Combining Testing and Proving in Dependent Type Theory

Peter Dybjer, Qiao Haiyan, and Makoto Takeyama

Department of Computing Science,
Chalmers University of Technology,
412 96 Göteborg, Sweden
[peterd,qiao,makoto]@cs.chalmers.se

Abstract. We extend the proof assistant Agda/Alfa for dependent type theory with a modified version of Claessen and Hughes' tool QuickCheck for random testing of functional programs. In this way we combine testing and proving in one system. Testing is used for debugging programs and specifications before a proof is attempted. Furthermore, we demonstrate by example how testing can be used repeatedly during proof for testing suitable subgoals. Our tool uses testdata generators which are defined inside Agda/Alfa. We can therefore use the type system to prove properties about them, in particular surjectivity stating that all possible test cases can indeed be generated.

1 Introduction

A main goal of the theorem proving community is to use proof assistants for producing correct programs. However, in spite of faster type-checkers, more automatic proof-search, better interfaces, larger libraries, proving programs correct is still very time consuming, and requires great skill of the user.

Testing has often been disregarded by the theorem proving community since, as Dijkstra emphasised, testing can only show the presence of errors, not the absence of them. However, testing is of course the method used in practice!

Most research on testing has concerned imperative programs. However, an interesting tool QuickCheck for random testing of functional programs (written in the lazy functional programming language Haskell) has recently been developed by Claessen and Hughes [5]. With this tool, correctness properties of functional programs can easily be checked for randomly generated inputs. Experience shows that this is a useful method for debugging programs [6].

Nevertheless, missing rare counterexamples is unacceptable for certain applications. Furthermore, not all correctness properties have a directly testable form. Since both testing and proving have their obvious shortcomings, it would be interesting to combine testing and proving in one system. The idea is to use testing to debug programs and specifications before a proof is attempted. Furthermore, we can debug subgoals which occur during proof, and we can also balance cost and confidence by testing assumptions instead of proving. It may

also be interesting to systematically study testing methods in the context of a full-fledged logic of functional programs.

To this end we have extended the proof assistant Agda/Alfa developed by Coquand [7] and Hallgren [13] with a testing tool similar to QuickCheck. The Agda/Alfa system is an implementation of a logical framework for Martin-Löf type theory. Intuitionistic logic is available via the Curry-Howard correspondence between propositions and types. Martin-Löf type theory is also a programming language which can be briefly described as a functional programming language with dependent types, where all programs terminate. Termination is ensured by only allowing certain recursion patterns. Originally only structural recursion over well-founded datatypes was allowed, but recent work on pattern matching with dependent types [8] and termination checking [22] has extended the class of programs accepted by the system. There is also a significant amount of work on the question of how to reason about general recursive programs in dependent type theory. For some recent contributions to this problem, see Bove [3].

As a first case study we are developing a certified library of classic data structures and algorithms. To illustrate our tool we consider the correctness of some simple search tree algorithms (search and insertion in binary search trees and AVL-trees). As Xi and Pfenning [23] have emphasized in their work on DML (Dependent ML), dependent types are useful for expressing invariants of such data structures. Our goals should be compared to those of Okasaki, who is currently developing Edison – a library of efficient functional data structures written in Haskell [18]. Okasaki uses QuickCheck to test his programs, and also includes QuickCheck test data generators to be used in applications of his data structures. We aim to build an analogous library for the Agda/Alfa system, where we can use dependent types to capture more invariants of the algorithms, and even provide full correctness proofs whenever this is feasible.

The idea of combining proving and testing is also part of the Programatica project currently under development at Oregon Graduate Centre [20]. Some early work on combining proving and testing was done by Hayashi, who used testing to debug lemmas while doing proofs in his PX-system [14]. Hayashi is currently pursuing the idea of testing proofs and theorems in his foundationally oriented project on "proof animation" [15]. AVL-insertion has been implemented and proven correct in Coq by Catherine Parent [19].

Plan. In Section 2 we introduce QuickCheck. In Section 3 we introduce the proof assistant Agda/Alfa. In Section 4 we extend Agda/Alfa with a QuickCheck-like tool. In Section 5 we discuss test case generation. In Section 6 we summarize our experiments with algorithms for insertion in AVL-trees and illustrate how testing helps during proof development. In Section 7 we briefly describe how AVL-tree insertion can be implemented by using dependent types, so that the type system ensures that the insertion preserves the AVL-tree property. Section 8 concludes with a brief discussion of further research. Appendix A contains some Haskell programs from Section 2.

