation

```interface I_GCD#(int) gcd <- mkGCD ();```

Module Syntax

Module declaration

```module mkGCD (I_GCD#(t));
...
endmodule```

Module instantiation

```interface I_GCD#(int) gcd <- mkGCD ();```
A rule is *declarative* specification of a state transition

- An action guarded by a Boolean condition

```
rule ruleName (<predicate>);
  <action>
endrule
```
Rule predicates

- The rule predicate can be any Boolean expression
  - Including function calls and method calls

- Cannot have a side-effect
  - This is enforced by the type system

- The predicate must be true for rule execution
  - But in general, this is not enough
  - Sharing resources with other rules may constrain execution
Next Lecture

Static elaboration and architectural exploration