Verifying Haskell Programs by Combining Testing and Proving

Peter Dybjer Qiao Haiyan Makoto Takeyama
Department of Computing Science,
Chalmers University of Technology, 412 96 Göteborg, Sweden
{peterd,qiao,makoto}@cs.chalmers.se

Abstract

We propose a method for improving confidence in the correctness of Haskell programs by combining testing and proving. Testing is used for debugging programs and specification before a costly proof attempt. During a proof development, testing also quickly eliminates wrong conjectures. Proving helps us to decompose a testing task in a way that is guaranteed to be correct. To demonstrate the method we have extended the Agda/Alfa proof assistant for dependent type theory with a tool for random testing. As an example we show how the correctness of a BDD-algorithm written in Haskell is verified by testing properties of component functions. We also discuss faithful translations from Haskell to type theory.

Keywords: program verification, random testing, proof-assistants, type theory, BDDs and Haskell.

1. Introduction

Haskell is a functional programming language with higher order functions and lazy evaluation. As a consequence it supports the writing of modular programs [18]. Haskell programs are often more concise and easier to read and write than programs written in more traditional languages.

Haskell programs are thus good candidates for formal verification. In particular, we are interested in verification in the framework of type theory [20], as we believe that its logic matches Haskell's functional style well. Our case studies are developed in the proof assistant Agda/Alfa [7, 15] for dependent type theory developed at Chalmers University of Technology. For a brief introduction, see Appendix A.

Despite the good match, current proof-assistant technology is such that a complete correctness proof of even a moderately complex program requires great effort of a user. If full correctness of the program is not vital, then the cost of such an effort may exceed its value. In a system where we can combine testing and proving we can keep the cost of verification down by leaving some lemmas to be tested only. In this way it is possible to balance the cost of verification and the confidence in the correctness of the program in a more appropriate way.

QuickCheck [6] is an automatic random testing tool for Haskell programs. It is a combinator library written in Haskell for an embedded specification language with test data generation and test execution. It can test whether a given function f satisfies a specification $\forall x \in A.\, P[x, f(x)]$ with a decidable property P by randomly generating values for x.

When f is defined in terms of component functions $f_1, f_2, \cdots$, the above is a top-level testing of f. It gives little information about how much f_i is tested, what sub-properties have been tested and *why* P follows from those sub-properties.

Our method combines testing and interactive proving to obtain such information. Testing and proving are complementary. Proving is used to decompose the property P of the function f into sub-properties P_i of components f_i and to show why P_i's are sufficient for P. Testing each f_i with respect to P_i increases confidence in test code coverage and locates potential bugs more precisely. Testing is used also during proving to quickly eliminate wrongly formulated lemmas.

We illustrate this methodology by verifying a Haskell implementation of a BDD algorithm by J. Bradley ([3], see Appendix B.) BDDs (Binary Decision Diagrams) [4] are a canonical representation for Boolean functions that makes testing of functional properties such as satisfiability and equivalence very efficient. BDD based model checkers are widely used. Various formulations of BDD algorithms have been verified in several theorem provers, e.g., [19, 22], but our interest is rather in trying an existing Haskell program not necessarily written with verification in mind.

All the code in the paper can be found at `http://www.cs.chalmers.se/~qiao/papers/BDDs/`.

Related Work: The idea of combining proving and testing is part of the Cover project [9] at Chalmers University of Technology, the goal of which is to develop a system that integrates a wide range of techniques and tools that are currently available only separately. It is partly inspired by the Programatica project [21] at Oregon Graduate Centre, which has similar goals. The combination of testing and proving has also been investigated by Chen, Tse and Zhou [5], who propose to check the correctness of a program by proving selected metamorphic relations with respect to the function. Some early work on combining proving and testing was done by Hayashi [17], who used testing to debug lemmas while doing proofs in his PX-system. Geller [14] argued that test data can be used in proving program correctness. This paper reports a case study based on our previous work on combining testing and proving in dependent type theory [11].

Plan of the paper: Section 2 describes our testing tool. After a general discussion on the benefits of the combination (Section 3), we discuss how to translate a Haskell program into Agda/Alfa (Section 4), and how to verify it (Section 5). Section 6 concludes the article with future directions for the work.

2. The Testing Tool

We have extended Agda/Alfa with a testing tool similar to QuickCheck, a tool for random testing of Haskell programs. However, our tool can express a wider range of properties and is integrated with the interactive reasoning capabilities of Agda/Alfa.

