Correctness of Pointer Manipulating Algorithms Illustrated by a Verified BDD Construction

Mathieu Giorgino and Martin Strecker

IRIT, Université de Toulouse*

Abstract. This paper is an extended case study using a high-level approach to the verification of graph transformation algorithms: To represent sharing, graphs are considered as trees with additional pointers, and algorithms manipulating them are essentially primitive recursive traversals written in a monadic style. With this, we achieve almost trivial termination arguments and can use inductive reasoning principles for showing the correctness of the algorithms. We illustrate the approach with the verification of a BDD package which is modular in that it can be instantiated with different implementations of association tables for node lookup. We have also implemented a garbage collector for freeing association tables from unused entries. Even without low-level optimizations, the resulting implementation is reasonably efficient.

Keywords: Verification of imperative algorithms, Pointer algorithms, Modular Program Development, Binary Decision Diagram.

1 Introduction

There is now a large range of verification tools for imperative and object-oriented (OO) languages. Most of them have in common that they operate on source code of a particular programming language like C or Java, annotated with pre- and post-conditions and invariants. This combination of code and properties is then fed to a verification condition generator which extracts proof obligations that can be discharged by provers offering various degrees of automation (see below for a more detailed discussion).

This approach has an undeniable success when it comes to showing that a program is well-behaved (no null-pointer accesses, index ranges within bounds, deadlock-freedom of concurrent programs etc.). Program verification and in particular static analysis often amounts to showing the absence of undesirable situations with the aid of a property language that is considerably more expressive than a traditional type system, but nevertheless has a restricted set of syntactic forms for program verification that cannot be user-extended unless the imperative programming language is embedded into a general purpose proof-assistant.

These limitations turn out to be a hindrance when one has to build up a larger "background theory" capable of expressing deeper semantic properties of

* Part of this research has been supported by the *Climt* project (ANR-11-BS02-016-02).

the data structures manipulated by the program (such as the notions of interpretation and validity of a formula used in this paper). Even worse, high-level mathematical notions (such as “sets” and “trees”) are often not directly available in the specification language. Even if they are, recovering an algebraic data type from a pointer structure in the heap is not straightforward: one has to ensure, for example, that a structure encoding a list is indeed an instance of a data type and not cyclic.

In this paper, we explore the opposite direction: we start from high-level data structures based on inductive data types, which allows for an easy definition of algorithms with the aid of primitive recursion and for reasoning principles based on structural induction and rewriting. References are added explicitly to these data structures, which makes it possible to express sharing of subtrees with a simple notion of reference equality as well as associating mutable content to nodes. The notion of state is manipulated with a state-exception monad (see Section 2), thus allowing for a restricted form of object manipulation (in particular object creation and modification).

We illustrate our approach with the development of a Binary Decision Diagram (BDD) package. After recalling the basic notions and the semantics of BDDs in Section 3, we describe a first, non-optimized version of the essential algorithms in Section 4 and the implementation of association tables in Section 6. Section 5 introduces a garbage collector and memoization, which lead to a substantial speed-up.

As formal framework, we use the Isabelle proof assistant [16] and its extension Imperative_HOL [8], together with its Isabelle-to-Scala code extractor. Our algorithms are therefore executable in Scala and, as witnessed by the performance evaluation of Section 7, within the realm of state-of-the-art BDD packages.

A further gain in efficiency might be achieved by mapping our still rather coarse-grained memory model to a fine-grained memory model, which would allow us to introduce bit-level optimizations. Even though this is compatible with our approach, we have refrained from it here because it would lead to a considerable increase in complexity and is not central to the approach of this paper. The formal development is available on the authors' home pages[1] and more detailed discussions of some topics will appear in the first author's forthcoming PhD thesis [11].

Related Work – Program Verification: There are roughly two broad classes of program verifiers - those aiming at a mostly automatic verification, as Spec# [2], VCC [9], Frama-C[2] or Why3 [4], or at mostly interactive proofs, such as the ones based on Dynamic Logic like KeY[3], KIV[3] or codings of programming languages and their associated Hoare logics in proof assistants [10,19]. The borderline is not clear-cut, since some of the “automatic” tools can also be interfaced with interactive proof assistants such as Coq and Isabelle, as in [5].

[1] http://www.irit.fr/~Mathieu.Giorgino/Publications/GiSt2012BDD.html
[2] http://frama-c.com/
[3] http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/

The work that comes closest to ours is the extension of Isabelle with OO features [6]. It is at the same time more complete and considerably more complex, since it has the ambition to simulate genuine OO capabilities such as late binding, which requires, among others, the management of dynamic type tags of objects. Our approach remains confined to what can be done within a conventional polymorphic functional type system. Our aim is not to be able to verify arbitrary programs in languages such as Java or Scala, but to export programs written and verified in a functional style with stateful features to a language such as Scala. We thus hope to reduce the proof burden, while still obtaining relatively efficient target code in an idiomatic style (using subtyping and inheritance) and compatible with a widely used language.

