

Verifying Haskell Programs by Combining Testing, Model Checking and Interactive Theorem Proving

Peter Dybjer, Qiao Haiyan *

*Department of Computing Science,
Chalmers University of Technology and Göteborg University
412 96 Göteborg, Sweden*

Makoto Takeyama

*Research Center for Verification and Semantics
National Institute of Advanced Industrial Science and Technology
Nakoji 3-11-46, Amagasaki, Hyogo, 661-0974 Japan*

Abstract

We propose a program verification method that combines random testing, model checking and interactive theorem proving. Testing and model checking are used for debugging programs and specifications before a costly interactive proof attempt. During proof development, testing and model checking quickly eliminate false conjectures and generate counterexamples which help to correct them. With an interactive theorem prover we also ensure the correctness of the reduction of a top level problem to subproblems that can be tested or proved. We demonstrate the method using our random testing tool and BDD-based (binary decision diagrams) tautology checker, which are added to the Agda/Alfa interactive proof assistant for dependent type theory. In particular we apply our techniques to the verification of Haskell programs. The first example verifies the BDD checker itself by testing its components. The second uses the tautology checker to verify bitonic sort together with a proof that the reduction of the problem to the checked form is correct.

Key words: program verification, random testing, proof-assistants, type theory, binary decision diagrams, Haskell.

* A preliminary version of this paper was presented at the 3rd International Conference on Quality Software (QSIC03)[12]

* Corresponding author. *Email address:* `qiao@cs.chalmers.se` (Qiao Haiyan).
Email addresses: `peterd@cs.chalmers.se` (Peter Dybjer),
`makoto.takeyama@aist.go.jp` (Makoto Takeyama).

1 Introduction

A main goal of the theorem proving community is to use proof assistants for producing correct programs. However, in spite of continuous progress in the area (better implementations, more automatic proof search, better user interfaces, larger proof libraries, etc.), 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 because, as Dijkstra emphasised, testing can only show the presence of errors, not the absence of them. Nevertheless, testing is still very useful as a tool for detecting and eliminating bugs.

Most research on testing has been concerned with traditional imperative programs. However, an interesting tool called Quick Check for random testing of functional programs (written in the lazy functional programming language Haskell) has recently been developed by Claessen and Hughes [6]. With this tool, correctness properties of functional programs can be tested by running it on 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. Therefore we thought that 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. More generally, we can debug sub-goals that occur during proof. Analysing the counter examples returned by failed tests, we can focus the search for error to smaller fragments of code and specifications. Moreover, we can balance cost and confidence by testing assumptions instead of proving. It may also be interesting to study testing methods systematically in the context of the logic of functional programs.

To demonstrate the methodology, we have extended the Agda/Alfa proof assistant for dependent type theory with a QuickCheck-like testing tool.

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, \dots , testing only the top-level of f is often inadequate. It gives little information about how much each 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 also used during proving to eliminate wrongly formulated lemmas quickly.

As a first example, we show how a Haskell implementation of the BDD algorithm by J. Bradley ([3], see Appendix B) is verified. 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., [21,25], but our interest is rather in trying to verify an existing Haskell program not necessarily written with verification in mind.

We then extend Agda/Alfa with a tool for checking if a boolean formula is a tautology. This tool is written in Haskell and is based on the BDD algorithm that we verified,

Finally, we show the verification of a sorting algorithm (bitonic sort) using a combination of interactive proving, model checking, and random testing. The main idea is that the combination can make a rigorous connection between a higher level description of a problem and what is actually model checked after modelling, abstraction, etc. A technique due to Day, Launchbury, and Lewis [10], uses parametricity to reduce the correctness of a polymorphic program to the correctness of its boolean instance. We show how to interactively prove the following results beyond the realm of model checking: (1) the parametricity result itself; (2) that the reduction is correct, that is, the symbolic expression fed to the checker correctly relates to the correctness of the boolean instance. We then show how to model check the correctness of the boolean instance.

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 [23] at the Oregon School of Science and Engineering, 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 [19], who used testing to debug lemmas while doing proofs in his PX-system. Geller [16] argued that test data can be used while proving programs correct. This paper

reports a case study based on our previous work on combining testing and proving in dependent type theory [13].

Plan of the paper: In Section 2 we describes our testing tool. Section 3 is a general discussion on the benefits of combining testing and proving. In Section 4 we discuss how to verify Haskell programs in Agda/Alfa. In Section 5 we show how to verify a BDD implementation written in Haskell. In Section 6 we discuss the tool using BDDs. In Section 7 we discuss how a sorting algorithm is verified. Section 8 concludes the article with future directions for the work.

2 The Random 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:

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

For decidability, we require the predicates (now themselves dependent types) to have the decidable form $\mathbb{T} (p[x_1, \cdots, x_n])$ (the term $p :: \text{Bool}$ lifted¹ to the

type level). We call properties of this form *testable*.

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. We use `taut(t[x:=0])` (and `taut(t[x:=1])`) to denote the expression `t` where the boolean variable `Var x` is replaced by the boolean value `False` (and `True`, respectively).

By making this a goal (`?o`), tests can be invoked from the Alfa menu (Fig. 1).

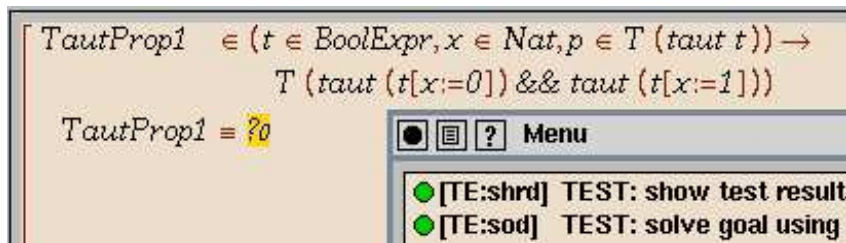


Fig. 1. Testing tool (`::` is displayed as `∈`)

The testing procedure is as follows:

```
do (at most maxCheck 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
```

where `maxCheck` is a predetermined parameter. The tool then reports the success of `maxCheck` tests, or the counterexample `(t, x)` found. In the displayed case the tool reports success.

