ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Dipl.-Inf. Achim D. Brucker
Dr. Burkhart Wolff

Computer-supported Modeling and Reasoning
http://www.infsec.ethz.ch/education/permanent/csmr/

(rev. 16802)
Submission date: –

HOL: Using Specifications for Code Generation and Testing

This exercises describes two advanced techniques for using formal specifications: *code generation* and *random testing*.

The former is a viable approach to achieve correct functional programs and fast evaluation of complex expressions, the latter may be used for early validations of definitions and formulas.

1 More on Isabelle

1.1 Isabelle's Code Generator

Isabelle has an own code generator that attempts to convert many constructs occurring in a specification (such as **primrec** or **datatype** definitions) into SML code. Code generation out of verified theories for efficient datatype implementations is a viable approach to achieve correct, non-trivial (functional) programs with Isabelle. For example, you can generate code for the term "foldl op + (0::int) [1,2,3,4,5]" and store it in the file `test.sml` via

generate_code

```
generate_code ("test.sml")
  test = "foldl op + (0::int) [1,2,3,4,5]"
```

The code generator can be configured both in more correctness oriented as well as pragmatic ways; it is possible, for example, to map the datatype nat on code resulting from the datatype definition in the theory Nat (thus on the free datatype generated by 0 and Suc) or simply on the SML-datatype int (thus reusing the machine integers based on two's complement representation).

Theories can contain highly generic function definitions that are not representable in a target programming language for a number of reasons:

1. a function may simply be not computable,
2. a function may have a type that is not representable in the target language.

An example for the former is a function definition involving a Hilbert-operator, an example for the latter is isord ('a :: ord) ('a tree) which is not representable in the SML type system but could be — in principle — represented in Haskell (note, however, that isord ('a :: order) ('a tree) could not even be represented in Haskell). In practice, the types of formulas to be converted into code must be sufficiently instantiated when configuring the code generator for a theory. You have mainly three options for configuring the code generator:

types_code

1. associate type constructors with specific SML code, e.g.:

```
types_code
  "*"     ("(_ */ _)")
```

consts_code

2. associate constants with specific SML code, e.g.:

```
consts_code
  "Pair"     ("(_,/ _)")
```

[code]

3. register theorems for code generation. This can be done using the **declare** statement, e.g.

   declare less_Suc_eq [code]

   or the code attribute:

   lemma [code]: "((n::nat) < 0) = False" **by**(simp)

   The used theorem should be either an equation (with only constructors and distinct variables on the left-hand side) or a horn-clause (in the same format as introduction rules of inductive definitions). The latter should denoted by using [code ind].

[code ind]

Finally note, if you omit the ("filename") part of the generate_+code statement, the generated code will be immediately available within Isabelle's ML-environment.

1.2 Quickcheck

Inspired by the success of random testing tools (e.g. Quickcheck for Haskell) a similar mechanism for testing lemmas was build into Isabelle: the **quickcheck** command. For example, if we try to prove

quickcheck

```
lemma rev_append: "rev (xs @ ys) = rev xs @ rev ys"
```

we will have a hard day (caused by a simple typo). Now we can try to find a counter example:

```
lemma rev_append: "rev (xs @ ys) = rev xs @ rev ys"
quickcheck
```

Doing this, Isabelle will respond with:

```
Counterexample found:
xs = [0]
ys = [1]
```

Thus our lemma does not even hold for lists of length one. After fully understanding why this assignment is a counter-example, we can reformulate our lemma:

```
lemma rev_append: "rev (xs @ ys) = rev ys @ rev xs"
```

and prove it.

Note that **quickcheck** uses internally the code generator which means that **quickcheck** can only be used if the code generator is already configured correctly!

2 Exercises

2.1 Exercise 49

Create a version of your AVL tree specification that works over integers, e.g., *insert* should have the type

```
consts
  insert :: "int ⇒ tree ⇒ tree"
```

and use it for code generation. Store your SML program in a file `avl.sml`. Create a file `avl-test.sml` with the following content:

```
Control.Print.printDepth := 100;  (* only for sml/NJ *)
Control.Print.printLength := 100; (* only for sml/NJ *)

use "avl.sml";
val elements = [1,5,3,4,8,2,4,6];
val t = foldl (fn (e,t) => insert e t) ET elements;
```

Now start open a shell (i.e., in a xterm) and start the SML Interpreter by typing SML and load your file by executing `use "avl-test.sml"`. Try to understand the shown tree representation and validate that your code produced a correct AVL tree with the elements $1, 2, 3, 4, 5, 6, 8$. Note, that 4 should be only stored once in your tree.

Hints:

- For datatype nat, please write Suc(n) instead of 1+n.
- The code generator will need some hints for the polymorphic max function. Therefore prove the following two theorems and declare them to the code generator:

  ```
  lemma [code]: "((x::nat) <= y) = ((x < y) ∨(x=y))"
  lemma [code]: "(max (a::nat) b) = (if (a <= b) then b else a)"
  ```

- The first two lines in your `avl-test.sml` file configure the pretty printer of New Jersey SML to show more details.

2.2 Exercise 50

Use the **quickcheck** command for testing your AVL tree specification "testing" your lemmas. Modify (i.e., introduce bugs) your specifications and try if **quickcheck** finds it. Find at least one example for a bug

- where quickcheck finds a non-trivial counter-example.
- where quickcheck fails in detecting the bug.