The form of properties that can be tested is

$$\begin{aligned}&\forall\, x_1 \in D_1.\ \cdots\ \forall\, x_n \in D_n[x_1,\cdots,x_{n-1}].\\ &P_1[x_1,\cdots,x_n] \Rightarrow\ \cdots\ \Rightarrow P_m[x_1,\cdots,x_n] \Rightarrow\\ &Q[x_1,\cdots,x_n]\ .\end{aligned}$$

Here $D_i[x_1,\cdots,x_{i-1}]$ is a type depending on $x_1,\cdots,x_{i-1}$; P_i is a precondition satisfied by relevant data; and Q is the target property, typically relating the outputs of functions being tested with the inputs $x_1,\cdots,x_n$. The predicates P_i and Q must be decidable.

Under the Curry-Howard correspondence between predicate logic and dependent type theory, such a property *is* a dependent function type of the Agda/Alfa language:

$$\begin{aligned}&(x_1 :: D_1) \texttt{ -> } \cdots\ (x_n :: D_n[x_1,\cdots,x_{n-1}]) \texttt{ -> }\\ &P_1[x_1,\cdots,x_n] \texttt{ -> } \cdots \texttt{ -> } P_m[x_1,\cdots,x_n] \texttt{ -> }\\ &Q[x_1,\cdots,x_n]\end{aligned}$$

For decidability, we require the predicates (now themselves dependent types) to have the decidable form `T` $(p[x_1,\cdots,x_n])$ (the term p `::` `Bool` lifted[1] to the type level). We call properties of this form *testable.*

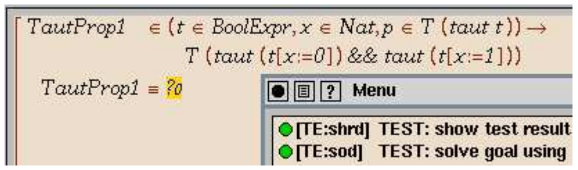
1 See Appendix A, but we mostly omit writing `T` in what follows.

An example is the following property of the function `taut::BoolExpr -> Bool`, which decides if a given boolean expression is a tautology (see Appendix B, C.)

```
(t::BoolExpr) -> (x::Nat) ->
 T (taut t) ->
 T (taut(t[x:=0]) && taut(t[x:=1]))
```

(If `t` is a tautology, so are the results of replacing the variable `Var x` by constants.)

By making this a goal ($?_0$), tests can be invoked from the Alfa menu:


Menu
[TE:shrd] TEST: show test result
[TE:sod] TEST: solve goal using

$$\begin{aligned}&\textit{TautProp1} \in (t \in \textit{BoolExpr}, x \in \textit{Nat}, p \in T\ (\textit{taut}\ t)) \rightarrow\\ &\qquad T\ (\textit{taut}\ (t[x{:=}0])\ \&\&\ \textit{taut}\ (t[x{:=}1]))\\ &\textit{TautProp1} \equiv ?_0\end{aligned}$$

The testing procedure in this case is as follows:

```
do (at most predetermined $N$ times)
  repeat generate test data (t, x) randomly
  until the precondition taut t computes to True
while taut(t[x:=0]) && taut(t[x:=0])
      computes to True
```

The tool then reports the success of N tests, or the counterexample `(t,x)` found. (A success, in this case.)

A test data generator $genD$ for the type D is a function written in Agda/Alfa. It maps random seeds of a fixed type to elements of D. The current implementation uses binary trees of natural numbers as seeds:

```
data BT = Leaf(n::Nat)
        | Branch(n::Nat)(l::BT)(r::BT)
```

randomly generates $t_1, t_2, \cdots$ `::` `BT` while using $genD\ t_1$, $genD\ t_2, \cdots$ `::` D as test data. For example, one possible way to write a `genBoolExpr` (see Appendix B for `BoolExpr`) is

```
genBoolExpr::BT -> BoolExpr
genBoolExpr (Leaf n) = Var n
genBoolExpr (Branch n l r) =
 let x  = genNat l
     v  = genBool r
     e1 = genBoolExpr l
     e2 = genBoolExpr r
 in choice4 n (Var x)  (Val v)
              (Not e1) (And e1 e2)
```

where `choice4` chooses one of the four constructs depending on `n` mod 4.