Related Work – Verification of BDDs: Binary Decision Diagrams (BDDs) [7] are a compact format for representing Boolean formulas, making extensive use of sharing of subtrees and thus achieving a canonical representation of formulas, and a verified BDD package might become useful for the formal verification of decision procedures

Even without such an application in mind, BDDs have become a favorite case study for the verification of pointer programs. As mentioned above, all the approaches we are aware of use a low-level representation of BDDs as linked pointer structures. The idea of representing the state space in monadic style is introduced in [13], but ensuring the termination of the functions poses a problem because termination and well-formedness of the state space are closely intertwined.

There is a previous verification [18] in the Isabelle proof assistant, starting from an algorithm written in a C-like language. As in our case, it is possible to take semantic properties of BDDs into account, but the proof of correctness has a considerable complexity. By a tricky encoding, the PVS formalization in [21] can avoid the use of the notion of "state" altogether, but the encoding creates huge integers even for a small number of BDD nodes, so that the approach might not scale to larger examples.

The most comprehensive verification [20] (apart from ours) describes a verification in the Coq proof assistant, including some optimizations and a garbage collector. The state space is explicitly represented and manipulated by a functional program, and also the OCaml code extracted from Coq is functional. This seems to account for the lower performance (slower execution and faster exhaustion of memory) as compared to genuine imperative code.

2 Memory and Object Models

We first present a shallow embedding of an OO management of references in Isabelle. As a basis we use the Imperative_HOL theory [8] belonging to the Isabelle library. This theory provides imperative features to Isabelle/HOL by defining a state-exception monad with syntax facilities like `do`-notation. We then add object-oriented features that should eventually improve code generation to Scala. In the following, we put the Isabelle keywords corresponding to the discussed concepts in parentheses.

Language. Imperative programs returning values of type $'a$ have type $'a$ *Heap*. They can manipulate references of type $'b$ *ref* using the usual ML syntax to allocate (*ref* a), read ($!r$) and write ($r := a$) references. On the logical level, the term *effect* $m\ h\ h'\ v$ states that if m terminates, it transforms the heap h (of type *heap*) into h' and returns the value v.

To get closer to an OO development, a reference to a record should be seen as a handle to an object, without giving the ability in the language to retrieve the record itself. To do this, we define accessors (of type $'a \rhd 'b$ where $'a$ and $'b$ are respectively the types of the record and the field) as a means to describe an abstract attribute in an object. These allow us to introduce primitives *lookup* (denoted $r{\cdot}ac$) and *update* ($r{\cdot}ac\ .{=}\ v$) to read and write fields of the record referenced by r, which can be seen as attributes of an object. A '\$' character will start accessor names to avoid name clashes with other identifiers.

For example, with the definition of accessors \$*fst* and \$*snd* for the first and second components of pairs, the definition $m\ p \equiv do\{\ a \leftarrow p{\cdot}\$fst;\ p{\cdot}\$snd\ .{=}\ a;\ p{\cdot}\$fst\ \}$ defines a monadic operation m replacing the second component of the pair referenced by p by its first component and returns it.

We also note that in Isabelle/HOL, implication is written at object or meta level as $\longrightarrow$ or $\Longrightarrow$ but can be read indifferently as implication.

Objects and Classes as Types. Hierarchical definition of data (with sub-typing) is provided by extensible records (**record**) as described in [15]. A record *runit* is used as a top element of a record hierarchy, in the same way as `Object` in Java or `Any` in Scala are the top classes of their class hierarchies. In contrast to the implicit object sub-typing, record types are explicitly parameterized by their extension types as for example $'a$ in $'a$ *runit-scheme*.

Methods and Classes as Modules. Locales (**locale**) [1] allows the creation of a context parameterized by constants (**fixes**) and assumptions (**assumes**). We use them to define functions in the context of a reference called *this* in the same way as for OO languages. Then the functions defined in this locale and used from the outside take an additional argument being a reference to the record.

locale *object* = **fixes** *this* :: $'a$ *ref*

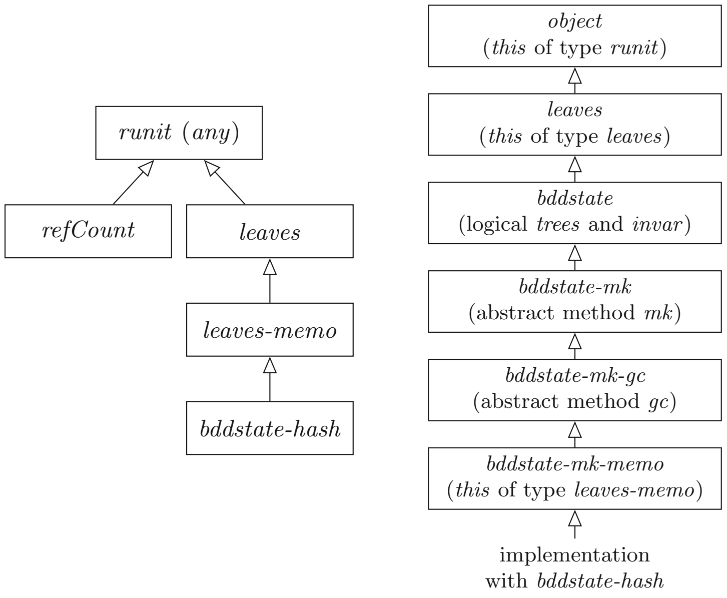
They can also be used as an equivalent of interfaces or abstract classes. They can be built upon each other with multiple inheritance (+) for which assumptions (including types of constants) can be strengthened (**for**). Finally they can be instantiated by several implementations.

