Disjunctive Interval Analysis

Graeme Gange[1 [0000−0002−1354−431X]], Jorge A. Navas[2 [0000−0002−0516−1167]], Peter Schachte[3 [0000−0001−5959−3769]], Harald Søndergaard[3 [0000−0002−2352−1883]], and Peter J. Stuckey[1 [0000−0003−2186−0459]]

[1] Faculty of Information Technology, Monash University, Melbourne, Australia
[2] SRI International, California, USA
[3] School of Computing and Information Systems, The University of Melbourne, Australia

Abstract. We revisit disjunctive interval analysis based on the Boxes abstract domain. We propose the use of what we call range decision diagrams (RDDs) to implement Boxes, and we provide algorithms for the necessary RDD operations. RDDs tend to be more compact than the linear decision diagrams (LDDs) that have traditionally been used for Boxes. Representing information more directly, RDDs also allow for the implementation of more accurate abstract operations. This comes at no cost in terms of analysis efficiency, whether LDDs utilise dynamic variable ordering or not. RDD and LDD implementations are available in the Crab analyzer, and our experiments confirm that RDDs are well suited for disjunctive interval analysis.

Keywords: Abstract interpretation · Boxes · Decision diagrams · Integer abstract domains

1 Introduction

The perennial challenge in the design of program analyses is to find an appropriate balance between precision and efficiency. A natural way to improve precision of analysis is to design an abstract domain that supports path-sensitive analysis, that is, allows for a degree of *disjunctive* information to be expressed. However, abstract domains are rarely closed under disjunction, as the cost of disjunctive closure usually leads to prohibitively expensive analysis.

In this paper we are concerned with the analysis of integer manipulating procedural programs. The abstract domain studied here is the `Boxes` domain [11], applied to $\mathbb{Z}$, the set of integers.[4] Assume we are given n integer variables $v_1, \ldots, v_n$. A *bounds constraint* takes one of the forms $v_i \leq k$ or $v_i \geq k$, where k is an integer constant. An *integer box* is any set $B \subseteq \mathbb{Z}^n$ that can be expressed as a (possibly empty) conjunction of bounds constraints. The `Boxes` domain consists of any set $S \subseteq \mathbb{Z}^n$ which can be written as a finite union $\bigcup_{i=1}^{j} B_i$, such

[4] With a little additional effort, the approach extends to rationals and floating point numbers.

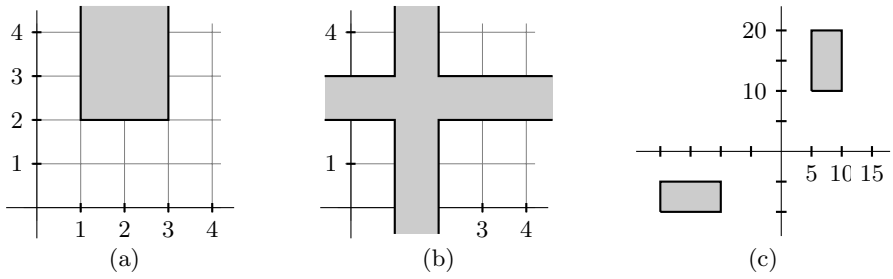

Fig. 1. Examples of `Boxes` elements.

that $B_1, \ldots, B_j$ are integer boxes. Some examples are given in Fig. 1: (a) shows an integer box, namely $x \in [1, 4) \wedge y \in [2, \infty)$; (b) shows the `Boxes` element $x \in [1, 3) \vee y \in [2, 4)$; and (c) shows the element

$$(x \in [-20, -9) \wedge y \in [-10, -4)) \vee (x \in [5, 11) \wedge y \in [10, 21))$$

Elements of `Boxes` are generally non-convex sets. The domain is closed under both (finite) intersection and union, as well as under complement. This clearly sets `Boxes` apart from more commonly studied *relational* abstract domains, such as zones [20], octagons [21], and convex polyhedra [7]. Note that, while it is a non-relational abstract domain, `Boxes` can still express conditional constraints, such as $x \geq 2 \Rightarrow (y \geq 0 \wedge y \leq 4)$.

To implement `Boxes`, Gurfinkel and Chaki [11] proposed the use of linear decision diagrams (LDDs) [3]. The inspiration for LDDs came from the better known binary decision diagrams (BDDs) [2]. But in an LDD, a decision node (a non-terminal node) no longer corresponds to a Boolean variable; instead it holds a primitive constraint in some theory. As with BDDs, non-terminal nodes in LDDs always have fan-out 2. LDDs can express any Boolean combination of primitive constraints, and in the `Boxes` case, "primitive constraint" means "bounds constraint" (only), such as $x \leq 42$. LDDs can utilise *sharing* of sub-structures, which reduces memory requirements and allows for a canonical representation.

However, LDDs (with primitive constraints taken from some theory T) come with a disadvantage: inability to precisely analyse expressions that fall outside the theory. This happens since many operations rely on a T-solver. With LDD-boxes, much precision is lost in the context of non-linear[5] expressions, as Example 1 will show. Moreover, with LDDs, many abstract operations are expensive, because of the need to preserve a node order that depends not only on variable ordering, but also on logical consequence. Gurfinkel and Chaki [11] define abstract operations in terms of constraint substitution, a relatively heavyweight approach which, again, limits transformers to the supported arithmetic fragment.

[5] Gurfinkel and Chaki [11] consider a restricted programming language with only linear expressions and guards.

