Boxes: A Symbolic Abstract Domain of Boxes

Arie Gurfinkel and Sagar Chaki

Carnegie Mellon University

Abstract. Numeric abstract domains are widely used in program analyses. The simplest numeric domains over-approximate disjunction by an imprecise join, typically yielding path-insensitive analyses. This problem is addressed by domain refinements, such as finite powersets, which provide exact disjunction. However, developing correct and efficient disjunctive refinement is challenging. First, there must be an efficient way to represent and manipulate abstract values. The simple approach of using "sets of base abstract values" is often not scalable. Second, while a widening must strike the right balance between precision and the rate of convergence, it is notoriously hard to get correct. In this paper, we present an implementation of the Boxes abstract domain – a refinement of the well-known Box (or Intervals) domain with finite disjunctions. An element of Boxes is a finite union of boxes, i.e., expressible as a propositional formula over upper- and lower-bounds constraints. Our implementation is symbolic, and weds the strengths of Binary Decision Diagrams (BDDs) and Box. The complexity of the operations (meet, join, transfer functions, and widening) is polynomial in the size of the operands. Empirical evaluation indicates that the performance of Boxes is superior to other existing refinements of Box with comparable expressiveness.

1 Introduction

Numeric abstract domains are widely used in Abstract Interpretation and Software Model-Checking to infer numeric relationships between program variables. To a large extent, the scalability of the most common domains, intervals, octagons, and polyhedra, comes from the fact that they represent convex sets – i.e., conjunctions of linear constraints. This inability to precisely represent disjunction (and disjunctive invariants) is also their main limitation. It means that analyses dependent on such domains are path-insensitive and produce a high-rate of false positives when applied to verification tasks. In practice, an analyzer based on such a domain uses a *disjunctive refinement* to extend the base domain with disjunctions (or a *disjunctive completion* [8] when all disjunctions are added), and thereby increase its precision.

There are several standard ways to build a disjunctive refinement. The simplest one is to allow a *bounded number of disjuncts* (e.g., [15, 17, 4, 11, 2, 12]). This is typically implemented via finite sets as abstract values (e.g., using $\{a, b\}$ for $a \vee b$), splitting locations in the control flow graph (e.g., unrolling loops, duplicating join points, etc.), or a combination of the two. It has an easy implementation based entirely on the abstract operations (i.e., image, widen, etc.) of the base domain. However, it does not scale to a large number of disjuncts.

The *finite powerset construction* [1, 2] represents disjunctions with finite sets and does not bound the number of disjuncts. Most abstract operations easily extend from the base domain. However, it does not scale to a large number of disjuncts, and widening is notoriously hard to get right (see [2] for examples).

If the base domain is finite, like in predicate abstraction [10], Binary Decision Diagrams (BDDs) [5] over the basis (i.e., predicates) of the base domain is the natural choice for the completion. This approach easily scales to a large number of disjuncts, which is particularly important for a "coarse" base domain. Additionally, the canonicity properties of BDDs eliminate redundant abstract values with the same concretization (which are common with the other approaches).

In this paper, we present a new abstract domain, BOXES, that is a disjunctive refinement of the well-known BOX [7] (or Intervals) domain. That is, each value of BOXES is a propositional formula over interval constraints. We make several contributions. *First*, BOXES values are represented by Linear Decision Diagrams [6] (LDDS), a data structure developed by us in prior work. LDDS are an extension of BDDs to formulas over Linear Arithmetic (LA), and are implemented on top of the state-of-the-art BDD package [18]. Thus, BOXES enjoys all of the advantages of a BDD-based disjunctive refinement, including canonicity of representation and scalability to many disjuncts. This is especially important since BOX is very coarse. BOXES is not only more expressive than BOX or a BDD-based domain, but, in practice, can effectively replace either.

Second, we implement algorithms for image computation of transfer functions for BOXES. *Third*, we develop a novel widening algorithm for BOXES, prove its correctness, and implement it via LDDS. Our widening does not fit any of the known widening schemes for disjunctive refinements, and is of independent interest. *Finally*, we evaluate BOXES on an extensive benchmark against state-of-the-art implementations [3] of BOX and its finite powerset.

There has been a significant amount of research on extending BDDs to formulas over LA, especially in the area of timed- and hybrid-verification (e.g., [19, 16, 13, 20]). In contrast, we concentrate on transfer functions common in program analysis applications; to our knowledge, we are the first to consider widening in this context; and, the first to conduct extensive evaluation of such an approach to a program analysis task.

The rest of the paper is structured as follows. Section 2 gives a brief overview of LDDs. Section 3 presents our abstract domain BOXES. Section 4 describes widening. Section 5 compares BOXES with the finite powerset of BOX. Section 6 presents our experimental results, and Section 7 concludes.

2 Linear Decision Diagrams

In this section, we briefly review Linear Decision Diagrams (LDDs) that are the basis for our abstract domain. For more details see [6].

A decision diagram (DD) is a directed acyclic graph (DAG) in which non-terminal nodes are labeled with decisions and terminal nodes are labeled with values. LDD is a DD with non-terminal nodes labeled by LA constraints and two terminal nodes representing TRUE and FALSE, respectively. LDDs are a natural representation for propositional formulas over LA.

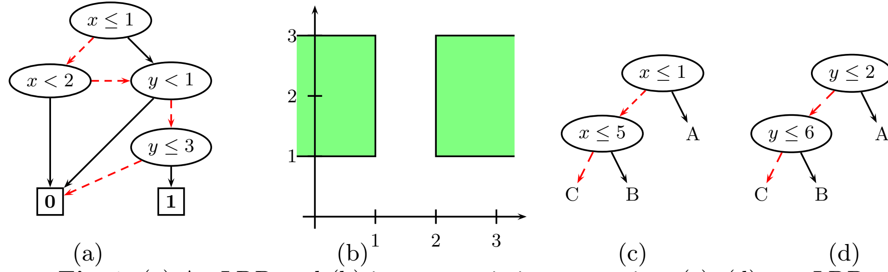

Fig. 1. (a) An LDD and (b) its geometric interpretation; (c), (d) two LDDs.

Example 1. An example of an LDD $(x \le 1 \lor x \ge 2) \land (1 \le y \le 3)$ and its geometric interpretation are shown in Fig. 1(a) and Fig. 1(b). Oval and boxed nodes represent non-terminal and terminal nodes, respectively. Solid and dashed edges represent high (TRUE) and low (FALSE) branches, respectively. □

Formally, an LDD over a fragment T of LA is a DAG with

- Two terminal nodes labeled by $\mathbf{0}$ and $\mathbf{1}$, respectively.
- Non-terminal nodes. Each non-terminal node u has two children, denoted by $H(u)$ and $L(u)$, and is labeled with a T-atom (i.e., an atomic predicate), denoted by $C(u)$.
- Edges, high $(u, H(u))$ and low $(u, L(u))$, for every non-terminal node u.

We write $attr(u)$ for $(C(u), H(u), L(u))$.

An LDD with a root node u represents the formula $\exp(u)$ over T defined by:

$$\exp(u) \triangleq \begin{cases} \text{FALSE} & \text{if } u = \mathbf{0} \\ \text{TRUE} & \text{if } u = \mathbf{1} \\ \text{ITE}(C(u), \exp(H(u)), \exp(L(u))) & \text{otherwise}, \end{cases} \tag{1}$$

where $\text{ITE}(a, b, c) \triangleq (a \land b) \lor (\neg a \land c)$.

For simplicity, we do not distinguish between a node u and $\exp(u)$.

