bluespec

BSV Training

Section: Types and type-checking 1

Role of types; syntax of types and type expressions; enums, structs, vectors, numeric types; polymorphic types

(Types are a major topic; here we only touch on basics, more in later sections)

www.bluespec.com

The role of types in modern programming languages

Most modern programming languages make strong use of *data types* to raise the level of abstraction and for correctness

- Types are an abstraction: ultimately, all computation, whether in SW or HW, is done on bits, but it is preferable to think in terms of integers, fixed point numbers, floating point numbers, booleans, symbolic state names, ethernet packets, IP addresses, employee records, vectors, and so on
- *Strong type-checking* is used to eliminate unintentional "mis-interpretation" of bits, such as taking the square of symbolic state, subtracting two IP addresses, indexing into an employee record, and so on

bluespec

Types in BSV

- BSV has basic scalar types just like Verilog
- BSV has SystemVerilog type mechanisms like typedefs, enums, structs, tagged unions, arrays and vectors, interface types, type parameterization, polymorphic types
- BSV also has types for static entities like functions, modules, interfaces, rules, and actions
  - (so you can write static-elaboration functions that compute with such entities)
- BSV has very powerful *systematic user-defined overloading*—typeclasses and instances (more powerful than C++)
  - This is used heavily by advanced users
- Type-checking in BSV is *very* strict
  - Even registers are strongly-typed
  - No silent extensions and truncations
  - Typical anecdote (observed by BSV and Haskell programmers, which have the same type system):
    "if it gets through the type-checker, it just works"

Syntax of types: Type Expressions

BSV uses SystemVerilog's notation for parameterized types

```
Type ::= TypeConstructor #(Type1, ..., TypeN)
       | TypeConstructor          // special case when N=0)
```

i.e., a type expression is a type constructor applied to zero or more other types. In the special case where it is applied to zero other types, the #() part can be omitted. Examples:

Type	*Comments*
Integer	Unbounded signed integers (static elaboration only)
Int#(18)	18-bit signed integers Note: 'int' is a synonym for Int#(32)
UInt#(42)	42-bit unsigned integers
Bit#(23)	23-bit bit vectors Note: 'bit[15:0]' is a synonym for Bit#(16)
Bool	Booleans, with constants True and False
Reg#(UInt#(42))	Interface of register that contains 42-bit unsigned integers
Mem#(A,D)	Interface of memory with address type A and data type D
Server#(Rq,Rsp)	Interface of server module with request type Rq and response type Rsp)

Note uppercase first letter in type names

Typedefs, enums, and structs

```
typedef  Bit #(32) Word;
typedef  Bit #(32) Addr;
typedef  Bit #(32) Data;

typedef  Bit#(4) RegName;
```

Simple typedefs (left) are just synonyms for readability; all these types are equivalent.

Enum and struct typedefs (below) define new types, not equivalent with any other type. (So, type-checking prevents misuse.)

```
typdef enum { Noop, Add, Bz, Ld, St } Opcode deriving (Bits, Eq);
```

```
typedef struct {
   Opcode   op;
   RegName  dest;
   RegName  src1;
   RegName  src2;
} Instr
  deriving (Bits);
```

```
typedef struct {
   Opcode    op;
   RegName   dest;
   Bit #(32) v1;
   Bit #(32) v2;
} DecodedInstr
  deriving (Bits);
```

"deriving (Eq)" tells bsc to pick a "natural" equality comparison operator for this type.

"deriving (Bits)" tells bsc to pick a "natural" bit-representation for this type.

(detailed treatment in a later section)

bluespec

structs

Enum and struct types are "first class" types. They can be stored in state elements. They can be passed as method and function arguments and results, etc. (unlike C, but like C++).

```
Reg #(Opcode) rg_op <- mkReg (Noop);
FIFOF #(DecodedInstr) buf <- mkFIFOF;
```

Strongly typed: they can never contain values of any other type, even if they are represented in the same number of bits.

Like C/C++, you can declare a variable with a struct type, and incrementally assign its members (fields). However, we often directly build entire struct values using struct expressions:

```
rule fetch (buf.notStall (instr));
   let di = DecodedInstr { op:   instr.op;
                           dest: instr.dest;
                           v1:   rf.sel1 (instr.src1);
                           v2:   rf.sel2 (instr.src2); };
   buf.enq (di);
   pc <= pc + 1;
endrule: fetch
```


bluespec

Vectors

We commonly use Vectors to express repeated structures.
In any package where we use them, we must first import the Vector package:

```
import Vector :: *;
```

Vectors are just type constructors, like any other. In particular, they can contain any types (including other Vectors):

```
interface EHR #(numeric type n, type t);
   interface Vector #(n, Reg #(t)) ports;
endinterface

typedef  Vector #(10, Vector #(5, Int #(16)))  Matrix;
```

Vectors are indexed with the usual square bracket "[ ]" notation:

```
Int #(5) new_val = extend (ctr.ports [p]) + extend (delta);
if (new_val > 7) ctr.ports [p] <= 7;
else if (new_val < -8) ctr.ports [p] <= -8;
else ctr.ports [p] <= truncate (new_val);
```


bluespec

Numeric types

- Some type constructors take *numeric types* in certain type-parameter positions. Examples:
  - 18-bit signed integers `Int #(18)`
  - Vector of sixteen 42-bit unsigned integers `Vector #(16, UInt #(42))`
- In a position where a numeric type is expected, you can provide:
  - Literal numeric values: 18, 16, 42, ...
  - Numeric type expressions: `TAdd #(18,16), TMul #(2,32), TLog #(19)`
- Although these have superficial similarity to numeric values and numeric value expressions:
  18 16 42 18+16 2*32 log2(19), ...
  they are not the same!
- Specifically, numeric types and type expressions are much weaker than full-blown numeric values and arithmetic expressions because:
  - Type-checking is performed in a separate, earlier phase of the compiler before any numeric value expression evaluation
  - Type-checking needs to be resolved statically

bluespec

Polymorphic types

- Any type-parameter in a type expression can be a *type variable* (identifier beginning with a lower-case letter). Examples:
  - *n*-bit signed integers `Int #(n)`
  - Vector of *m* elements, each of of type *t* `Vector #(16, UInt #(t))`
- This allows for writing highly parameterized designs
- [C++ users: BSV type variables are like *template types*]

bluespec

End

Questions?
Join online forums at www.bluespec.com, and ask your question,
or send an e-mail to support@bluespec.com