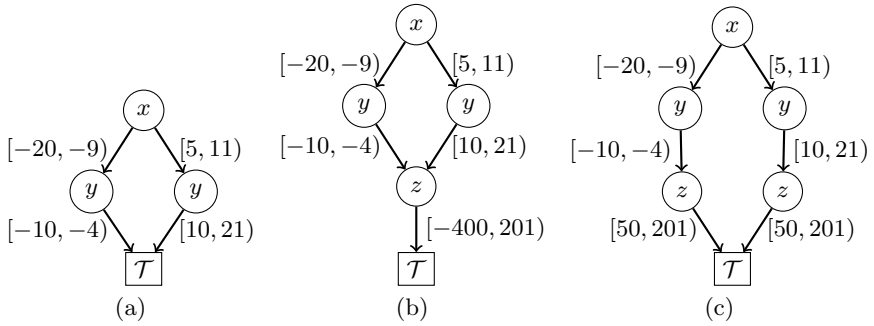

Fig. 2. (a) A `Boxes` state with disjunctive information, and after applying `z := x * y` (b) using interval approximation, and (c) using a precise transformer.

Our aim is to improve `Boxes` analysis through the use of a dedicated data structure that can support precise analysis, including analysis of programs involving non-linear operations. As it turns out, our proposed representation of disjunctive intervals also tends to speed up the analysis of *linear* programs, mainly because it allows *bounds-propagation* to take the place of calls to a theory solver. We use what we call "range decision diagrams" (RDDs), a variant of multi-valued decision diagrams (MDDs) [24]. Non-terminal nodes in these structures can have varying fan-out, each with each edge corresponding to a range of values for some variable. RDDs also generalize BDDs, but in a different manner to LDDs.

In this paper, we use RDDs in the context of integer predicates. We note that LDDs have a wider scope: they can support more complex theories. But for the Boxes application, in which LDDs use monadic predicates only, RDDs have exactly the same expressiveness as LDDs.

We define RDDs formally in the next section. Until then, we ask the reader to rely on intuition and the diagrams shown in Fig. 2.

Example 1. To appreciate the limitations flowing from reliance on a theory solver, consider the `Boxes` set from Fig. 1(c). Its RDD[6] is shown in Fig. 2(a). Given the statement `z := x * y`, the LDD approach must collapse the information to its non-disjunctive interval form $-20 \leq x < 11 \wedge -10 \leq y < 21$. Since the theory solver used does not understand multiplication, the analysis engine must collapse the representation, in order to calculate bounds on the multiplication. The result is the `Boxes` element shown in RDD form in Fig. 2(b). A precise transformer, on the other hand, computes the possible values of `z` on each branch in the diagram, resulting in the much more precise Fig. 2(c) with a better lower bound for `z`. □

In summary, the benefit of using RDDs for `Boxes` analysis is improved expressiveness *and* efficiency. We introduce RDDs in Section 2, and Section 3 provides

[6] It is straightforward to translate an RDD to an LDD over bounds constraints, and vice versa.

algorithms for the abstract operations. We report on an experimental evaluation in Section 4. Related work is discussed in Section 5, and Section 6 concludes.

2 Range decision diagrams

By *range* we mean an integer interval of form $[i, j)$ (that is, $\{k \in \mathbb{Z} \mid i \leq k < j\}$), $[i, \infty)$, or $(-\infty, i)$. A set S of ranges is *fitting* iff every pair $I_1, I_2 \in S$ is disjoint ($I_1 \cap I_2 = \emptyset$) and $\bigcup S = \mathbb{Z}$. We assume a given finite set *Var* of variables. The set of *range decision diagrams*, or RDDs, is defined recursively, as follows.

- $\mathcal{F}$ and $\mathcal{T}$ are RDDs.
- Let $M = \{(I_1, r_1), (I_2, r_2), \ldots (I_n, r_n)\}, n > 1$ be a set of pairs whose first components are ranges and whose second components are RDDs. If the set $\{I \mid (I, r) \in M\}$ is fitting, and $v \in \mathit{Var}$, then $\langle v, M\rangle$ is an RDD.

We may sometimes relax the fitting requirement to allow $\bigcup S \subseteq \mathbb{Z}$, in which case each missing range is understood to be paired with $\mathcal{F}$.

The meaning of an RDD r is a predicate $[\![r]\!]$, defined as follows:

$$\begin{aligned} [\![\mathcal{F}]\!] &= \mathit{false} \\ [\![\mathcal{T}]\!] &= \mathit{true} \\ [\![\langle v, M\rangle]\!] &= \bigvee\{[\![r]\!] \wedge v \in I \mid (I, r) \in M\} \end{aligned}$$

We can view the RDD as a directed acyclic graph in the obvious manner: $\mathcal{T}$ and $\mathcal{F}$ are sinks. An RDD $\langle v, M\rangle$ has a node labelled v as its root, and for each $(I, r) \in M$, an edge (with label I) from v to the root of r. We draw graphs so that arrows point downwards. We will assume a (total) precedence order $\prec$ on *Var* and construct RDDs where the path variables earlier in the ordering always appear above variables later in the ordering (this condition may be temporarily violated in algorithms).

In algorithms, it is sometimes useful to utilize a different view of a fitting RDD. We may write a non-sink RDD r as $\langle v, [r_0, k_1, r_1, \ldots, k_n, r_n]\rangle$. Here $r_0, \ldots, r_n$ are RDDs and $k_1, \ldots, k_n$ are the *split points*, with $k_1 < k_2 < \cdots < k_n$. The intention is that r_i is the co-factor of r with respect to v, over the interval $[k_i, k_{i+1})$, implicitly taking $k_0 = -\infty$ and $k_{n+1} = \infty$.[7] For a fixed variable ordering this representation is canonical, provided we ensure $r_i \neq r_{i+1}$ for all i.