Let V be a set of variables. We write $\mathbb{UBQ}$ for the set $\{x \lesssim k \mid x \in V, k \in \mathbb{Q}, \lesssim \in \{<, \le\}\}$ of rational[1] upper bound constraints, and $\mathbb{IVQ}$ for the set $\{x \sim k \mid x \in V, k \in \mathbb{Q}, \sim \in \{<, \le, =, \ge, >\}\}$ of rational interval constraints. In this paper, we restrict LDDs to $\mathbb{UBQ}$. This is sufficient to represent any propositional formula over $\mathbb{IVQ}$. For example, $x \le 5$ and $x > 5$ correspond to LDDs $\text{ITE}(x \le 5, \mathbf{1}, \mathbf{0})$, and $\text{ITE}(x \le 5, \mathbf{0}, \mathbf{1})$, respectively. For $(x \sim k) \in \mathbb{IVQ}$, we write $\text{VAR}(x \sim k)$ for x.

Let $\preceq \subseteq V \times V$ be a total order on V. We extend it to $\mathbb{UBQ}$ in a natural way:

$$(x_1 \lesssim_1 k_1) \preceq (x_2 \lesssim_2 k_2) \text{ iff } (x_1 \preceq x_2) \lor ((x_1 \lesssim_1 k_1) \Rightarrow (x_2 \lesssim_2 k_2)), \tag{2}$$

and to LDD nodes:

$$u \preceq v \text{ iff } (v \in \{\mathbf{0}, \mathbf{1}\}) \lor (u \notin \{\mathbf{0}, \mathbf{1}\} \land C(u) \preceq C(v)). \tag{3}$$

An LDD u is *ordered* w.r.t. $\preceq$ iff for every node v reachable from u, $v \preceq H(v)$ and $v \preceq L(v)$. An LDD over $\mathbb{UBQ}$ is *locally reduced* iff the following five

[1] While we use rationals for ease of presentation, our results extend to integers.

Operation	Semantics	Complexity	Operation	Semantics	Complexity
AND(f,g)	$f \wedge g$	$O(\lvert f\rvert \cdot \lvert g\rvert)$	OR(f,g)	$f \vee g$	$O(\lvert f\rvert \cdot \lvert g\rvert)$
ITE(h,f,g)	$(h \wedge f) \vee (\neg h \wedge g)$	$O(\lvert h\rvert \cdot \lvert f\rvert \cdot \lvert g\rvert)$	LEQ(f,g)	$f \Rightarrow g$	$\lvert f\rvert \cdot \lvert g\rvert$
NOT(f)	$\neg f$	$O(\lvert f\rvert)$	EXIST(U,f)	$\exists U \cdot f$	$O(\lvert f\rvert \cdot 2^{\lvert U\rvert})$

Table 1. Basic LDD operations. U is a set of variables.

```
1: function LEQ (LDD f, LDD g)
2:     if (f = g) ∨ (f = 0) ∨ (g = 1) then
3:         return TRUE
4:     if (f = 1) ∨ (g = 0) then
5:         return FALSE
6:     if C(f) ⪯ C(g) then v ← C(f)
7:     else   v ← C(g)
8:     return LEQ(f|_v, g|_v) ∧ LEQ(f|_¬v, g|_¬v)
```

```
Require: FNPOS and FNNEG map constraints into LDDs
1: function RC(var x, LDD f, fun FNPOS, fun FNNEG)
2:     if (f = 0) ∨ (f = 1) then return f
3:     v ← C(f)
4:     t ← RC(x, f|_v, FNPOS, FNNEG)
5:     e ← RC(x, f|_¬v, FNPOS, FNNEG)
6:     if x ≠ VAR(v) then return ITE(v, t, e)
7:     t ← AND(FNPOS(v), t)
8:     e ← AND(FNNEG(v), e)
9:     return OR(t, e)
```

Fig. 2. LDD algorithms: LEQ – decides implication, and RC replaces constraints.

conditions hold on every internal node u and v: (1) *No duplicate nodes.* $attr(u) = attr(v) \Rightarrow u = v$; (2) *No redundant nodes.* $L(v) \neq H(v)$; (3) *Normalized labels.* $C(v) \in \mathbb{UBQ}$; (4) *Imply high.* $\neg(C(v) \Rightarrow C(H(v)))$; (5) *Imply low.* If $C(v) \Rightarrow C(L(v))$ then $H(v) \neq H(L(v))$.

For a fixed variable order, ROLDDs are canonical for propositional formulas over $\mathbb{IVQ}$. Specifically, if u and v represent semantically equivalent expressions $(\exp(u) \Leftrightarrow \exp(v))$, then $u = v$. From here on, we fix a total order $\preceq$ on V and say LDD to mean Reduced Ordered LDDs (ROLDD).

Like BDDs, LDDs provide polynomial time algorithms for the basic propositional operations: disjunction (union), conjunction (intersection), negation (complement), and existential quantification (projection). These are summarized in Table 1. Like BDDs, in the worst case, the size of an LDD is exponential in the number of variables. In some implementations (e.g., in [6]) negation $(\neg f)$ is a constant time operation. For completeness, the pseudo code for LEQ is shown in Fig. 2. This, and all other DD algorithms in this paper, are implicitly memoized – results of all intermediate operations are cached and reused as needed. For an LDD f (or its corresponding ITE-expression) and a constraint $v \in \mathbb{UBQ}$, we write $f|_v$ and $f|_{\neg v}$ for, respectively, the positive and the negative cofactor of f w.r.t. v. Let $f[u/w]$ denote the result of the substitution of constraint w for every occurrence of constraint u in f. Then, the cofactors are defined as follows:

$$f|_v \triangleq f[u/\text{TRUE} \mid v \Rightarrow u] \qquad f|_{\neg v} \triangleq f[u/\text{FALSE} \mid u \Rightarrow v]\,. \tag{4}$$

In this paper, we do not distinguish between propositional formulas over $\mathbb{IVQ}$ the corresponding LDDs. For example, we write $f \wedge (1 \leq x \leq 10)$ to mean an LDD obtained by conjunction of an LDD for f and an LDD for $1 \leq x \leq 10$.

3 The BOXES Abstract Domain

Let $\mathbb{R}^n$ be an n-dimensional real vector space. A set $\mathcal{B} \subseteq \mathbb{R}^n$ is a *rational box* iff it is expressible by a finite system of rational interval constraints. The set of all rational boxes of $\mathbb{R}^n$ is denoted by $\mathbb{B}^n$. The BOX abstract domain [7] is a tuple $(\mathbb{B}^n, \subseteq, \emptyset, \mathbb{R}^n, \uplus, \cap)$, where $\subseteq$ is the subset ordering, $\emptyset$ is the empty set, $\uplus$ is the

box hull (i.e., for any two boxes $\mathcal{B}_1$ and $\mathcal{B}_2$, $\mathcal{B}_1 \uplus \mathcal{B}_2 = \mathcal{B}_3$ is the smallest rational box s.t. $\mathcal{B}_1 \cup \mathcal{B}_2 \subseteq \mathcal{B}_3$), and $\cap$ is set intersection. Note that since $\mathbb{B}^n$ is not closed under union, union is over-approximated by the box hull.

A set $\mathcal{BS} \subseteq \mathbb{R}^n$ is a *set of rational boxes* iff there exist rational boxes $\mathcal{B}_1, \ldots, \mathcal{B}_k$ such that $\mathcal{BS} = \bigcup_{i=1}^{k} \mathcal{B}_i$. The set of all sets of rational boxes of $\mathbb{R}^n$ is denoted by $\mathbb{BS}^n$. The BOXES abstract domain is a tuple $(\mathbb{BS}^n, \subseteq, \emptyset, \mathbb{R}^n, \cup, \cap)$, where $\subseteq$ is the subset ordering, $\emptyset$ is the empty set, and $\cup$ and $\cap$ are set union and intersection, respectively. BOXES abstract domain is a disjunctive refinement of BOX domain. Since $\mathbb{BS}^n$ is closed under union, intersection, (and complement) all basic operations are exact. In the rest of this section, we describe our implementation of BOXES using LDDs.