In This Development. Objects and classes are used at two levels:

– for the state of the BDD factory containing the two *True* and *False* leaves and the association tables for maximal sharing and memoization. This state and its reference is unique in the context of the algorithms and provided by the locale *object* as a *this* constant parameter.

– for the nodes, each one containing a reference to a mutable extension of itself. This extension is initially empty and called *runit* to be extended later to *refCount* to store the reference counter for the garbage collection.

Figures 1a and 1b present the hierarchies of records and locales used in this development. We also take advantage of locales to specify the logical functions used only in proofs (locale *bddstate*) and the abstract methods (locales *bddstate-mk* and *bddstate-mk-gc*).



(a) Data (records) (b) Methods/Logic (locales)

Fig. 1. Hierarchies of data and methods

3 Binary Decision Diagrams

3.1 Tree Structure and Interpretation

BDDs are used to represent and manipulate efficiently Boolean expressions. We will use them as starting point of our algorithms, by defining a function constructing BDDs from their representation of type $('v,\ bool)\ expr$ in which $'v$ is the type of variable names. The definition of expressions is rather standard:

datatype $('v,'a)\ expr =$
  $Var\ 'v \mid Const\ 'a \mid BExpr\ ('a \Rightarrow 'a \Rightarrow 'a)\ (('v,'a)\ expr)\ (('v,'a)\ expr)$

and their interpretation is done by *interp-expr* taking as extra argument the variable instantiations represented as a function from variables to values:

fun $\mathit{interp\text{-}expr} :: ('v,\ 'a)\ \mathit{expr} \Rightarrow ('v \Rightarrow 'a) \Rightarrow 'a$ **where**
$\quad \mathit{interp\text{-}expr}\ (\mathit{Var}\ v)\ \mathit{vs} = \mathit{vs}\ v$
$|\ \mathit{interp\text{-}expr}\ (\mathit{Const}\ a)\ \mathit{vs} = a$
$|\ \mathit{interp\text{-}expr}\ (\mathit{BExpr}\ \mathit{bop}\ e_1\ e_2)\ \mathit{vs} = \mathit{bop}\ (\mathit{interp\text{-}expr}\ e_1\ \mathit{vs})\ (\mathit{interp\text{-}expr}\ e_2\ \mathit{vs})$

We now define BDDs as binary trees where the two subtrees represent the BDDs resulting from the instantiation of the root variable to *False* or *True* :

datatype $('a,\ 'b)\ \mathit{tree} = \mathit{Leaf}\ 'a \mid \mathit{Node}\ 'b\ (('a,\ 'b)\ \mathit{tree})\ (('a,\ 'b)\ \mathit{tree})$
type-synonym $('a,'b,'c)\ \mathit{rtree} = ('a \times 'c\ \mathit{ref},\ 'b{::}\mathit{linorder} \times 'c\ \mathit{ref})\ \mathit{tree}$

$('a,\ 'b,\ 'c)\ \mathit{rtree}$ is the type of referenced trees with leaf content of type $'a$, node content of type $'b$ and mutable extension of type $'c$. These trees contain a reference to this mutable extension that will be used as an identifier. Each node contains a variable index whose type is equipped with a linear order (as indicated by Isabelle's sort annotation ::*linorder*) and each leaf contains a value of any type instantiated later in the development (for interpretations) to Booleans.

BDDs can be interpreted (*i. e.* evaluated) by giving values to variables which is what the *interp* function does (l and h abbreviate *low* and *high*):

fun $\mathit{interp} :: ('a,\ 'v,\ 'r)\ \mathit{rtree} \Rightarrow ('v \Rightarrow \mathit{bool}) \Rightarrow 'a$ **where**
$\quad \mathit{interp}\ (\mathit{Leaf}\ (b,r))\ \mathit{vs} = b$
$|\mathit{interp}\ (\mathit{Node}\ (v,r)\ l\ h)\ \mathit{vs} = (\mathit{if}\ \mathit{vs}\ v\ \mathit{then}\ \mathit{interp}\ h\ \mathit{vs}\ \mathit{else}\ \mathit{interp}\ l\ \mathit{vs})$

3.2 Sharing

We first illustrate the concept of subtree-sharing by an example. A non-shared BDD (thus, in fact, just a decision tree) representing the formula $(x \wedge y) \vee z$ is given by the tree on the left of Figure 2.

There is a common subtree (shaded) which we would like to share. We therefore adorn the tree nodes with references, using the same reference for structurally equal trees. The result of sharing is illustrated on the right of Figure 2.