The two views are for presentation only; each is faithful to the data structure used in implementation. To translate between them, we use two functions **ser** and **des** (for "serialize" and "deserialize"). **des** takes an RDD representation $\langle v, [r_0, k_1, r_1, \ldots, k_n, r_n]\rangle$ in split-point form and turns it into the deserialised $\langle v, \{((-\infty, k_1), r'_0), ([k_1, k_2), r'_1), \ldots, ([k_n, \infty), r'_n)\}\rangle$, where r'_i is the deserialised form of r_i (base cases $\mathcal{F}$ and $\mathcal{T}$ are left unchanged).

The function **ser** is its inverse, defined only for fitting RDDs; **ser**, in particular, adds edges from v to $\mathcal{F}$ for any "missing" intervals.

[7] This view explains our tendency to use notation like $[3, 4)$ for what is obviously a (closed) singleton integer interval.

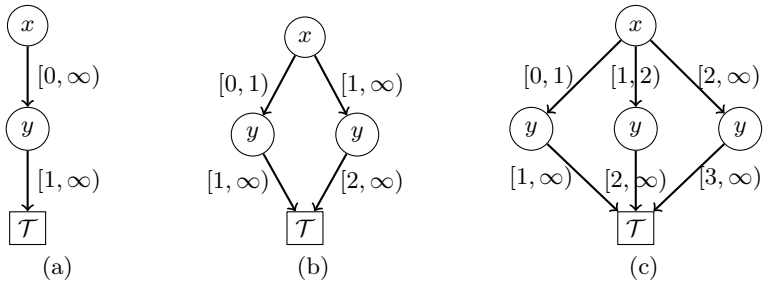

Fig. 3. A sequence of better approximations of $p(x, y) \equiv 0 \leq x \wedge x < y$.

In diagrams we omit the node $\mathcal{F}$ and its incident edges. The (serial form) RDD $\langle x, [\mathcal{F}, 0, \langle y, [\mathcal{F}, 1, \mathcal{T}]\rangle, 1, \langle y, [\mathcal{F}, 2, \mathcal{T}]\rangle]\rangle$ is shown in Fig. 3(b). Deserializing the RDD representation yields

$$\left\langle x, \quad \left\{ \begin{array}{l} ((-\infty, 0), \mathcal{F}), \\ ((0, 1), \langle y, \{((-\infty, 1), \mathcal{F}), ([1, \infty), \mathcal{T})\rangle), \\ ([1, \infty), \langle y, \{((-\infty, 2), \mathcal{F}), ([2, \infty), \mathcal{T})\rangle) \end{array} \right\} \right\rangle$$

Applying a Boolean operator to RDDs is similar to how that is done for binary or (classical) multi-valued decision diagrams: we apply the operator pointwise on the co-factors of the nodes, and collect the result into a new node. However, unlike usual BDD or MDD operations, the intervals for the children of each node may not coincide. Thus we must first introduce additional separators that *refine* the generated intervals, enabling pointwise application of operators (we exemplify this in Section 3.2).

3 Implementing `Boxes` with RDDs

We now describe how to implement the `Boxes` domain with RDDs. Note again that `Boxes` forms a Boolean lattice, but not a complete one: there are necessarily predicates for which there can be no *best* RDD representation, instead admitting infinite chains of better and better RDD approximations. Consider the predicate $p(x, y) \equiv 0 \leq x \wedge x < y$. Fig. 3(a) shows an over-approximation, $x \geq 0 \wedge y \geq 1$. We can separate the case $x = 0$ from $x \geq 1$ and obtain a marginally more precise over-approximation which excludes the infeasible valuation $\{x \mapsto 1, y \mapsto 1\}$, see Fig. 3(b). Indeed, we can split the domain of x arbitrarily often, in each step obtaining a slightly more precise RDD. Similarly there are infinite ascending chains of ever closer under-approximations.

Hence, for some arithmetic operations, we cannot hope to define optimal abstract transformers. In the context of RDDs there is, however, a natural criterion for precision, akin to the concept of interval consistency [1, 4] from the constraint programming literature. Consider a constraint c over set X of variables. A *domain* D maps each variable $x \in X$ to a set of possible values for x.

Here we assume that $D(x)$ must be an interval and we say that D is *interval consistent* for c, iff, for each variable $x \in X$ with $D(x) = [\ell, u]$, each of the valuations $\{x \mapsto \ell\}$ and $\{x \mapsto u\}$ can be extended to a valuation (over all of X) satisfying $D \wedge c$. In this context, the role of *bounds-propagation* is to narrow variable domains as far as possible without breaking interval consistency.

For RDDs we need to refine the concept of consistency slightly. Note that each path ρ through the RDD induces a domain D_ρ.

Definition 1. ***All-paths interval consistency.*** *RDD r is all-paths interval consistent for constraint c iff, for each path ρ in r, the induced domain D_ρ is interval consistent.*

Our abstract operations strive to maintain all-paths interval consistency. Loosely this means we produce the strongest information that can possibly be inferred without resorting to speculative introduction of new disjunctive splits. Only our algorithm for inequality fails to maintain all-paths interval consistency—Example 5 will show a case of this.

3.1 Lattice operations

The standard lattice operations are fairly straightforward. $\sqsubseteq, \sqcap, \sqcup$ coincide with the standard Boolean operators $\rightarrow, \wedge, \vee$, and can all be implemented *pointwise*: for $u \bowtie v$, we scan the children of u and v in order, applying $\bowtie$ recursively to children with overlapping intervals, and rebuild the results into a new node. As with the corresponding BDD/MDDs operations, these can be performed in $O(|u||v|)$ time. All lattice operations are optimally precise.