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

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

So $genD$ has type $Rand \rightarrow D$. The idea is that the tool randomly generates $t_1, t_2, \dots :: Rand$ while using $genD\ t_1, genD\ t_2, \dots :: D$ as test data. For example, one possible way to write `genBoolExpr` (see Appendix B for `BoolExpr`) is

```
genBoolExpr :: Rand -> 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 `mod n 4`.

The advantage of writing $genD$ in Agda/Alfa is that we may formally prove its properties. For example, we can prove the important property of surjectivity (“any $x :: D$ could be generated.”)

$$(x :: D) \rightarrow \exists r :: Rand. genD\ r == x$$

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

The reader is referred to [13] for more information about the testing tool and test data generation.

3 Combining Testing and Proving

Our method interleaves proof steps and tests, 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 false 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. Furthermore, applying interactive proof steps to properties that are known to be false is an effective way to analyse the failure.

- All goals (properties to be proved) do not have a form that 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.

We will demonstrate how Haskell programs can be verified by using the system.

4 Verifying Haskell Programs in Agda/Alfa

Haskell is a functional programming language with higher order functions and lazy evaluation. As a consequence it supports the writing of modular programs [20]. 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 [22], as we believe that its logic matches Haskell's functional style well. Our case studies are developed in the proof assistant Agda/Alfa [7,17] 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 user effort. 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.

In this section we 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 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` is undefined on open expressions, which evaluates boolean expressions to their values (see Appendix B).

```
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 way² 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 :: BoolExpr -> Bool
closed t =
  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: in that case `T (closed t)` is an empty type, so there can be no well-typed value `p` to make `totalEvalBoolExpr t p` well-typed.

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 the sense that the two versions have the same computational behaviour.

The domain constraint for one function propagates to others. For example,

² Another is to redesign the data structure using dependent types. This is often more elegant but requires changes in program structure as well.

`getNextBddTableNode h` has a constraint that the argument `h :: BddTable` be non-null. This propagates to the same constraint for `insertBdd h ...` 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 do 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 translation.

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

```
buildBddTable3 :: (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).fst4
```

A test immediately reveals this to be false (see Fig. 2 for a counter example, where 0 is displayed as `zer`).

Further, counterexamples always have `hi.fst = []`. Analysing the code in this light, we realise that `buildBddTable` is never called with `hi.fst == []`.

³ The function `buildBddTable` 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.

⁴ 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.

```

Counterexample where
t := Var zer
vs := suc zer : (zer : (zer : []))
p := tt
hi := struct {
  [fst ≡ []]
  [snd ≡ zer]
}

```

Fig. 2. A counter example

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.

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.⁵

```

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

```

⁵ `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.

Here we omitted the `T` on the last line. 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 a 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, because 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) ->
(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) ->
```

$$\begin{aligned}
&v0 == 1 \ \&\& \ v1 == 1 \ \rightarrow \\
&(\text{makeBddTableNode } h \ p \ (i, \ v0, \ v1)).\text{snd} == 1
\end{aligned}
\tag{5.1}$$

$$\begin{aligned}
&(t :: \text{BoolExpr}) \ \rightarrow \ (x :: \text{Nat}) \ \rightarrow \\
&\text{taut } t \ \rightarrow \ \text{taut } (t[x:=0]) \ \&\& \ \text{taut } (t[x:=1])
\end{aligned}
\tag{5.2}$$

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

$$\begin{aligned}
&(t :: \text{BoolExpr}) \ \rightarrow \\
&(\text{vs} :: [\text{Nat}]) \ \rightarrow \ (p :: \text{VarIn } t \ \text{vs}) \ \rightarrow \\
&(\text{hi} :: \text{BddTI}) \ \rightarrow \ (q :: \text{NotNull } \text{hi.fst}) \ \rightarrow \\
&\text{isBddOne } (\text{buildBddTable } t \ \text{vs } p \ \text{hi } q) \ \rightarrow \ \text{taut } t
\end{aligned}
\tag{5.3}$$

Testing this fails. This time we 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 (5.3) give little information except that they are not reachable from `initBddOne`. Interactive steps show that (5.3) follows from the reverse direction of (5.1) and (5.2). At least one of them must fail, and tests reveal that the former, i.e.,

$$\begin{aligned}
&(h :: \text{BddTable}) \ \rightarrow \ (p :: \text{NotNull } h) \ \rightarrow \\
&(i, \ v0, \ v1 :: \text{BddTableIndex}) \ \rightarrow \\
&(\text{makeBddTableNode } h \ p \ (i, \ v0, \ v1)).\text{snd} == 1 \ \rightarrow \\
&\quad v0 == 1 \ \&\& \ v1 == 1
\end{aligned}
\tag{5.4}$$

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

$$\begin{aligned}
&(h :: \text{BddTable}) \ \rightarrow \ (p :: \text{NotNull } h) \ \rightarrow \\
&(e :: \text{BddTableEntry}) \ \rightarrow \\
&(\text{insertBddTable } h \ p \ e).\text{snd} /= 1
\end{aligned}
\tag{5.5}$$

```

(h :: BddTable) -> (p :: NotNull h) ->
(e :: BddTableEntry) -> (q :: isJust(findBddTableEntry h e) ->
  fromJust(findBddTableEntry h e) q /= 1) (5.6)

```

Both (5.5) and (5.6) are false. The false statements and counterexamples are specific enough to help us define a predicate, `Pos`, that characterises the set of possible BDD values. In all counterexamples to (5.5), `h` has the form⁶ $(0, \dots) : h$, and in counterexamples to (5.6) `h` always contains a node of the form $(1, \text{Just } \dots)$. These are impossible because a BDD table is built up from the initial one $[(1, \text{Nothing}), (0, \text{Nothing})]$ by `insertBddTable`, which only adds nodes of the form $(i, \text{Just } e)$ 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 strong enough, but testing again can quickly give a good indication.

Adding `Pos hi.fst` as a precondition to (5.3), a strengthening of the soundness property becomes the following:

```

(t :: BoolExpr) ->
(vs :: [Nat]) -> (p :: VarIn t vs) ->
(hi :: BddTI) -> (q :: NotNull hi.fst) ->
Pos hi.fst -> (5.3')
isBddOne(buildBddTable t vs p hi q) -> taut t

```

Now (5.3') passes the tests. So do (5.4), (5.5), and (5.6) with precondition `Pos h` added (let us call them (5.4'), (5.5'), and (5.6'), respectively).

The proofs of implications $(5.5) \wedge (5.6) \Rightarrow (5.4)$ and (reverse direction of 5.2) $\wedge (5.4) \Rightarrow (5.3)$ can easily be adapted to the primed versions $(5.5') \wedge (5.6') \Rightarrow (5.4')$ and (reverse direction of 5.2) $\wedge (5.4') \Rightarrow (5.3')$ respectively. It uses the lemma

