Bddl: A Type System for Binary Decision Diagrams

Yousra Lembachar[1], Ryan Rusich[2], Iulian Neamtiu[3], and Gianfranco Ciardo[4]

[1] Blockdaemon
`yousra@blockdaemon.com`
[2] University of California, Riverside, USA
`rusichr@cs.ucr.edu`
[3] New Jersey Institute of Technology, USA
`ineamtiu@njit.edu`
[4] Iowa State University, USA
`ciardo@iastate.edu`

Abstract. Binary Decision Diagrams (BDDs) are compact data structures used to efficiently store and process boolean functions. BDDs have many uses, from system design to model checking to efficiently storing context information for context-sensitive analysis. The use of BDDs in verification and program analysis has been facilitated by the recent emergence of many open source BDD libraries. The correctness of BDD-based system design and verification hinges upon the correctness of the BDD library implementations, and the correct use of these libraries. Surprisingly, for a technology so prevalent in system design and formal verification, there has been little research effort on formally verifying the correctness of BDD library implementations or their use. For BDD libraries that do perform some correctness checks, these are mostly confined to runtime assertion checking, which slows down BDD operations and might still be unable to reveal errors until deployment. To address these issues and take a step toward provably correct, yet efficient, BDD-handling code, we propose a formal system called Bddl to describe, reason about, and prove the correctness of BDD operations. Bddl extends lambda calculus with support for BDD operations (e.g., creation, manipulation), expressing BDD structural properties (e.g., canonicity, proper ordering), and BDD semantics (e.g., sets, relations). Bddl uses a type system based on refinement types to statically check BDD manipulation. We have proved our system correct using a small-step semantics and standard notions of progress and preservation. Bddl is the first attempt to provide a well-defined syntax and semantics to BDD operations; we show how it could prevent bugs and semantic errors in the implementation and use of three mature DD libraries.

Keywords: binary decision diagrams, type checking, BDD library, correctness by construction

1 Introduction

Formal methods for hardware and software verification have been facilitated by reliable and efficient methods for expressing and checking hardware as well as software behavior. For instance, in digital circuit design, where chips can have billions of transistors, symbolic model checking was made possible primarily by the introduction of canonical and efficient data structures such as BDDs, which often provide a compact representation of very large state spaces. Essentially, BDDs can be used to symbolically represent boolean functions. This *symbolic*, rather than explicit representation of the state space is a main strength of BDDs and decision diagrams in general.[5] In addition to symbolic model checking, BDDs are extensively used in quantitative risk assessment; for example, QRAS, a commercial system used by NASA to perform Probabilistic Risk Assessment (PSA)[11], allows systems engineers to quantify risks, identify risk scenarios, as well as reason about how risk is affected by changes to the system or organization—failure for NASA operations can have unacceptable costs.

Numerous decision diagram library implementations support BDDs [8, 6, 1, 12]. Yet, formal method support for checking BDD correctness is lacking. The aim of this paper is to provide a formal system that verifies the validity of BDD construction and manipulation. The core of our approach consists of a calculus and type system that support BDD terms, BDD operations and BDD semantics. Our current system performs type safety checks for BDD manipulation, but is general enough that we envision it can be extended to support other kinds of decision diagrams. We analyzed three mature DD libraries to drive the design of Bddl. CUDD [8] is a popular DD library, used in the NuSMV model checker. MDDL is part of SMART [6], which has been used to verify the NASA runway safety monitor [20]. JavaBDD [12] is used in `bddbddb` [22], a Datalog-based framework for specifying, and efficiently performing, program analysis.

To illustrate how our system statically prevents semantic errors, we present two examples of BDD library implementation and BDD library usage errors that cause BDD-based programs to crash or silently produce erroneous outcomes. These examples are drawn from CUDD and MDDL; in Section 3 we provide the actual code for these, and other, examples. First, consider the `BDD::Compose(g,v)` operation from the CUDD library; `BDD::Compose` returns the result of splicing BDD `g` into the slot currently occupied by variable with index `v` in the BDD represented by `this`. Clients can crash the program by passing in an incorrect index `v`; recent versions of CUDD generate an `Unexpected error`, while older versions crash with a `Segmentation fault`. Second, consider the method

[5] Other kinds of decision diagrams operate on integers and reals to encode algebraic, arithmetic, and relational functions. Decision diagrams have been employed in areas as diverse as optimization [2], electronic design [24], VLSI CAD [5], Genetics (gene expression analysis [25], data-mining DNA subsequences) [15], NASA safety operations [20], and reliability [23].

`RelationalProduct(p,r)` from MDDL. The method computes the relational product of BDDs `p` and `r`, and requires that `p` have L levels and `r` have $2 * L$ levels, as `p` encodes a set and `r` encodes a relation. However, library clients can invoke `RelationalProduct` in incorrect ways: first, they can invoke it with two sets or two relations, in which case the library silently returns an incorrect result; or, they can invoke it with a set and a relation where `r`'s number of levels is not twice `p`'s number of levels, which leads to a runtime error (assertion failure). We present a detailed account of these and other errors in Section 3.

In general, BDD libraries do not check the higher-level semantics of library implementations and client-supplied data, or perform such checks at runtime; as a result, they silently return an incorrect result, or fail with a runtime error; another disadvantage of runtime checks is that they slow down the execution. In this paper, we make progress toward provably correct and efficient BDD-handling code using BDDL, a calculus we developed. Our approach consists of two main steps. First, BDD library and client code must be expressed in BDDL, e.g., C, C++, or Java code translated to BDDL, and library function types expressed as BDDL types; currently, this approach is manual, though we found the translation to be straightforward, as evidenced by the translations in Sections 3. Second, the DDL type inference and checking system statically checks the BDDL code and reports typing errors. Section 3 shows how, when using BDDL, we would get a static typing error in the semantic/assertion failure cases we previously discussed—our system prevents certain ill-typed operations on BDDs that may cause programs to crash or produce erroneous outcomes.

Our approach consists of BDDL, a calculus with an associated type system and operational semantics. BDDL's expressiveness and effectiveness derive from its ability to model and verify two kinds of BDD semantic properties: structural and logical. *Structural* semantic properties, e.g., constraints on node and edge manipulation, are captured by our terms and typing system—we model BDD nodes as terms (Section 4), and use a refinement-based type system (Section 5) to express integrity properties (e.g., the number of levels, set vs. relation encoding, quasi- vs. fully-reduced form). *Logical* semantic properties are captured by refined function types: the BDDL types assigned to library functions permit concise expression of logical properties (pre- and post-conditions), because refinement predicates on our types allow for conjunctions of arithmetic expressions. At the same time, our subtyping system allows polymorphism over BDD structures, which increases expressiveness, especially for generic BDD-manipulating functions. We employ a small-step operational semantics to prove our system correct using standard notions of progress and preservation (Section 5.4).

In summary, our work makes two main contributions:

– An exposé of implementation and usage errors in BDD libraries.

– BDDL, a formal system to express and statically verify the safety of BDD library implementations and BDD library uses.