As $genD$ is written in Agda/Alfa, we can prove properties about it in Agda/Alfa. For example, we can prove the important property of surjectivity ("any x `::` D could be generated.")

$$(x :: D) \texttt{ -> } \exists\, r :: \mathsf{BT}.\ genD\, r == x$$

The above `genBoolExpr` is surjective when `genNat` and `genBool` are, and proving so in Agda/Alfa is immediate.

The reader is referred to [11] for more information about the testing tool and test data generation.

3. Combining Testing and Proving

Our method interleaves proving steps and testing steps, as we see in the next sections. The benefits are the following:

- The essence of creative user interaction is the introduction of lemmas, including strengthening of induction hypotheses. This is often a speculative process. If a user fails to prove a possible lemma or its hypotheses, she must backtrack and try another formulation. Testing before proving is a quick way to discard wrong conjectures and inapplicable lemmas.
- Analysis of failed tests gives useful information for proving. Tests can fail both because of bugs in programs and because of bugs in specifications. Concrete counterexamples help our search for bugs of either type. Further, applying interactive proof steps to properties which are known to be false is an effective way to analyse the failure.
- All goals (properties to be proved) do not have a form which makes them testable, and even if they have testable form it may be hard to test them properly because it is difficult to write good test data generators. 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 logical gaps are introduced.
- Interactive proving increases confidence in the code coverage 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 to derive appropriate subgoals for the different components and branches. Testing these subgoals individually reduces the risk of missing test cases in the top-level testing.

4. Translating Haskell Programs to Agda/Alfa

In this section we will briefly describe how we faithfully translate Haskell programs into Agda/Alfa. Testing is already helpful at this stage.

The non-dependent fragment of the Agda/Alfa language is already close to the basic part of Haskell, but the function definitions are required to be total.

Two causes of partiality in Haskell are non-terminating computation and explicit use of the `error` function. Here we concentrate on the latter (The BDD implementation is structurally recursive). For example, `totalEvalBoolExpr`[2] is undefined on open expressions.

```
totalEvalBoolExpr :: BoolExpr -> Bool
totalEvalBoolExpr t = case t of
 (Var x) -> error "vars still present"
 (Val v) -> v
        ...
```

Such a partial function is made total in Agda/Alfa by giving it a more precise dependent type. One way[3] is to require, as an extra argument, a proof that the arguments are in the domain. With the *domain constraint* `closed` characterising the domain, the translation becomes:

```
closed(t::BoolExpr)::Bool = case t of
 (Var x    ) -> False
 (Val v    ) -> True
 (Not t1   ) -> closed t1
 (And t1 t2) -> closed t1 && closed t2

totalEvalBoolExpr ::
 (t::BoolExpr) -> T(closed t) -> Bool
totalEvalBoolExpr t p = case t of
 (Var x) -> case p of {}
 (Val v) -> v
        ...
```

Type-checking statically guarantees that `totalEvalBoolExpr` is never applied to an open `t` at run-time, as there is no well-typed value of type `T(closed t)`, which is the empty type.

The only modification we allow in translating from Haskell to Agda/Alfa is the addition of those extra proof-arguments (and extra type arguments, see Appendix A). This is a faithful translation in that the two versions have the same computational behaviour.

The domain constraint for one function propagates to others. For example, `getNextBddTableNode h` has a constraint that the argument `h::BddTable` be non-null. This propagates to the same constraint for `insertBdd h` $\cdots$ and `makeBddTableNode h` through a chain of function calls.

This propagation represents both benefits and difficulties. On the one hand, we locate a source of run-time errors when constraints on a caller function does not imply those on a called function. On the other hand, this may be a result of wrong characterisation of domains, which are not known a priori. Testing can help during the translations.

2 It evaluates boolean expressions to their values (Appendix B)

3 Another is to redesign data structure with dependent types, which is often more elegant but requires changes in program structure as well.

At one stage, we characterised the constraint on `buildBDDTable` by `VarIn t vs` (free variables of expression `t` are contained in the list `vs`):

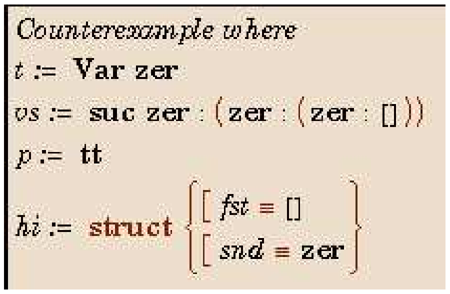
```
buildBddTable[4] :: (t::BoolExpr) ->
   (vs::[Nat]) -> VarIn t vs ->
   BddTI -> BddTI
```

The function constructs an intermediate value `h1 :: BddTable` with two recursive calls and then calls `makeBddTableNode h1 ···`. As above, this `h1` must be non-null, but it is not immediately clear whether `VarIn t vs` implies that. Now we might attempt a proof, but this may be costly if our guess turns out to be wrong. So we test instead.