Acknowledgments. We wish to thank Koen Claessen and John Hughes for many discussions.

2 QuickCheck

The basic idea of QuickCheck is to test whether a boolean property

```
p[x1, ..., xn] :: Bool
```

is `True` for random instances of the variables `x1 :: D1, ..., xn :: Dn`. (The notation `p[x1,...,xn]` means that the expression `p` may contain occurrences of the free variables `x1,...,xn`. The reader is warned not to confuse this notation with Haskell's list notation!)

For example, if we wish to test that

```
reverse (reverse xs) == xs
```

for arbitrary integer lists `xs`, we write a property definition

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

Then we call QuickCheck

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

QuickCheck here uses a library test data generator for integer lists. It is also possible for the user to define her own test data generator.

More generally, QuickCheck can test conditional properties written

```
p[x1,...,xn] ==> q[x1,...,xn]
```

where `p[x1,...,xn], q[x1,...,xn] :: Bool`. QuickCheck performs a sequence of tests as follows (at least conceptually):

1. A random instance `r1 :: D1, ..., rn :: Dn` is generated.
2. `p[r1, ..., rn]` is computed. If it is `False`, the test is discarded and a new random instance is generated. If on the other hand it is `True`, then
3. `q[r1, ..., rn]` is computed. If it is `False`, QuickCheck stops and reports the counterexample. If it is `True` the test was successful and a new test is performed.

QuickCheck repeats this procedure 100 times, by default. Only tests which are not discarded at step 2 are counted.

Another example of a QuickCheckable property is the following correctness property of a search algorithm `binSearch` for binary search trees. The property states that the algorithm correctly implements membership in binary search trees:

```
isBST lb ub t ==> binSearch t key == member t key
```

Here `t` is a binary tree of type `BT`, the type of binary trees with integers in the nodes; in Haskell:

```
data BT = Empty | Branch Int BT BT
```

`isBST lb ub t` holds if `t` is a binary search tree with elements strictly between `lb` and `ub`, (see Appendix A for the definitions in Haskell).

Before we can use QuickCheck we need a suitable test data generator. A generator for `BT` could be used, but is inappropriate. The reason is that most randomly generated binary trees will not be binary search trees, so most of them will be discarded. Furthermore, the probability of generating a binary search tree decreases with the size of the tree, so most of the generated trees would be small. Thus the reliability of the testing would be low. A better test case generator generates binary search trees only.

For more information about QuickCheck, see Claessen and Hughes [5] and the homepage `http://www.cs.chalmers.se/~rjmh/QuickCheck/`. Much of the discussion about QuickCheck, both about pragmatics and concerning possible extensions seems relevant to our context.

3 The Proof Assistant Agda/Alfa

This section briefly describes the proof assistant Agda/Alfa. The reader familiar with Agda/Alfa can skip it.

Agda [7] is the latest version of the ALF-family of proof systems for dependent type theory developed in Göteborg since 1990. Alfa [13] is a window interface for Agda. We quote from the Alfa home page [13]:

> Alfa is a successor of the proof editor ALF, i.e., an editor for direct manipulation of proof objects in a logical framework based on Per Martin-Löf's Type Theory. It allows you to, interactively and incrementally, define theories (axioms and inference rules), formulate theorems and construct proofs of the theorems. All steps in the proof construction are immediately checked by the system and no erroneous proofs can be constructed.

That "no erroneous proofs can be constructed" only means that a completed proof is indeed correct. It does not help you to avoid blind alleys.

The syntax of Agda/Alfa has been strongly influenced by the syntax of Haskell and also of Cayenne [1], a functional programming language with dependent types. In addition to the function types `a -> b` available in ordinary functional languages, there are dependent function types written `(x :: a) -> b`, where the type `b` may depend on `x :: a`.

Agda/Alfa also has dependent record types (signatures) written

```
sig {x1 :: a1; x2 :: a2; ...; xn :: an}
```

where `a2` may depend on `x1 :: a1` and `an` may depend on `x1 :: a1, x2 :: a2`, etc. Elements of signatures are called structures written

```
struct{x1 = e1; x2 = e2; ...; xn = en}
```

Signatures are much like iterated Σ-types Σ`x1 :: a1.`Σ`x2 :: a2. ... an.` and structures are much like iterated tuples `(e1, (e2, ..., en))` inhabiting them.