```

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

```

⁶ A table `h` is a list of nodes, each of that is (node index, `Just`(var index, high branch, low branch)) except for $(0, \text{Nothing})$, $(1, \text{Nothing})$ for the constants.

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

that is tested and proved. (5.5') is trivial to prove. A proof of (5.6'), 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 (5.3') is instantiated with `hi = initBddOne` and trivial proofs of `NotNull` and `Pos`. This concludes the proof of soundness.

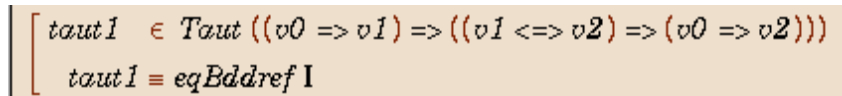
6 Combining Testing and Model Checking

We often need to check if a boolean formula is a tautology. We could use `taut`, but it is computationally expensive. BDDs (Binary Decision Diagrams) is an efficient method for this purpose. We augment Agda/Alfa with a plug-in that is based on the BDD implementation in Haskell we verified before. When the current goal has a type of the form `Taut e` in Alfa, the user can invoke the BDD tool via an Alfa menu to check if `e` is a tautology. It computes the BDD for a given boolean expression, and signals success or returns a counter model.

For example, if we model check the following goal (`v0` abbreviates `Var 0`):

```
Taut ((v0 => v1) => ((v1 => v2) => (v0 => v2))) (6.1)
```

then the BDD tool returns "success". The goal can also be solved interactively by the proof object `eqBddref I`, a proof that the equality on `Bdd` is reflexive (see Fig. 3; `Taut t = T (eqBdd t I)`).



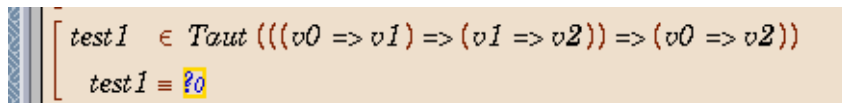
```

┌ taut1 ∈ Taut ((v0 => v1) => ((v1 <=> v2) => (v0 => v2)))
└ taut1 ≡ eqBddref I

```

Fig. 3. Solved by reflexivity

Now suppose we want to check the goal in Fig. 4 parenthesised differently from (6.1):



```

┌ test1 ∈ Taut (((v0 => v1) => (v1 => v2)) => (v0 => v2))
└ test1 ≡ ?0

```

Fig. 4. A false property

In this case, we get a counter model. Fig. 5 shows how this is displayed as an assignment of boolean values to variables that makes the property in Fig. 4 false. In this example, `v2 = 0`, `v1 = 0`, `v0 = 1`.

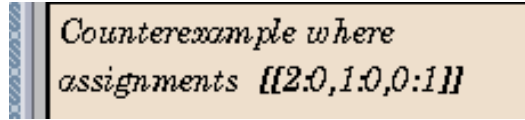


Fig. 5. A counter model

Our testing tool combines testing and model checking in one system. If a boolean formula contains parameters, then we can use our random testing tool to test if the formula is a tautology on randomly generated parameters.

More generally, our model checking tool can check properties of the following form by using both testing and model checking:

$$\begin{aligned}
 &(x_1 :: D_1) \rightarrow \cdots \rightarrow (x_n :: D_n[x_1, \dots, x_{n-1}]) \rightarrow \\
 &P_1[x_1, \dots, x_n] \rightarrow \cdots \rightarrow P_m[x_1, \dots, x_n] \rightarrow \qquad (6.2) \\
 &\text{Taut } (q[x_1, \dots, x_n])
 \end{aligned}$$

Here $D_i[x_1, \dots, x_{i-1}]$ is a type depending on x_1, \dots, x_{i-1} ; P_i is a precondition satisfied by relevant data; and q is the boolean formula being tested with the parameters x_1, \dots, x_n . The predicates P_i must be decidable.

The user chooses an option “BDD-check” in one of the menus provided by Alfa. The plug-in checks the property in the following way:

For predefined maxCheck (the number of maximum checks) and maxFail (the number of maximum test cases that fail the preconditions), the procedure is

```

s = 1; f = 1
while (s ≤ maxCheck and f ≤ maxFail)
  generate test data (x1, ..., xn)
  if some Pi[x1, ..., xn] is false then
    f = f + 1
  else if q[x1, ..., xn] computes to BDD 1 then
    s = s + 1
  else return (x1, ..., xn) as a counterexample and stop
if s ≥ maxCheck then
  report “success” and stop
else report “argument exhausted” and stop.

```

The test data x_i is generated by a user-defined generator $genD_i :: \text{Rand} \rightarrow D_i$ as explained in Section 2.

Users can also define special purpose generators to generate only and all those data satisfying preconditions (and prove them so). This is particularly useful when data satisfying preconditions are sparse among all data of appropriate

types, or when filtering by preconditions distort the distribution of data. See also [13].

When model checking succeeds for a goal without parameters, it is guaranteed to be solved by the proof object `eqBddref I` as in Fig. 3.

7 Verifying a Sorting Algorithm

In this section, we demonstrate how to use the system to verify a polymorphic bitonic sort algorithm. We follow the approach taken by Day, Launchbury and Lewis [10] but formalise it in Agda/Alfa. First a parametricity theorem for the function `bitonic` is proved interactively. This reduces the correctness of polymorphic bitonic sort to that of bitonic sort on lists of booleans. The latter is then checked by the BDD based testing tool.

7.1 The Bitonic Sort Algorithm

Bitonic sort is an efficient divide-and-concur sorting algorithm. In the first phase, an input list is divided into two halves, the first of which is recursively sorted in ascending order, and the second descending. The concatenation of the two is a ‘bitonic’ list. The second phase sorts bitonic lists. It divides a bitonic list into two and compares and swaps their elements at each position pairwise. The result is a pair of bitonic lists where each element in one is no smaller than any element of the other. Applying the second phase recursively to each list, we obtain the sorted result.

Bitonic sort is parameterised by a comparison-swap function. We took the Haskell program in [10] (Appendix E) and implemented a dependently typed version in Agda/Alfa: it takes a list of length n and returns a list of the same length.⁷

In what follows, we use letters a, b for type variables. We write a^n for the type `Vec a n` of lists of length n with elements in a , and $a \times b$ for the cartesian product type `Prod a b`. A typical object in a^n is written (x_1, \dots, x_n) . (See appendix A.)

⁶ Paul Taylor’s `diagrams` package was used to prepare this section.

⁷ The bitonic sort algorithm assumes lists of length 2^n . We could implement the algorithm on lists of length 2^n in Agda/Alfa using dependent types. Here we implement the algorithm on lists of any length following [10].

Bitonic sort is defined by the following function:

```

bitonic_sort :: (a :: Set) -> (f :: a × a -> a × a) ->
              (n :: Nat) -> an -> Bool -> an
bitonic_sort a f Zero      () up = ()
bitonic_sort a f (Succ Zero) (x) up = (x)
bitonic_sort a f (Succ (Succ n)) xs up =
  let n1 = fst (splitN n)
      n2 = snd (splitN n)
      yzs :: an1 × an2 = splitV n xs
      ys' :: an1 = bitonic_sort a f n1 (fst yzs) True
      zs' :: an2 = bitonic_sort a f n2 (snd yzs) False
  in bitonic_to_sorted a f n (ys' ++ zs') up

```

Here we list the types of some functions used; Appendix F has their definitions.

```

bitonic_to_sorted :: (a :: Set) -> (f :: a × a -> a × a) ->
                    (n :: Nat) -> an -> Bool -> an

pairwise :: (a :: Set) -> (f :: a × a -> a × a) ->
           (n :: Nat) -> an1 × an2 -> an1 × an2

splitV :: (a :: Set) -> (n :: Nat) -> an -> an1 × an2

splitN :: Nat -> Nat × Nat

```

Later we will instantiate the parameter `f` of the function `bitonic_sort` to a comparison-swap function. Assuming a total ordering \leq on the set `a`, the comparison-swap function `f` is

```
f (x, y) = if (x ≤ y) (x, y) (y, x)
```

Then `bitonic_sort f n xs up` sorts `xs` in increasing order if `up` is `True`, and in decreasing order if `up` is `False`.

We define the increasing order bitonic sort function:

```

bitonic :: (a :: Set) -> (f :: a × a -> a × a) ->
          (n :: Nat) -> an -> an

bitonic a f n xs = bitonic_sort a f n xs True

```

7.2 The Specification of Sorting Functions

A list `bs` is a (the) sorted list of `as`, if `bs` is a permutation of `as` and if elements in `bs` are in increasing order, that is, sorted. A function `sort` is a correct

sorting function if $sort\ as$ is a sorted list of as for any list as . However, we will only discuss the sortedness in this paper, ignoring the (easier) permutation property.

One way to define the sortedness of a list with respect to a given order relation is the following.

```
sorted :: (a :: Set) -> ((≤) :: a -> a -> Bool) ->
        (n :: Nat) -> an -> Bool

sorted a (≤) Zero      ()          = True
sorted a (≤) (Succ Zero) (x)      = True
sorted a (≤) (Succ (Succ n)) (x1,x2,xs') =
        (x1 ≤ x2) && (sorted a (≤) (Succ n) (x2, xs'))

Sorted :: (a :: set) -> ((≤) :: a -> a -> Bool) ->
        (n :: Nat) -> an -> Set

Sorted a (≤) n xs = T (sorted a (≤) n xs)
```

The formal statement that $sort :: (n :: Nat) \rightarrow a^n \rightarrow a^n$ is a sorting function with respect to an order (\leq) is

$$(n :: Nat) \rightarrow (xs :: a^n) \rightarrow Sorted\ a\ (\leq)\ n\ (sort\ n\ xs)$$

7.3 Proving The Parametricity Theorems

Knuth has a theorem on those sorting algorithms that are based only on comparison-swaps: if such a sorting algorithm can sort booleans correctly, then it can sort integers correctly. His theorem is an instance of the parametricity theorem [24], that is, one of “theorems for free” [26] derivable from the type of a function. In this section we state the parametricity theorem for the function `bitonic`.

The parametricity theorem for the typing of `bitonic` is that, for all functions f , g , and h making the left diagram commute, the right diagram commutes too, for each n .

$$\begin{array}{ccc}
 \begin{array}{ccc}
 a \times a & \xrightarrow{f} & a \times a \\
 \downarrow \langle h, h \rangle & & \downarrow \langle h, h \rangle \\
 b \times b & \xrightarrow{g} & b \times b
 \end{array} & \Rightarrow &
 \begin{array}{ccc}
 a^n & \xrightarrow{\text{bitonic } a\ f} & a^n \\
 \downarrow \text{map } h & & \downarrow \text{map } h \\
 b^n & \xrightarrow{\text{bitonic } b\ g} & b^n
 \end{array}
 \end{array}$$

Here we used functions $\langle h, h \rangle$ and $\text{map } h$ defined by $\langle h, h \rangle(x_1, x_2) = (h x_1, h x_2)$ and $\text{map } h(x_1, \dots, x_n) = (h x_1, \dots, h x_n)$ ($\text{map } h$ actually takes a, b , and n as arguments. Here and elsewhere, we sometimes suppress such arguments for readability). The formal statement in Agda/Alfa is