3.2 Variable hoisting

An operation which will be useful for operators defined below is *hoisting* (or *interchanging*) variables. This is necessary when we would like to construct a function representing $\langle x, [r_0, k_1, \ldots, k_n, r_n]\rangle$, but where the root of r_i is earlier than x in the precedence order (so just building the node would be malformed).

For this definition, we restrict ourselves to the case where, for all r_i, every variable except (possibly) the root y are before x, so we merely need to interchange the decisions for x and y. For RDDs, this is straightforward and detailed in Fig. 4. We sort all the split points at the second level of the tree, removing duplicates and create a set $\mathcal{I}$ of covering intervals for the variable y. We fill in the matrix *Cofac* of cofactors, based on the intervals for y and x. We then construct a new node for each x interval using *Cofac*. Finally we construct a new root linked appropriately to these nodes.

Example 2. Fig. 5 shows an almost-ordered RDD, where top levels x_2 and x_1 must be transposed. Fig. 6 shows the matrix of cofactors $Cofac[I, I']$ generated by the algorithm. Fig. 7 shows the RDD that results from hoisting. □

function HOIST-VAR($y, \langle x, M\rangle$)
  $E = \text{SORT\_NODUP}(\bigcup\{\{l, u\} \mid (\_, \langle x, M'\rangle) \in M, ([l, u), \_) \in M'\})$
  $\mathcal{I} = [(E[i], E[i+1]) \mid i \in 1 \,..\, |E| - 1]$
  for $(I, \langle x, M'\rangle) \in M$ **do**
    for $I' \in \mathcal{I}$ **do**
      let (I^s, r') be the element in M' where $I^s \supseteq I'$
      $Cofac[I, I'] \leftarrow r'$
  for $I' \in \mathcal{I}$ **do**
    $r_{I'} \leftarrow \langle y, \{(I, Cofac[I, I']) \mid (I, r) \in M\}\rangle$
  return $\langle x, \{(I', r_{I'}) \mid I' \in \mathcal{I}\}\rangle$

Fig. 4. Variable hoisting: How to construct a node rooted at y, representing the decision structure $\langle x, M\rangle$.

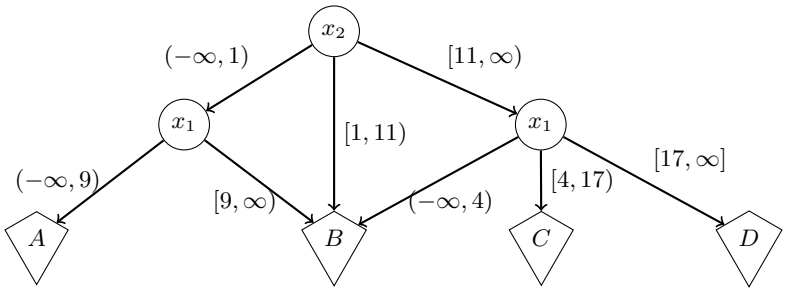

Fig. 5. A mis-ordered RDD r.

3.3 Arithmetic operators

Gurfinkel and Chaki [11] implemented the arithmetic operators for `Boxes` using constraint substitution over LDDs. This has some drawbacks, relying as it does on having a theory solver for a sufficiently expressive theory of arithmetic. Instead, we construct arithmetic abstract transformers that operate directly on the RDD representations. Each transformer is formulated as a recursive traversal of the RDD, carrying with it the projection of the operation along the path to the current node. This makes implementing operators more involved, but it avoids the need for a (frequently expensive) theory solver and offers more flexibility in terms of expressiveness and the level of precision we can support. As with conventional BDD operations, we save previously computed results in a cache to avoid repeated work; nevertheless worst-case complexity of the arithmetic operators is exponential in the size of the RDD, as each path through the RDD may yield a different projected arithmetic expression.

Interval Computation. A basic step for many algorithms will be computing the interval of possible values for an expression E, given RDD r. The pseudo-code in Fig. 8 shows how to do this. We walk the RDD, substituting each variable as

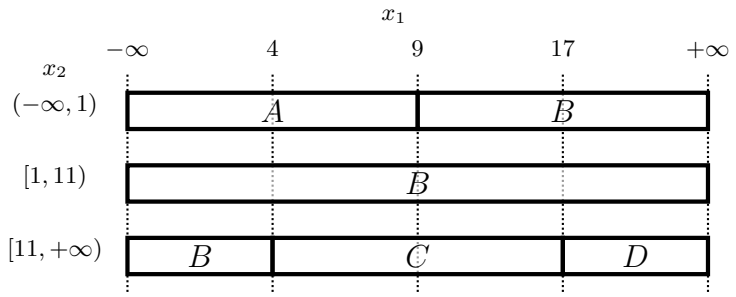

Fig. 6. Matrix of cofactors used to interchange x_1 and x_2

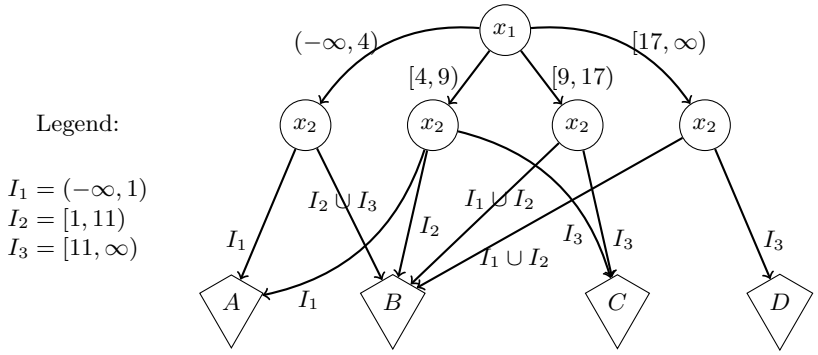

Fig. 7. The correctly ordered RDD r' after hoisting r from Fig. 5.

reached, by its possible intervals, collecting their union. Once all variables in E have been replaced by intervals, we use the function INTERVAL(E) to return the smallest interval containing the possible values of E. In the figure we use $\perp$ to denote the empty interval. MIN-VAR(E) produces the variable (from E) with the earliest precedence ($\emptyset$ if E is variable-free).