Furthermore, Agda/Alfa has a type `Set` of sets in Martin-Löf's sense. Such sets are built up from basic inductive data structures, using dependent function types and signature types. A basic example is the set of natural numbers. Its definition in Agda/Alfa is

```
data Nat = Zero | Succ (n :: Nat)
```

More generally, constructors for sets may have dependent types, see for example the definition of AVL-trees in Section 7.

Remark. The reader is warned that the dependent type theory code given here is not accepted verbatim by the Agda/Alfa system, although it is very close to it. To get more readable notation and avoiding having to discuss some of the special choices of Agda/Alfa we borrow some notations from Haskell, such as writing `[a]` for the set of lists of elements in `a`. In particular, we use Haskell-style overloading although this feature is not present in Agda/Alfa.

Predicates on a set `D` are propositional functions with the type `D -> Set` by the identification of propositions as sets. Decidable predicates have the type `D -> Bool`. To convert from decidable to general predicates we use the function

```
T :: Bool -> Set
T True  = Truth
T False = Falsity
```

where `Truth = Unit` represents the trivially true proposition and `Falsity` is the empty set representing the false proposition.

For a more complete account of the logical framework underlying Agda/Alfa including record types see the paper about structured type theory [9] and for the inductive definitions available in type theory, see Dybjer [10] and Dybjer and Setzer [11, 12].

4 A Testing Tool for Agda/Alfa

Our testing tool can test properties of the following form:

```
(x1 :: D1) -> ... -> (xn :: Dn[x1, ..., x(n-1)]) ->
 T (p1[x1, ..., xn]) -> ... -> T (pm[x1, ..., xn]) ->
 T ( q[x1, ..., xn])
```

Under the identification of 'propositions as types', this reads

$$\forall x_1 \in \mathtt{D1}.\ \cdots \forall x_n \in \mathtt{Dn}[x_1, \cdots, x_{n-1}].$$
$$\mathtt{p1}[x_1, \cdots, x_n] \Longrightarrow \mathtt{pm}[x_1, \cdots, x_n] \Longrightarrow$$
$$\mathtt{q}[x_1, \cdots, x_n]$$

This is essentially the form of properties that QuickCheck can test, except that in dependent type theory the data domains `Di` can be dependent types.

The user chooses an option "test" in one of the menus provided by Alfa. The plug-in calls a test data generator and randomly generates a number of cases (at the moment 50) for which it checks the property.

Consider again the correctness property of binary search. It is the following Agda/Alfa type (using `Nat` rather than `Int` for simplicity):

```
(lb, ub, key :: Nat) -> (t :: BT) ->  T (isBST lb ub t) ->
 T (binSearch t key == member t key)
```

Fig. 1 shows how this property is displayed in an Alfa-window: here *bst* is a

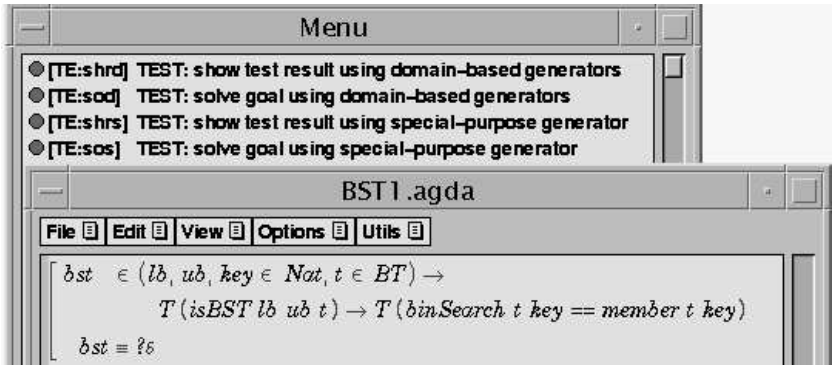

Fig. 1. Testing binary search

proof object for the correctness of binary search; *bst* is yet to be defined by instantiating the highlighted question mark $?_5$. This can now be done either by proving or testing. If we choose to prove it, $?_5$ should be instantiated to a proof term which we can build interactively by pointing and clicking (see the Alfa homepage for details [13]). If we instead want to test it, we choose one of the testing options found in one of Alfa's menus (see Fig. 1). We can either use a *domain-based* or a *special purpose* test data generator. The domain-based option looks for generators for the data domains `Di`. The special purpose option looks for a generator for the lemma in question; it can therefore take the conditions into account. We can also choose whether we want to see the results of testing in a separate window, or just try to solve the goal, and if successful replace $?_5$ by