Representation and basic operations. Let $V = \{x_1, \ldots, x_n\}$ be n variables, and $\preceq$ be some total order on V. We assume that each variable is bound to a unique dimension. We use $\boldsymbol{x}$ to denote an element of $\mathbb{R}^n$. Implicitly, $\boldsymbol{x}$ is also a valuation of V, where $\boldsymbol{x}(i)$ is the value of the variable bound to the ith dimension. Then, there is a one-to-one correspondence between Reduced $\preceq$-Ordered LDDs over V and rational boxes over $\mathbb{R}^n$ – each $\mathcal{BS} \in \mathbb{BS}^n$ corresponds to the unique LDD f such that $\boldsymbol{x} \in \mathcal{BS} \Leftrightarrow \boldsymbol{x} \models \exp(f)$. Thus, the domain BOXES is implemented by a tuple $(\mathrm{LDD}(V), \textsc{leq}, \mathbf{0}, \mathbf{1}, \textsc{or}, \textsc{and})$, where $\mathrm{LDD}(V)$ is the set of all LDDs over V. All of the operations are linear in the size of their operands (see Table 1). Note, however, that the size of an LDD over V is in the worst case exponential in $|V|$. In the rest of this paper, we do not distinguish between a set of boxes $\mathcal{BS} \subseteq \mathbb{R}^n$, a corresponding LDD, and a corresponding propositional formula.

In addition to the base operations described above, static analysis applications typically require operations to check for equality and satisfiability (non-emptiness), to compute set-theoretic difference, projection (or unconstraining), images of assignments and guards, and widening (e.g., see [9]). Except for the last two, the operations follow easily from the existing LDD operations. Image computation requires new (but simple) operations, and widening is the most complex one. In the rest of the section, we summarize implementations of the basic and image operations. Widening is deferred to Section 4.

Basic operations. LDDs are canonical for $\mathbb{BS}^n$, hence equality is a constant time operation – two LDD nodes are equivalent iff they have the same attributes. Similarly, satisfiability (and universality) are checked by comparing an LDD to $\mathbf{0}$ (and $\mathbf{1}$). Sometimes (e.g., [2]) it is useful to compute an over-approximation of a set-theoretic difference of two abstract values. BOXES domain is closed under complement and intersection, hence set-theoretic difference is computed exactly using the equivalence: $\mathcal{BS}_1 \setminus \mathcal{BS}_2 \triangleq \mathcal{BS}_1 \cap \neg\mathcal{BS}_2$, where $\mathcal{BS}_1, \mathcal{BS}_2 \in \mathbb{BS}^n$. Note that set-complement is also computed exactly via LDD negation. Projection of a variable x is done via existential quantification EXIST (see last row of Table 1).

Guards and Assignments. Let c be an assignment or a guard. We write $||c||$ for the concrete semantics of c as a function from $\mathbb{R}^n$ to $\mathbb{R}^n$. For example,

$$||x_i \leq 4||(\mathcal{BS}) = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{BS} \wedge \boldsymbol{x}(i) \leq 4\}\,.$$

We write $||\cdot||_{\mathbb{BS}} : \text{LDD} \rightarrow \text{LDD}$ for an abstract transformer that over-approximates $||\cdot||$ in $\mathbb{BS}^n$. That is, given an LDD f for a set of boxes $\mathcal{BS}$, $||c||_{\mathbb{BS}}(f)$ is an LDD representing the smallest set of boxes over-approximating $||c||(\mathcal{BS})$.

The simplest guard is $k_1 \lesssim_1 x_i \lesssim_2 k_2$. The corresponding abstract transformer just adds the appropriate constraint:

$$||k_1 \lesssim_1 x_i \lesssim_2 k_2||_{\mathbb{BS}}(f) \triangleq f \wedge k_1 \lesssim_1 x_i \lesssim_2 k_2\,, \tag{5}$$

where $k_1, k_2 \in \mathbb{Q}$ and $\lesssim_1, \lesssim_2 \in \{<, \leq\}$. Either the lower bound (k_1) or the upper bound (k_2) can be omitted. The simplest assignment is $x_i \leftarrow v$ where v is a symbolic constant such that $k_1 \lesssim_1 v \lesssim_2 k_2$. The abstract transformer is constructed by projecting away the current value of x_i and assuming that x_i is in the same interval as v:

$$||x_i \leftarrow v||_{\mathbb{BS}}(f) \triangleq ||k_1 \lesssim_1 x_i \lesssim_2 k_2||_{\mathbb{BS}}(\exists x_i \cdot f)\,, \tag{6}$$

where k_1, k_2, $\lesssim_1$, and $\lesssim_2$ are as above.

For the next class of transformers, we introduce the function RC shown in Fig. 2. RC takes a variable x, an LDD f, and two functions fnPos and fnNeg that map constraints to LDDs, and returns an LDD obtained by replacing every constraint u on x in f by fnPos(u) on the high-branch of u and by fnNeg(u) on the low branch of u.

For example, let $\text{ID} \triangleq \lambda u \cdot u$ and $\text{COMP} \triangleq \lambda u \cdot \neg u$ be the identity and the complement functions, respectively. Then, $\text{RC}(x, f, \text{ID}, \text{COMP})$ is an identity function – every constraint on x is replaced by itself.

Transfer functions implemented with RC are shown in Table 2, where the columns are: 1st – the command, 2nd – assumptions made, 3rd – the implementation as a call to RC, and 4th and 5th – fnPos and fnNeg functions used by RC, respectively. Throughout, we assume that $a, a_1, a_2, k_1, k_2 \in \mathbb{Q}$, and x and y are two distinct variables. Furthermore, for the guard (the last row), we require that constraints on x precede those on y in the diagram ordering. This is not a limitation – any guard can be rewritten to satisfy this restriction.

The transfer function for $x \leftarrow x + a \cdot y$ is implemented with xpy (see Fig. 3):

$$||x \leftarrow x + a \cdot y||_{\mathbb{BS}}(f) \triangleq \text{XPY}(f, x, a, y)\,. \tag{7}$$

The intuition behind xpy is to traverse the DD and reduce the transfer function to a simpler one as soon as a bound for either x or y is found. There are two cases based on whether $x \preceq y$ or $y \preceq x$.

Example 2. As a simple example, consider applying the transfer function to LDD f shown in Fig. 1(c). Here, we assume that $x \preceq y$, and A, B, and C are sub-diagrams that do not contain x. Note that f is equivalent to:

$$(x \leq 1 \wedge A) \vee (1 < x \leq 5 \wedge B) \vee (5 < x \wedge C) \tag{8}$$

The transformer distributes over disjunction. First, compute $x \leftarrow a \cdot y$ on the sub-diagrams A, B, and C, to get:

$$A' = ||x \leftarrow a \cdot y||(A) \qquad B' = ||x \leftarrow a \cdot y||(B) \qquad C' = ||x \leftarrow a \cdot y||(C)\,. \tag{9}$$

Second, update the results to reflect the bounds on x in A, B, and C:

$$||x \leftarrow x + v_a||(A') \vee ||x \leftarrow x + v_b||(B') \vee ||x \leftarrow x + v_c||(C')\,, \tag{10}$$

Action	Assume	Implementation	$\text{FNPOS}(z \lesssim b)$	$\text{FNNEG}(z \lesssim b)$
$x \leftarrow x + v$		$\text{RC}(x, f, \text{FNPOS}, \text{FNNEG})$	$x \lesssim b + k_2$	$\neg(x \lesssim b + k_1)$
$x \leftarrow a \cdot x$	$a > 0$	$\text{RC}(x, f, \text{FNPOS}, \text{FNNEG})$	$x \lesssim a \cdot b$	$\neg(x \lesssim a \cdot b)$
$x \leftarrow a \cdot x$	$a < 0$	$\text{RC}(x, f, \text{FNPOS}, \text{FNNEG})$	$a \cdot b \lesssim x$	$\neg(a \cdot b \lesssim x)$
$x \leftarrow a \cdot y$	$a > 0$	$\text{RC}(y, \exists x \cdot f, \text{FNPOS}, \text{FNNEG})$	$(z \lesssim b) \wedge$ $(x \lesssim a \cdot b)$	$\neg(z \lesssim b) \wedge$ $\neg(x \lesssim a \cdot b)$
$x \leftarrow a \cdot y$	$a < 0$	$\text{RC}(y, \exists x \cdot f, \text{FNPOS}, \text{FNNEG})$	$(z \lesssim b) \wedge$ $(a \cdot b \lesssim x)$	$\neg(z \lesssim b) \wedge$ $\neg(a \cdot b \lesssim x)$
$a_1 \cdot x +$ $a_2 \cdot y \lesssim k$	$a_1 > 0$	**let** $g = \text{RC}(x, f, \text{FNPOS}, \text{FNNEG})$**in**	$(z \lesssim b)$	$\neg(z \lesssim b) \wedge$ $(a_2 \cdot y \lesssim k - a_1 \cdot b)$
	$a_2 < 0$	$\text{RC}(y, g, \text{FNPOS}, \text{FNNEG})$	$(z \lesssim b) \wedge$ $(a_1 \cdot x \lesssim k - a_2 \cdot b)$	$\neg(z \lesssim b)$