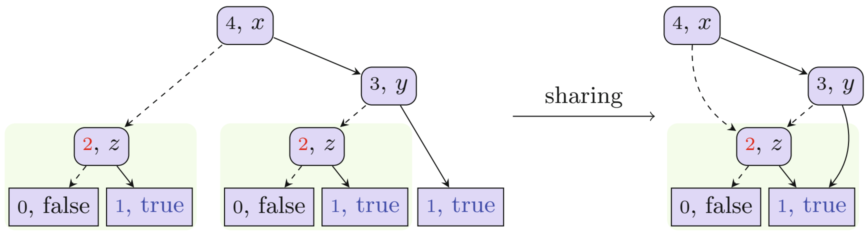


Fig. 2. Sharing nodes in a tree

In this way, as long as subtrees having identic references are the same, we can represent sharing. To ensure this property giving meaning to references, we use the predicate *ref-unique ts*:

definition *ref-unique* :: $('a,\ 'v,\ 'r)$ *rtree set* $\Rightarrow$ *bool* **where** *ref-unique ts* $\equiv$
$\forall t_1\ t_2.\ t_1 \in ts \longrightarrow t_2 \in ts \longrightarrow (\textit{ref-equal}\ t_1\ t_2 \longleftrightarrow \textit{struct-equal}\ t_1\ t_2)$

in which *ref-equal* means that two trees have the same reference attribute, and *struct-equal* is structural equality neglecting references, thus corresponding to the typical notion of equality of data in functional languages.

While the left-to-right implication of this equivalence is the required property (two nodes having the same reference are the same), the other implication ensures maximal sharing (same subtrees are shared, *i. e.* have the same reference).

3.3 Ordering and Reducedness

With this definition, and without any other property, BDDs would be rather hard to manipulate. For one, same variable indices could appear several times on paths from root to leaves. Also, variables would not be in the same order, making comparison of BDDs harder. Moreover, a lot of space would be wasted. To circumvent this problem, one often imposes a strict order on variables, the resulting BDDs being called ordered (OBDDs). We define this property using the *tree-vars* constant to collect all variables of a tree:

fun *ordered* :: $('a,\ 'v{::}linorder,\ 'r)$ *rtree* $\Rightarrow$ *bool* **where**
ordered (*Leaf b*) = *True*
| *ordered* (*Node* (*i*, *r*) *l h*) =
$((\forall\ j \in (\textit{tree-vars}\ l \cup \textit{tree-vars}\ h).\ i < j) \wedge \textit{ordered}\ l \wedge \textit{ordered}\ h)$

An additional important property is to avoid redundant tests, which occur when the two children of a node have the same interpretation. All the nodes satisfying this property can be removed. In this case, the OBDD is said to be reduced (ROBDD).

fun *reduced* :: $('a,\ 'v,\ 'r)$ *rtree* $\Rightarrow$ *bool* **where**
reduced (*Node vr l h*) = $((\textit{interp}\ l \neq \textit{interp}\ h) \wedge \textit{reduced}\ l \wedge \textit{reduced}\ h)$
| *reduced* (*Leaf b*) = *True*

This property uses a high-level definition (*interp*), but it can be deduced (*cf.* Lemma 1) from the three low-level properties *ref-unique*, *ordered* (already seen) and *non-redundant*:

fun *non-redundant* :: $('a,\ 'v,\ 'r)$ *rtree* $\Rightarrow$ *bool* **where**
non-redundant(*Node vr l h*)=$((\neg \textit{ref-equal}\ l\ h) \wedge \textit{non-redundant}\ l \wedge \textit{non-redundant}\ h)$
|*non-redundant*(*Leaf b*) = *True*

We then merge these properties into two definitions *robdd* (high-level) and *robdd-refs* (low-level):

definition *robdd t* $\equiv$ (*ordered t* $\wedge$ *reduced t*)
definition *robdd-refs t* $\equiv$ (*ordered t* $\wedge$ *non-redundant t* $\wedge$ *ref-unique* (*treeset t*))

From these definitions, we finally show that ROBDDs are a canonical representation of Boolean expressions, *i. e.* that two equivalent ROBDDs are structurally equal at high (*robdd*) and low (same with *robdd-refs*) level:

Theorem 1 (canonic_robdd)

$robdd\ t_1 \wedge robdd\ t_2 \wedge interp\ t_1 = interp\ t_2 \Longrightarrow struct\text{-}equal\ t_1\ t_2$

Proof. By induction on the pair of trees: the leaves case is trivial, heterogeneous cases (leaf and node or nodes of different levels) lead to contradictions, and the remaining case (two nodes of same level) is proved by applying the induction hypothesis on subtrees.

Also high- and low-level properties are related in Theorem 2 as a consequence of Lemma 1:

Lemma 1 (non_redundant_imp_reduced)

$ordered\ t \wedge non\text{-}redundant\ t \wedge ref\text{-}unique\ (treeset\ t) \Longrightarrow reduced\ t$

Proof. By induction on t: the leaf case is trivial and the node case is proved by applying the induction hypothesis on the subtrees and proving that trees with different references are structurally different (from definitions of *non-redundant* and *ref-unique*) and then have different interpretations (with contrapositive of Theorem 1).