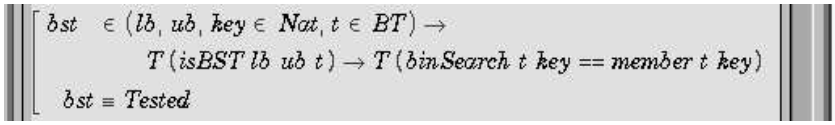

Fig. 2. The goal is 'Tested'

the pseudo proof term "`Tested`" indicating that the goal has been successfully tested, but not proved (see Figure 2).

If testing fails, a counterexample is returned. For example, if we remove the condition `T (isBST lb ub t)`, then the property above is not true any more, and the plug-in will report a counterexample, as in Fig. 3.

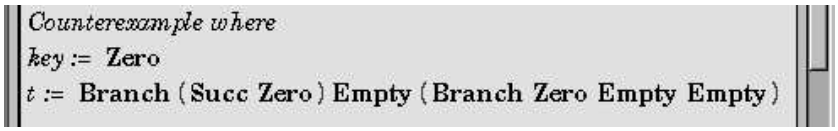

Fig. 3. A counterexample

5 Test Data Generators

In principle we could generate test data for a type `D`, by writing a function which enumerates its elements:

```
enum :: Nat -> D
```

Then we could either use `enum` for exhaustive testing, that is, the plug-in could test the property for `enum 0, enum 1, enum 2, ...`. Or we could generate a sequence of random natural numbers $r_0, r_1, r_2, \ldots$ and test the property for `enum` r_0, `enum` r_1, `enum` $r_2, \ldots$.

However, rather than starting with randomly generated natural numbers we shall start with randomly generated binary trees of natural numbers. This is a somewhat more practical way to write test data generators for a wide variety of types. For example, it is easy to get the next random seed when writing a generator.

Thus a test data generator for the type `D` has type

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

The Agda/Alfa system does not have a built in random number generator. So the plug-in first calls Haskell to generate a random element of the Agda/Alfa type `BT`. Then it applies `genD` to convert it to an element of `D`. An alternative approach would be to write test data generators for `D` directly in Haskell. However, using

Agda/Alfa for this purpose has several advantages. The user can stay inside the language of Agda/Alfa when testing as well as when proving. Moreover, we can use Agda/Alfa for enforcing dependent type correctness of `genD`, and also for showing that it is a *surjective* function, expressing that every element of `D` can indeed be produced by the test data generator.

The drawback is efficiency: Haskell's evaluator is much more efficient than Agda/Alfa's. However, for our present experiments this is not a major problem; we expect that later versions of Agda/Alfa will be as efficient as Cayenne [1].

Let us see some generator examples.

Example 1. The following function defines a generator for `[Nat]`:

```
genList :: BT -> [Nat]

genList Empty               = []
genList (Branch root lt rt) = root:genList lt
```

We can prove that the generator `genList` is surjective, that is

$$\forall xs \in [\mathtt{Nat}].\ \exists\, seed \in \mathtt{BT}.\ \mathtt{genList}\ seed\ =\ xs.$$

Existential quantification becomes dependent sum under the propositions as types identification, so what we prove in Agda/Alfa is `Surj genList`, where

```
Surj :: (g :: BT -> a) -> Set
Surj g = (x :: a) -> sig { seed :: BT; prf :: T (g seed == x) }
```

Example 2. Here is a generator for binary search trees:

```
genBST :: Nat -> Nat -> BT -> BT

genBST lb ub Empty = Empty
genBST lb ub (Branch rnd l r) =
  let newroot = lb + 1 + (rnd `mod` (ub - lb - 1))
      lt = genBST lb newroot l
      rt = genBST newroot ub r
  in if (ub > lb + 1) (Branch newroot lt rt) Empty
```

We can now prove (or test!) that `genBST` only generates binary search trees:

```
(lb, ub :: Nat) -> T (lb < ub) ->
(  seed :: BT ) -> T (isBST lb ub (genBST lb ub seed))
```

We can also prove that `genBST` is surjective on binary search trees:

```
(lb, ub :: Nat) -> T (lb < ub) ->
(     t :: BT ) -> T (isBST lb ub t) ->
 sig { seed :: BT;  prf :: T (genBST lb ub seed) == t }
```

Now we can choose "using special generator" to test the property in Fig. 1 and the result is "passed 50 successful tests" . If we choose the option "solve the goal", then the goal is filled with "Tested" (see Fig. 2).

