Übung zur Vorlesung Architekturen und Entwurf von Rechnersystemen

Prof. Dr-Ing. A. Koch
Jaco Hofmann, MSc.

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 16/17

Übungsblatt 3 - Lösungsvorschlag

Der sprachliche Unterbau von Bluespec ist Haskell. Dementsprechend lassen sich viele Techniken, die aus Haskell bekannt sind, auch auf Bluespec anwenden. Eine dieser Techniken, QuickCheck, wird in dieser Übung vorgestellt. QuickCheck erlaubt es Funktionen automatisiert zu testen. Dabei kann man Eigenschaften der zu testenden Funktion abstrakt beschreiben und QuickCheck kümmert sich um das Erstellen der Testfälle.

Eine Bluespec Variante von QuickCheck heißt BlueCheck. Clonen Sie das Repository von Github und kopieren Sie das Bluespec Paket `BlueCheck.bsv` in Ihr Codeverzeichnis. Einige Beispiele finden Sie in den Paketen `SimpleExample` und `StackExample`.

In dieser Übung lernen Sie die Verwendung von BlueCheck zum Testen von Bluespec Modulen kennen.

Aufgabe 3.1 Testen gegen Bedingungen

Bluespec erlaubt es Tests von Bedingungen durchzuführen. Dies kann zum Beispiel zum Überprüfen von arithmetischen Operationen verwendet werden (Beispiel von BlueCheck SimpleExample):

```
1  module [BlueCheck] mkArithSpec ();
2    function Bool addComm(Int#(4) x, Int#(4) y) =
3      x + y == y + x;
4
5    function Bool addAssoc(Int#(4) x, Int#(4) y, Int#(4) z) =
6      x + (y + z) == (x + y) + z;
7
8    function Bool subComm(Int#(4) x, Int#(4) y) =
9      x - y == y - x;
10
11   prop("addComm"  , addComm);
12   prop("addAssoc" , addAssoc);
13   prop("subComm" , subComm);
14 endmodule
15
16 module [Module] mkArithChecker ();
17   blueCheck(mkArithSpec);
18 endmodule
```

Bauen Sie das Modul `mkArithChecker` und lassen Sie es im Simulator laufen. BlueCheck wird bei der Ausführung automatisch Parameter für die Funktionen generieren und versuchen, die Aussagen zu widerlegen.

Erweitern Sie die Testfälle um sinnvolle Tests für Multiplikation und Division.

Lösungsvorschlag

Das Modul wird in entsprechender Art und Weise um drei weitere Spezifikationsfunktionen erweitert:

```
1  package StrangeModule;
2
3  import BlueCheck :: *;
4
5  module [BlueCheck] mkArithSpec ();
```

```
6       function Bool addComm(Int#(4) x, Int#(4) y) =
7           x + y == y + x;
8
9       function Bool addAssoc(Int#(4) x, Int#(4) y, Int#(4) z) =
10          x + (y + z) == (x + y) + z;
11
12      function Bool oneNeutralMul(Int#(4) x) =
13          x * 1 == x;
14
15      function Bool zeroTimesX(Int#(4) x) =
16          x * 0 == 0;
17
18      function Bool divIsMostlyNotComm(Int#(4) x, Int#(4) y) =
19          x == 0 || y == 0 || (x == y) || (-x == y) || x / y != y / x;
20
21      prop("addComm"  , addComm);
22      prop("addAssoc" , addAssoc);
23      prop("oneNeutralMul" , oneNeutralMul);
24      prop("zeroTimesX" , zeroTimesX);
25      prop("divIsMostlyNotComm" , divIsMostlyNotComm);
26  endmodule
27
28  module [Module] mkArithChecker ();
29      blueCheck(mkArithSpec);
30  endmodule
31
32  endpackage
```

Aufgabe 3.2 Die FIFO

Für die nächste Teilaufgabe entwickeln Sie eine FIFO auf der Basis eines Ringspeichers. Die FIFO soll 16 Elemente aufnehmen können. Benutzen Sie als Basis des Stacks einen `Vector#(16, Reg#(Int#(16)))`. Verwenden Sie folgendes Interface, denken Sie an passende Guards, um das Überschreiben von Werten zu verhindern.

```
1  interface FIFO
2    method Action                 put(Int#(16) e); // Put Element on FIFO
3    method ActionValue#(Int#(16)) get(); // Get Element from FIFO
4  endinterface
```

Lösungsvorschlag

```
1   module mkMyFIFO(MyFIFO);
2           Reg#(UInt#(4)) writePntr <- mkReg(0);
3           Reg#(UInt#(4)) readPntr <- mkReg(0);
4
5           Vector#(16, Reg#(Int#(16))) buffer <- replicateM(mkRegU());
6
7           method Action put(Int#(16) e) if((writePntr + 1) != readPntr);
8                   writePntr <= writePntr + 1;
9                   buffer[writePntr] <= e;
10          endmethod
11
12          method ActionValue#(Int#(16)) get() if(readPntr != writePntr);
13                  readPntr <= readPntr + 1;
14                  return buffer[readPntr];
```

```
15          endmethod
16    endmodule
```

Aufgabe 3.3 FIFO testen

Als weitere Möglichkeit zum Testen von Bluespec Modulen erlaubt es Bluespec, eine Referenzimplementierung (auch z.B. in C++ geschrieben) mit der Hardwareimplementierung zu vergleichen. Testen Sie die FIFO aus Aufgabe 3.2 mit Hilfe von BlueCheck gegen die von Bluespec bereitgestellte `FIFO`:

```
1  FIFO#(Int#(16)) goldenFIFO <- mkSizedFIFO(16);
```

Sie können dafür BlueCheck anweisen, die Methoden der Module zu vergleichen. Für einen Stack könnte so ein Modul folgendermaßen aussehen:

```
1  module [BlueCheck] checkStack ();
2    /* Specification instance */
3    Stack#(8, Bit#(4)) spec <- mkStackSpec();
4
5    /* Implmentation instance */
6    Stack#(8, Bit#(4)) imp <- mkBRAMStack();
7
8    equiv("pop"    , spec.pop    , imp.pop);
9    equiv("push"   , spec.push   , imp.push);
10   equiv("isEmpty", spec.isEmpty, imp.isEmpty);
11   equiv("top"    , spec.top    , imp.top);
12 endmodule
```

Erstellen Sie eine randomisierte Testbench für Ihre FIFO.

Lösungsvorschlag

```
1   module [BlueCheck] mkFIFOSpec ();
2           FIFO#(Int#(16)) spec <- mkSizedFIFO(16);
3           MyFIFO impl <- mkMyFIFO();
4
5           function ActionValue#(Int#(16)) pop(FIFO#(Int#(16)) e);
6                   actionvalue
7                           e.deq();
8                           return e.first();
9                   endactionvalue
10          endfunction
11
12          equiv("put", spec.enq, impl.put);
13          equiv("get", pop(spec), impl.get);
14  endmodule
```