Table 2. Abstract transformers; v is a symbolic constant bounded by $k_1 \lesssim v \lesssim k_2$, t_1 is $a_1 \cdot x + k_1$, and t_2 is $a_2 \cdot x + k_2$.

where $v_a \leq 1$, $1 < v_b \leq 5$, and $5 < v_c$.

Alternatively, lets apply the same transformer to LDD g shown in Fig. 1(d). Here, we assume $y \preceq x$, and A, B, and C are sub-diagrams that do not contain y. g is equivalent to:

$$(y \leq 2 \wedge A) \vee (2 < y \leq 6 \wedge B) \vee (6 < y \wedge C) \tag{11}$$

In each disjunct the value of y is known:

$$A' = ||x \leftarrow x + v_a||(A) \quad B' = ||x \leftarrow x + v_b||(B) \quad C' = ||x \leftarrow x + v_c||(C)\,,$$

where $v_a \leq 2 \cdot a$, $2 \cdot a < v_b \leq 6 \cdot a$, and $6 \cdot a < v_6$. The final result is: $\text{ITE}(y \leq 2, A', \text{ITE}(y \leq 6, B', C'))$. □

Finally, an abstract transformer for a linear assignment $x \leftarrow a_1 \cdot x_1 + \cdots + a_n \cdot x_n + v$ is computed as a sequence of simpler transformers. For example $||x \leftarrow a \cdot y + b \cdot z + k||_{\mathbb{BS}}$ is reduced to

$$||x \leftarrow k||_{\mathbb{BS}} \circ ||x \leftarrow x + a \cdot y||_{\mathbb{BS}} \circ ||x \leftarrow x + b \cdot z||_{\mathbb{BS}}\,. \tag{12}$$

Theorem 1. *Let c be an action of the form above, and f be the LDD corresponding to $\mathcal{BS} = \{\mathcal{B}_1, \ldots, \mathcal{B}_k\}$. Then, $||c||_{\mathbb{BS}}(f)$ is equivalent to $\cup_{i=1}^{k} ||c||_{\mathbb{B}}(\mathcal{B}_i)$, where $||\cdot||_{\mathbb{B}} : \mathbb{B}^n \to \mathbb{B}^n$ is the abstract transformer of* BOX.

Proof. (Sketch) For simplest transformers, the result follows trivially. The rest are equivalent to applying $||\cdot||_{\mathbb{B}}$ to each **1**-path of f. □

Join and Box Hull of Boxes. Figure 4 presents algorithms for two other operations on LDDs. $\text{BOXJOIN}(f, g)$ returns $f \uplus g$, while $\text{BOXHULL}(f, g)$ returns the box hull of f and g. BOXHULL invokes BOXJOIN as a subroutine. The complexity of both algorithms is in $O(|f| \cdot |g|)$.

4 Widening

In static analysis, widening is used to ensure that the analysis always terminates, even in the presence of infinite ascending chains in the underlying abstract domain [9]. Let $\hat{D} = (D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be an abstract domain. An operation $\nabla_d : D \times D \to D$ is a *widening* for $\hat{D}$ iff it satisfies two conditions: (1)

Require: $x \neq y$
1: **function** XPY(LDD f, var x, $\mathbb{Q}$ a, var y)
2: **if** $x \preceq y$ **then return** XPY1$(f, x, a, y, \text{TRUE})$
3: **else return** XPY2$(f, x, a, y, \text{TRUE})$

4: **function** XPY1(LDD f, var x, $\mathbb{Q}$ a, var y, cons c)
5: **if** $f = \mathbf{0} \vee f = \mathbf{1}$ **then return** f
6: $u \leftarrow C(f)$
7: **if** $x \preceq \text{VAR}(u)$ **then**
8: **if** $c = \text{TRUE}$ **then return** f
9: **return** $||x \leftarrow a \cdot y + v||_{\mathbb{BS}}(f)$, where $v \models c$
10: **if** $x \neq \text{VAR}(u)$ **then**
11: $t \leftarrow \text{XPY1}(f, x, a, y, c)$
12: $e \leftarrow \text{XPY1}(f, x, a, y, c)$
13: **return** ITE(u, t, e)
14: **Assert** u is $x \lesssim b$
15: $t \leftarrow ||x \leftarrow a \cdot y||_{\mathbb{BS}}(f|_u)$
16: $t \leftarrow ||x \leftarrow x + v||_{\mathbb{BS}}(t)$, where $v \models (c \wedge v \lesssim b)$
17: $e \leftarrow \text{XPY1}(f|_{\neg u}, x, a, y, \neg(u[x/v]))$
18: **return** OR(t, e)

19: **function** XPY2(LDD f, var x, $\mathbb{Q}$ a, var y, cons c)
20: **if** $f = \mathbf{0} \vee f = \mathbf{1}$ **then return** f
21: $u \leftarrow C(f)$
22: **if** $y \preceq \text{VAR}(u)$ **then**
23: **if** $c = \text{TRUE}$ **then return** EXIST(x, f)
24: **return** $||x \leftarrow x + v||_{\mathbb{BS}}(f)$, where $v \models c$
25: **if** $y \neq \text{VAR}(u)$ **then**
26: $t \leftarrow \text{XPY2}(f, x, a, y, c)$
27: $e \leftarrow \text{XPY2}(f, x, a, y, c)$
28: **return** ITE(u, t, e)
29: **Assert** u is $y \lesssim b$
30: $t \leftarrow ||x \leftarrow x + v||_{\mathbb{BS}}(f|_u)$, where $v \models (c \wedge v \lesssim a \cdot b)$
31: $e \leftarrow \text{XPY2}(f|_{\neg u}, x, a, y, \neg(v \lesssim a \cdot b))$
32: **return** ITE(u, t, e)

Fig. 3. Algorithm to compute abstract transfer function for $x \leftarrow x + a \cdot y$.

1: **function** BOXHULL (LDD f)
2: **if** $(f = \mathbf{0}) \vee (f = \mathbf{1})$ **then**
3: **return** f
4: $fh \leftarrow \text{BOXHULL}(H(f))$
5: $fl \leftarrow \text{BOXHULL}(L(f))$
6: **if** $L(f) = \mathbf{0}$ **then**
7: **return** ITE$(C(f), fh, \mathbf{0})$
8: **if** $H(f) = \mathbf{0}$ **then**
9: **return** ITE$(C(f), \mathbf{0}, fl))$
10: **return** BOXJOIN$(C(f) \wedge fh, fl)$

Require: f and g are singletons.
1: **function** BOXJOIN (LDD f, LDD g)
2: **if** $(f = g) \vee (f \in \{\mathbf{0}, \mathbf{1}\}) \vee (g \in \{\mathbf{0}, \mathbf{1}\})$ **then**
3: **return** OR(f, g)
4: **if** $C(g) \preceq C(f)$ **then return** BOXJOIN(g, f)
5: $u \leftarrow C(f)$
6: **if** $(f|_u = \mathbf{0}) \wedge (g|_u = \mathbf{0})$ **then**
7: **return** ITE$(u, \mathbf{0}, \text{BOXJOIN}(L(f), L(g)))$
8: **if** $(f|_{\neg u} = \mathbf{0}) \wedge (g|_{\neg u} = \mathbf{0})$ **then**
9: **return** ITE$(C(g), \text{BOXJOIN}(H(f), H(g)), \mathbf{0})$
10: **return** BOXJOIN$(f|_u = \mathbf{0}\ ?\ L(f) : H(f), g)$

Fig. 4. Algorithms to compute a box hull and a box join ($\uplus$) of LDDs.

over-approximation: $x \sqsubseteq y \Rightarrow (x \sqcup y) \sqsubseteq (x \nabla_d y)$, and (2) *stabilization*: for every increasing sequence $x_1 \sqsubseteq x_2 \cdots$, the (widening) sequence

$$y_1 = x_1\,, \qquad y_i = y_{i-1} \nabla_d (y_{i-1} \sqcup x_i)\,,\ \text{for } i > 1 \tag{13}$$

stabilizes, i.e., $\exists k \cdot y_k = y_{k+1}$.