```
(f :: a × a -> a × a) ->
(g :: b × b -> b × b) ->
(h :: a -> b) ->
(q :: (x :: a × a) -> T ((h,h) (f x) == g ((h,h) x)))
(n :: Nat) ->
(xs :: an) ->
T (map h (bitonic a f n xs) == bitonic b g n (map h xs))
```

where we overloaded the equality symbol `==`.

This theorem is proved by induction on the size n ⁸. While proving this theorem, we need to prove similar parametricity theorems for other component functions. For example, the theorem for the function `pairwise` is

$$\begin{array}{ccc}
 a \times a & \xrightarrow{f} & a \times a \\
 \langle h, h \rangle \downarrow & & \langle h, h \rangle \downarrow \\
 b \times b & \xrightarrow{g} & b \times b
 \end{array}
 \Rightarrow
 \begin{array}{ccc}
 a^n & \xrightarrow{\text{pairwise } a f} & a^n \\
 \text{map } h \downarrow & & \text{map } h \downarrow \\
 b^n & \xrightarrow{\text{pairwise } b g} & b^n
 \end{array}$$

Another for the function `splitV` is

$$\begin{array}{ccc}
 a^n & \xrightarrow{\text{splitV } a} & a^{n_1} \times a^{n_2} \\
 \text{map } h \downarrow & & \text{map } h \times \text{map } h \downarrow \\
 b^n & \xrightarrow{\text{splitV } b} & a^{n_1} \times b^{n_2}
 \end{array}$$

where $(n_1, n_2) = \text{splitN } n$.

7.4 The Correctness of Bitonic Sort

To prove that `bitonic f` is a sorting function, we need that the parameter `f` is really a comparison-swap function. The comparison-swap function parame-

⁸ It is also possible to prove it at once for all appropriate functions, by induction on type- and term-expressions that are suitably internalised.

terised on an order le on set a is

```
cmpSwap :: (a -> a -> Bool) -> a × a -> a × a
cmpSwap le (x, y) = if (le x y) (x, y) (y, x)
```

Bitonic sort parameterised in set a of list elements and an order leA (\leq) is

```
bitonicA :: (a :: Set) -> (a -> a -> Bool) ->
           (n :: Nat) -> an -> an
bitonicA a leA = bitonic a (cmpSwap leA)
```

Bitonic sort on booleans is an instance with leB being the implication.

```
bitonicB :: (n :: Nat) -> Booln -> Booln
bitonicB = bitonicA Bool leB
```

Our goal now is to formally prove that if $bitonicB$ is a sorting function, then $bitonicA a leA$ is also a sorting function:

```
(a :: Set) -> (n :: Nat) ->
((bs :: Booln) -> Sorted Bool leB (bitonicB n bs)) ->      (7.1)
(as :: an) -> Sorted a leA (bitonicA a leA n as)
```

We note that this goal is not of the form amenable to model checking or testing.

The proof uses the parametricity theorem for the function $bitonic$ in the following argument. Suppose that the list $bitonicA a leA n xs$ is not sorted for some $xs :: a^n$. Then it must contain an x occurring before y such that $x > y$ (not $(leA x y)$). Let h be the function $h z = z > y$. The premise of the parametricity theorem for $bitonic$ holds with this h .

$$\begin{array}{ccc}
 a \times a & \xrightarrow{\text{cmpSwap } leA} & a \times a \\
 \langle h, h \rangle \downarrow & & \downarrow \langle h, h \rangle \\
 Bool \times Bool & \xrightarrow{\text{cmpSwap } leB} & Bool \times Bool
 \end{array}$$

So we have the diagram below commuting.

$$\begin{array}{ccc}
 a^n & \xrightarrow{\text{bitonicA } a \text{ } leA \text{ } n} & a^n \\
 \text{map } h \downarrow & & \downarrow \text{map } h \\
 Bool^n & \xrightarrow{\text{bitonicB } n} & Bool^n
 \end{array}$$

Now, in the list $ys = \text{map } h (\text{bitonicA } a \text{ leA } n \text{ } xs)$, `True` occurs before `False`, because $h \ x = \text{True}$ and $h \ y = \text{False}$. Hence ys is not sorted with respect to `leB`. But the diagram shows ys is equal to $\text{bitonicB } n (\text{map } h \ xs)$, which is sorted by assumption, whatever the list $\text{map } h \ xs$ may be. This is a contradiction.

We have thus reduced the correctness of `bitonic` on any type (that the result is sorted) to checking of the following.

$$\begin{aligned} & (n :: \text{Nat}) \rightarrow \\ & (bs :: \text{Bool}^{2^n}) \rightarrow \text{Sorted } \text{leB } (\text{bitonicB } 2^n \text{ } bs) \end{aligned} \tag{7.2}$$

This property is easy to test using our testing tool. We can randomly generate values for n and bs , and check if `bitonic sort` sorts bs .

Using our BDD tool, however, we can do much better. For each n , we can model check the whole of universally quantified proposition (type)

$$(bs :: \text{Bool}^{2^n}) \rightarrow \text{Sorted } \text{leB } (\text{bitonicB } 2^n \text{ } bs)$$

We do so by constructing a boolean expression such that it is a tautology if and only if this type is inhabited. The boolean expression, which we call `ok n`, is parameterised in $n :: \text{Nat}$.

$$\text{ok} :: \text{Nat} \rightarrow \text{BoolExpr}$$

Combining random generation of values for n and model checking of `ok n` by the BDD tool, we can check whether `bitonic sort` on lists of booleans is correct for all inputs of size 2^n . This is of course much stronger than just testing it for some of inputs out of 2^{2^n} possibilities!

The boolean expression `ok n` reflects (internalises) in its structure the construction of `Sorted leB (bitonicB 2n bs)`. To define `ok n`, we first generalises the function `sorted` to `sorted'` with more parametrisation.

$$\begin{aligned} \text{sorted}' & :: (a, b :: \text{Set}) \rightarrow b \rightarrow (b \rightarrow b \rightarrow b) \rightarrow \\ & \quad (a \rightarrow a \rightarrow b) \rightarrow \\ & \quad (n :: \text{Nat}) \rightarrow a^n \rightarrow b \\ \text{sorted}' \ a \ b \ \text{true} \ (*) \ (\leq) \ \text{Zero} \quad & () \quad = \ \text{true} \\ \text{sorted}' \ a \ b \ \text{true} \ (*) \ (\leq) \ (\text{Succ } \text{Zero}) \quad & (x) \quad = \ \text{true} \\ \text{sorted}' \ a \ b \ \text{true} \ (*) \ (\leq) \ (\text{Succ } (\text{Succ } n)) \ (x_1, x_2, xs') & = \\ & (x_1 \leq x_2) * (\text{sorted}' \ a \ b \ \text{true} \ (*) \ (\leq) \ (\text{Succ } n) \ (x_2, xs')) \end{aligned}$$

While `sorted` computes a value in `Bool` using `True` and `(&&)`, the function `sorted'` takes the the return type b as a parameter along with appropriately

typed operations on it. So `sorted` as previously defined can be recovered from `sorted'` using `True` and `(&&)`.