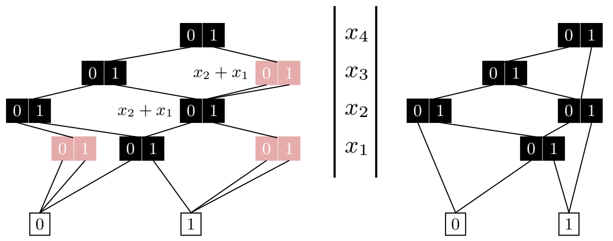

Fig. 1. A quasi-reduced (left) vs. a fully-reduced (right) BDD.

2 Background: Binary Decision Diagrams

We now provide a brief overview of BDDs.

2.1 Binary Decision Diagrams

A Binary Decision Diagram (BDD) is a rooted directed acyclic graph encoding a boolean function of the form $f : \mathbb{B}^L \to \mathbb{B}$, for some $L \in \mathbb{N}^+$. Each non-terminal node is labeled with a variable $x_k \in \{x_L, ..., x_1\}$, and placed at level j (where $j = k$ for BDDs that have the order $x_1 < x_2 < ... < x_L$) , while there are two terminal nodes, $\boxed{1}$ and $\boxed{0}$ placed at level 0. A non-terminal node has two outgoing edges labeled "false", or "0", and "true", or "1", respectively, pointing to a node at a level h satisfying $0 \le h < k$. The value of the function encoded by a BDD for a particular truth value assignment for the L variables is obtained by following the corresponding path from the root to a terminal node. For example, the BDD on the left of Figure 1 encodes $f(x_4, x_3, x_2, x_1) = \overline{x_4}\,\overline{x_3}x_2x_1 + (x_4+x_3)(x_2+x_1)$; note that product denotes logical AND, while sum denotes logical OR. Observe that $f(0, 0, 0, 0) = 0$, since the path from the root corresponding to the choices $x_4 = 0$, $x_3 = 0$, $x_2 = 0$, and $x_1 = 0$ ends at $\boxed{0}$. The BDD on the right encodes the same function, but x_1 is not tested on that path, as it does not affect the value of f when $x_4 = 0$, $x_3 = 0$, and $x_2 = 0$. These two BDDs correspond to quasi-reduced (left) and fully-reduced (right) forms, discussed next; essentially, in fully-reduced BDDs edges can skip levels associated with a variable, whereas in quasi-reduced BDDs edges never skip levels. While the size of a BDD can be exponential in the number of variables, many functions can be encoded compactly.

2.2 Canonical Binary Decision Diagrams

Decision diagrams can be reduced to different forms, so that they still denote the same function in a more compact *canonical* representation. Most library implementations support only canonical forms, having the property that no two nodes encode the same function. This is achieved by first *eliminating duplicate nodes* and then either *retaining* all redundant nodes (quasi-reduced form, no edge skips levels) or *removing* them all (fully-reduced form, edges skip levels whenever possible), where a node is *redundant*, shown in light color in Figure 1, if its outgoing edges point to the same node.

While other, more specialized, reduction forms have been defined, we only consider the quasi-reduced and fully-reduced forms as they are widely used in decision diagrams libraries. To illustrate why we are interested in these reduction forms, the following section presents some functions from MDDL, the library used in SMART [6], which take BDDs reduced using a specific form as inputs, and we show how BDDL prevents illegal use of such operations.

2.3 Encoding Sets and Relations with BDDs

BDDs can encode sets and relations (we limit the discussion to binary relations, the most widely used in practice). A BDD encodes a set $Y \subseteq \mathbb{B}^L$ through its characteristic function f_Y:

$$i = (i_L, ..., i_1) \in Y \text{ iff } f_Y(i_L, ..., i_1) = 1$$

Analogously, a BDD encodes a relation $R \subseteq \mathbb{B}^L \times \mathbb{B}^L$ through its characteristic function f_R:

$$(i,i') = ((i_L,...,i_1),(i'_L,...,i'_1)) \in R \text{ iff } f_R(i_L,i'_L,...,i_1,i'_1) = 1$$

If the library implementation allows for the same BDD to encode a set or a relation, i.e., a BDD with $2L$ levels may encode a subset of $\mathbb{B}^{2L}$ or a relation over $\mathbb{B}^L$, then the user may use the library in the wrong way: a BDD encoding a set is used as the argument of a function that expects a relation, or vice versa.

For example, given an L-level BDD on $(x_L, ..., x_1)$ rooted at `p` encoding a set $Y \subseteq \mathbb{B}^L$ and a $2L$-level BDD on $(x_L, x'_L ..., x_1, x'_1)$ rooted at `r` encoding a relation $R \subseteq \mathbb{B}^L \times \mathbb{B}^L$, the relational product of `p` and `r` returns the root of the L-level BDD encoding the set $\{i' : \exists i \in Y, (i, i') \in R\}$.

In Section 3, we consider an implementation of the relational product in MDDL; since root nodes are not checked, the user can pass any ill-typed arguments. We show how BDDL captures these type errors statically.

3 Motivation

The design of BDDL and its type system was driven by examining the source code and evolution (history of bug fixes) of three mature DD libraries: CUDD, MDDL, and JavaBDD. We now proceed to showing the actual code and errors from the examined libraries, the equivalent code in BDDL and the BDDL types that would prevent these errors at compile time.

We begin with the BDD composition code from CUDD (Figure 2). The left side shows the actual C++ code from the library. Function `Cudd_bddCompose` (lines 1

```
 1 DdNode * Cudd_bddCompose(
 2   DdManager * dd, DdNode * f,
 3   DdNode * g, int v)
 4 {
 5   DdNode *proj, *res;
 6   /* Sanity check. */
 7   if (v < 0 || v >= dd→ size)
 8     return(NULL);
 9   proj = dd→ vars[v];
10   do {
11     dd→ reordered = 0;
12     res = cuddBddComposeRecur(
13           dd,f,g,proj);
14   } while (dd→ reordered == 1);
15   return(res);
16 }
17 BDD BDD::Compose(BDD g,int v)
18 { ...
19   return BDD(...,
20          Cudd_bddCompose(
21          mgr.node, g.node, v));
22 }
23 int main ()
24 {
25   Cudd mgr(0,2);
26   BDD x = mgr.bddVar();
27   BDD y = mgr.bddVar();
28   BDD h = x * y;
29   BDD j = x + y;
30   BDD k = h.Compose(j,2);
31   /* runtime error or crash */
32 }
```

C++ code

```
letrec two : {ν : nat | ν = 2} =
  (succ (succ 0)) in

letrec bddCompose : bdd[l, r, c]
  →
  bdd[l′, r, c] → {ν : nat | ν ≤ l − 1} =
  λ f .
    λ g .
      λ v .
      <body>
in
letrec h = Bnode(two, ...) in
letrec j = Bnode(...) in
letrec k =
  bddCompose h j two // type error
```

BDDL **code**

Fig. 2. Crashing CUDD: the C++ code leads to `Segmentation fault` in CUDD 2.3.1 and `Unexpected error` in CUDD 2.4.2.

```
BddNode *RelationalProduct(
     BddNode *p, BddNode *r)
{ ...
  ASSERT(( r→ GetLevel()+1)/2
           == p→ GetLevel());
  ...
}
BddNode* construct_set(int l)
{
  BddNode *bdd =
  malloc(sizeof(BddNode)); ...
  return bdd;
}
int main ()
{
  BddNode *f=construct_set(2);
  BddNode *g=construct_set(3);
  BddNode *res =
       RelationalProduct(f,g);
  /* no exception is raised */
}
```