6 Combining Testing and Proving

In this section we show some concrete examples to illustrate the following general points about how testing and proving help each other:

[a] The essence of creative user interaction is the introduction of lemmas. This is often a speculative process. If a user fails to prove a conjecture or its hypotheses, she must backtrack and try another formulation. Testing before proving is a quick and effective way to discard wrong or inapplicable conjectures.

[b] Analysis of failed tests gives useful information for proving. We call a counterexample to a conjecture *spurious* if it lies outside the intended domain of application of the conjecture. Having those at hand, the user can formulate a sharper lemma that excludes them. Genuine counterexamples on the other hand helps locating bugs in programs or in the formalisation of intended properties.

[c] A given goal may not be (efficiently) testable. When interaction with the proof assistant produces testable subgoals, it is guaranteed that testing all of them is at least as good as testing the original goal; we know that no unintended logical gaps are introduced.

[d] Interactive proving increases confidence in the coverage and rationality of testing. Suppose a program consists of various components and branches, and it passes a top-level test for a property. When we try to prove the goal, the proof assistant typically helps us derive appropriate subgoals for the different components or branches. Testing these subgoals individually reduces the risk of missed test cases in the top-level testing.

Example 3 (list reverse). Consider again the example of reversing a list twice. We proceed as follows:

1. Test the `main` goal using a domain-based generator for `[Nat]`, to check for a bug in the program or the specification ([b]).

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

2. Start proving by induction on `xs`. The `nil_case` is trivial. The testable subgoal `cons_case` is automatically derived by Alfa.

```
cons_case :: (x :: Nat) -> (xs' :: [Nat]) ->
                  T (reverse (reverse xs' ++ [x]) == x:xs')
```

   It is testable in principle, although there is little point in doing so; most test cases for the top-level goal probably already had `cons`-form.

3. The normalization in Agda/Alfa was blocked by a non-reducing `++`. Therefore we speculate the `lemma` by changing variables:

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

This is a creative step not forced by logic. Therefore it is worth testing `lemma` before trying to prove it. Although a brief thought shows its equivalence to `cons_case` under the induction hypothesis, running a test is cheaper ([a]).

4. Proceed by proving `lemma` by induction on `ys`, which finishes the proof.

Example 4 (AVL-tree insertion). Recall that an AVL-tree `t` is a binary search tree which is balanced (`Bal t`): the height difference between the left and right subtrees of each node is at most 1.

```
Bal :: BT -> Bool
Bal Empty            = True
Bal (Branch n lt rt) = |#lt - #rt| <= 1 && Bal lt && Bal rt
```

where `#t` is the height of `t` and `|x|` is the absolute value of the integer `x`.

The following algorithm inserts a key in an AVL-tree and is adapted from a textbook on functional data structures [21]. We show relevant parts only:

```
insert :: BT -> Nat -> BT

insert Empty            k           = Branch   k      Empty Empty
insert (Branch n lt rt) k | k == n = Branch   n         lt    rt
                          | k <  n = insert_l n (insert lt k) rt
                          | k >  n = insert_r n lt (insert rt k)
insert_l :: Nat -> BT -> BT -> BT

insert_l n newlt@(Branch n' lt' rt') rt
  | #newlt - #rt == 2 = if #lt' > #rt' then rotateLeft t'
                        else          doubleRotateLeft t'
  | otherwise         = t'
                        where t' = Branch n newlt rt
insert_l n Empty      = Empty  -- unreachable
```

where the pattern `var@pat` expresses that `var` is a name for the value being matched by `pat`.

One of the required properties of `insert` is to maintain the height balance:

```
main :: (t :: BT) -> Bal t -> (k :: Nat) -> Bal (insert t k)
```

(We omit writing `T` in `T (Bal t)` etc. in this section.)

We now summarize how testing interacts with proving while verifying a part of `main`. Note that our aim is not to show a well-organised development with much afterthought; rather, our point is to show how the combination cost-effectively helps in a real life 'first-try'. (Ti is here a testing step and Pi is a proving step.)