```
sorted a = sorted' a Bool True (&&)
```

The function `sortedBE` instead uses the constant expression `Val True` and the syntactic operation `(*)` that forms conjunction expressions.

```
sortedBE a = sorted' a BoolExp (Val True) (*)
```

Given a function $R :: a \rightarrow a \rightarrow \text{BoolExp}$, `sortedBE a R 4 (x1, x2, x3, x4)` is the boolean expression $(R\ x_1\ x_2) * ((R\ x_2\ x_3) * ((R\ x_3\ x_4) * (\text{Val True})))$ stating that the list is sorted with respect to R .

We define `ok n` as follows, where `(+)` and `(=>)` are the syntactic operations to form disjunction and implication expressions, respectively.

```
nVars :: (n :: Nat) -> BoolExpn
nVars Zero      = ()
nVars (Succ n) = (Var n, nVars n)

cmpSwapBE :: BoolExp -> BoolExp -> BoolExp
cmpSwapBE e1 e2 = (e1 * e2, e1 + e2)

ok :: Nat -> BoolExp
ok n = sortedBE BoolExp (=>) n
      (bitonic BoolExp cmpSwapBE n (nVars n))
```

This is a form of symbolic evaluation. We start from the list `nVars n` of n distinct input variables. Each element of the list `(bitonic ...)` is a boolean expression that represents the output of `bitonic` at that position as the function of those input variables. The expression `ok n` is the one stating that those outputs are sorted with respect to `(=>)`, which is a complex function of the input variables.

Our task now is divided in two parts. The first is to check that

$$(n :: \text{Nat}) \rightarrow \text{Taut } (\text{ok } 2^n) \tag{7.3}$$

This fits in the form (6.2), and our BDD tool can check this by combining random generation and model checking.

The second is to prove that (7.3) implies (7.2). The proof, given in Appendix D, uses the parametricity theorem for `sorted'`. We remark that the relationship between the program to be verified and the formulas to be input to model checkers is usually an informal one. We can formally reason this because the

program, the specification, and moreover the tool (the function `ok`) to construct the formula are all in a single framework. We believe this to be a unique advantage of our integrated approach based on an expressive type theory.

Note that `bitonic` is correct only when `n` is a power of 2. Replacing `Bool2n` by `Booln` in (7.3) yields counterexamples. In Fig. 6, where `n = 3`, the second list `[1:1, 0:0]` represents the assignment `v1 = 1` and `v0 = 0`, which is a counter model.

```

Counterexample where
n := suc (suc (suc zer))
assignments [[2:1,1:0,0:1],[1:1,0:0],[2:1,1:0,0:0]]

```

Fig. 6. A counter model to `ok n` for `n = 3`

In general, we can test and model check a parameterised boolean expression `ok :: A -> BoolExpr` in goals of the form

$$(x :: A) \rightarrow T (p \ x) \rightarrow \text{Taut } (\text{ok } x)$$

by randomly generating values for `x`.

Admittedly, the present example of generating sizes randomly is not very convincingly useful; it is probably just as good in practise to systematically try `n = 0, 1, \dots`.

Nevertheless, we have illustrated the principle, and hope to find more useful applications in the future, and thus show that having a system combining testing, model checking and interactive proving is useful.

8 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 extension to the proof-assistant Agda/Alfa in the context of Haskell program verification, we have illustrated some 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.

Model checking and theorem proving are also complementary. One typical method of combining model checking and theorem proving is to use abstraction. Day, Launchbury and Lewis [10] connected Haskell with a BDD package and illustrated how the parametric nature of Haskell’s polymorphism is used to lift the result of the BDD analysis to arbitrary datatypes. This is done informally. By comparison, using an extended version of Agda/Alfa, this can also be done formally in one system.

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.

Another direction is 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 [18] 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.

We are looking at a tableau prover in Haskell in that lazy evaluation and IO operations are essential. The experiment is promising. The Cover project at Chalmers University of Technology is actively studying approaches to bridge remaining gaps between Agda/Alfa and Haskell, both foundational (e.g., implicit quantification, internal representations of logic of functional programs) and technological (e.g., automatic translation tools).

Logical aspects of testing also need further investigation. What it means to be a “faithful” translation for, e.g., a partial Haskell functions? Can we have a logic in Agda/Alfa that gives some guarantee on the quality of testing? Can tests be designed to fulfil such guarantees, similarly to terms constructed against goal types?

Although each of testing, model checking, and interactive theorem proving is a highly developed research area, the combination of the three will expose many more exciting research topics of their own right. We hope to contribute to software quality through such research, developing further the methodologies and tools reported in this article.

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] Ana Bove: General Recursion in Type Theory. PhD thesis. Chalmers University of Technology. 2002.
- [3] Jeremy Bradley: Binary Decision Diagrams - A Functional Implementation. <http://www.cs.bris.ac.uk/~bradley/publish/bdd/>.
- [4] Randal E. Bryant: Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. In *ACM Computing Surveys*, volume 24, number 3, pages 293-318, 1992.
- [5] T. Y. Chen, T. H. Tse, and Zhiquan Zhou: Semi-Proving: an Integrated Method Based on Global Symbolic Evaluation and Metamorphic Testing. In *Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis ISSA-02* volume 27.4, pages 191-195. ACM Press, 2002.
- [6] 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.
- [7] Catarina Coquand: Agda, available from <http://www.cs.chalmers.se/~catarina/agda>.
- [8] Thierry Coquand, Randy Pollack, and Makoto 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://www.coverproject.org/>.
- [10] Nancy A. Day, John Launchbury, and Jeff Lewis: Logical Abstractions in Haskell. In *Proceedings of the 1999 Haskell Workshop*, Utrecht University Department of Computer Science, Technical Report UU-CS-1999-28, 1999.
- [11] Peter Dybjer: Inductive Families. In *Formal Aspects of Computing*, volume 6, pages 440-465, 1994.
- [12] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama: Verifying Haskell Programs by Combining Testing and Proving. In *Proceedings of Third International Conference on Quality Software*, pages 272-279, IEEE Press, 2003.
- [13] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama: Combining Testing and Proving in Dependent Type Theory. In *Proceedings of Theorem Proving in Higher Order Logics*, pages 188-203, volume 2758 of Lecture Notes in Computer Science, Springer-Verlag, 2003.

- [14] 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.
- [15] Peter Dybjer and Anton Setzer: Indexed Induction-Recursion. In *Proof Theory in Computer Science, LNCS* 2183, pages 93–113, 2001.
- [16] Matthew N. Geller: Test Data as an Aid in Proving Program Correctness. In *Communications of the ACM*, pages 368–375, volume 21.5, 1978.
- [17] Thomas Hallgren: Alfa, available from <http://www.cs.chalmers.se/~hallgren/Alfa>.
- [18] Peter Hancock and Anton Setzer: Interactive programs in dependent type theory. *Computer Science Logic. 14th international workshop, CSL*, pages 317–331, volume 1863 of Lecture Notes in Computer Science, 2000.
- [19] 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.
- [20] John Hughes: Why functional programming matters. In *The Computer Journal*, volume 32(2), pages 98–107, 1989.
- [21] Sava Krstic and John Matthews: Verifying BDD Algorithms through Monadic Interpretation, In *Lecture Notes in Computer Science*, pages 182–195, volume 2294, 2002.
- [22] Bengt Nordström, Kent Petersson, and Jan M. Smith: Programming in Martin-Löf Type Theory: an Introduction. Oxford University Press, 1990.
- [23] Programatica: Integrating Programming, Properties, and Validation. <http://www.cse.ogi.edu/PacSoft/projects/programatica>.
- [24] John C. Reynolds: Types, Abstraction and Parametric Polymorphism. In *Proc. of 9th IFIP World Computer Congress, Information Processing '83*, pages 513–523, North-Holland, 1983.
- [25] Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar: Reflecting BDD in Coq. In *Proc. 6th Asian Computing Science Conference (ASIAN'2000)*, pages 162–181, volume 1961 of Lecture Notes in Computer Science, Springer-verlag, 2000.
- [26] Philip Wadler: Theorem for Free. In *Functional Programming Languages and Computer Architecture*, pages 347–359, ACM, 1989.

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 [17] 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 as in Haskell, for example, the types of booleans and natural numbers are