Example 3. Consider computing $\mathsf{eval}(\texttt{x*y}, r)$, the possible interval values of the expression `x*y` given the RDD r from Fig. 2(a). The initial call generates a call $\mathsf{eval}([-20, 9)\texttt{*y}, r')$ where r' is the left child of the root. This in turn generates a call $\mathsf{eval}([-20, 9)*[-10, -4), \mathcal{T})$ which returns $[50, 201)$. The initial call generates a second call $\mathsf{eval}([5, 11)\texttt{*y}, r'')$ where r'' is the right child of the root. This in turn generates a call $\mathsf{eval}([5, 11)*[10, 21), \mathcal{T})$ which returns $[50, 201)$. The initial call thus returns the union which is again $[50, 201)$. □

Assignments. The abstract transformers for assignments all operate in a similar manner. Let $E[y \mapsto I]$ denote the (interval-valued) expression obtained by partially evaluating E assuming $y = I$. To apply `z := E` to $r = \langle y, E \rangle$, we iterate over each non-$\mathcal{F}$ child (I, r') of r, recursively applying `z :=` $E[y \mapsto I]$, and then rebuild the resulting node.

```
function EVAL(r, E)
  match r with
    case ⊥ ⇒ return ⊥
    case ⊤ ⇒ return INTERVAL(E[v ↦ (−∞, +∞) | v ∈ vars(E)])
    case ⟨x, M⟩ ⇒
      match MIN-VAR(E) with
        case ∅ ⇒
          return INTERVAL(E)
        case y where y ≺ x ⇒
          return EVAL(r, E[y ↦ (−∞, +∞)])
        case y where y ≻ x ⇒
          return ⋃{EVAL(r′, E) | (I, r′) ∈ M, r′ ≠ ⊥}
        case x ⇒
          return ⋃{EVAL(r′, E[x ↦ I]) | (I, r′) ∈ M, r′ ≠ ⊥}
      end
  end
```

Fig. 8. Evaluating the interval approximation of E on RDD r.

```
function EVAL-SPLIT(r, E)
  match r with
    case ⊥ ⇒ return ⊥
    case ⊤ ⇒
      I ← INTERVAL(E[v ↦ (−∞, +∞) | v ∈ vars(E))})
      return ⟨ϵ, {(I, ⊤)}⟩
    case ⟨x, M⟩ ⇒
      match MIN-VAR(E) with
        case ∅ ⇒
          return (ϵ, {(INTERVAL(E), r)}))
        case y where y ≺ x ⇒
          return EVAL-SPLIT(r, E[y ↦ (−∞, +∞))
        case y where y ≽ x ⇒
          let M′ = {(I, EVAL-SPLIT(r′, E[y ↦ I])) | (I, r′) ∈ M}
          return HOIST-VAR(ϵ, ⟨x, M′⟩)
      end
  end
```

Fig. 9. Constructing a node, rooted at variable ϵ, encoding the possible valuations of E on RDD r. We use HOIST-VAR to percolate the valuations of E up to the root.

Once we reach (or skip) variable `z`, we have two options. We can compute the interval I containing the possible values of the residual E, and apply $\mathtt{z} := I$ at the current node. Alternatively, we can construct the resulting RDD *as if* `z` were below all variables in E, then percolate `z` back up to the correct location. The latter is the analogue of the substitution-based approach used by Chaki, Gurfinkel and Strichman [3]; the former is less precise, but reduces the growth of the RDD. In practice the less precise version loses too much precision. Pseudo-code for the latter approach is shown in Fig. 9.

```
function APPLY(r, z := E)
  match r with
    case F ⇒ return F
    case T ⇒
      I ← INTERVAL(E[v ↦ (−∞, +∞) | v ∈ vars(E))])
      return ⟨ϵ, {(I, T)}⟩
    case ⟨x, C⟩ ⇒
      if x ⪯ z then
        r ← EVAL-SPLIT(r, E)                    ▷ Constructs a node rooted at ϵ
        if x = z then
          r ← FORGET(r, x)
        return r[ϵ ↦ z]
      else
        match MIN-VAR(E) with
          case ∅ ⇒                                          ▷ E fully evaluated
            return ⟨z, {(INTERVAL(E), r)}⟩
          case y where y ≺ x ⇒                          ▷ y unconstrained in r
            E′ ← E[y ↦ (−∞, +∞)]
            C′ ← {(I, APPLY(r′, z := E′)) | (I, r′) ∈ C}
          case y where y ≻ x ⇒                          ▷ E independent of x
            C′ ← {(I, APPLY(r′, z := E)) | (I, r′) ∈ C}
          case x ⇒
            C′ ← {(I, APPLY(r′, z := E[y↦I])) | (I, r′) ∈ C}
        end
        return ⟨x, C′⟩
  end
```

Fig. 10. Abstract version of `z := E` given RDD r, for some arithmetic expression E.