T1 Test `main` using the special purpose generator for balanced trees: A bug in `insert` or in `Bal` is likely to be discovered early.

P2 Do induction on `t`, and split the `Branch` case according to the structure of `insert`: correct subgoals are automatically generated by Alfa.

Here we take the simplest subgoal as an example. In case `t = Branch n lt rt`, `k < n`, and not `#newlt - #rt == 2` (writing `newlt` for `insert lt k`), we get

```
subgoal :: Bal t -> Bal (Branch n newlt rt)
```

When we unfold the definition of `Bal` to the right we get three conjuncts, where only `|#newlt - #rt| <= 1` is not immediate. Thus we need to prove the following:

```
subgoal' ::  |   #lt - #rt| <= 1
         ->   #newlt - #rt  /= 2
         ->  |#newlt - #rt| <= 1
```

P3 Abstracting from tree heights to plain numbers, we speculate the following

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

The intended instantiation is $(\mathtt{x},\mathtt{y},\mathtt{z}) = (\mathtt{\#newlt},\mathtt{\#lt},\mathtt{\#rt})$. The speculated preconditions here abstract away much of the relationship among those heights.

T4 Testing `lemmaA` shows a counterexample $(\mathtt{x},\mathtt{y},\mathtt{z}) = (3,1,0)$.

A moment's reflection reveals that this counterexample is spurious; `newlt` contains at most one more element than `lt` and rotations do not increase height, so `#newlt` cannot increase by two from `#lt`.

P5 Therefore we add an extra hypothesis `x-y <= 1` to `lemmaA`:

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

lemmaB  :: (lt :: BT) -> Bal lt -> #(insert lt k) - #lt <= 1
```

`lemmaB` is needed to discharge the added hypothesis `x-y <= 1` in `lemmaA'`.

T6 Test lemmas: `lemmaB` passes the test, but `lemmaA'` fails again with $(\mathtt{x},\mathtt{y},\mathtt{z}) = (0,1,2)$. This reminds us that `#newlt >= #lt` also must be used in the proof.

P7 Add yet another hypothesis to `lemmaA'`:

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

lemmaC   :: (lt :: BT) -> Bal lt -> #(insert lt k) >= #lt
```

T8 Test lemmas: Both now pass tests.

P9 Prove `lemmaA''`, `lemmaB`, and `lemmaC`. This finishes the proof of this particular `subgoal`.

When a proof proceeds with nested cases, the context of a subgoal may quickly become unmanageably large and essential information may be obscured. Besides the general points, this example demonstrates how testing lets us first try reckless abstractions and then recover the needed pieces of information.

7 Using Dependently Typed Data Structures

We also experimented with several ways of writing a dependently typed version of AVL-insertion. Let `AVL h lb ub` be the set of AVL-trees with height `h` and bounds `lb` and `ub`. The insertion function then gets the type:

```
insertAVL :: (h, lb, ub :: Nat) -> AVL h lb ub ->
             (k :: Nat) -> Between lb k ub ->
             (AVL h lb ub  +  AVL (Succ h) lb ub)
```

expressing that the height is either unchanged or increased by one. `Between lb k ub` abbreviates `T (lb <= k && k < ub)`. The definition of `AVL` is as follows.

```
AVL  Zero           lb ub = T (lb <= ub)
AVL (Succ    Zero)  lb ub = data Leaf (k :: Nat)
                                     (p :: Between lb k ub)
AVL (Succ (Succ h)) lb ub =
  data LH (root :: Nat) (lt :: AVL (Succ h) lb          root)
                        (rt :: AVL  h       (Succ root) ub  )
     | EQ (root :: Nat) (lt :: AVL (Succ h) lb          root)
                        (rt :: AVL (Succ h) (Succ root) ub  )
     | RH (root :: Nat) (lt :: AVL  h       lb          root)
                        (rt :: AVL (Succ h) (Succ root) ub  )
```

Furthermore, we get informative types of the rotations. For example,

```
rotLeft :: (n, lb, ub, root :: Nat)
             -> (lt :: AVL (Succ (Succ n)) lb          root)
             -> (rt :: AVL n               (Succ root) ub  )
             ->        AVL (Succ (Succ n)) lb          ub
```

A complete version of `insertAVL` which does not deal with the search tree condition, but only with the balance condition can be found at `http://www.cs.chalmers.se/~qiao/papers/TestingProving/`.

It is about one page long.