Theorem 2 (robdd_refs_robdd)

$ref\text{-}unique\ (treeset\ t) \Longrightarrow robdd\text{-}refs\ t = robdd\ t$

4 Constructing BDDs

The simplest BDDs are the leaves corresponding to the *True* and *False* values. These ones have to be unique in order to permit sharing of nodes. We put them in the BDD factory whose data is this record:

record $('v,\ 'c)$ *leaves* = *runit* +
  leafTrue :: $(bool,\ 'v,\ 'c)$ *rtree*
  leafFalse :: $(bool,\ 'v,\ 'c)$ *rtree*

We define the context of this state by constraining the type of the referenced record *this*. This context together with the *leaves* record would be equivalent to a class definition `class Leaves extends Object` in Java where type of `this` is constrained from `Object` to `Leaves`.

locale *leaves* = *object this* **for** *this* :: $('v,\ 'c,\ 'a)$ *leaves-scheme ref*

Then we extend it to add logical abstractions *trees* and *invar* that will be instantiated during the implementation to provide the correctness arguments we will rely on in the proofs. The *trees* parameter abstracts the set of trees already constructed in the state. The *invar* parameter is the invariant of the data-structures that will be added to the heap by the implementation and that will have to be preserved by BDD operations.

```
locale bddstate = leaves +
  fixes trees :: heap ⇒ (bool, 'v, 'c) rtree set
  fixes invar :: heap ⇒ bool
```

To be well-formed (*wf-heap*), the heap needs to follow the abstract implementation invariant *invar* and its trees need to contain the leaves and to be maximally shared and closed for the subtree relation.

```
definition  wf-heap :: heap ⇒ bool where
wf-heap s ≡ (invar s ∧ ref-unique (trees s) ∧ subtree-closed (trees s) ∧ leaves-in s)
```

Finally we add an abstract function *mk* and its specification (*mk-spec*) especially ensuring that *mk i l h* constructs a ROBDD whose interpretation is correct under the precondition that the heap is well-formed, level i is consistent with levels of l and h and trees in the heap are already ROBDDs. It uses the function *levelOf* returning the level of a BDD.

```
locale bddstate-mk = bddstate +
  fixes mk :: 'v ⇒ (bool, 'v, 'r) rtree ⇒ (bool, 'v, 'r) rtree ⇒ (bool, 'v, 'r) rtree Heap
  assumes mk-spec: effect (mk i l h) s s' t ∧ wf-heap s ∧ {l,h} ⊆ trees s ⟹ (
  (LevNode i < Min (levelOf ` {l,h}) ∧ (∀t' ∈ trees s. robdd-refs t') ⟶ robdd-refs t)
  ∧ (∀vs. interp t vs = (if vs i then interp h vs else interp l vs))
  ∧ (wf-heap s') ∧ (trees s' = insert t (trees s)))
```

In this context we define the *app* function which applies a binary Boolean operator to two BDDs. If these BDDs are both leaves, it returns a leaf corresponding to the application of the binary Boolean operator to their contents. Else it returns a new BDD constructed with *mk* from its recursive calls to the left and right subtrees of BDDs with the same level. For this purpose it uses the *select* function which returns two pairs of BDDs corresponding to the subtrees (*split-lh*) of the BDD(s) with the smallest level and the duplication (*dup*) of the other (if any). It also uses the function *varOfLev* retrieving the variable corresponding to the level of a node.

```
function app :: (bool ⇒ bool ⇒ bool)
  ⇒ ((bool, 'v, 'r) rtree * (bool, 'v, 'r) rtree) ⇒ (bool, 'v, 'r) rtree Heap where
  app bop (n₁, n₂) = do {
    if tpair is-leaf (n₁, n₂) then (constLeaf (bop (leaf-contents n₁) (leaf-contents n₂)))
      else (do {
         let ((l₁, h₁), (l₂, h₂)) = select split-lh dup (n₁, n₂);
         l ← app bop (l₁, l₂); h ← app bop (h₁, h₂);
         mk (varOfLev (min-level (n₁, n₂))) l h })}
```

This is the only function whose termination proof is not automatic, but still very simple: it suffices to show that *select split-lh dup* decreases the sum of the sizes of the trees in the pair. Indeed by representing BDDs as an inductive structure instead of pointers in the heap, the termination condition does not appear anymore in the implicit nested recursion on the heap like in [13] and we do not need to add a phantom parameter as a bound like in [20].

Finally, we define the *build* function which is a simple traversal recursively constructing BDDs for sub-expressions and then joining them with *app*.

primrec $\mathit{build} :: ('v,\ \mathit{bool})\ \mathit{expr} \Rightarrow (\mathit{bool},\ 'v,\ 'r)\ \mathit{rtree}\ \mathit{Heap}$ **where**
$\mathit{build}\ (\mathit{Var}\ i) = (\mathit{do}\{\ \mathit{cf} \leftarrow \mathit{constLeaf}\ \mathit{False};\ \mathit{ct} \leftarrow \mathit{constLeaf}\ \mathit{True};\ \mathit{mk}\ i\ \mathit{cf}\ \mathit{ct}\})$
$|\ \mathit{build}\ (\mathit{Const}\ b) = (\mathit{constLeaf}\ b)$
$|\ \mathit{build}\ (\mathit{BExpr}\ \mathit{bop}\ e_1\ e_2) = (\mathit{do}\{\ n_1 \leftarrow \mathit{build}\ e_1;\ n_2 \leftarrow \mathit{build}\ e_2;\ \mathit{app}\ \mathit{bop}\ (n_1,\ n_2)\})$

The verification of these functions involves the preservation of the well-formedness of the heap (Theorems 3 and 4) – implying that the returned BDD (as well as the others in the heap) is a ROBDD and that it is interpreted like the expression – and the construction of canonical BDDs (Theorem 5) – implying for example that a tautology constructs *Leaf True*.