The algorithm in Fig. 10 is instantiated for the cases where E is a linear expression ($\sum_i c_i x_i + I$), n-ary product ($\prod_i x_i \times I$) or (binary) division (x/y). In each case, we specialize the generic algorithm slightly:

- For linear expressions, we terminate if some variable in E is skipped (in which case E, and therefore z, is unconstrained).
- For products, we handle negative, zero, and positive ranges separately, and apply early termination when some variable is zero or unbounded.
- For division, we again perform case-splits on sign, and terminate early on zero and unbounded inputs.

Example 4. Consider the RDD in Fig. 11(a). APPLY(r, `z := E`) constructs the RDD shown in Fig. 11(b). For x_2 the split points are the extreme values of $3x_1 + 8x_3 + 10$, along the possible paths, that is, 77, 93, 113, and 241; hence the x_2 fan-out of three. In general, for each path ρ of r, APPLY(r, `z := E`) constructs an all-paths interval consistent RDD, introducing an edge for z that is tight with respect to the projection of E along ρ. For linear expressions, APPLY constructs the smallest such RDD (though not for multiplication and division, owing to our speculative splits on sign). □

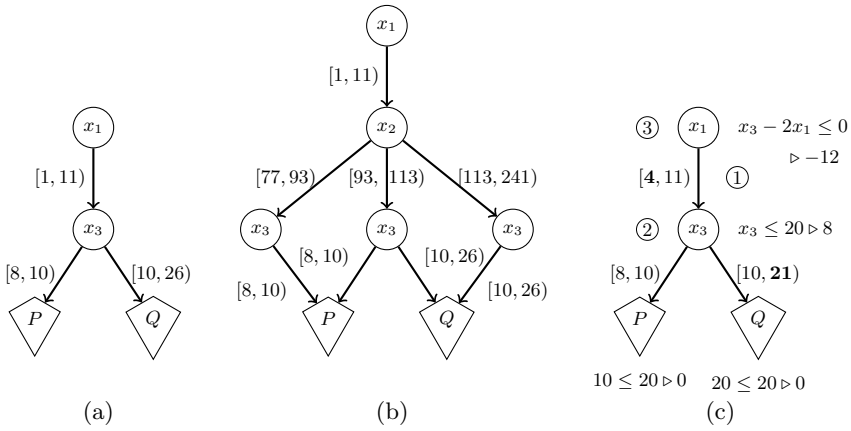

Fig. 11. With full splitting, evaluating x_2 := $3x_1+8x_3+10$ on (a) yields (b). Applying $x_3-2x_1 \leq 0$ on (a) yields (c), with tightened bounds shown in bold: ① the downwards phase finds upper bound 20 for x_3; ② the upwards phase notes the lower bound 8 and ③ uses this information ($2x_1 \geq 8$) to improve the lower bound on x_1.

Arithmetic Constraints. The abstract transformer for an arithmetic constraint $\mathsf{apply}(r, E \leq k)$ is very similar to that for assignment. Again we traverse the RDD depth-first, passing the projection of our constraint onto the current valuation, then reconstruct the result on the way back up. But during reconstruction, we also return the projection of E onto the variable *beneath* the current node which we use to perform bounds-propagation on edge ranges. The pseudo-code is given in Fig. 12. Each call returns the resulting RDD, together with an upper bound to be applied to the RDD above. At each node involved in the expression (lines 33–36), we recursively apply the projected constraint along each outgoing edge, and use the returned upper bound to prune the bounds of the edge.

Example 5. Fig. 11(c) shows the effect of applying $x_3 - 2x_1 \leq 0$ to the RDD from Fig. 11(a). Each node is annotated with $C \triangleright \ell$, where C is the projected constraint we constructed in the downward phase, and ℓ the lower-bound which was returned upwards. □

Unlike assignment, APPLY$(r, E \leq k)$ does not in general yield an all-paths interval consistent RDD—it (implicitly) introduces splits on the way down, but TRIM-LEQ only uses the returned lower bound on E for pruning, rather than introducing new splits. A slightly more precise RDD for the case considered in Example 5 would have an additional split point for x_1, namely 5. That would bar some spurious solutions, such as $Q \wedge x_1 = 4 \wedge x_3 = 13$. An algorithm that maintains all-paths interval consistency also in the case of inequalities is perfectly possible, but in practice we find the cost of doing so outweighs any advantage.

Bounds extraction/box hull. The *box hull* operation takes boxes $\mathcal{B}$ and produces a *stick* mapping each variable to the smallest interval covering its feasible valu-

```
1: function TRIM-LEQ(cx + E ≤ k, I, r)
2:     u_r, r' ← APPLY(r, E ≤ k − c min(I))
3:     I_r ← I ∩ (−∞, u_r/c)
4:     if I_r = ∅ then
5:         return ∞, (∅, F)
6:     else
7:         return l_r − c min(I), (I_r, r')
8: function APPLY(r, E ≤ k)
9:     match r, E with
10:        case F, _ ⇒ return ∞, F
11:        case _, 0 ⇒                                   ▷ E fully evaluated
12:            if k < 0 then
13:                return ∞, F
14:            else
15:                return 0, r
16:        case T, ax ⇒              ▷ One unconstrained variable, can be bounded
17:            return −∞, ⟨x, {((−∞, k/a), r)}⟩
18:        case T, ax + E ⇒      ▷ At least two unconstrained, cannot infer anything
19:            return −∞, T
20:        case ⟨x, M⟩, ay + E' where y ≺ x ⇒      ▷ y currently unconstrained in r,
21:                                                  ▷ check if E' is bounded
22:            I_E' ← EVAL(r, E')
23:            y_ub ← ⌈(k − min I_E')/a⌉
24:            if y_ub is finite then
25:                return −∞, ⟨y, {((−∞, y_ub), r)}⟩
26:            else
27:                return −∞, r
28:        case ⟨x, M⟩, ay + E' where y ≻ x ⇒              ▷ E independent of x
29:            R ← {l_c, (I, c') | (I, c) ∈ M and l_c, c' = APPLY(c, E ≤ k)}
30:            l_r ← min{l_c | _, (l_c, _) ∈ R}
31:            r' ← ⟨x, {(I, c') | _, (I, c') ∈ R}⟩
32:            return l_r, r'
33:        case ⟨x, M⟩, ax + E' ⇒
34:            R ← {TRIM-LEQ(ax + E ≤ k, I, r') | (I, r') ∈ M}
35:            l_c ← min{l_c | l_c, _ ∈ R}
36:            return l_c, ⟨x, {(I_c, c') | _, (I_c, c') ∈ R}⟩
37:    end
```