When writing these programs in Alfa, we start with their dependent type. While building the programs the type-checker ensures that we do not do anything wrong, hence no need for testing! However, we should keep in mind that this algorithm was written after a certain amount of experimentation with Haskell-style algorithms, where both testing and proving helped us to gain insight into the logical structure of the problem. Moreover, even for this dependently typed `insertAVL`, which has a type that shows that it maintains the AVL-tree property, testing is still useful for making sure that it satisfies the insertion axiom:

```
(m :: s) -> (x, y :: a) ->
 T (member (insert m x) y == (x == y) || member m y)
```

when `s` is the set of all AVL-trees, `a = Nat`, `member` is binary search for AVL-trees, and `insert` is implemented using `insertAVL`.

When building a library of datastructures we may go further and show that concrete datastructures, such as AVL-trees with associated operations, satisfy all properties of suitable abstract data types. We can use Agda/Alfa's dependent records (signatures and structures) for this purpose. For example, a signature for the ADT of finite sets, including both operations and axioms, can be formalized as follows:

```
AbsSet :: (a, s :: Set) -> Eq a ->
     sig  empty    :: s
          member   :: s -> a -> Bool
          insert   :: s -> a -> s
          single   :: a -> s
          union    :: s -> s -> s
          ...
          insertAx :: (m :: s) -> (x, y :: a) ->
                      T (member (insert m x) y
                         == (x == y) || member m y)
          singleAx :: (m :: s) -> (x, y :: a) ->
                      T (member (single x) y == (x == y))
          unionAx  :: (m,m' :: s) -> (x :: a) ->
                      T (member (union m m') x
                          == member m x || member m' x)
          ...
```

where the dots indicate that we may include all the usual set operations and their axioms, and `Eq a` specifies that the equality `(==)` on the set `a` is an equivalence relation:

```
Eq :: Set -> Set

Eq a = sig  (==) :: a -> a -> Bool
            ref  :: (x     :: a) -> T (x == x)
            sym  :: (x,y   :: a) -> T (x == y) -> T (y == x)
            tra  :: (x,y,z :: a) -> T (x == y) ->
                                      T (y == z) -> T (x == z)
```

Note that both `Eq a` and `AbsSet a s eq` are "testable signature" since all axioms are testable properties.

8 Discussion and Further Research

This paper is a first progress report on combining testing and proving in dependent type theory. We have built a simple tool for random testing of goals occurring during proof construction in the proof assistant Agda/Alfa. We have also carried out some case studies.

Like Hayashi, we feel that it is indeed fruitful to combine proving and testing. In formulating theorems and planning proofs, testing is helpful for avoiding false

starts and wrong turns, while recovering from these is costly in traditional proof assistants.

Claessen and Hughes report three common kinds of errors discovered by testing: errors in programs, errors in specifications, and errors in test data generators which sometimes are complex programs themselves. Since our test data generators are written in Agda/Alfa, we can prove properties, such as surjectivity, about them. In this way we increase confidence in successful test runs. We can also use the dependent type system to ensure that generated test data have the expected property.

Testing encourages an experimental frame of mind. You try this and you try that. Once you have access to suitable test data generators it is easy to test different lemmas. Proving on the other hand requires that you think very clearly about the problem. It is a familiar observation that you pay a high price for lack of elegance when trying to prove a property. So you are forced to think more deeply about the reasons why a program works.

A consequence of this is that you feel inclined to improve the design of your program. Much work on program verification and derivation has been based on what seems like a too simple model of programming. First you write a specification. Then you write a program. Then you prove that the program meets the specification. Alternatively, you may try to systematically derive the program from the specification in such a way that it meets the specification. (This latter view has been popular in the type theory community.) This seems to be an unrealistic model of program development, however. What happens in practice seems to be that both the specification and the program evolve gradually. We believe that both testing and formal proof can help during this process, and it is advantageous to have a system where you can easily switch between the two.

We believe that combining testing and proving will give rise to new research problems distinct from those which arise when doing testing and proving separately. Here are some questions that we asked ourselves, while carrying our case studies:

- How often in a proof do we encounter (efficiently) testable subgoals? In the small examples in this paper testable subgoals appeared sufficiently often to make testing a useful guide for proving. But will this be the case when doing larger and more complex proofs?
- Can we automate test data generation more? At the moment test data generation has to be done manually, but the technique of generic (or polytypic) programming [2] may help us write domain-based test data generators uniformly for a large class of data types. Can even the activity of writing special purpose generators be automated to some extent? Another question is to consider alternatives to the approach of the present paper where random binary trees are first generated and then converted to appropriate data types.
- Should we consider systematic test data generation instead? Can we use the structure of the specification term `q[x1, ..., xn]` to generate test data with good coverage properties?