```
Bool :: Set = data True | False
Nat  :: Set = data Zero | Succ (n :: Nat)
```

Dependent types may refer to terms of other types. For example, one may have the type `Vec A n :: Set` for lists of specified length $n :: \text{Nat}$ with elements in $A :: \text{Set}$.

A dependent function type $(x :: A) \rightarrow B[x]$ is the type of functions that sends argument $a :: A$ to a value of type $B[a]$, which may depend on a . As a special case in the above dependent function type, if the variable x does not appear in B , then we get the ordinary function type $A \rightarrow B$. As an example, we show how dependent types are used for representing the type of all lists of length n :

```
Vec :: (A :: Set) -> (n :: Nat) -> Set
Vec A Zero      = Unit
Vec A (Succ n) = Prod A (Vec A n)
```

where `Prod A B` is the set of pairs of elements of A and elements of B :

```
data Prod (A, B :: Set) = Pair (x :: A) (y :: B)
```

Now we can define the head function for nonempty lists:

```
head :: (A :: Set) -> (n :: Nat) -> Vec A (Succ n) -> A
head A n (Pair x y) = x
```

As above, polymorphic functions take explicit type arguments (e.g., `head Nat` and `head Bool`). Agda can infer them in most cases, and Alfa has an option that hides them from the user's view. In the paper, when showing Agda/Alfa code, we will use this feature and omit some type arguments when the omitted arguments are clear from the context. For example, we may write `head n xy`

for some `xy :: Vec Bool (Succ n)` instead of `head Bool n xy`.

A dependent record type, also called a signature, `sig {x1 :: A1, ..., xn :: An}`, is the type of records (tuples) `struct {x1 = v1, ..., xn = vn}` labelled with x_i 's, where $v_i :: A_i$ and A_i may depend on x_1, \dots, 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 $P :: D \rightarrow \mathbf{Set}$, which is regarded as a propositional function because sets represent propositions. Logical connectives map to type forming operations; e.g., a function of type $A \rightarrow B$ is a proof of an implication, sending a proof of A to a proof of B ; similarly, a function of type $(x :: D) \rightarrow Px$ 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 $\forall x \in D. Px \vee \neg Px$ constructively. It is equivalent to having some $p :: D \rightarrow \mathbf{Bool}$ such that $\forall x \in D. Px \Leftrightarrow T(p(x))$. 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 [11] and Dybjer and Setzer [14,15].

Remark 1 *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 the BDD Implementation in Haskell ([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 (Thierry Coquand)

```

module Bdt where
  data Bdd = 0 | I | (/ \) Bdd Bdd

  neg :: Bdd -> Bdd
  neg 0 = I
  neg I = 0
  neg (b / \ d) = neg b / \ neg d

  and_bdd :: Bdd -> Bdd -> Bdd
  and_bdd 0 h' = 0
  and_bdd I h' = h'
  and_bdd h 0 = 0
  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 0 0 = 0
  mkt I I = I

```

```

mkt h1 h2 = h1 /\ h2

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

var :: Nat -> Bdd
var Zero    = I /\ 0
var (Succ n) = next (var n)

bdt :: BoolExpr -> Bdd
bdt (Val True)  = I
bdt (Val False) = 0
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

Taut t = T (taut t)

```

D Proving that (7.3) implies (7.2)

We first state the parametricity theorem for `sorted'`: if $h' \text{ true}_1 = \text{true}_2$ and the following diagrams commute

$$\begin{array}{ccc}
a_1 \times a_1 & \xrightarrow{\leq_1} & b_1 \\
\langle h, h \rangle \downarrow & & \downarrow h' \\
a_2 \times a_2 & \xrightarrow{\leq_2} & b_2
\end{array}
\quad \text{and} \quad
\begin{array}{ccc}
b_1 \times b_1 & \xrightarrow{*_1} & b_1 \\
\langle h', h' \rangle \downarrow & & \downarrow h' \\
b_2 \times b_2 & \xrightarrow{*_2} & b_2
\end{array}$$

then the following diagram also commutes.

$$\begin{array}{ccc}
a_1^n & \xrightarrow{\text{sorted}' a_1 b_1 \text{ true}_1 (*_1) (\leq_1)} & b_1 \\
\text{map } h \downarrow & & \downarrow h' \\
a_2^n & \xrightarrow{\text{sorted}' a_2 b_2 \text{ true}_2 (*_2) (\leq_2)} & b_2
\end{array}$$

This theorem allows us to lift a property on `Bool` to a property on `BoolExpr` that can be model checked.

We take `BoolExpr` as a_1 and b_1 , `Bool` as a_2 and b_2 . Both h and h' are the evaluation function