A sufficient condition for the implication to hold is:

```
(t ::BoolExpr) ->
(vs::[Nat]) -> (p::VarIn t vs) ->
(hi::BddTI) ->
NotNull (buildBddTable t vs p hi).fst[5]
```

A test immediately reveals this to be false.



Counterexample where
$t :=$ **Var zer**
$vs :=$ **suc zer** : (**zer** : (**zer** : []))
$p :=$ **tt**
$hi :=$ **struct** $\left\{\begin{array}{l}[\ fst \equiv []\ ] \\ [\ snd \equiv \textbf{zer}\ ]\end{array}\right.$

Further, counterexamples always have `hi.fst = []`. Analysing the code in this light, we realize that `buildBddTable` is never called with `hi.fst` null. So we revise the constraint in the Agda/Alfa version to

```
buildBddTable:: (t::BoolExpr)->
 (vs::[Nat]) -> VarIn t vs ->
 (hi::BddTI) -> NotNull hi.fst -> BddTI
```

The revised sufficient condition

```
(t ::BoolExpr) ->
(vs::[Nat]) -> (p::VarIn t vs) ->
(hi::BddTI) -> (q::NotNull hi.fst) ->
NotNull (buildBddTable t vs p hi q).fst
```

passes the tests and the proof-argument required for that call to `makeBddTableNode` is constructed.

4 It computes `t`'s BDD with a variable ordering given by `vs`. A BDD is represented by a pair of type `BddTI = (BddTable, BddTableIndex)`. The first component is the list of linked, shared nodes of the BDD. The second is the index into the root node of the BDD. The function's last BddTI argument is an accumulating argument of 'currently constructed' BDD, threaded through recursion.

5 The record `struct{fst = a; snd = b}` in Agda/Alfa is used to represent the pair `(a, b)` in Haskell, and the dot notation is used to select a field value in a record.

5. Checking the Correctness of the BDD Implementation

The correctness of the BDD implementation is verified through a combination of testing and proving.

The informal statement of the correctness is: "the implementation computes a boolean expression to the BDD 1 (truth) if and only if it is a tautology." We formalise this as the following equivalence.[6]

```
(t ::BoolExpr) ->
(vs::[Nat]) -> (p::VarIn t vs) ->
iff
 (isBddOne
  (buildBddTable t vs p initBddOne tt))
 (taut t)
```

The function `taut::BoolExpr -> Bool` is the direct BDD implementation without sharing (see Appendix C). It can be seen as a definition of tautology in terms of the boolean algebra on binary decision trees.

The correctness statement has testable form, and it passes the test. Being reasonably sure that this statement is formulated correctly, we start proving it by decomposing it to a completeness (if part) and a soundness (only if part). The completeness part is

```
(t ::BoolExpr) ->
(vs::[Nat]) -> (p::VarIn t vs) ->
taut t ->
isBddOne
 (buildBddTable t vs p initBddOne tt)
```

A naive induction on `vs` does not go through, since the step case `vs` = `v`:`vs′` requires

```
isBddOne(buildBddTable t[v:=0] vs′ p1 h1 q1)
isBddOne(buildBddTable t[v:=1] vs′ p2 h2 q2)
```

where `h1` and `h2`, the values for the accumulating argument, are different from the initial value `initBddOne`. So we need to strengthen the induction hypothesis, generalising with respect to this accumulating argument.