C++ code

```
letrec relationalProduct :
 bdd[l, r, s] → {ν : bdd[l′, r, e]
     | l′ = l + l − 1} → bdd[l, r, s] =
       λ p. λ r. <body>
in
letrec f : bbd[2, q, s] = Bnode(...)
in
letrec g : bbd[3, q, s] = Bnode(...)
in
       relationalProduct f g
       /* type error */
```

BDDL **code**

Fig. 3. The MDDL relational product code silently outputs an incorrect result when two BDDs encoding a set are passed as arguments.

through 16) takes BDDs `f` and `g` as arguments, and returns the result of splicing `g` into `f` at the position indicated by variable with index `v`. Note the runtime sanity check on line 7, which verifies that the index of `v` is positive, but less than the BDD size (the number of levels L, as there is one variable per level), i.e., `v` is within the BDD. The method `BDD:Compose` (lines 17–22) is the C++ library interface for the clients; it invokes `Cudd_bddCompose` on `this.node`, which represents `f`, and `g.node`, which represents `g`, and returns a BDD containing the result of the composition. On lines 23–32 we show the actual code of a program we wrote to crash the library. We first construct a BDD `h` with two levels (`x` at level 0 and `y` at level 1, lines 26–28), another BDD `j` (the number of levels in `j` is not important in this example), and then invoke `h.Compose(j,2)`, which should splice-in `j` at index 2. As index 2 does not exist in the BDD, the check on line 7 fails. In the current version, CUDD 2.4.2, this error leads to the program halting with `Unexpected error`; in a previous version we tested, CUDD 2.3.1, it leads to the program crashing with `Segmentation fault`, because the check on line 7 uses `>` rather than `>=`. In either case, the error manifests itself only during execution, and the CUDD client is left with little information as to what the cause of the error is. On the right side of Figure 2 we show the BDDL code that models the BDD composition via function `bddCompose`; while we have not discussed the BDDL syntax and typing yet, note that our refinement type system

```
BddNode* Union_QQ(
  BddNode *p, BddNode *q) {
 ASSERT((k = p→ GetLevel())
      == q→ GetLevel());
 ...
 ka = answer→ GetLevel();
 ASSERT(ka == k);
 return answer;
}
int main ()
{
  BddNode *f = new_bdd(1);
  BddNode *g = new_bdd(2);
  BddNode *res = union(f,g);
     /* runtime error */
}
```

C++ code

```
letrec union : bbd(l, q, c) → bbd[l, q, c]
  → {ν : bbd[l′, q, c]) | l′ ≤ l} =
     λ p . λ r . <body>
in letrec f : bbd[1, q, s]
   = Bnode(1, ...) in
letrec g : bbd[2, q, s]
   = Bnode(2, ...) in
 union f g // type error
```

BDDL code

Fig. 4. The union of two BDDs leads to a runtime error when two BDDs of different levels are passed as arguments (from MDDL).

```
public class BasicTests extends BDDTestCase {...
  public void testCrash() {
    reset(); Assert.assertTrue(hasNext());
    BDDFactory bdd = nextFactory();
    BDD a = bdd.one();
    bdd.reorder(bdd.getReorderMethod());
// java.lang.NullPointerException
}
```

Java code

```
letrec reorder :
{ν : bbd[l, r, c] | l ≥ 1} → {ν : bbd[l, r, c] | l ≥ 1} =
  λ p : bbd[l, r, c]. <body>
in
reorder [1] //type error
```

BDDL code

Fig. 5. JavaBDD: reordering a terminal-only BDD leads to a `nullPointerException`.

allows us to specify that (a) the argument `v` should be less than the number of levels in `f` (line 5), and (b) that the argument `v` has value 2, which is the number of levels in `h`. Trying to apply `bddCompose` to arguments `h j two` (line 14) results in a static typing error.

As a second example, we focus on the `RelationalProduct(p,r)` function from MDDL [6], which computes the relational product of BDDs `p` and `r` (Figure 3). As mentioned in Section 2.3, the argument `p` must be an L-level BDD encoding a set, whereas the argument `r` must be a $2L$-level BDD encoding a relation. MDDL does not differentiate between sets and relations in its actual implementation, therefore, the same BDD may encode a set and a relation and the correct use of these two forms is left to the user.[6] The left side of Figure 3 contains an excerpt of the library function `RelationalProduct(p,r)` (lines 1-7); the `ASSERT` ensures that the number of levels in `r` is equal to $2L$ (the `+1` is due to levels starting at 0) but no set vs. relation check is performed. The `main` function shows the C++ client code: it uses `construct_set()` to build two BDDs `f` and `g` encoding sets (lines 17-18); next, we pass `f` and `g` as inputs to `RelationalProduct(p,r)`, which computes the result considering the second argument as a relation. Upon completion (line 20), no error is signaled, and the incorrect result is silently returned to the client. On the right side of Figure 3 we show the BDDL translation. Note how (a) the refinement types of l and l_e on `relationalProduct`'s signature express the runtime checks in the C++ code, and (b) the `s` and `e` on `p` and `r`'s types force them to be a set and relation, respectively. When a client tries to apply `relationalProduct` on `f` and `g` (line 10) a typing error is raised at compile-time, because the encoding of `g` is `s`, rather than `e`, the expected encoding.

As a third example, we present the function `Union_QQ(p,q)` from MDDL, which computes the union set of two quasi-reduced BDDs `p` and `q` and requires them

[6] To eliminate ambiguity and prevent a potential incorrect use of functions that support both forms of encoding [1], some libraries do not leave this choice to the user.

to have the same number of levels. A portion of the C++ code of this function is shown in Figure 4 left (lines 1–9); the `ASSERT`s are used to check that both the input BDDs `p` and `q`, and the output BDD, `answer`, have the same number of levels, `k`. The client (lines 10–16) tries to compute the union of `f` and `g`, two BDDs with different numbers of levels. The library generates a runtime error because the `ASSERT` on line 3 fails. On the right side of Figure 4 we show the BDDL equivalent of the library and client code; the l on `union`'s type forces the input BDDs to have the same number of levels and the output to have at at most l levels. The application of `union` to `f` and `g`, which have 1 and 2 levels, respectively, is ill-typed and results in a compile-time error.

As a fourth example, we show how reordering a terminal-only BDD leads to a `nullPointerException` in JavaBDD. The left side of Figure 5 shows a simple method, `testCrash`, that we wrote as an addition to the JavaBDD test suite. The method first performs some initialization/sanity checks (lines 3–4), then creates a BDD, containing just the terminal $\boxed{1}$, and invokes `reorder`, which generates a `nullPointerException`. Reordering a terminal-only BDD should not be allowed, or at most should be a no-op. On the right side we show the BDDL code for this scenario; the type of `reorder` (lines 2–3) stipulates that it can only be invoked on BDDs with at least one level. The application of `reorder` to $\boxed{1}$ (line 5) is ill-typed and will be rejected, because $\boxed{1}$ is at level 0.

In the companion technical report [14] we provide more examples of how BDDL can be used to specify interfaces of CUDD, MDDL and JavaBDD libraries.