Theorem 3 (wf_heap_app)

$$\mathit{wf\text{-}heap}\ s \wedge \{t_1,\ t_2\} \subseteq \mathit{trees}\ s \wedge \mathit{effect}\ (\mathit{app}\ f\ (t_1,\ t_2))\ s\ s'\ t \Longrightarrow$$
$$\mathit{interp}\ t\ \mathit{vs} = f\ (\mathit{interp}\ t_1\ \mathit{vs})\ (\mathit{interp}\ t_2\ \mathit{vs}) \wedge \mathit{insert}\ t\ (\mathit{trees}\ s) \subseteq \mathit{trees}\ s' \wedge \mathit{wf\text{-}heap}\ s'$$

Proof. We use the induction schema generated from the termination proof of *app* working on a pair of trees – following the order relation infered from *select split-lh dup*. If both trees are leaves, the BDD is a leaf already in the unchanged state. Else the induction hypotheses hold for the subtrees provided by *select*. The specification of *mk* and the transitivity of ⊆ finish the proof.

Theorem 4 (wf_heap_build)

$$\mathit{effect}\ (\mathit{build}\ e)\ s\ s'\ t \wedge \mathit{wf\text{-}heap}\ s \Longrightarrow$$
$$\mathit{interp}\ t = \mathit{interp\text{-}expr}\ e \wedge \mathit{insert}\ t\ (\mathit{trees}\ s) \subseteq \mathit{trees}\ s' \wedge \mathit{wf\text{-}heap}\ s'$$

Proof. By induction on the expression: In the cases of *Const* or *Var*, the result is immediate from the specification of *mk* and the definition of *constLeaf.* In the case of *BExpr*, the induction hypotheses hold for the sub-expressions and the result is obtained from Theorem 3.

Theorem 5 (build_correct)

$$(\forall\, t {\in} \mathit{trees}\ s_1.\ \mathit{robdd\text{-}refs}\ t) \wedge \mathit{wf\text{-}heap}\ s_1 \Longrightarrow$$
$$(\forall\, t {\in} \mathit{trees}\ s_2.\ \mathit{robdd\text{-}refs}\ t) \wedge \mathit{wf\text{-}heap}\ s_2 \Longrightarrow$$
$$\mathit{effect}\ (\mathit{build}\ e_1)\ s_1\ s_1'\ t_1 \wedge \mathit{effect}\ (\mathit{build}\ e_2)\ s_2\ s_2'\ t_2 \Longrightarrow$$
$$\mathit{struct\text{-}equal}\ t_1\ t_2 = (\mathit{interp\text{-}expr}\ e_1 = \mathit{interp\text{-}expr}\ e_2)$$

Proof. In the same way as for Theorem 4, by proving a similar property for *app.*

5 Optimizations: Memoization and Garbage Collection

The *app* and *build* functions have been presented in their simplest form and without optimizations. We present in this section the two optimizations we have made to them.

Memoization During the BDD construction, several identical computations can appear. This happens mostly within the recursive calls of the *app* function during which the binary operation stays the same and identical pairs of BDDs can arise by simplifications and sharing. In order to avoid these redundant computations, the immediate solution is to use a memoization table – recording the arguments and the result for each of its calls and returning directly the result in case the arguments already have an entry in the table. This optimization is essential as it cuts down the complexity of the construction of highly shared BDD.

We add this memoization table to the state by extending the record containing the leaves. Then the only changes to the *app* function are the memoization table lookup before the eventual calculation and the table update after.

By adding an invariant on all the trees in the memoization table ensuring the properties desired for the resulting tree (mostly the conclusion of Theorem 3), the changes in the proof follow the changes of the function. With a case distinction on the result of the table lookup for the arguments, if there is an entry for them, the result follows the invariant, else the original proof remains and the result following the invariant is stored in the table.

Garbage Collection. Using an association table avoids duplication of nodes and allows us to share them. However, recording all created nodes since the start of the algorithm can lead to a very huge memory usage. Indeed keeping a reference to a node in an association table prevents the JVM garbage collector to collect nodes that could have been discarded during BDD simplifications.

We chose to remove these unused nodes from the association table by a reference counting variant. The principle of reference counting is simply to store for each node the number of references to it. Instead of counting references for all nodes, we only count them for the BDD roots. This allows us to keep the *mk* function independent of the reference count. Then, we parametrized the development with a garbage collection function *gc* whose specification ensures the preservation of used nodes (*i. e.* nodes reachable from a node with a non-null reference count). We call it in the *build* function when the association table becomes too large.