In this situation, we typically need to analyse values that can occur (are reachable from the initial value) in the course of `buildBddTable` recursion and formulate an appropriate property of them. However, testing is cheap, so before such an analysis we try the strongest form (the most general form, within the constraint on `buildBddTable` from Section 4):

```
(t ::BoolExpr) ->
(vs::[Nat]) -> (p::VarIn t vs) ->
```

6 `isBddOne(`*table*, *index*`)` is true if *index* points to the node '1' in *table*. `initBddOne` is the initial value for the accumulating argument, containing nodes '1' and '0'. `tt` is the trivial proof that this is non-null.

```
(hi::BddTI) -> (q::NotNull hi.fst) ->
taut t ->
isBddOne (buildBddTable t vs p hi q)
```

Surprisingly, testing returns no counterexample. So we start verifying this in more detail. The base case amounts to

```
(t::BoolExpr) -> (p::Closed t) ->
taut t ->
totalEvalBoolExpr t p == true
```

This requires a logically straightforward but tedious proof: we content ourselves with successful tests and move on. With the strengthened induction hypothesis, the step case is reduced to the following properties:

```
(h::BddTable) -> (p::NotNull h) ->
(i, v0, v1::BddTableIndex) ->
v0 == 1 && v1== 1 ->                        (1)
(makeBddTableNode
  h p (i, v0, v1)).snd == 1

(t::BoolExpr) -> (x::Nat) ->
taut t ->                                   (2)
taut (t[x:=0]) && taut (t[x:=1])
```

We prove the first property. The second property, which can be considered as a natural property that a definition of tautology must satisfy in any case, is however only tested.

The proof of the soundness also requires a strengthening of the induction hypothesis. Again, we start by testing the strongest form.

```
(t ::BoolExpr) ->
(vs::[Nat]) -> (p::VarIn t vs) ->           (3)
(hi::BddTI) -> (q::NotNull hi.fst) ->
isBddOne(buildBddTable t vs p hi q) ->
taut t
```

Testing this fails. This time we do need to formulate an appropriate property of the reachable values for the accumulating argument `hi` in the course of `buildBddTable` recursion.

Here we aim to find an appropriate decidable predicate `Pos::BddTable -> Bool` on `hi.fst` that makes the induction go through. Combined testing and proving is useful in this context too. Counterexamples from a failed test hint at what should be excluded. When this is not specific enough, we may interactively decompose the failing property into more specific properties of components that fail. Counterexamples to these give better information for defining `Pos`, as well as a better understanding of the program.

The values of `hi.fst` in counterexamples to (3) give little information except that they are not reachable from `initBddOne`. Interactive steps show that (3) follows from the reverse direction of (1) and (2). At least one of them must fail, and tests reveal that the former, i.e.,

```
(h::BddTable) -> (p::NotNull h) ->
(i, v0, v1::BddTableIndex) ->
(makeBddTableNode                           (4)
  h p (i, v0, v1)).snd == 1 ->
v0 == 1 && v1 == 1
```

is false. Counterexamples to this are still not very informative. So we continue to decompose it to possible properties of component functions:

```
(h::BddTable) -> (p::NotNull h) ->
(e::BddTableEntry) ->                       (5)
(insertBddTable h p e).snd /= 1

(h::BddTable) -> (p::NotNull h) ->
(e::BddTableEntry) ->                       (6)
(q::isJust(findBddTableEntry h e) ->
  fromJust(findBddTableEntry h e) q
   /= 1
```

Both are false. The false statements and counterexamples are specific enough to help us define `Pos`. In all counterexamples to (5), `h` has the form[7] `(0,`$\cdots$`):h'`, and in counterexamples to (6) `h` always contains a node of the form `(1, Just` $\cdots$`)`. These are impossible since a BDD table is built up from the initial one `[(1,Nothing), (0,Nothing)]` by `insertBddTable`, which only adds nodes of the form `(i, Just e)` at the head, with increasing node index `i` > 1. It is easy to see this from the program, but only after we know what to look for.

To exclude these cases, we define `Pos` as follows:

```
Pos::BddTable -> Bool
Pos [x1,x2]    = [x1,x2] == initBddTable
Pos (x1:x2:xs) = x1.fst > 1 && Pos (x2:xs)
Pos  _         = false
```

This is weaker but much simpler than an exact characterisation of reachable values. There is no guarantee that this is enough, but testing again can quickly give a good indication.

Adding `Pos hi.fst` as a precondition to (3), a strengthening of the soundness property

```
(t ::BoolExpr) ->
(vs::[Nat]) -> (p::VarIn t vs) ->
(hi::BddTI) -> (q::NotNull hi.fst) ->
Pos hi.fst ->                               (3')
isBddOne(buildBddTable t vs p hi q) ->
taut t
```

passes the tests. So do (4), (5), and (6) with precondition `Pos h` added (let us call them (4'), (5'), and (6'), respectively).