4 The BDDL Language

We now present BDDL, our core language for BDDs. We designed BDDL to support abstractions for key BDD operations, functionality, and semantic properties, based on our examination of several mature decision diagram libraries. BDDL provides forms to build a BDD (via `Bnode (...)`), use a BDD `t` (`t.level`, `t.index`, `t.var`, `t.tchild`, `t.fchild`), and types, possibly with refinements, to capture semantic properties of BDDs ($bdd[l, r, c]$).

The syntax is shown in Figure 6. *Indexes* i are unique id's associated with each BDD node: ID_0 and ID_1 are reserved for terminals while id_2, id_3, and higher are used for non-terminals; index uniqueness is a enforced by our typing system. *Strings* g are used to hold variable names; for simplicity, we only allow $x_1, x_2, x_3, \ldots$ as indexes. The *level* l specifies the level of BDD nodes; it can be $\perp$ (unspecified level), or a natural number nv. The *reduction* r of a BDD can be $\perp$ (unspecified), `f` (fully reduced), or `q` (quasi reduced). The *encoding* c of a BDD can be $\perp$ (unspecified), `s` (set), or `e` (relation). *Predicate variables* π are used to construct the predicates on refinement types—either terms ν, or variables l, r, or c, which are universally quantified over their respective domains and $\perp$.

$i :=$	$id_2, id_3, id_4, \ldots$	*index*
$g :=$	$x_1, x_2, x_3, \ldots$	*string*
$l :=$	$\bot,\ nv$	*level*
$r :=$	$\bot$, `f`, `q`	*fully or quasi reduced*
$c :=$	$\bot$, `s`, `e`	*set or relation*
$\pi :=$	$\nu \mid l \mid r \mid c$	*predicate variable*
$\Gamma :=$	$ID_0 : Id, ID_1 : Id \mid \Gamma, x : \tau \mid \Gamma, h : \tau$	*typing context*
$\mu :=$	$\emptyset \quad \mid \quad \mu, h \mapsto v$	*location binding*
$t ::=$		***Terms***
	v	*value*
$\mid$	x	*variable*
$\mid$	`succ t` $\mid$ `pred t`	*successor, predecessor*
$\mid$	`iszero t`	*zero test*
$\mid$	`t t`	*application*
$\mid$	`letrec` $x : \tau$ `= t in t`	*(recursive) let*
$\mid$	`if t then t else t`	*if statement*
$\mid$	`ref t` $\mid$ `!t`	*reference, dereference*
$\mid$	`Bnode (t,` i`, t, t, t)`	*BDD node*
$\mid$	`t.level`	*level of a node*
$\mid$	`t.index`	*index of a node*
$\mid$	`t.var`	*variable of a node*
$\mid$	`t.tchild` $\mid$ `t.fchild`	*true, false child of a node*
$nv ::=$		***Numeric values***
	0	*zero constant*
$\mid$	$S(nv)$	*successor value*
$v ::=$		***Values***
	`true` $\mid$ `false`	*boolean value*
$\mid$	nv	*numeric value*
$\mid$	$i \quad \mid \quad g$	*index, string*
$\mid$	$\lambda x : \tau.$`t`	*abstraction value*
$\mid$	h	*heap location*
$\mid$	$\boxed{0} \quad \mid \quad \boxed{1}$	*boolean terminal*
$\mid$	`Bnode`(v, v, v, v, v)	*boolean node*
$\tau ::=$		***Types***
	$bool \quad \mid \quad nat$	*booleans, naturals*
$\mid$	$string \quad \mid \quad Id$	*strings, identifiers*
$\mid$	$\tau \rightarrow \tau$	*function type*
$\mid$	$ref\, \tau$	*reference type*
$\mid$	$bdd[l, r, c]$	*node types*
$\mid$	$\{\nu : \tau \mid p(\pi)\}$	*refined type*
$exp ::=$		***Arithmetic expressions***
	$nv \mid \quad exp + exp \quad \mid \quad exp - exp$	
$p(\pi) ::=$		***Refinement predicates***
	$\pi \quad = \quad exp$	*constant type*
	$\mid \pi \leq exp \mid \pi \geq exp \mid \pi \neq exp$	*restrained type*
$\mid$	$p(\pi) \quad \wedge \quad p(\pi)$	*conjunction*

Fig. 6. BDDL syntax.

The typing context (environment) Γ contains variable names and their associated types $(x : \tau)$, as well as heap locations and their types $(h : \tau)$. We augment Γ with the unique indexes ID_0 and ID_1 assigned to $\boxed{0}$ and $\boxed{1}$, respectively.

The heap μ is a map from references h to values v.

A Bddl *term* can be a value v; a variable x; the `succ`essor or `pred`ecessor of another term, whose semantics is the successor and predecessor of natural numbers, e.g., the term `succ S(S(S(0)))`, successor of 3 in other words, reduces to $S(S(S(S(0))))$ and corresponds to numerical value 4, whereas `pred S(0)` reduces to 0, meaning the predecessor of 1 is 0; an `iszero` test for zero; an application; a `letrec` binding that allows recursion; an `if` statement; a reference `ref t` or dereference `!t`. Note the presence of references, but the lack of assignment in our system: this is by choice, as we want to allow sharing BDD nodes but avoid mutation to preserve a BDD's structural integrity. We represent BDDs via 5-tuples, i.e., terms `Bnode(`$\mathtt{t}_1$`,`i`,`$\mathtt{t}_3$`, `$\mathtt{t}_4$`,`$\mathtt{t}_5$`)`; a `Bnode` is a boolean node in a BDD; $\mathtt{t}_1$ is the node level (a natural number), i represents the index of the node (a unique identifier), $\mathtt{t}_3$ holds the variable associated with a node (a string), $\mathtt{t}_4$ represents a reference to the true ('1') child of the node and $\mathtt{t}_5$ is a reference to its false ('0') child. To extract the elements of the 5-tuple, we use `t.index`, `t.level`, `t.var`, `t.tchild`, and `t.fchild`; i.e., `(Bnode(`$\mathtt{t}_1$`,`i`,`$\mathtt{t}_3$`,`$\mathtt{t}_4$`,`$\mathtt{t}_5$`)).level = `$\mathtt{t}_1$, `(Bnode(`$\mathtt{t}_1$`,`i`,`$\mathtt{t}_3$`,`$\mathtt{t}_4$`,`$\mathtt{t}_5$`)).index = `i, and so on.

The top of Figure 9 in the companion technical report [14] illustrates the use of the Bddl language to model a BDD with 3 levels. The whole BDD is represented by $\mathtt{t}_6$`=Bnode(3,`id_6`,`x_2`,`$\mathtt{t}_4$`,`$\mathtt{t}_5$`)` since $\mathtt{t}_4$ and $\mathtt{t}_5$ are references to the children that are themselves `Bnode` terms, i.e., $\mathtt{t}_4$ contains information about its children $\mathtt{t}_2$ and $\mathtt{t}_1$ which in turn contain information about their children. In this case, we reach the terminal nodes, hence we have the information about the entire diagram. We also show the Bddl types of these terms, described in the next section. Recall that we use `t.level`, `t.index`, `t.var`, `t.tchild` and `t.fchild` to extract components of a `Bnode`. For the previous example, we have `t.level(`id_6`) = 3`, `t.index(`id_6`) = `id_6, `t.tchild(`id_6`) = `id_4, and `t.fchild(`id_6`) = `id_5.