In this section, we describe a widening for BOXES. We proceed in stages. First, we introduce a new domain construction $\text{STEP}(\hat{D})$, called *step (function)* construction, that lifts a domain $\hat{D}$ to (step) functions from $\mathbb{R}$ to D. Second, we give a procedure to lift a widening ∇_d of $\hat{D}$ to a widening ∇_s of $\text{STEP}(\hat{D})$. Finally, we show that n-dimensional BOXES are step constructions of $(n-1)$-dimensional BOXES and implement ∇_s on top of LDDs.

Step function construction. A function $f : \mathbb{R} \to D$ is a *step function* if it can be written as a finite combination of intervals. That is,

$$f(x) = (v_1 \sqcap f_1(x)) \sqcup \cdots \sqcup (v_n \sqcap f_n(x))\,, \tag{14}$$

where $v_i \in D$, and there exists a partitioning $F_1, \ldots, F_n$ of $\mathbb{R}$ by intervals such that $f_i(x) = \top$ if $x \in F_i$ and $f_i(x) = \bot$ otherwise. A step function f induces an

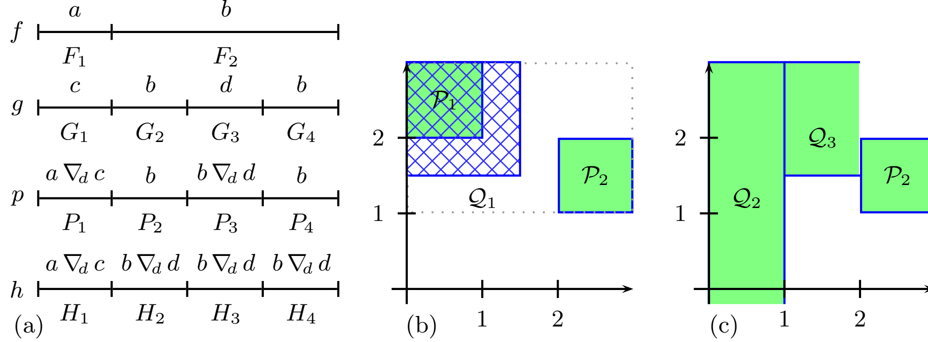

Fig. 5. Example of a widening.

equivalence relation $\equiv_f$ on R:

$$x \equiv_f y \Leftrightarrow \forall z \cdot ((x \leq z \leq y) \vee (y \leq z \leq x)) \Rightarrow f(x) = f(z)\,. \tag{15}$$

We write $[x]_f$ for the equivalence class of x w.r.t. $\equiv_f$. Note that the index of $\equiv_f$ is finite and the equivalence classes are naturally ordered: $[x] \leq [y] \Leftrightarrow x \leq y$. We assume the classes are enumerated, so the $\leq$-least equivalence class is *first*, the next one is *second*, etc. For step functions f, g, we write $\equiv_{f,g}$ for the relation:

$$x \equiv_{f,g} y \quad \Leftrightarrow \quad (x \equiv_f y) \wedge (x \equiv_g y)\,, \tag{16}$$

and $[x]_{f,g}$ for the corresponding equivalence class of x.

The set of all step functions from $\mathbb{R}$ to a domain $\hat{D}$, denoted $\mathbb{R} \rightarrow_s D$, forms an abstract domain $\text{STEP}(\hat{D}) \triangleq (\mathbb{R} \rightarrow_s D, \dot{\sqsubseteq}, \dot{\bot}, \dot{\top}, \dot{\sqcup}, \dot{\sqcap})$. The dot above an operator denotes pointwise extension, e.g., $f \,\dot{\sqsubseteq}\, g \triangleq \forall x \in \mathbb{R} \cdot f(x) \sqsubseteq g(x)$.

One-dimensional Boxes is $\text{STEP}(\{\text{TRUE}, \text{FALSE}\})$ – the step construction applied to the Boolean domain. Similarly, n-dimensional Boxes is a step construction of $(n-1)$-dimensional Boxes. Since the Boolean domain is finite – it's join and widening coincide. Thus, to get a widening for Boxes, we only need to show how to lift a widening from a base domain to its step construction. We use 1- and 2-dimensional Boxes for examples in the rest of this section.

Lifting widening to $\text{STEP}(\hat{D})$. Clearly, the pointwise extension $\dot{\nabla}_d$, of the widening ∇_d of $\hat{D}$, is not a widening of $\text{STEP}(\hat{D})$. As a counterexample, the divergent sequence $\{(0 \leq x \leq i)\}_{i=1}^{\infty}$ of Boxes values is it's own pointwise widening sequence. Let us examine this in more detail.

Example 3. Let f and g be step functions as illustrated in Fig. 5(a). Each function is shown as a partitioning of the number line with the value above and the name below the line, respectively. Thus, f has two partitions F_1 and F_2 with values a, and b, respectively. We assume that lower case letters represent distinct elements of some domain $\hat{D}$, ordered alphabetically. Note that $G_1 = F_1$ and F_2 is refined by G_2, G_3, and G_4. Consider $p = f \,\dot{\nabla}_d\, g$ as shown in Fig. 5(a) and compare to f. Clearly, p is on a divergent path. In it, partition F_2 is split into three parts, but both P_2 and P_4 have the same value as F_2. Thus, they can be refined again. A way to ensure convergence is to assign to P_2 and P_4 the value of P_3, as shown by h in Fig. 5(a). This is the intuition for our approach. □

In summary, pointwise widening $f \mathbin{\dot{\nabla}_d} g$ diverges whenever it refines a partition in f without updating its value. Thus, to guarantee convergence, we assign to the offending partition a value of its neighbor that refines the same partition of f. The formal definition is given below:

Definition 1. *Let $\hat{D} = (D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be an abstract domain with a widening ∇_d. Let $f, g \in \mathbb{R} \rightarrow_s D$ be two step functions s.t. $f \mathrel{\dot{\sqsubseteq}} g$, and $[y_1], \ldots, [y_n]$ be the equivalence classes of $\equiv_{f,g}$ enumerated by their natural order $\leq$. Then, the* step widening, *∇_s, for $\mathrm{STEP}(\hat{D})$ is defined as follows:*

$$(f \mathbin{\nabla_s} g)(x) \triangleq \bigsqcup_{i=1}^{n} (v_i \sqcap h_i(x)) \,, \tag{17}$$

where n is the index of $\equiv_{f,g}$, $h_i(x) = \top$ if $x \in [y_i]$ and $h_i(x) = \bot$ otherwise, and

$$v_i \triangleq \begin{cases} f(y_i) \;\; \nabla_d \;\; g(y_i) & \text{if } f(y_i) \neq g(y_i) \text{ or } [y_i]_{f,g} = [y_i]_f \\ f(y_i) \;\; \nabla_d \;\; g(y_{i+1}) & \text{else if } i < n \text{ and } [y_{i+1}]_{f,g} \subseteq [y_i]_f \\ f(y_i) \;\; \nabla_d \;\; g(y_{i-1}) & \text{otherwise} \end{cases} \tag{18}$$