7 A table `h` is a list of nodes, each of which is (node index, `Just`(var index, high branch, low branch)) except for (0, `Nothing`), (1, `Nothing`) for the constants.

The proofs of the implication (5) ∧ (6) ⇒ (4) and (reverse direction of 2) ∧ (4) ⇒ (3) can easily be adapted to the primed versions (5') ∧ (6') ⇒ (4') and (reverse direction of 2) ∧ (4') ⇒ (3') respectively, using the following lemma

```
(t ::BoolExpr) ->
(vs::[Nat]) -> (q::VarIn t vs) ->
(hi::BddTI) -> (p::NotNull hi.fst) ->
Pos hi.fst ->
Pos (buildBddTable t vs q hi p).fst
```

which is tested and proven. (5') is trivial to prove. A proof of (6'), which is 'clear' from a code inspection, would take slightly more effort than it is worth, so it is left as a tested assumption. Finally, the proof of (3') is instantiated with `hi = initBddOne` and trivial proofs of `NotNull` and `Pos`. This concludes the proof of soundness.

6. Conclusions and Future Work

Random testing and interactive theorem proving benefit each other. Though obvious as a general idea, few systems make it easy to combine the two. Using our testing extention to the proof-assistant Agda/Alfa in the context of a Haskell program verification, we have illustrated some concrete aspects of the benefits: testing before proving; testing to help formulating domain-constraints of functions; starting optimistically from a strong but false statement and then using counterexamples and interactive proof steps to obtain the right formulation; decomposing testing tasks; balancing the cost of proof-efforts against the gain in confidence; etc.

One direction of future work is to add various automatic theorem proving techniques to our system. Currently, a user must write a decidable predicate in the special form accepted by our tool. This is inflexible and inefficient both in terms of human efforts and testing implementation. There are many other forms of predicates with efficient decision procedures. We believe that the combination of such decision procedures with random test data generation and support for interactive proving has a great potential to be explored. For a start, we are integrating the very BDD program we verified here in our tool.

For another direction, we plan to extend and automate the method, covering more features of functional programming not present in type theory: possibly non-terminating general recursion, lazy evaluation manipulating infinite objects, IO operations, etc. Bove [2] provides a uniform method for representing general recursive partial programs in type theory. Hancock and Setzer [16] model interactive systems in type theory with a structure related to the Haskell's IO-monad. Applying these and other works, we hope to make our method more practical.

References

[1] L. Augustsson. Cayenne: a language with dependent types. In M. Berman and S. Berman, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP-98)*, volume 34, 1 of *ACM SIGPLAN Notices*, pages 239–250. ACM Press, 1998.

[2] A. Bove. *General Recursion in Type Theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2002.

[3] J. Bradley. Binary decision diagrams - a functional implementation. `http://www.cs.bris.ac.uk/~bradley/publish/bdd/`.

[4] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.

[5] T. Y. Chen, T. H. Tse, and Z. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In P. G. Frankl, editor, *Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis (ISSTA-02)*, pages 191–195. ACM Press, 2002.

[6] K. Claessen and J. 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 of *ACM Sigplan Notices*, pages 268–279, N.Y., Sept. 18–21 2000. ACM Press.

[7] C. Coquand. The Agda homepage. `http://www.cs.chalmers.se/~catarina/agda`.

[8] T. Coquand, R. Pollack, and M. Takeyama. A logical framework with dependently typed records. In *Proceedings of TLCA 2003*, volume 2701 of *LNCS*, pages 105–119, 2003.

[9] Cover - combining verification methods in software development. `http://dilbert.cs.chalmers.se/Cover/`.

[10] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

[11] P. Dybjer, Qiao Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[12] P. Dybjer and A. Setzer. A finite axiomatization of inductive and inductive-recursive definitions. In J.-Y. Girard, editor, *Proceedings of TLCA'99*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer-Verlag, 1999.

[13] P. Dybjer and A. Setzer. Indexed induction-recursion. In *PTCS: International Seminar on Proof Theory in Computer Science, LNCS*, 2001.

[14] M. M. Geller. Test data as an aid in proving program correctness. *Communications of the ACM*, 21(5):368–375, May 1978.

[15] T. Hallgren. The Alfa homepage. `http://www.cs.chalmers.se/~hallgren/Alfa`.