Values denote the possible final results of an evaluation. A value in Bddl is either a boolean constant `true` or `false`; a string or an identifier; the number 0 or a non-zero natural number $S(nv)$; an abstraction $\lambda x : \tau.\mathtt{t}$; a heap location h; BDD terminal nodes $\boxed{0}$ or $\boxed{1}$, or a non-terminal node `Bnode`$(v_1, v_2, v_3, v_4, v_5)$ where v_1, v_2, v_3, v_4, and v_5 are values. The typing system, and the syntax of refinements are defined next.

5 Typing, Semantics, and Soundness

5.1 Types

We use primitive types *bool* and *nat* to denote the sets of boolean values and natural numbers, respectively; *string*s are used only for representing variable names (accessible via `.var`) in BDD nodes. We use *Id* to represent identifier types; we define this as a type distinct from *string* or *nat* to account for different DD implementations using different representations (e.g., int, string); the only

Basic typing

$$\Gamma \vdash \texttt{true}\colon bool \quad (\text{T-True}) \qquad \Gamma \vdash \texttt{false}\colon bool \quad (\text{T-False})$$

$$\Gamma \vdash nv : \{\nu : nat \mid \nu = nv\} \quad (\text{T-Nat})$$

$$\Gamma \vdash i : Id \quad (\text{T-Id})$$

$$\Gamma \vdash g : string \quad (\text{T-String})$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\text{T-Var}) \qquad \frac{\Gamma \vdash \texttt{t}_1 : nat}{\Gamma \vdash \texttt{iszero t}_1 \texttt{:} bool}\ (\text{T-IsZero})$$

$$\frac{\Gamma \vdash \texttt{t}_1 : nat}{\Gamma \vdash \texttt{succ t}_1\texttt{:}\ nat}\ (\text{T-Succ}) \qquad \frac{\Gamma \vdash \texttt{t}_1 : nat}{\Gamma \vdash \texttt{pred t}_1 \texttt{:} nat}\ (\text{T-Pred})$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \tau_{11} \to \tau_{12} \qquad \Gamma \vdash \texttt{t}_2 : \tau_{11}}{\Gamma \vdash \texttt{t}_1 \quad \texttt{t}_2 : \tau_{12}}\ (\text{T-App})$$

$$\frac{\Gamma, x : \tau_1 \vdash \texttt{t}_2 : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . \texttt{t}_2 : \tau_1 \to \tau_2}\ (\text{T-Abs})$$

$$\frac{\Gamma \vdash \texttt{t} : \tau_1 \ \tau_1 <: \tau_2}{\Gamma \vdash \texttt{t} : \tau_2}\ (\text{T-Sub})$$

$$\frac{\Gamma, x : \tau_1 \vdash \texttt{t}_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash \texttt{t}_2 : \tau_2}{\Gamma \vdash \texttt{letrec} \quad x : \tau_1 = \texttt{t}_1 \ \texttt{in} \ \texttt{t}_2 : \tau_2}\ (\text{T-Letrec})$$

$$\frac{\Gamma \vdash \texttt{t}_1 : bool \qquad \Gamma \vdash \texttt{t}_2 : \tau \qquad \Gamma \vdash \texttt{t}_3 : \tau}{\texttt{if t}_1 \quad \texttt{then t}_2 \quad \texttt{else t}_3 : \tau}\ (\text{T-If})$$

$$\frac{\Gamma(h) = \tau_1}{\Gamma \vdash h : ref\ \ \tau_1}\ (\text{T-Loc}) \qquad \frac{\Gamma \vdash \texttt{t}_1 \ : \ \tau_1}{\Gamma \vdash \texttt{ref t}_1 \texttt{:}\ ref\ \ \tau_1}\ (\text{T-Ref})$$

$$\frac{\Gamma \vdash \texttt{t}_1 \ : ref\ \ \tau_1}{\Gamma \vdash \texttt{!t}_1 \ \texttt{:}\tau_1}\ (\text{T-Deref})$$

BDD typing

$$\Gamma \vdash \boxed{0} : bdd[0, r, c] \qquad (\text{T-Terminal0})$$

$$\Gamma \vdash \boxed{1} : bdd[0, r, c] \qquad (\text{T-Terminal1})$$

$$\frac{\begin{array}{c} \Gamma \vdash id : Id \quad id \notin dom(\Gamma) \quad \Gamma \vdash v_{var} : string \\ \Gamma \vdash \texttt{t}_0 : \{\nu : nat \mid \nu \geq 1 \wedge \nu = l\} \\ \Gamma \vdash \texttt{t}_1\texttt{:}\ ref\ \ bdd[l', r, c] \\ bdd[l', r, c] <:_B \{\nu : bdd[p, r, c] \mid l = p + 1\} \\ \Gamma \vdash \texttt{t}_2\texttt{:}\ ref\ \ bdd[l'', r, c] \\ bdd[l'', r, c] <:_B \{\nu : bdd[p', r, c] \mid l = p' + 1\} \end{array}}{\Gamma, id : Id \vdash \texttt{Bnode (t}_0\texttt{,}id\texttt{,}v_{var}\texttt{,t}_1\texttt{,t}_2\texttt{)} : bdd[l, r, c]}\ (\text{T-Bnode})$$

$$\frac{\Gamma \vdash \texttt{t} : \{\nu : bdd[l, r, c] \mid l \geq 0 \wedge l \neq \bot\}}{\Gamma \vdash \texttt{t.level}\texttt{:}\ \{\nu : nat \mid \nu = l\}}\ (\text{T-Level})$$

$$\frac{\Gamma \vdash \texttt{t} : \{\nu : bdd[l, r, c] \mid l \geq 0 \wedge l \neq \bot\}}{\Gamma \vdash \texttt{t.index} \ : \ Id}\ (\text{T-Index})$$

$$\frac{\Gamma \vdash \texttt{t} : \{\nu : bdd[l, r, c] \mid l \geq 1 \wedge l \neq \bot\}}{\Gamma \vdash \texttt{t.var}\texttt{:}\ string}\ (\text{T-BVar})$$

$$\frac{\begin{array}{c} \Gamma \vdash \texttt{t} : \{\nu : bdd[l, r, c] \mid l \geq 1 \wedge l \neq \bot\} \\ \tau' <: ref\ \ bdd[l - 1, r, c] \end{array}}{\Gamma \vdash \texttt{t.tchild} \ : \ \tau'}\ (\text{T-TrueChild})$$

$$\frac{\begin{array}{c} \Gamma \vdash \texttt{t} : \{\nu : bdd[l, r, c] \mid l \geq 1 \wedge l \neq \bot\} \\ \tau' <: ref\ \ bdd[l - 1, r, c] \end{array}}{\Gamma \vdash \texttt{t.fchild} \ : \ \tau'}\ (\text{T-FalseChild})$$

$$\frac{\Gamma \vdash \texttt{t} : bdd[l, r, c] \qquad fully(\texttt{t})}{\Gamma \vdash \texttt{t} : bdd[l, f, c]}\ (\text{T-Fully})$$