Fig. 12. Applying a constraint $\sum_i c_i x_i \leq k$ on RDD r. For simplicity, we restrict consideration to positive c_i.

ations. The box hull algorithm of Gurfinkel and Chaki [11] proceeds by merging all feasible children of the root node (using $\sqcup$), then recursively building the hull of the single remaining child. However, while the final result is compact, the successive joins may cause an exponential growth in the intermediate results. Instead, we construct the box hull in two stages: we first traverse the RDD to

```
function LOWER-BOUNDS(⟨x, M⟩)
    let [d_0, k_1, ..., k_n, d_n] = ser(M)
    i_0 ← if (d_0 = F) then 1 else 0
    B ← LOWER-BOUNDS(d_{i_0})
    for i ∈ i_0 + 1 ... n do
        if d_i ≠ F then
            B ← LOWER-BOUNDS-R(d_i, B)
    seen(⟨x, M⟩) ← true
    if d_0 = F then
        return [(x, k_1)|B]
    else
        return B
```

```
function LOWER-BOUNDS-R(⟨r, M⟩, B)
    let [d_0, k_1, ..., k_n, d_n] = ser(M)
    match B with
        case [] ⇒
            return []
        case [(x', k')|B'] where x ≻ x' ⇒
            return LOWER-BOUNDS-R(⟨x, M⟩, B')
        case [(x', k')|B'] ⇒
            if seen(⟨x, M⟩) then
                return [(x', k')|B']
            seen(⟨x, M⟩) ← true
            for i ∈ 0 ... n do
                if d_i ≠ F then
                    B' ← LOWER-BOUNDS-R(d_i, B')
            if x = x' ∧ d_0 = F then
                return [(x', min(k', k_1))|B']
            else
                return B'
end
```

Fig. 13. Extracting lower bounds of all variables from RDD $\langle x, M\rangle$. Upper bound extraction is similar. The *seen* markers are used to avoid re-processing a previously explored node.

collect lower and upper bounds of each variable, then construct the hull RDD directly.

The algorithm for extracting bounds is given in Fig. 13. On the leftmost feasible path in r, it constructs an ordered list of variables having finite lower bounds. On the remaining paths, it updates the current set of bounds, removing any variables that are skipped or are unbounded. This operation takes time linear in the size of the input RDD. Unfortunately, the operation is not cached across calls (as we update bounds information in-place, rather than merge bounds from subgraphs).

3.4 Widening

The last operator we need is a widening, $\triangledown$, to ensure convergence. A standard approach to constructing a sound widening is based on the notion of *stability*: we decompose the domain into a finite set of individual properties, and any *unstable* properties—those which are not preserved (under entailment) from the previous iteration—are weakened (usually discarded) to eliminate infinite ascending chains.

If we were working with pure BDDs or classical MDDs (with finite domains), the join would be sufficient, as there are only finitely many cofactors, and each cofactor can increase at most once. But with RDDs, a difficulty arises when the *position* of a split changes.

Example 6. Consider the following sequence of iterates:

$$\begin{aligned} r_0 &\equiv \langle x, \{((-\infty, 0), \mathcal{T}),\ ([0, +\infty), \mathcal{F})\}\rangle \\ r_1 &\equiv \langle x, \{((-\infty, 1), \mathcal{T}),\ ([1, +\infty), \mathcal{F})\}\rangle \\ r_i &\equiv \langle x, \{((-\infty, i), \mathcal{T}),\ ([i, +\infty), \mathcal{F})\}\rangle \end{aligned}$$

If we apply a 'widening' pointwise, we get the chain:

$$\begin{aligned} w_0 = r_0 &\equiv \langle x, \{((-\infty, 0), \mathcal{T}),\ ([0, +\infty), \mathcal{F})\}\rangle \\ w_1 = w_0 \nabla r_1 &\equiv \langle x, \{((-\infty, 0), \mathcal{T}),\ ([0, 1), \mathcal{T}\nabla\mathcal{F}),\ ([1, +\infty), \mathcal{F})\}\rangle = r_1 \\ w_2 = w_1 \nabla r_2 &\equiv \langle x, \{((-\infty, 1), \mathcal{T}),\ ([1, 2), \mathcal{T}\nabla\mathcal{F}),\ ([2, +\infty), \mathcal{F})\}\rangle = r_2 \\ w_i = w_{i-1} \nabla r_i &\equiv \langle x, \{((-\infty, i-1), \mathcal{T}),\ ([i-1, i), \mathcal{T}\nabla\mathcal{F}),\ ([i, +\infty), \mathcal{F})\}\rangle = r_i \end{aligned}$$

Looking at the result for any one fixed value of x, there are no infinite chains. But the overall widening sequence is nevertheless an infinite ascending chain. □

The problem just exemplified arises when the *target* of a child remains stable, but its *range* shrinks. The widening of Gurfinkel and Chaki [11] handles the situation by detecting when this has occurred, and instead taking the value of (one of) its unstable siblings. For Example 6, we notice that the transition to $\mathcal{F}$ was unchanged but its range decreased, so we take the neighbouring $\mathcal{T}$ value instead.