Example 4. Consider two sets of boxes $\mathcal{BS}_1 = \{\mathcal{P}_1, \mathcal{P}_2\}$ and $\mathcal{BS}_2 = \{\mathcal{Q}_1, \mathcal{P}_2\}$ shown in Fig 5(b), where

$$\mathcal{P}_1 = (0 \leq x \leq 1) \wedge (2 \leq y \leq 3) \qquad \mathcal{P}_2 = (2 \leq x \leq 3) \wedge (1 \leq y \leq 2)$$
$$\mathcal{Q}_1 = (0 \leq x \leq 1.5) \wedge (1.5 \leq y \leq 3) \,.$$

The result of $\mathcal{BS}_1 \mathbin{\nabla_s} \mathcal{BS}_2$ is shown in Fig. 5(c). Note that even though $\mathcal{BS}_1$ and $\mathcal{BS}_2$ have the same box hull (shown by a doted frame), their widening is larger. This shows that widening makes it very hard to analytically compare difference in precision between Boxes and Box. □

Theorem 2. *The operator ∇_s defined in Def. 1 is a widening on $\mathrm{STEP}(\hat{D})$.*

Proof. Over-approximation. Let $h = f \mathbin{\nabla_s} g$. We show that for any $i \in [1, n]$, $h(y_i) \sqsupseteq g(y_i)$. Based on (18) there are 3 cases. In case 1, $h(y_i) = f(y_i) \mathbin{\nabla_d} g(y_i) \sqsupseteq g(y_i)$. In case 2, $[y_{i+1}]_{f,g} \subseteq [y_i]_f \Rightarrow f(y_i) = f(y_{i+1})$. Also, $f \mathrel{\dot{\sqsubseteq}} g \Rightarrow f(y_i) = f(y_{i+1}) \sqsubseteq g(y_{i+1})$. Finally, $h(y_i) = f(y_i) \mathbin{\nabla_d} g(y_{i+1}) \sqsupseteq f(y_i) = g(y_i)$. In case 3, we have $[y_{i-1}]_{f,g} \subseteq [y_i]_f$, from which the results follows as in case 2..

Stabilization. Let $f : \mathbb{R} \rightarrow_s D$ be a step function, and $\{f_i\}_{i=1}^{\infty}$ be an infinite sequence defined as follows:

$$f_1 \triangleq f \,, \qquad f_i \triangleq f_{i-1} \mathbin{\nabla_s} g_i \,, \text{ for } i > 1 \,, \tag{19}$$

where $\{g_i\}_{i=1}^{\infty}$ is any sequence of step function such that $f_{i-1} \mathrel{\dot{\sqsubseteq}} g_i$. We show that the sequence stabilizes, i.e., for some k, $f_k \doteq f_{k+1}$.

We write $\equiv_i$ for $\equiv_{f_i}$ and $[x]_i$ for $[x]_{f_i}$. For $i \geq 1$, let $\equiv_{\leq i}$ be the equivalence relation: $x \equiv_{\leq i} y \Leftrightarrow \forall 1 \leq j \leq i \cdot x \equiv_j y$, and $[\cdot]_{\leq i}$ be its equivalence classes.

Let $T = (V, E)$ be a tree with $V \triangleq (0, \mathbb{R}) \cup \{(i, [x]_{\leq i}) \mid i \geq 1, x \in \mathbb{R}\}$ and

$$E \triangleq \{((0, \mathbb{R}), (1, [x]_{\leq 1})) \mid x \in \mathbb{R}\} \cup$$
$$\{((i, [x]_{\leq i}), (j, [x]_{\leq j})) \mid j > i \wedge f_j(x) \neq f_i(x) \wedge \forall i < k < j \cdot f_i(x) = f_k(x)\} \,.$$

That is, T is a tree of refined equivalence classes with edges corresponding to differences in f_i's. Let T_i be the subtree of T restricted to the nodes (j, X) where $j \leq i$. Then the leaves of T_i correspond to f_i, i.e., $(i, [x]_{\leq i})$ is a leaf iff $f_i(x) \neq f_{i-1}(x)$. T is finitely-branching because all edges from an equivalence class at level i only go to equivalence classes at some other level j, and there are finitely many classes at each level. Formally,

$$((i, [x]_{\leq i}), (j, [x]_{\leq j})) \in E \wedge ((i, [y]_{\leq i}), (k, [y]_{\leq k})) \in E \wedge ([x]_{\leq i} = [y]_{\leq i}) \Rightarrow j = k$$

which follows from cases 2 and 3 of (18).

Suppose $\{f_i\}_{i=1}^{\infty}$ is not stable. Then, T is infinite. By König's lemma, there is an infinite path $\pi = (0, \mathbb{R}), (1, [x]_{\leq 1}), (i_2, [x]_{\leq i_2}), \ldots$ in T. By Def. 1, for any consecutive nodes $(i_k, [x]_{\leq i_k})$ and $(i_{k+1}, [x]_{\leq i_{k+1}})$ on π, there is a $d \in D$, s.t. $f_{i_{k+1}}(x) = f_{i_k}(x) \nabla_d d$. This contradicts that ∇_d is a widening. □

Widening for Boxes. Recall that 1-dimensional Boxes are step functions into $\{\text{TRUE}, \text{FALSE}\}$. Thus, ∇_s where $\nabla_d = \vee$ is a widening for them. Widening, ∇_{bs}^n for n-dimensional Boxes is defined recursively by letting ∇_{bs}^n be ∇_s parameterized by $\nabla_d = \nabla_{bs}^{n-1}$. We write ∇_{bs} when the dimension is clear or irrelevant.

Theorem 3. *The operation ∇_{bs} is a widening for* Boxes.

We now describe our implementation of ∇_{bs} with LDDs. It is not hard to show that the last two cases of (18) are equivalent to v_{i+1} and v_{i-1}, respectively. That is, the value of the partition i is either a widening of the corresponding partitions of the arguments, or the value of an adjacent partition. Thus, if we assume that the step functions are given as a linked list of partitions, ∇_s is computable by a recursive traversal of this list. Conveniently, this is how Boxes are represented by LDDs. For example, in Fig. 1(c) the low edges form the linked list of partitions of dimension x. However, there are no back-edges, and it is hard to access the value of the "previous" partition. We overcome this problem by sending the value of the "current" partition down the recursion chain.

Our algorithm WR implementing $f \nabla_{bs}^n g$ is shown in Fig. 6. The inputs are LDDs f and g, a variable x bound to dimension n, and an LDD h representing the value of "previous" partition or **nil**. When the dimension of f and g is not known apriori, $f \nabla_{bs} g$ is implemented by WR$(f, g, x, \textbf{nil})$, where x is the $\preceq$-least variable of f and g, and h is **nil** since the algorithm starts at the first partition. This is done by Widen shown in Fig. 6. The WR proceeds exactly as the simple recursive algorithm described above. Comments indicate which lines correspond to the three cases of (18).