Subtyping

$$\frac{}{\tau <: \tau}\ (\text{S-Refl}) \qquad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}\ (\text{S-Trans})$$

$$\frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_1}{ref\ \ \tau_1 <: ref\ \ \tau_2}\ (\text{S-Ref}) \qquad \frac{\tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}\ (\text{S-Fun})$$

$$\frac{}{\{\nu : \tau \mid p(\nu)\} <: \tau}\ (\text{S-Refin})$$

$$\frac{p(\nu_1) \quad \Rightarrow \quad p(\nu_2)}{\{\nu_1 : \tau \mid p(\nu_1)\} <: \{\nu_2 : \tau \mid p(\nu_2)\}}\ (\text{S-Pred})$$

Auxiliary judgments

$$\frac{}{bdd[l, r, c] <:_B bdd[l, r, c]}\ (\text{S-BRefl})$$

$$\frac{l' \leq l}{bdd[l', \texttt{f}, c] <:_B bdd[l, \texttt{f}, c]}\ (\text{S-BFully})$$

$$\frac{bdd[l_1, r_1, c_1] <:_{LAT} bdd[l_2, r_2, c_2]}{bdd[l_1, r_1, c_1] <: bdd[l_2, r_2, c_2]}\ (\text{S-Lat})$$

Node redundancy

$$\frac{}{fully(\boxed{1})}\ (\text{W-Fully1}) \qquad \frac{}{fully(\boxed{0})}\ (\text{W-Fully0}) \qquad \frac{\begin{array}{c} fully(\texttt{t}_4) \qquad fully(\texttt{t}_5) \\ \texttt{(!t}_4\texttt{).index} \neq \texttt{(!t}_5\texttt{).index} \end{array}}{fully(\texttt{Bnode(t}_1\texttt{, i, t}_3\texttt{, t}_4\texttt{, t}_5\texttt{))}}\ (\text{W-Fully})$$

Fig. 7. BDDL typing.

values that inhabit this type are the unique *id*'s associated with BDD nodes (accessible via `.index`). Function types have the standard representation, $\tau \to \tau$, as do references, $ref\ \ \tau$.

$bdd[l, r, c]$ is the fundamental type in our language. A BDD node has type $bdd[l, r, c]$, where l refers to the level of the node, r refers to the reduction form of the decision diagram, and c refers to the encoding of the diagram (a set or a relation). For example, node $\texttt{t}_6$ = `Bnode(`3`,`id_6`,`x_2`,`$\texttt{t}_4$`,`$\texttt{t}_5$`)` (Figure 9 [14]) has type $bdd[3, r, c]$, i.e., it is a node at level 3 of a BDD with no restriction on the reduction or the encoding.

To express semantic BDD properties, we use refinement types [9], in the form proposed by Rondon et al. [19]. A refined type $\{\nu : \tau \mid p(\pi)\}$ expresses a refinement of the primitive type τ. For example, the refined type $\{\nu : nat \mid \nu \geq 1\}$ describes the set of natural numbers that are greater or equal to 1; the type $\{\nu : nat \mid \nu = n\}$ describes the type of the natural number n; the type of function `Union_QQ` from Figure 4 in Section 3 is:

$$bdd[l, \mathtt{q}, \mathtt{s}] \rightarrow bdd[l, \mathtt{q}, \mathtt{s}] \rightarrow \{\nu : bdd[l', \mathtt{q}, \mathtt{s}] \mid l' \leq l\})$$

meaning it takes two quasi-reduced set-encoding BDDs with arbitrary (but equal) numbers of levels l, and returns a quasi-reduced set-encoding BDD with at most the same number of levels l. Note how r and c are quantified over their domains; for instance, the type of $\boxed{1}$ is $bdd[0, r, c]$, meaning that a terminal can only be at level 0, but can be part of fully- or quasi-reduced BDDs, encoding sets or relations.

Finally, the language of predicates $p(\pi)$ used on refinement types allows us to express equality and inequality refinements, e.g., $\{\nu : nat \mid \nu = 2\}$, and conjunctions of simple arithmetic expressions, e.g., $\{\nu : nat \mid \nu \geq 1 \wedge \nu = 5\}$.

5.2 Typing Rules

The BDDL typing rules shown in Figure 7 can be split into two categories: basic typing and BDD-specific typing. The basic typing rules are the standard rules for lambda calculus extended with booleans and unary representation of natural numbers, as presented by Pierce [18], with the following modifications: when type-checking natural values nv via (T-NAT), we use refinements to represent their value, i.e., $\{\nu : nat \mid \nu = nv\}$; we add the rules (T-ID) and (T-STRING) to type-check id's and variable names.

The top right side of Figure 7 shows our BDD typing rules. Rules (T-TERMINAL0) and (T-TERMINAL1) stipulate that the $\boxed{0}$ and $\boxed{1}$ terminals can be found in any BDD at level 0.

(T-BNODE) is the fundamental rule in our system. To enforce identifier uniqueness, we require $id \notin dom(\Gamma)$. We require that $\mathtt{t}_0 : \{\nu : nat \mid \nu \geq 1 \wedge \nu = l\}$ i.e., that the node's level be at least 1 (to ensure that `Bnode`s are non-terminals), and the same level l as in the conclusion of the judgment. To allow sharing, we store the children $\mathtt{t}_1$ and $\mathtt{t}_2$ by reference; they have types $bdd[l', r, c]$ and $bdd[l'', r, c]$, respectively. We now explain how `BNode`s can store both fully and quasi-reduced BDDs, and do so safely. Consider the $<:_B$ subtyping premise for the true child:

$$bdd[l', r, c] <:_B \{\nu : bdd[p, r, c] \mid l = p + 1\}$$

If this is a quasi-reduced BDD which allows no level skipping, then we have $l' = p = l - 1$. If this is a fully-reduced BDD which allows level skipping, then we have $l' \leq p$ where $p = l - 1$, and by applying the (S-BFULLY) subtyping rule, we can actually have $l' < p$; for instance, a `Bnode` at level 3 can have its children be at level 1; the premises for the false-child $\texttt{t}_2$ are similar. Note the use of the same variables r and c in both the children and current node (premises and conclusion of the rule) which forces the BDD to be consistent: either fully-reduced or quasi-reduced, and, respectively, either set-encoding or relation-encoding. Note that the $<:_B$ relationship is different from the subtyping relationship, as it only allows us to establish a relationship between reduction forms.

The rule (T-LEVEL) type-checks extracting a node's level. The level is a refined *nat* type (refinement: $l \geq 0 \wedge l \neq \bot$) with the same l as in the refinement of the BDD node; we allow $l = 0$ because asking for the level of a terminal is permitted; we do not allow $l = \bot$ because $\bot$ means unspecified level. The rule (T-INDEX) is similar—we can ask for the level of any node (including terminals, whose indexes are ID_0 and ID_1) as long as its level is not $\bot$. The rule (T-BVAR) is similar, though it only works on non-terminals ($l \geq 1$), as terminals cannot encode variables. The rule (T-TCHILD) type-checks the extraction of the true-child; the premises are the same as for (T-BVAR), since we can only extract children of non-terminals; note again the subtyping in the premises, which allows level skipping for fully-reduced, but not for quasi-reduced, BDDs; (T-FCHILD) is similar.