For this improvement, the proof additions were substantial. Indeed, several mutations to the reference counters appear in the functions, causing inner modifications in proofs. Moreover the invariant $insert\ t\ (trees\ s) \subseteq trees\ s'$ for *build* had to be weakened to $insert\ t\ (reachable\ s) \subseteq reachable\ s'$. These difficulties attributable to mutability highlight the simplifications provided by the encoding of BDDs as inductive datatypes instead of nodes and pointers.

6 Implementation of Abstract Functions

It is now time to implement the abstract function *mk* as well as the logical functions *invar* and *trees*. We wrote two implementations and present the most efficient one using a hash-map provided by the Collection Framework [14].

Following its specification, *mk* needs to ensure the maximal sharing of nodes. To do this, we add in the state a table associating the components of a node (its children and variable name) to itself. Then by looking in this table, we know whether a BDD that could be returned has already been created.

```
record ('v, 'c) bddstate-hash =
  ('v,('c ref × 'c ref, (bool,'v,'c) rtree) hashmap, 'c) leaves-memo +
  hash :: ('v × 'c ref × 'c ref, (bool, 'v, 'c) rtree) hashmap
```

We also define two auxiliary monadic functions *add* and *lookup* adding and looking for nodes of the table in the state. For example, the lookup function is:

```
definition lookup where
  lookup i l h = do{ hm ← this·$hash; return (ahm-lookup (i, ref-of l, ref-of h) hm) }
```

They are used in the definition of *mk*:

```
definition mk where
  mk i l h = (if ref-equal l h then return l else
    do{ to ← lookup i l h; (case to of None ⇒ add i l h | Some t ⇒ return t)  })
```

The garbage collector *gc* is then also implemented using two auxiliary monadic functions *referencedSet* – computing the set of nodes reachable from a node with a non-null reference count – and *hash-restrict* – restricting the domain of the hash table to the set given as argument:

```
definition gc :: unit Heap where gc = do { hs ← referencedSet; hash-restrict hs }
```

To avoid too frequent calls to the garbage collector, it is triggered only when the table size exceeds 10000 which is an acceptable condition for preliminary tests but that could be improved by adding a counter in the state.

We finally use these functions satisfying the specifications of the locales to obtain instantiated *app* and *build* functions for which we can generate code.

7 Performance Evaluation

Finally we evaluate the performance of our BDD construction development.

As a comparison point we developed a BDD package directly in Scala whose code would be naively expected from the code generation from the Isabelle theories. This allows us to evaluate the efficiency of the default code generation of Isabelle into Scala wrt our encoding of objects. We also compare these two implementations with a third one being a highly optimized BDD library called JavaBDD[4] providing a Java interface to several BDD libraries written in C or Java. The results are given in Figure 3.

For this evaluation we construct BDDs for two kinds of valid formulas both of which are standard benchmarks. The first one is the Urquhart's formulae U_n defined by $x_1 \Leftrightarrow (x_2 \Leftrightarrow \dots (x_n \Leftrightarrow (x_1 \Leftrightarrow \dots (x_{n-1} \Leftrightarrow x_n))))$. The second one is a formulae P_n stating the pigeonhole principle for $n+1$ pigeons in n holes *i. e.* given that $n+1$ pigeons are in n holes, at least one hole contains two pigeons.

[4] `http://javabdd.sourceforge.net/`

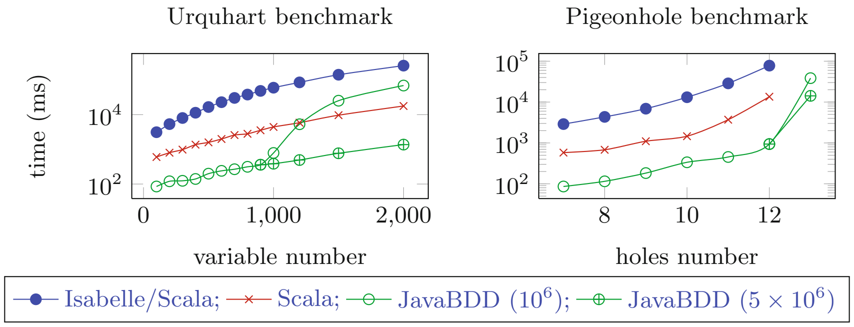


Fig. 3. Evaluation of the generated code efficiency by comparison with a direct implementation and the JavaBDD library

In the Scala version, we use the standard hash map of the Scala library (`scala.collection.mutable.HashMap`) which has an adaptable size. Its garbage collection is triggered when the table size exceeds a threshold value initially set to 1000 and increased by one half when unavoidable.

On the other side, JavaBDD lets the user choose the right table size which is increased, if necessary, after garbage collections by an initially fixed value. In the benchmarks, we set it to 10^6 and 5×10^6. We can see that increasing the initial table size for the JavaBDD version leads to better performances for large expressions but then more space is needed even for smaller ones.