[16] P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic. 14th international workshop, CSL*,

volume 1862 of *Lecture Notes in Computer Science*, pages 317–331, 2000.

[17] S. Hayashi, R. Sumitomo, and Ken-ichiro Shii. Towards animation of proofs -testing proofs by examples. *Theoretical Computer Science*, 272:177–195, 2002.

[18] J. Hughes. Why functional programming matters. *The Computer Journal.*, 32(2):98–107, February 1989.

[19] S. Krstic and J. Matthews. Verifying BDD algorithms through monadic interpretation. *Lecture Notes in Computer Science*, 2294:182–195, 2002.

[20] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf type theory: an introduction*. Oxford University Press, 1990.

[21] Programatica: Integrating programming, properties, and validation. `http://www.cse.ogi.edu/PacSoft/projects/programatica/`.

[22] K. N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *Proc. 6th Asian Computing Science Conference (ASIAN'2000), Penang, Malaysia, Nov. 2000*, volume 1961 of *Lecture Notes in Computer Science*, pages 162–181. Springer-Verlag, 2000.

A. The Proof Assistant Agda/Alfa

Agda [7] is the latest version of the ALF-family of proof systems for dependent type theory developed in Göteborg since 1990. Alfa [15] is a window interface for Agda. They assist users as structure editors for *well-typed* programs and *correct* proofs, by checking each step of the construction on the fly, by presenting possible next steps, by solving simple problems automatically, etc.

Roughly speaking, the language of Agda/Alfa is a typed lambda calculus extended with the type `Set` of (small) types, which is equipped with formation of inductive data types, dependent function types, and dependent record types. Its concrete syntax comes to a large extent from Cayenne [1], a Haskell-related language with dependent types.

Basic data types are introduced much like in Haskell, e.g., the types of booleans and natural numbers are

```
Bool::Set = data true | false
Nat ::Set = data zer  | suc(n::Nat)
```

Dependent types may refer to terms of other types: e.g., one may have the type `Vect` $A\ n$ `::` `Set` for lists of specified length n `::` `Nat` with elements in A `::` `Set`.

A dependent function type `(`x `::` A`)` `->` $B[x]$ is the type of functions that sends argument a `::` A to a value of type $B[a]$, which may depend on a. The ordinary function type A `->` B is a special case. As an example we show how dependent types are used for representing information about the length of a list in the type of the `cons`-function for `Vect`

```
cons :: (A::Set) -> (n::Nat) ->
   A -> Vect A n -> Vect A (suc n)
```

As above, polymorphic functions explicitly take type arguments (e.g., `cons Nat`$\cdots$, `cons Bool`$\cdots$). Agda can infer them in most cases, and Alfa has an option to hide them from the user's view.

A dependent record type, also called a signature, `sig` $\{x_1$ `::` $A_1, \cdots, x_n$ `::` $A_n\}$, is the type of records (tuples) `struct` $\{x_1 = v_1, \cdots, x_n = v_n\}$ labelled with x_i's, where v_i `::` A_i and A_i may depend on $x_1, \cdots, x_{i-1}$. Haskell tuples are a non-dependent special case. The x_i component of record r is written $r.x_i$.

(Constructive) logic is represented in type theory by recognising propositions as types of their proofs. A proposition holds when it has a proof, i.e., when it is a non-empty type. A predicate P on type D is a function P`::`D `-> Set`, which is regarded as a propositional function since sets represent propositions. Logical connectives map to type forming operations; e.g., a function of type A `->` B is a proof of an implication, sending a proof of A to a proof of B; similarly, a function of type `(`x `::` D`)` `->` $P\,x$ is a proof of a universally quantified statement.

Truth values in `Bool` lift to propositions (sets) by

```
T::Bool -> Set
T True  = data tt  (singleton of trivial proof)
T False = data     (empty type; no proofs)
```

A predicate P on D is decidable if $\forall\, x \in D.\ P\,x \vee \neg P\,x$ constructively. It is equivalent to having some p `::` D `-> Bool` such that $\forall\, x \in D.P\,x \Leftrightarrow \mathsf{T}\,(p(x))$. In our tool, we require decidable predicates to have the latter form.

For a more complete account of the logical framework underlying Agda/Alfa including record types see the paper about structured type theory [8] and for the inductive definitions available in type theory, see Dybjer [10] and Dybjer and Setzer [12, 13].

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`.

