Combining Testing and Proving in Dependent Type Theory

Peter Dybjer Qiao Haiyan
Makoto Takeyama

Department of Computing Science
Chalmers University of Technology

TPHOL 2003, Rome

10 September

Dijkstra:

Testing can never prove the absence of errors
– only the presence of them ...

Program specification in Martin-Löf type theory

The program $f : A \to B$ satisfies the input-output relation R under the precondition P:

$$\forall x : A. P\,x \supset R\,x\,(f\,x)$$

In Martin-Löf type theory (used as an *external logic*) this becomes the type:

```
(x :: A) -> P x -> R x (f x)
```

Here

```
P :: A -> Set
R :: A -> B -> Set
```

do not need to be computable; they can e g use quantifiers and inductive definitions.

Testable specifications

If we have shown that `R` is computable by defining

```
r :: A -> B -> Bool
```

such that

```
R x y <-> T (r x y)
```

where

```
T :: Bool -> Set

T True  = Unit  -- one-element set
T False = Empty -- empty set
```

then the specification is *testable* provided we have a complete enumeration

```
a0, a1, a2, ...
```

of all correct inputs `a :: A` such that `P a` is true.

Random testing

Pragmatically, it might be better to choose random inputs, as long as all inputs in the enumeration indeed have a chance to be chosen:

QuickCheck a tool for random testing of Haskell programs, K. Claessen and J. Hughes, ICFP 2000.

We here extend

Agda proof assistant for Martin-Löf style type theory, C. Coquand.

Alfa window interface for Agda, T. Hallgren.

by a QuickCheck-like testing tool. But specification language and test-data generation now becomes internal to Martin-Löf type theory!

Generating test-cases

- Test-data generator for a datatype `D` has type

  ```
  genD :: BT -> D
  ```

  rather than `genD :: N -> D` .

- Library of generators for common datatypes (opportunity for generic programming?)

- If the precondition is computable, i e there is `p : A -> Bool` such that `P x <-> T (p x)` , then we can overestimate and automatically discard test-cases that do not satisfy the precondition (cf QuickCheck)

- If the precondition `P` is given as an inductively defined predicate (à la Prolog) then we can use a Prolog-like technique for generating test-cases

Three kinds of errors

- error in the *program*
- error in the *specification*
- error in the *generation of test-data*

The last is most treacherous! This is a reason for writing test-data generators inside Agda/Alfa, so that we can *prove surjectivity* i e *correctness of test-data generators*.

Combining testing and proving

- Testing is helpful during proof development
  - Debug *programs* and *specifications*
  - Check speculative steps
- Proving helps testing:
  - Decompose a testing task into simpler testing tasks
  - Build consequences of tested properties
  - Correctness of test-data generation

Testing Example

In Haskell:

```
reverse::[a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

A property in QuickCheck:

```
prop_RevRev xs = reverse (reverse xs) == xs
  where types = xs::[Int]
```

QuickCheck the property:

```
Main> quickCheck prop_RevRev
  ...    ...
OK, passed 100 tests.
```

Testing does not guarantee correctness (Dijkstra ...)

Proving The Property

The property in Agda/Alfa:

```
(xs::[Nat]) -> T (reverse(reverse xs) == xs)
```

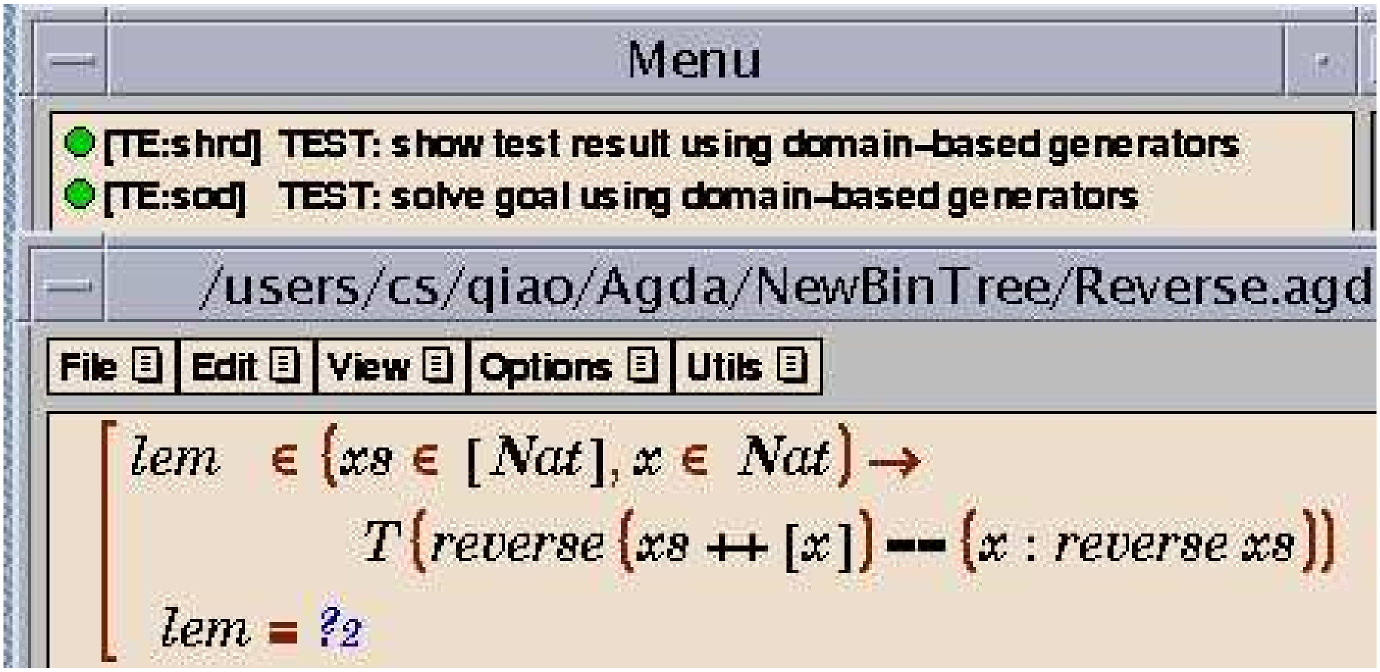
The property can be proved by induction. In the step case, it follows from the following lemma:

```
 (xs::[Nat]) -> (x::Nat) ->
  T (reverse (xs++[x]) == x:(reverse xs))
```

We can try to prove it.

But, it has testable form so why not test it before proving it? Helps us avoid trying to prove false goals.

Our Testing Tool for Agda/Alfa



1. looks for *standard generators* for lists and natural numbers and generates test data (`xs, x`)

2. computes

   `reverse (xs++[x]) == x:(reverse xs)`

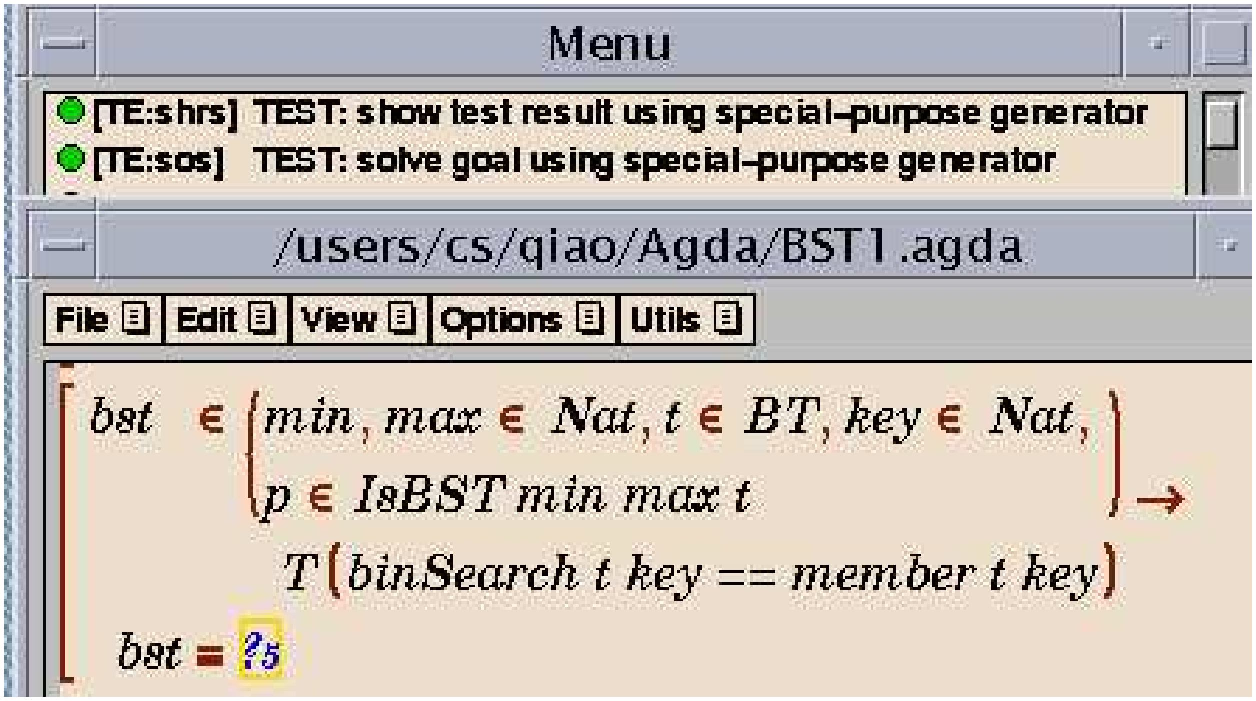
3. if false, returns the counterexample (`xs, x`)

4. if true, repeat the process a given number of times

Generation of test data

In Haskell	In Agda/Alfa		
BT	BT	genD	D
rnd	rnd'		d
randomly generated by QuickCheck	binary tree in Agda/Alfa		elemen

Testing Conditional Properties



- With standard "domain-based" generator, most test data are discarded (do not satisfy the precondition)

- Interested in those `(min, max, t, key)` which satisfies the condition `(isBST min max t)`

- "Special" generator generates dependent records: `(min, max, t, key, p::T (isBST min max t))`

A larger example: AVL-insertion

AVL-tree: balanced binary search tree

```
Balanced Empty = True
Balanced (Branch root lt rt) =
   |#lt - #rt| <= 1
   && Balanced lt && Balanced rt

insert::BT -> a -> BT
insert (Branch root lt rt) key
  | key < root =
      insert_l key root (insert lt key) rt
   ...
insert_l key root (Branch root' lt' rt') rt
  =  let newlt = Branch root' lt' rt'
         t' = Branch root newlt rt
     in  if (#newlt - #rt  == 2)
         then if (#lt' > #rt')
              then       rotateLeft t'
              else doubleRotateLeft t'
         else t'
```

`insert` **preserves** ***balanced*** **(1)**

We show a testing-proving interaction for proving

```
(t::BT) -> Balanced t -> (key::Nat)
        -> Balanced (insert t key)
```

1. We first do the top-level testing of the property. (No point to prove a property with bugs!)

2. Then we start proving by induction on `t` and case-analysis. We will look at the case where

   - `t` = `Branch root lt rt`
   - `key < root`
   - `#(insert lt key) - #rt /= 2`

   Agda generates the subgoal

   ```
   Balanced (Branch root lt rt)
   -> T (#(insert lt key) - #rt /= 2)
   -> Balanced (Branch root (insert lt key) rt)
   ```

`insert` preserves *balanced* (2)

3. Disposing easy parts, it becomes

```
T (|#lt - #rt| <= 1)
 -> T ( #(insert lt key) - #rt  /= 2)
 -> T (|#(insert lt key) - #rt| <= 1)
```

4. Abstracting from heights to numbers, we *speculate* a lemma

```
 (x,y,z::Nat)
  -> T (|y - z| <= 1)
  -> T ( x - z  /= 2)
  -> T (|x - z| <= 1)           (A)
```

$((\mathrm{x}, \mathrm{y}, \mathrm{z}) \leftarrow (\texttt{\#(insert lt key)}, \texttt{\#lt}, \texttt{\#rt}))$

`insert` preserves *balanced* (3)

5. Test of the speculated lemma (A) failed with a counterexample $(\mathtt{x}, \mathtt{y}, \mathtt{z}) = (3, 1, 0)$.

6. Analysing the counterexample, we realise that `#(insert lt key)` $-$ `#lt` $= 2$ cannot happen.
   We revise the speculation (A) to two subgoals:

```
(lt::BT)-> Balanced lt
  -> #(insert lt key) - #lt <= 1      (B1)

(x,y,z::Nat)
  -> T (|y - z| <= 1)
  -> T ( x - z  /= 2)
  -> T ( x - y  <= 1)
  -> T (|x - z| <= 1)                 (B2)
```

7. Test: (B1) passed the test, but (B2) failed with a counterexample $(\mathtt{x}, \mathtt{y}, \mathtt{z}) = (0, 1, 2)$.
   But `#(insert lt key)` $<$ `#lt` cannot happen.

`insert` **preserves** ***balanced*** **(4)**

8. Reformulating (B2) to

```
(lt::BT)-> Balanced lt
  -> #(insert lt key) >= #lt          (C1)

(x,y,z::Nat)
  -> T (|y - z| <= 1)
  -> T ( x - z  /= 2)
  -> T ( x - y  <= 1)
  -> [T (x >= y)]
  -> T (|x - z| <= 1)                 (C2)
```

9. Test (C1) and (C2), no counterexample is returned.

Finally

Testing and proving guided us from the original goal to proving the following simpler properties:

(B1)
```
(lt::BT)-> Balanced lt
    -> #(insert lt key) - #lt <= 1
```

(C1)
```
(lt::BT)-> Balanced lt
    -> #(insert lt key) >= #lt
```

(C2)
```
(x,y,z::Nat)
   -> T (|y - z| <= 1)
   -> T ( x - z  /= 2)
   -> T ( x - y  <= 1)
   -> T (x >= y)
   -> T (|x - z| <= 1)
```

The Generators (1)

The generator for type `D` is an Agda/Alfa function:

```
genD::BT -> D
  where
data BT = Empty
       | Branch (root::Nat)(lt::BT)(rt::BT)
```

Example: Generating balanced trees

```
genBBT::Nat -> BT -> BT
genBBT Zero Empty = Empty
genBBT (Succ Zero) (Branch root l r) =
    Branch root Empty Empty
genBBT (Succ (Succ n)) (Branch root l r) =
  let lt = genBBT n l; rt = genBBT n r
      lt' = genBBT (Succ n) l
      rt' = genBBT (Succ n) r
  in choice3 root
     (Branch root lt rt')
     (Branch root lt' rt)
     (Branch root lt' rt')
```

The Generators (2)

Then we can define the following generator:

```
genBBT' :: BT -> BT
genBBT' Empty = Empty
genBBT' (Branch root l r) =
  genBBT root l
```

and prove that only balanced trees are generated:

```
  (r::BT) -> Balanced (genBBT' r)
```

Furthermore, we can prove all balanced trees can be generated, that is, the generator is surjective.

Proving Surjectivity

Define surjectivity:

```
Surj (genD::BT->D)::Set
 = (x::D) -> ∃ rnd::BT. genD rnd == x
```

i.e., any object in the type can be generated.

The generator `genBBT'` is surjective:

```
 Surj genBBT'
```

The proof can be done by induction.

Related Work

Hayashi pioneering work in the 1980-ies where he used testing to debug lemmas while doing proofs in his PX-system

Programatica project at Oregon Graduate Institute: building a Haskell-based system that integrates testing and proving (informal and formal, interactive and automatic)

Cover project at Chalmers: similar goals

Also

Okasaki is developing Edison (an efficient functional data structure library) by using QuickCheck.

Parent Proof of correnctness of AVL insertion in Coq.

Conclusions

- More case studies have been done: proving properties of BDDs and a tableau prover.
- Testing is helpful during proof development, for example, for finding correct formulations of lemmas.
- Proving can decompose a property into simpler properties to be tested
- Proving can improve "coverage" of testing.
- Can prove properties of generators (surjectivity, satisfaction of preconditions)