Our system lacks type polymorphism, which we omit for simplicity at the expense of expressivity. Note however that the variables l, r and c used on type rules allow specification of quasi-reduced, fully-reduced, or generic BDDs. For example, we can construct BDDs whose reducing and encoding are unspecified, and allow functions to operate on them (albeit the range of operations is constrained, as with any polymorphic function). Consider a simple BDD, $\texttt{BNode(1,}id_2\texttt{,}x_1\texttt{,ref }\boxed{0}\texttt{,ref }\boxed{1}\texttt{)}$, that contains one non-terminal node with two children, $\boxed{0}$ and $\boxed{1}$; its type is $\{\nu : bdd[l, r, c] \mid l = 1\}$, or, in short, $bdd[1, r, c]$. This BDD is "generic", in that it can be safely attached to both a fully- or quasi-reduced BDD, that encodes either a set or a relation. Then the BDD can be safely used in a concrete context, e.g., with reduction `f` and an encoding `s`.

To express the absence of redundant nodes in fully-reduced BDDs, we introduce the rules (W-FULLY0), (W-FULLY1), (W-FULLY), and (T-FULLY). The first two rules express the non-redundancy of terminal nodes—they are unique and do not have children nodes. A fully-reduced BDD must not contain any redundant nodes, i.e., its outgoing edges must not point to the same node. We capture this property by applying (W-FULLY) recursively and by checking at each recursion that the indexes of the children nodes are different (this prevents a node from being redundant by prohibiting the true edge and the false edge from pointing to the same node) and that they are fully-reduced. Finally, (T-FULLY) prevents passing a BDD with the type $bdd[l, r, c]$ when a BDD with type $bdd[l, \texttt{f}, c]$ is expected when the BDD is not fully-reduced.

$$\begin{tikzcd}
 & \mathit{bdd}[\bot,\bot,\bot] \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\mathit{bdd}[l,\bot,\bot] \arrow[d, dash] \arrow[dr, dash] & \mathit{bdd}[\bot,r,\bot] \arrow[d, dash] \arrow[dr, dash] & \mathit{bdd}[\bot,\bot,c] \arrow[dll, dash] \arrow[d, dash] \\
\mathit{bdd}[l,\bot,c] \arrow[dr, dash] & \mathit{bdd}[l,r,\bot] \arrow[d, dash] & \mathit{bdd}[\bot,r,c] \arrow[dl, dash] \\
 & \mathit{bdd}[l,r,c] &
\end{tikzcd}$$

Fig. 8. Subtyping lattice (the $<:_{LAT}$ relation).

5.3 Subtyping

Subtyping is essential in our system, as it allows types that contain more information about a BDD node (e.g., fixed number of levels) to be used in contexts where knowing the number of levels is not required; also, it allows refined types with strong conditions to be used in contexts where types' conditions are more relaxed. The subtyping rules are shown on the bottom of Figure 7. (S-Refl), (S-Ref), (S-Trans), and (S-Fun) are standard. (S-BFully) allows level skipping for fully-reduced BDDs and (S-BRefl) allows the B-subtyping relationship to be reflexive. Rule (S-Refin) indicates that a refinement of a type τ is a subtype of τ. Rule (S-Pred) allows us to use a more constrained refinement type, i.e., with a stronger predicate $p(\nu_1)$ (e.g., $\nu \geq 0 \wedge \nu \neq 3$) in a context that only requires a weaker predicate $p(\nu_2)$ (e.g., $\nu_2 \geq 1$). Note that this rule is only used for ν predicates but not for the other predicates since weakening the l, r and c constraints in a $\mathit{bdd}[l, r, c]$ has a different semantics. As an example, $\{\nu : \mathit{bdd}[l, r, c] \mid l = 0\}$ is not a subtype of $\{\nu : \mathit{bdd}[l, r, c] \mid l \geq 0\}$ even though $(l = 0) \Rightarrow (l \geq 0)$ since a terminal node can only be at level 0.

(S-Lat) stipulates that a BDD with type $\mathit{bdd}[l_1, r_1, c_1]$ can be used in a context where a $\mathit{bdd}[l_2, r_2, c_2]$ is required, as long as $\mathit{bdd}[l_1, r_1, c_1]$ is on a downward path from $\mathit{bdd}[l_2, r_2, c_2]$ in the subtyping lattice. Figure 8 shows the lattice of subtyping relationships in our system; we use $\bot$ to represent unspecified values. Essentially, $\bot$ is a union type over the domains of l, r, and c.

In the companion technical report [14] we provide two examples of how type checking and inference are performed in Bddl.

5.4 Semantics and Soundness

The operational semantics is standard, small-step, defined as reduction on terms $\mathtt{t}$ and memory stores μ, i.e.,

$$\mu; \mathtt{t} \longrightarrow \mu'; \mathtt{t}'$$

The full semantics can be found in the companion technical report [14]. We now state the progress and preservation lemmas for our system.

Lemma 1 (Progress). *If* `t` *is a closed, well-typed term (such that* `t` $: \tau$*) then either* `t` *is a value or else, for any store* μ *such that* $\Gamma \vdash \mu$*, there exists a term* $\mathtt{t}'$ *and store* μ' *with* $\mu;\mathtt{t} \longrightarrow \mu';\mathtt{t}'$

Proof. By induction on the typing derivation `t` $: \tau$.

Lemma 2 (Preservation). *If* $\Gamma \vdash \mathtt{t}:\tau, \Gamma \vdash \mu$ *and* $\mu;\mathtt{t} \longrightarrow \mu';\mathtt{t}'$ *then for some* Γ'*, we have* $\Gamma' \vdash \mathtt{t}':\tau$ *and* $\Gamma' \vdash \mu'$*.*

Proof. By induction on the typing derivation $\Gamma \vdash \mathtt{t}:\tau$.

We can now state the soundness theorem: well-typed programs do not go wrong.

Theorem 1 (Soundness). *If* $\Gamma \vdash \mathtt{t}:\tau$*, and* $\Gamma \vdash \mu$*, then either* `t` *is a value, or there exist* Γ'*,* μ'*,* $\mathtt{t}'$*, such that* $\mu;\mathtt{t} \longrightarrow \mu';\mathtt{t}'$ *and* $\Gamma' \vdash \mathtt{t}' : \tau$ *and* $\Gamma' \vdash \mu'$*.*