B. Part of BDD Haskell Implementation [3]

```
module BddTable where

type BddTableIndex = Int

type BddTableEntry =
 (BddTableIndex,BddTableIndex,BddTableIndex)
-- (Variable Index, Low branch pointer,
                    High branch pointer)
type BddTable =
       [(BddTableIndex, Maybe BddTableEntry)]

type BddTI = (BddTable, BddTableIndex)
```

```
initBddTable::BddTable
initBddTable =
  (1, Nothing) : (0, Nothing) : []

initBddOne::BddTI
initBddOne = (initBddTable, 1)

toIndex::Bool -> BddTableIndex
toIndex v = if v then 1 else 0

makeBddTableNode::BddTable -> BddTableEntry
                -> BddTI
makeBddTableNode h (i, v0, v1)
  | (v0 == v1)    = (h, v0)
  | (isJust f)    = (h, fromJust f)
  | otherwise     =
       (insertBddTable h (i, v0, v1)) where
          f = findBddTableEntry h (i, v0, v1)

insertBddTable::BddTable -> BddTableEntry
                   -> BddTI
insertBddTable [] _ =
                  error "table not initialised"
insertBddTable hs e = ((ni, Just e):hs, ni)
      where ni = getNextBddTableNode hs

getNextBddTableNode::BddTable -> BddTableIndex
getNextBddTableNode [] =
                  error "table not initialised"
getNextBddTableNode ((i,_):_) = (i + 1)

findBddTableEntry::BddTable -> BddTableEntry
                    -> Maybe BddTableIndex
findBddTableEntry h e
 | null h2    = Nothing
 | otherwise  = Just (fst $ head h2) where
   h2 = dropWhile (f e) h
   f::BddTableEntry ->
    (BddTableIndex, Maybe BddTableEntry) -> Bool
   f _ (_, Nothing) = True
   f e1 (_, Just e2) = (e1 /= e2)

buildBddTable::BoolExpr -> [BoolVar]
                -> BddTI -> BddTI
buildBddTable t [] (h, _) =
   (h, toIndex $ totalEvalBoolExpr t)
buildBddTable t (x:xs) (h, i) =
   makeBddTableNode h1 (i, v0, v1) where
  (h0, v0) =
    buildBddTable (rewriteBoolExpr t (False,x))
                xs (h, i + 1)
  (h1, v1) =
    buildBddTable (rewriteBoolExpr t (True,x))
                xs (h0, i + 1)

module BoolAlgebra where

type BoolVar = Int

data BoolExpr = Var BoolVar
              | Val Bool
              | Not BoolExpr
              | And BoolExpr BoolExpr

totalEvalBoolExpr::BoolExpr -> Bool
totalEvalBoolExpr t =
  case t of
    Var x  ->
   error "variables still present in expression"
    Val v  -> v
    Not (Val v) -> if v then False else True
    Not (Not x) -> (totalEvalBoolExpr x)
    Not x -> not $ totalEvalBoolExpr x
    And (Val v) y ->
     if v then (totalEvalBoolExpr y) else False
    And x (Val v) ->
     if v then (totalEvalBoolExpr x) else False
    And x y -> (totalEvalBoolExpr x)
             && (totalEvalBoolExpr y)
```

C. The Tautology Checker

```
module Bdt where
 data Bdd = O | I | (/\) Bdd Bdd

 neg::Bdd -> Bdd
 neg O = I
 neg I = O
 neg (b /\ d) = neg b/\ neg d

 and_bdd::Bdd -> Bdd -> Bdd
 and_bdd O h’ = O
 and_bdd I h’ = h’
 and_bdd h O  = O
 and_bdd h I  = h
 and_bdd (b /\ d)(b’ /\ d’) =
             mkt (and_bdd b b’) (and_bdd d d’)

 mkt::Bdd -> Bdd -> Bdd
 mkt O O   = O
 mkt I I   = I
 mkt h1 h2 = h1 /\ h2

 next::Bdd -> Bdd
 next h =  h /\ h

 var::Nat -> Bdd
 var Zero     = I /\ O
 var (Succ n) = next (var n)

 bdt::BoolExpr -> Bdd
 bdt (Val True)  = I
 bdt (Val False) = O
 bdt (Var n)     = var n
 bdt (Not t)     = neg (bdt t)
 bdt (And t1 t2) = and_bdd (bdt t1)(bdt t2)

 taut::BoolExpr -> Bool
 taut t = bdt t == I
```