As it can be seen on the pigeonhole benchmark, the memory consumption is still a limiting factor of the Scala versions compared to the JavaBDD one which manages to construct the BDD for 13 pigeon-holes. Also while the generated code is 100 times slower than the JavaBDD one (using low-level optimizations), it is only 10 times slower than the hand-written code that was the lower bound for its efficiency – the algorithms being identical. We suspect several causes of inefficiency and space usage introduced by the code extraction:

– Monad operations are converted into method calls. The presence of monadic operators at each line could explain some performance penalties.
– A "Ref" class is introduced to allow reference manipulations in Scala. This is unnecessary for objects as long as we don't use references on primitive types and referenced values are accessed only through accessors.
– Record extensions are translated to class encapsulations leading to waste of space and several indirections at the time of attribute accesses.

Improving on these points is current work and we think that these optimizations in the code generation could improve the general performances, to the point that the generated code would be comparable to the hand-written code. However, the confidence in the code generator is an essential component of the whole process that makes it hard to modify. More details on possible solutions will be discussed in [11].

8 Conclusions

This paper has presented a verified development of a BDD package in the Isabelle proof assistant, with fully operational code generated for the programming language Scala. It represents BDDs by trees containing references allowing for easy definitions and proofs – done by natural induction schemas and rewriting. The development time for the formalization itself (around 6 person months) is difficult to estimate exactly, because it went hand in hand with the development of the methodology. In the light of the performance of the code obtained, the result is encouraging, and we expect to explore the approach further for the development of verified decision procedures.

As mentioned in the outset, bit-level optimizations could be introduced, at the price of adding one or several refinement layers, with corresponding simulation proofs. Even though feasible, this is not our current focus, since we aim at a method for producing reasonably efficient verified code with a very moderate effort. Indeed this development stretches over about 7500 lines – 5000 before optimizations – among them about 1500 are generic and concern object management. This compares very favorably with the verification in Coq of the same algorithm including optimizations [20] (about 15000 lines), and with the verification of normalization of BDDs in Isabelle/HOL [18] (about 10000 lines).

Consequently, our method is not a panacea. As far as the class and object model is concerned: The type system has intentionally been kept simple in the sense that classes are essentially based on record structures and inductive data types as found in ML-style polymorphism. Such a choice is incompatible with some OO features such as late method binding, which appears to be acceptable in the context of high-integrity software. As mentioned in Section 7, we are aware of some inefficiencies that arise during code extraction to Scala, and which have as deeper cause a mismatch between pointer-manipulating languages (as incorporated in the Imperative_HOL framework) and "all is object" languages, such as Java and Scala. We will address this issue in our future work.

Finally, even though the representation "trees with sharing" appears to be a severe limitation at first glance, its combination with cute functional data structures [17] allows to represent quite general pointer meshes (see for example the verification of the Schorr-Waite algorithm [12] using a "zipper" data structure).

References

1. Ballarin, C.: Locales and Locale Expressions in Isabelle/Isar. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 34–50. Springer, Heidelberg (2004)
2. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM 54(6), 81–91 (2011)
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)

4. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd Your Herd of Provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011)
5. Böhme, S., Moskal, M., Schulte, W., Wolff, B.: HOL-Boogie — An Interactive Prover-Backend for the Verifying C Compiler. Journal of Automated Reasoning 44(1-2), 111–144 (2010)
6. Brucker, A.D., Wolff, B.: Semantics, Calculi, and Analysis for Object-Oriented Specifications. Acta Informatica 46(4), 255–284 (2009)
7. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers C-35, 677–691 (1986)
8. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative Functional Programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008)
9. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
10. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
11. Giorgino, M.: Proofs of pointer algorithms by an inductive representation of graphs. PhD thesis, Université de Toulouse (forthcoming, 2012)
12. Giorgino, M., Strecker, M., Matthes, R., Pantel, M.: Verification of the Schorr-Waite Algorithm – From Trees to Graphs. In: Alpuente, M. (ed.) LOPSTR 2010. LNCS, vol. 6564, pp. 67–83. Springer, Heidelberg (2011)
13. Krstić, S., Matthews, J.: Verifying BDD Algorithms through Monadic Interpretation. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 182–195. Springer, Heidelberg (2002)
14. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 339–354. Springer, Heidelberg (2010)
15. Naraschewski, W., Wenzel, M.T.: Object-Oriented Verification Based on Record Subtyping in Higher-Order Logic. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 349–366. Springer, Heidelberg (1998)
16. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
17. Okasaki, C.: Purely functional data structures. Cambridge University Press (1998)
18. Ortner, V., Schirmer, N.W.: Verification of BDD Normalization. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 261–277. Springer, Heidelberg (2005)
19. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL, PhD thesis, Technische Universität München (2006)
20. Verma, K.N., Goubault-Larrecq, J., Prasad, S., Arun-Kumar, S.: Reflecting BDDs in Coq. In: Kleinberg, R.D., Sato, M. (eds.) ASIAN 2000. LNCS, vol. 1961, pp. 162–181. Springer, Heidelberg (2000)
21. von Henke, F.W., Pfab, S., Pfeifer, H., Rueß, H.: Case Studies in Meta-Level Theorem Proving. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 461–478. Springer, Heidelberg (1998)