6 Related work

The work closest to ours, on which we partially build, is Kawaguchi et al. [13]'s liquid types (which in turn built upon Rondon et al.'s work [19]). We borrow their syntax of refinement types expressed as logically quantified predicates over program variables. In addition, we add formal support for expressing and checking structural constraints on BDDs. They can check a broad-range of properties on various data structures, such as lists, trees, heaps, maps, and vectors. In contrast, we focus on one data structure, BDD, but express and verify a broader range of structural and logical properties. For example, they can check BDDs for ordering: `mk_not::x:bdd`$\rightarrow \{\nu$`:bdd`$|$ $\mathit{var}\ x \leq \mathit{var}\ \nu\}$.
The same type can be expressed in our formalism as:
`mk_not`:$\{\nu_1 : \mathit{bdd}[l,r,c]\} \rightarrow \{\nu_2 : \mathit{bdd}[l',r',c'] \mid \nu_1.\mathtt{var} \leq \nu_2.\mathtt{var}\}$.
However, this type does not guarantee preservation of structural integrity, such as number of levels, reduction, or encoding status. We believe that their system can capture some of these properties but, as our focus is on BDDs, we can express the properties more directly and concisely. For instance, the type of `reorder` (with the natural order $x_1 \leq x_2 \leq \cdots \leq x_n$) is, in our system:
$\{\nu_1 : \mathit{bdd}[l,r,c] \mid l \geq 1\} \rightarrow \{\nu_2 : \mathit{bdd}[l,r,c] \mid \nu_2.\mathtt{var} \leq \mathtt{!}(\nu_2.\mathtt{tchild}).\mathtt{var} \wedge \nu_2.\mathtt{var} \leq \mathtt{!}(\nu_2.\mathtt{fchild}).\mathtt{var}\}$, which enforces that the levels, reduction, and encoding of input BDD and output BDD are the same, hence preserving structural integrity. Our system is less expressive in general, as we do not allow type-level polymorphism or nested refinements. We have pursued one data structure, BDD, and its manipulation in three DD libraries; they have verified the implementations of a wide-range of data structures from OCaml's standard library.

Drechsler [7] presented a run-time technique for BDD verification, using a recursive checksum method to verify BDD integrity. Their approach has been motivated by memory errors, e.g., copy faults and errors due to aliasing; in their approach, such memory errors are detected at runtime. Our approach is intended to capture and verify higher-level properties, and to do so at compile time; we do not have a memory model, though we can prevent errors such as use-before-allocation, because we do not allow uninitialized references.

Stanković and Astola [21] take a syntactic approach to checking the structural integrity of BDDs. A `Treetype` XML wrapper stores information about a particular BDD instance (e.g., number of variables, levels, edges) as attribute values, and these values are used to reason about what operations are permitted for this particular BDD instance—the BDD's body and collection of attributes constitute a type which can be type-checked by an XML parser. Their approach can be readily extended to more elaborate decision diagram flavors, such as EVBDD, by merely adding attribute fields. In contrast to this syntactic approach, our method uses a type system and semantics to check BDD-manipulating code, so BDDs created by Bddl-checked code are correct by construction.

Bernasconi et al. [4] as well as Bernasconi and Ciriani [3] have proposed dynamic approaches for detecting and repairing index or pointer errors in Ordered Binary Decision Diagrams. Their approach differs from ours in the same way as described for the previous two approaches, i.e., our approach is enforcing a different set of properties, and does so statically.

Giorgino and Strecker [10] as well as Ortner and Schirmer [16, 17] provide mechanized proofs in Isabelle for Isabelle/HOL-expressed implementations, e.g., proving certain BDD properties by casting the verification of BDD (normalization) as a particular case of verifying pointer-manipulating programs. Our goal was not a mechanized proof, but rather Bddl is a type system for static checking. For example a CUDD checker, or a checker for any substantial implementation written in popular imperative languages, could be built by adding a simple front-end that maps C++ to Bddl. It is unclear how a substantial imperative implementation such as CUDD can be transformed into Isabelle to be checked.

7 Conclusion

We have presented Bddl, a calculus to verify BDD correctness. Bddl combines language support to build and safely manipulate BDDs with refinement types that allow programmers to concisely express structural and logical invariants. We formalized Bddl using a type system and small-step operational semantics, and proved it sound. We have presented examples of how frequently-used BDD library functions can be expressed and statically verified using Bddl. We plan to extend this work in two directions: (1) automatic verification of library code, and (2) extend Bddl to reason about, and verify, properties of more general variants of decision diagrams, such as multi-way, multi-terminal, and edge-valued.

References

1. Andrew Miner and others: MEDDLY: Multi-terminal and Edge-valued Decision Diagram LibrarY, https://meddly.sourceforge.io/
2. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.: Decision diagrams for optimization. Springer (2016)
3. Bernasconi, A., Ciriani, V.: Index-resilient zero-suppressed bdds: Definition and operations. ACM Trans. Des. Autom. Electron. Syst. **21**(4) (may 2016)
4. Bernasconi, A., Ciriani, V., Lago, L.: On the error resilience of ordered binary decision diagrams. Theoretical Computer Science **595**, 11–33 (2015)
5. Bryant, R.: On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. IEEE Transactions on Computers **40**(2), 205–213 (1991)
6. Ciardo, G., Miner, A.S.: SMART: Stochastic Model-checking Analyzer for Reliability and Timing, available at https://asminer.github.io/smart/
7. Drechsler, R.: Verifying integrity of decision diagrams. Integr. **32**(1-2), 61–75 (2002)
8. Fabio Somenzi: CUDD: CU Decision Diagram Package, https://github.com/ivmai/cudd
9. Freeman, T., Pfenning, F.: Refinement types for ML. p. 268–277. PLDI '91, Association for Computing Machinery, New York, NY, USA (1991)
10. Giorgino, M., Strecker, M.: Correctness of pointer manipulating algorithms illustrated by a verified bdd construction. In: FM 2012: Formal Methods. pp. 202–216
11. Groen, F., Smidts, C., Mosleh, A., Swaminathan, S.: Qras - the quantitative risk assessment system. In: Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318). pp. 349–355 (2002)
12. John Whaley: JavaBDD, http://javabdd.sourceforge.net/
13. Kawaguchi, M., Rondon, P., Jhala, R.: Type-based data structure verification. p. 304–315. PLDI '09 (2009)
14. Lembachar, Y., Rusich, R., Neamtiu, I., Ciardo, G.: BDDL: A type system for binary decision diagrams. Tech. rep., Department of Computer Science, NJIT (May 2022), https://web.njit.edu/~ineamtiu/pubs/bddl-tr.pdf
15. Loekito, E., Bailey, J., Pei, J.: A binary decision diagram based approach for mining frequent subsequences. Knowledge and Information Systems **24**(2), 235–268 (2010)
16. Ortner, V., Schirmer, N.: Verification of bdd normalization. In: Hurd, J., Melham, T. (eds.) Theorem Proving in Higher Order Logics. pp. 261–277 (2005)
17. Ortner, V., Schirmer, N.: Bdd normalisation. Archive of Formal Proofs (Feb 2008), https://isa-afp.org/entries/BDD.html, Formal proof development
18. Pierce, B.C.: Types and programming languages. MIT Press (2002)
19. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. p. 159–169. PLDI '08 (jun 2008)
20. Siminiceanu, R.I., Ciardo, G.: Formal verification of the nasa runway safety monitor. Int. J. Softw. Tools Technol. Transf. **9**(1), 63–76 (feb 2007)
21. Stanković, S., Astola, J.: Xml framework for various types of decision diagrams for discrete functions. IEICE Transactions **90-D**, 1731–1740 (11 2007)
22. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. p. 131–144. PLDI '04
23. Xing, L., Amari, S.V.: Binary Decision Diagrams and extensions for system reliability analysis. John Wiley & Sons (2015)
24. Yanushkevich, S.N., Miller, D.M., Shmerko, V.P.., Stankovic, R.S.: Decision Diagram Techniques for Micro- and Nanoelectronic Design Handbook (2006)
25. Yoon, S., De Micheli, G.: An application of zero-suppressed binary decision diagrams to clustering analysis of dna microarray data. pp. 2925–2928. EMBC'04