```
eval :: Booln -> BoolExpr -> Bool
eval (b1,...,bn) t = t[b1/v1,...,bn/vn]
```

where v_i are the free variables⁹ in t . Furthermore, $e1 \leq_1 e2 = e1 \Rightarrow e2$ and $e1 \leq_2 e2 = \text{leB } e1 \ e2$. We can check that the premises of the parametricity theorem for `sorted'` hold, that is,

```
eval bs (e1 ≤1 e2) = (eval bs e1) ≤2 (eval bs e2)
```

```
eval bs (e1 * e2) = (eval bs e1) && (eval bs e2)
```

Hence, for any $bs = (b1, \dots, bn) :: \text{Bool}^n$, we have

```
eval bs (sorted' (Val True) (*) (=>) n
          (bitonic cmpSwapBE n (v1,...,vn)))
= sorted' True (&&) leB n
  (map (eval bs) (bitonic cmpSwapBE n (v1,...,vn)))
= sorted' True (&&) leB n
  (bitonic cmpSwapB n (map (eval bs) (v1,...,vn)))
= sorted leB n (bitonicB n bs)
```

where the first equality follows from the parametricity theorem for `sorted'`, and the second equality follows from that for `bitonic`.

It is semantically clear, and can be proved so too, that `Taut t` is equivalent to $(bs :: \text{Bool}^n) \rightarrow \text{T } (\text{eval } bs \ t == \text{True})$. Therefore, if (7.3) is a tautology, then `sorted leB n (bitonic cmpSwap n bs)` is always equal to `True` for any bs , that is, (7.2) holds.

E Bitonic Sort: the Haskell Version ([10])

```
bitnoic_to_sorted cmpSwap [] up = []
bitnoic_to_sorted cmpSwap [x] up = [x]
bitnoic_to_sorted cmpSwap xs up =
  let k = length xs `div` 2
      (ys, zs) = pairwise cmpSwap (splitAt k xs)
      (ys', zs') = if up then (ys, zs) else (zs, ys)

  in bitnoic_to_sorted cmpSwap ys' up ++
```

⁹ In this Appendix, we display (lists of) variables informally for readability.


```

    bitnoic_to_sorted cmpSwap zs' up

pairwise f ([] , [])      = ([] , [])
pairwise f (x:xs , y:ys) =
  let (x' , y') = f x y
      (xs' , ys') = pairwise f (xs , ys)
  in (x':xs' , y':ys')
pairwise f (xs , ys)      = (xs , ys)

bitonic_sort cmpSwap []   up = []
bitonic_sort cmpSwap [x] up = [x]
bitonic_sort cmpSwap xs   up =
  let k = length xs `div` 2
      (ys , zs) = splitAt k xs
      ys' = bitonic_sort cmpSwap ys True
      zs' = bitonic_sort cmpSwap zs False

  in bitnoic_to_sorted cmpSwap (ys' ++ zs') up

cmpSwap x y = if x < y then (x , y) else (y , x)

sort :: Ord a => [a] -> [a]
sort xs = bitonic_sort cmpSwap xs True

sorted :: Boolean b => (a -> a -> b) -> [a] -> b
sorted test []      = true
sorted test [x]     = true
sorted test (x:ys@(y:_)) =
  (test x y) <&> sorted test ys

```

F Bitonic Sort: the Agda/Alfa Version

```

bitonic_to_sorted :: (a :: Set) -> (a × a -> a × a) ->
  (n :: Nat) -> an -> Bool -> an

bitonic_to_sorted f Zero      ()      up = xs
bitonic_to_sorted f (Succ Zero) (x)    up = (x)
bitonic_to_sorted f (Succ m)  (x , xs) up =
  let n1 :: Nat = fst (splitN n)
      n2 :: Nat = snd (splitN n)
      yzs :: an1 × an2 = pairwise a f n (splitV a n xs)
      ys  :: an1 = fst yzs
      zs  :: an2 = snd yzs
      yzs1 :: an = app1 a n (bitonic_to_sorted a f n1 ys up)
              (bitonic_to_sorted a f n2 zs up)

```

```

    yzs2 :: an = app2 a n (bitonic_to_sorted a f n2 zs up)
                      (bitonic_to_sorted a f n1 ys up)
  in if up yzs1 yzs2

pairwise :: (a :: Set) -> (f :: a × a -> a × a) ->
           (n :: Nat) -> an1 × an2 -> an1 × an2

pairwise a f Zero          xs          = xs
pairwise a f (Succ Zero)  xs          = xs
pairwise a f (Succ (Succ m)) ((x,y), (x', y')) =
  let n1 :: Nat = fst (splitN m)
      n2 :: Nat = snd (splitN m)
      xy' :: a × a = f (x, x')
      xys' :: an1 × an2 = pairwise a f m (y, y')
      zs1 :: an1 = fst xys'
      zs2 :: an2 = snd xys'
  in ((fst xy', zs1), (snd xy', zs2))

splitV :: (a :: Set) -> (n :: Nat) -> an -> an1 × an2

splitV a Zero          ()          = ((), ())
splitV a (Succ Zero)   (x)         = ((x), ())
splitV a (Succ (Succ n)) (x1,x2,xs) =
  let n12 = splitN n
      n1 = fst n12
      n2 = snd n12
      ts :: an × a = shiftr a n (x2, xs)
      hs :: an1 × an2 = splitV a n (fst ts)
      ts' :: a × an2 = shiftl a n2 ((snd hs), snd ts)
  in ((x1, fst hs), ts')

splitN :: Nat -> Nat × Nat

splitN Zero          = (Zero, Zero)
splitN (Succ Zero)   = (Succ Zero, Zero)
splitN (Succ (Succ n)) =
  let n12 = splitN n
  in (Succ (fst n12), Succ (snd n12))

shiftl :: (a :: Set) -> (n :: Nat) -> an × a -> a × an

shiftr :: (a :: Set) -> (n :: Nat) -> a × an -> an × a

```