Theorem 4. *Algorithm* Widen *implements ∇_{bs} in time $O(|f| \cdot |g|)$.*

5 Boxes and Finite Powerset of Box.

The finite powerset of Box [1, 2], which we call PowerBox, is the main alternative to Boxes as a refinement of Box. An advantage of a finite powerset construction is its applicability to any base domain. However, this makes it hard

```
Require: LEQ(f, g)
 1: function WIDEN (LDD f, LDD g)
 2:     if (f = 0) ∨ (f = g) ∨ (g = 1) then
 3:     else    return g
 4:     if C(f) ⪯ C(g) then return WR(f, g, VAR(C(f)), nil)
 5:     else    return WR(f, g, VAR(C(g)), nil)
 6: function WR  (LDD f, LDD g, VAR x, LDD h)
 7:     if f = g then
 8:         if (VAR(C(f)) ≠ x) ∧ (h ≠ nil) then return h                ▷ (case 3)
 9:         return g                                                    ▷ (case 1)
10:     if (VAR(C(f)) ≠ x) ∧ (VAR(C(g)) ≠ x) then return WIDEN(f, g)   ▷ (case 1)
11:     if C(f) ⪯ C(g) then v ← C(f)
12:     else   v ← C(g)
13:     t ← WR(f|_v, g|_v, x, nil)
14:     e ← WR(f|_¬v, g|_¬v, x, nil)
15:     if v = C(f) then
16:         if (g|_v = f|_v) ∧ (h ≠ nil) then
17:             return ITE(v, h, e)                                     ▷ (case 3)
18:         else
19:             return ITE(v, t, e)                                     ▷ (case 1)
20:     if g|_v = f|_v then return e                                    ▷ (case 2)
21:     return ITE(v, t, WR(f|_¬v, g|_¬v, x, t))                        ▷ (case 1)
```

Fig. 6. Widening for BOXES.

(if not impossible) to leverage the power of domain-specific data structures. In contrast, our BOXES implementation is based on a specific data-structure – LDDs – but does not extend to other base domains. In the rest of the section, we compare the two domains analytically. Results of extensive empirical evaluation are presented in Section 6.

Finite powerset construction. Let $\hat{D} = (D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be an abstract domain. For any $S \subseteq D$, let $\Omega(S)$ be the set of the $\sqsubseteq$-maximal elements of S, and $S \subseteq_{fn} D$ mean that S is a *finite* subset of D. The *finite powerset* domain over $\hat{D}$ is:

$$\hat{D}_P = (\mathcal{P}^{\Omega}_{fn}(\hat{D}), \sqsubseteq_P, \emptyset, \Omega(D), \sqcup_P, \sqcap_P)\,, \tag{20}$$

where $\mathcal{P}^{\Omega}_{fn}(\hat{D}) \triangleq \{S \subseteq_{fn} D \mid \Omega(S) = S\}$, $S_1 \sqsubseteq_P S_2$ iff $\forall d_1 \in S_1 \cdot \exists d_2 \in S_2 \cdot d_1 \sqsubseteq d_2$, $S_1 \sqcup_P S_2 \triangleq \Omega(S_1 \cup S_2)$, and $S_1 \sqcap_P S_2 \triangleq \Omega(\{s_1 \sqcap s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2\})$.

Comparing Representation. BOXES and POWERBOX differ in their element representation. Let φ be a Boolean formula over $\mathbb{IVQ}$. POWERBOX represents φ by its (unshared) DNF, while BOXES represents φ by its BDD. Thus, there exists a φ whose POWERBOX representation is exponentially bigger than its BOXES representation, and vice versa. Of course, deciding between a DNF or a BDD representation of a Boolean formula is a long-standing open problem.

Comparing Basic Operations. The $\subseteq$ operation of BOXES is exact, while the corresponding $\sqsubseteq_P$ operation of POWERBOX is not. For example, let $S_1 = \{0 \leq x < 2\}$ and $S_2 = \{0 \leq x < 1, 1 \leq x < 2\}$ be elements of POWERBOX. Then, $(S_1 \not\sqsubseteq_P S_2)$, but $S_1 \subseteq S_2$. The complexity of the operations in both domains is polynomial in the sizes of the representations of their arguments. Complexities of the LDD operations used by BOXES are shown in Table 1. For

PowerBox, most expensive operations are Ω and meet ($\sqcap_P$). Ω is quadratic and has no analogue in Boxes. $\sqcap_P$ has the same complexity, relative to the size of its arguments, as AND. The complexity of join ($\sqcup_P$) is similar to OR, but is more efficient if irreducibility of the result is not required.

Comparing Widening. Bagnara et al. [2] suggest three schemes to extend a widening from the base domain (in this case, Box) to the finite powerset (i.e., Boxes): k-bounded, connector, and certificate-based. Our widening does not fit any of these categories. It does not bound the number of disjuncts apriori, and hence is not k-bounded. It does not compute certificates, or a box hull of its arguments, and hence is not certificate-based. It is close in spirit to connector-widening, but is not itself based on widening of a base-domain. Thus, our widening is not easily comparable to any of the suggestions of [2]. Note that extending a PowerBox widening to Boxes is difficult. One possibility is to convert between a Boxes and a PowerBox value, apply PowerBox widening, and convert the value back. But, this involves an exponential blowup – number of paths in an LDD is exponential in its size. The alternative is to adapt PowerBox widening algorithm to work directly on an LDD. This is non-trivial.

In summary, it is not obvious which of Boxes and PowerBox is superior. In Section 6, we present empirical evidence that suggests that in practice the Boxes domain does scale better.

6 Experiments

To evaluate Boxes, we implemented a simple abstract interpreter, IRA, on top of the LLVM compiler infrastructure [14]. For every function of a given program, IRA computes invariants over all SSA variables at all loop heads using a given abstract domain. We compared four abstract domains: LDD Boxes – the domain described here; LDD Box – Box implemented with LDDs using BoxJoin and the standard widening instead of OR and Widen, respectively; PPL Box – Box implemented by `Rational_Box` class of PPL [3]; and, PPL Boxes – PowerBox implemented by `Pointset_Powerset<Rational_Box>` of PPL. For LDD-based domains, we used dynamic variable ordering.

The benchmark. We applied IRA to 25 open source programs, including `mplayer`, `CUDD`, and `make`, with over 16K functions in total. All experiments were ran on a 2.8GHz quad-core Pentium machine. Running time and memory for each function was limited to 1 minute and 512MB, respectively. Here, we report on the 5,727 functions which at least one domain required 2 or more seconds to analyse. The first two columns of Table 3(a) summarize key characteristics of the benchmark: on average there are 238 variables and 7 loop heads per function. The last 3 columns summarize the size of the invariants computed as either LDDs, number of paths in a LDD, or number of elements in a PowerBox value. Note that the large standard deviations indicate that the benchmark was quite heterogeneous. Overall, Table 3(a) shows that our analysis was non-trivial.

The results. Our experimental results are summarized in Table 3(b). The first two columns show the percentage of (the 5,727) functions analyzed successfully, and

	Vars	Loop	DD	Path	Box
MIN	9	0	1	0	1
MAX	9,052	241	87,692	2.15E09	7,296
AVG	238	7	1,011	2.46E08	802
STDEV	492	12	3,754	5.75E08	761
MEDIAN	97	3	149	5,810	589

(a)

Domain	%S	T(m)	%B	%I	%∇
LDD Box	99.8	4	77	23	0
PPL Box	96.1	117	86	14	0
LDD Boxes	87.9	118	61	38	1
PPL Boxes	14.2	201	95	1	3

(b)

Table 3. (a) Benchmark summary: Vars – # of variables; Loop – # of loop heads; Invariant Sizes: DD – # of nodes in a DD, Path – # of paths, Box – # of elements in a PPL Boxes value. (b) Summary of the experimental results: %S – % Solved, T – total time, %B – % time in basic ops, %I – % time in image, %∇ – time in widen.

the time taken, respectively. The time includes only the cost of abstract domain operations, and only counts the successful cases for the corresponding domain. Each Box domain solved over 90% of the cases. Surprisingly, LDD Box was significantly faster. We conjecture that this is due to the large number of tracked variables in our benchmark. The size of an LDD Box value is proportional to the number of bounded variables (dimensions), whereas that size of PPL Box value is proportional to the, much larger, number of tracked variables.

Our LDD Boxes domain did quite well, solving close to 90% of the cases. PPL Boxes domain did not scale at all: solved under 20% and took almost double the time of LDD Boxes.

The last three columns of Table 3(b) break down the time between the basic ($\sqsubseteq, \sqcap, \sqcup$) domain operations (Basic), image computation (Image), and widening (Widen). Again, both Box domains perform similarly, with Basic being the most expensive, while Widen is negligible. For LDD Boxes, the time is divided more evenly between Basic and Image, with a non-negligible Widen. For PPL Boxes, the time is dominated by Basic, and Widen is also significant.