– Can we generalize the class of testable properties that our tool accepts? In a sense Martin-Löf's meaning explanations for type theory say that all judgements are testable! (A discussion of this point is outside the scope of this paper.) Can we make use of this observation?

References

1. Lennart Augustsson: Cayenne: a Language with Dependent Types. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP-98)*, ACM SIGPLAN Notices, 34(1), pages 239-250, 1998.
2. R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens: Generic Programming — An Introduction. LNCS 1608, pages 28–115, 1999.
3. Ana Bove: General Recursion in Type Theory. PhD thesis. Chalmers University of Technology. 2002.
4. Magnus Carlsson and Thomas Hallgren: Fudgets - Purely Functional Processes with applications to Graphical User Interfaces. PhD thesis. Chalmers University of Technology. 1998.
5. Koen Claessen and John Hughes: QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)* volume 35.9, pages 18–22. ACM Press, 2000.
6. Koen Claessen and John Hughes: QuickCheck: Automatic Specification-Based Testing: http://www.cs.chalmers.se/~rjmh/QuickCheck/.
7. Catarina Coquand: Agda, available from http://www.cs.chalmers.se/~catarina/agda.
8. Thierry Coquand: Pattern Matching with Dependent Types. In Bengt Nordström, Kent Petersson and Gordon Plotkin, editors, *Proceedings of The 1992 Workshop on Types for Proofs and Programs* pages 71–84. Båstad, 1992.
9. Thierry Coquand: Structured Type Theory. draft, 1999, available from http://www.cs.chalmers.se/~coquand/type.html.
10. Peter Dybjer: Inductive Families. In *Formal Aspects of Computing*, volume 6, pages 440–465, 1994.
11. Peter Dybjer and Anton Setzer: A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications, Lecture Notes in Computer Science* 1581, pages 129–146. 1999.
12. Peter Dybjer and Anton Setzer: Indexed Induction-Recursion. In *Proof Theory in Computer Science, LNCS* 2183, pages 93-113, 2001.
13. Thomas Hallgren: Alfa, available from http://www.cs.chalmers.se/~hallgren/Alfa.
14. Susumu Hayashi and Hiroshi Nakano: PX, a Computational Logic. The MIT Press. 1988.
15. Susumu Hayashi, Ryosuke Sumitomo and Ken-ichiro Shii: Towards Animation of Proofs - testing proofs by examples. In *Theoretical Computer Science*, volume 272, pages 177–195, 2002.
16. Per Martin-Löf: Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153-175. North-Holland, 1982.
17. Per Martin-Löf: Intuitionistic Type Theory. Bibliopolis, 1984.
18. Chris Okasaki: An Overview of Edison. In *Haskell Workshop*, pages 34–54, September 2000.
19. Catherine Parent: A collection of examples using the Program tactic. available from http://pauillac.inria.fr/coq/contribs-eng.html.

20. Programatica: Integrating Programming, Properties, and Validation. http://www.cse.ogi.edu/PacSoft/projects/programatica/.
21. F.A. Rabhi and G. Lapalme: Algorithms: a functional programming approach. Addison-Wesley Press, 1999.
22. David Wahlstedt: Detecting termination using size-change in parameter values. Master thesis. Chalmers University of Technology, 2000.
23. Hongwei Xi and Frank Pfenning: Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages*, pages 214–227, ACM Press, 1999.

A Haskell Programs in Section 2

```
isBST :: Int -> Int -> BT -> Bool
isBST lb ub Empty = lb < ub
isBST lb ub (Branch root left rt) = lb < root && root < ub
              && isBST lb root left && isBST root ub rt
```

The membership algorithm for general binary trees is

```
member :: BT -> Int -> Bool
member Empty               key = False
member (Branch root lt rt) key = key == root
                                 || member lt key || member rt key
```

The binary search algorithm is

```
binSearch :: BT -> Int -> Bool
binSearch Empty                key = False
binSearch (Branch root lt rt)  key
          | key < root  = binSearch lt key
          | key == root = True
          | key > root  = binSearch rt key
```