Fig. 7(a) compares LDD Box (the fastest and least precise analysis) and LDD Boxes. Clearly, additional expressivity of LDD Boxes costs additional (often, several orders of magnitude) complexity. Fig. 7(b) compares PPL Boxes and LDD Boxes (only successful cases for PPL are shown). Here, LDD Boxes is several orders of magnitude faster.

In order to understand whether the increased expressivity of LDD Boxes yields more precise results, and to evaluate the effectiveness of our widening, we measured the number of times widening points are visited during the analysis. We conjecture that a very aggressive (and, thus, imprecise) widening results in a very quick convergence and, hence, few repeated applications of widening. Fig. 7(c) compares LDD Box and LDD Boxes. In all but 23 cases, analysis with LDD Boxes visits widening points as many (and often significantly more) times than LDD Box. In the remaining 23 cases, LDD Boxes converges faster – often, before the widening is ever applied[2] – but to a more precise invariant.

Fig. 7(d) compares LDD Boxes and PPL Boxes (on the cases where PPL Boxes was successful). In most cases, both domains converge after similar number of iterations. In general, the convergence rate is within a factor of 2. We

[2] in IRA, we delay widening until the 3rd iteration of a loop.

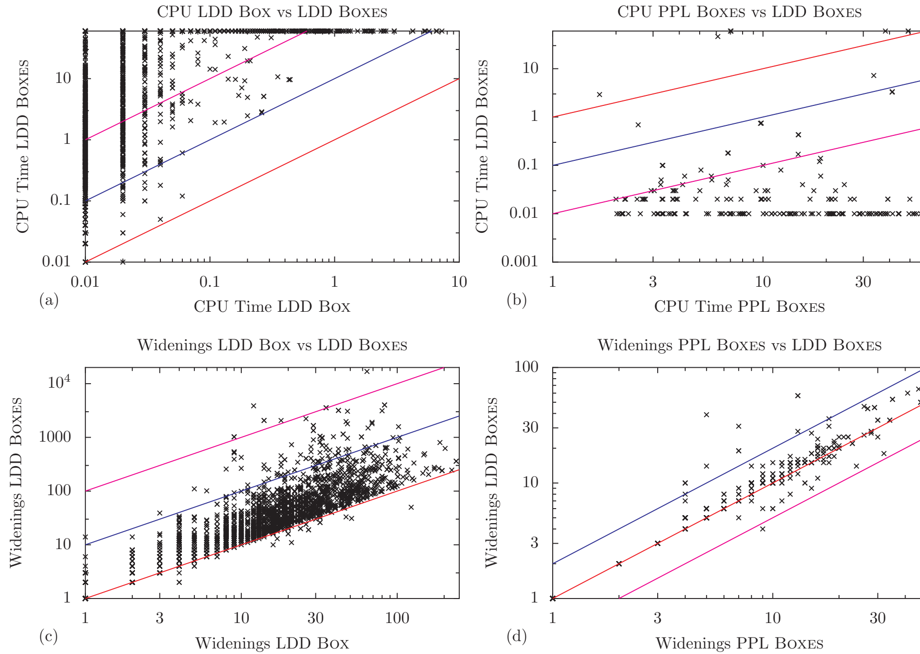

Fig. 7. Running time: (a) LDD Box vs. LDD Boxes; (b) PPL Boxes vs. LDD Boxes. Number of widenings: (c) LDD Box vs. LDD Boxes; (d) PPL Boxes vs. LDD Boxes.

conjecture that this indicates that our widening is similar in its precision to the finite powerset widening used by PPL Boxes.

Overall, our evaluation indicates that LDDs provide a solid backbone for implementing Box and its disjunctive refinements. LDD Box is competitive with PPL Box, and scales much better as the number of variables increases. The performance degradation when moving from Box to its disjunctive refinement is milder for LDDs than for PPL. Finally, LDD Boxes performs better than PPL Boxes, while maintaining a similar precision level.

7 Conclusion

In this paper, we presented Boxes, a symbolic abstract domain that weds disjunctive refinement of Box with BDDs. Boxes is implemented on top of LDDs, an extension of BDDs to linear arithmetic. We present a novel widening algorithm for Boxes that is different from known schemes for implementing widening for disjunctive refinements. Empirical evaluation indicates that Boxes is more scalable than existing implementations of the finite powerset of Box.

An area of future work is to study applicability and scalability of Boxes in a practical software verification setting. In particular, Boxes offers a promising platform for combining model-checking and abstract interpretation as in [12]. Another direction is to extend the approach to weakly-relational domains. The main challenge is developing an effective and efficient widening.

Acknowledgements. We thank Ofer Strichman for numerous insightful discussions.

References

1. Bagnara, R.: "A Hierarchy of Constraint Systems for Data-Flow Analysis of Constraint Logic-Based Languages". Science of Computer Programming 30(1-2), 119–155 (1988)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: "Widening Operators for Powerset Domains". International Journal on Software Tools for Technology Transfer (STTT) 8(4), 449–466 (August 2006)
3. Bagnara, R., Hill, P.M., Zaffanella, E.: "The Parma Polyhedra Library: Towards A Complete Set of Numerical Abstractions for The Analysis and Verification of Hardware and Software Systems". Science of Computer Programming 72(1-2), 3–21 (2008)
4. Beyer, D., Henzienger, T.A., Theoduloz, G.: Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In: CAV'07. LNCS, vol. 4590, pp. 504–518 (2007)
5. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers (TC) 35(8), 677–691 (1986)
6. Chaki, S., Gurfinkel, A., Strichman, O.: "Decision Diagrams for Linear Arithmetic". In: FMCAD'09 (2009)
7. Cousot, P., Cousot, R.: "Static Determination of Dynamic Properties of Programs". In: Proceedings of the 2nd Internaitional Symposium on Programming (ISOP'76). pp. 106–130 (1976)
8. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Proceedings of the 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '79). pp. 269–282 (1979)
9. Cousot, P., Cousot, R.: "Abstract Interpretation Frameworks". Journal of Logic and Computation (JLC) 2(4), 511–547 (1992)
10. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97). LNCS, vol. 1254, pp. 72–83 (1997)
11. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: "Automatically Refining Abstract Interpretations". In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08). LNCS, vol. 4963, pp. 443–458 (2008)
12. Gurfinkel, A., Chaki, S.: "Combining Predicate and Numeric Abstraction for Software Model Checking". In: FMCAD'08. pp. 127–135 (2008)
13. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: "Clock Difference Diagrams". Nord. J. Comput. 6(3), 271–298 (1999)
14. Lattner, C., Adve, V.: "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: CGO'04 (2004)
15. Mauborgne, L., Rival, X.: "Trace Partitioning in Abstract Interpretation Based Static Analyzers". In: Proceedings of the 14th European Symposium On Programming (ESOP '05). LNCS, vol. 3444, pp. 5–20 (2005)
16. Møller, J.B., Lichtenberg, J., Andersen, H.R., Hulgaard, H.: Difference Decision Diagrams. In: Proceedings of the 13th International Workshop on Computer Science Logic (CSL '99). LNCS, vol. 1683, pp. 111–125 (1999)
17. Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., Gupta, A.: "Static Analysis in Disjunctive Numerical Domains". In: Proceedings of the 13th International Static Analysis Symposium (SAS '06). LNCS, vol. 4134, pp. 3–17 (2006)

18. Somenzi, F.: CU Decision Diagram Package, vlsi.colorado.edu/~fabio/CUDD
19. Strehl, K., Thiele, L.: “Symbolic Model Checking of Process Networks Using Interval Diagram Techniques”. In: ICCAD’98. pp. 686–692 (1998)
20. Wang, F.: “Efficient Data Structure for Fully Symbolic Verification of Real-Time Software Systems”. In: Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS ’99). LNCS, vol. 1785, pp. 157–171 (2000)