We can adapt the same widening strategy for the RDD representation. Fig. 14 gives the detailed widening algorithm; WIDEN(u, v) is the function that calculates $u \nabla v$. As with other lattice operations, we walk over both operands in lock-step. But as ∇ is asymmetric, the main case of WIDEN(u, v) (lines 34–39) iterates over the edges of u, and, for each edge, calls WIDEN-EDGE to compute the (possibly refined) widening of the corresponding ranges of v. WIDEN-EDGE walks over the edges of v applying widening pointwise (lines 12–19), substituting stable children with their left unstable sibling (line 16). The *first* edge of course has no such sibling, so the algorithm starts (lines 5–8) by finding the first unstable successor if one exists (if not, the entire edge was stable, so it can be returned)[8].

The argument for termination of the widening algorithm is the following.

1. The operator is increasing by construction.
2. Note that (a) it is not possible to have an infinite ascending chain of refined split positions and (b) for any one (fixed) split position there is no infinite ascending chain. Namely, each co-factor leads to a finite ascending chain: whenever a new split location is introduced, the co-factors on both sides increase strictly.

[8] As presented, this differs slightly from [11] in that we select the *left* sibling as replacement in WIDEN-EDGE, where [11] selects the right. We also implemented a right-biased variant, and differences are minimal.

```
1: function WIDEN-EDGE(x, M_x, [y_0, k_1, ..., k_m, y_m])
2:     d_y ← y_0
3:     i ← 1
4:     d ← x▽d_y
5:     while d = x ∧ i ≤ m ∧ k_i < M_x do                ▷ Find first unstable child d
6:         d_y ← y_i
7:         i ← i + 1
8:         d ← x▽d_y
9:     if i > m ∨ k_m ≥ M_x then                          ▷ Only one child, no subdivision
10:        return [d], [d_y, k_i, y_i, ..., k_m, y_m]
11:    E_out ← [d]          ▷ Replace any stable children with d to ensure convergence
12:    while i ≤ m ∧ k_m < M_x do
13:        d_y ← y_i
14:        d' ← x▽d_y
15:        if d' = x then
16:            d' ← d
17:        E_out ← E_out ++ [k_i, d']
18:        d ← d'
19:        i ← i + 1
20:    return (E_out, [d_y, k_i, y_i, ..., k_m, y_m])
21: function WIDEN(u, v)
22:    match (u, v) with
23:        case (⊤, _) ⇒ return ⊤
24:        case (⊥, v) ⇒ return v
25:        case (⟨x, M⟩, v) ⇒
26:            let [r_0, k_1, r_1, ..., k_n, r_n] = ser(M)
27:            let ⟨x', E_v⟩ = v
28:            if x ≺ x' then
29:                return ⟨x, [r_0▽v, k_1, r_1▽v, ..., k_n, r_n▽v]⟩
30:            else if x' ≺ x then
31:                (E_out, _) ← WIDEN-EDGE(u, +∞, ser(E_v))
32:                return ⟨x', des(E_out)⟩
33:            else
34:                E_out ← []
35:                for i ∈ 1...n do
36:                    (E'_i, E_v) ← WIDEN-EDGE(r_{i−1}, k_i, ser(E_v))
37:                    E_out ← E_out ++ E'_i
38:                (E_n, _) ← WIDEN-EDGE(r_n, +∞, ser(E_v))
39:                return ⟨x, des(E_out ++ E_n)⟩
40:    end
```

Fig. 14. Widening on `Boxes`, adapted to the RDD representation.

4 Experimental evaluation

We have implemented all the RDD operations required by the `Boxes` domain, following the algorithms described in this paper. Apart from the node representation, the architecture of the underlying RDD package is relatively standard: a

unique table mapping node structures to canonical representations, and a cache to record the results of recent computations. The implementation is available at `https://bitbucket.org/gkgange/mdd-boxes`.

The evaluation that we now report on has had two aims: First, to compare scalability of `rdd-boxes` with the existing `ldd-boxes` implementation. Second, to compare precision of the two implementations in order to assess the impact of the more precise abstract transformers provided by `rdd-boxes`.

4.1 Experimental setup

For the evaluation, we integrated our RDD-based implementation of `Boxes` into the `Crab`[9] abstract interpreter, which already provides LDD-based `Boxes`. We evaluated both implementations on a collection of C programs using `Clam`[10], an LLVM frontend for `Crab`.

The programs used for testing were taken from the 2019 Software Verification Competition. We chose 190 programs from the `ControlFlow` and `Loops` categories. These programs are already annotated with assertions. `ControlFlow` is a challenging set of programs for abstract interpretation because the instances generally require path-sensitive reasoning. However, they do not require a deep memory analysis. They constitute a good test suite for `Boxes` because this abstract domain is expressive enough to prove the majority of assertions. Nevertheless, both `rdd-boxes` and `ldd-boxes` needed to use a widening delay of 15 to produce precise results. The second selected category, `Loops`, is quite different from `ControlFlow`: the programs are much smaller and neither memory analysis nor path sensitivity is required. However, the majority of programs have many nested loops or require complex (although typically linear) loop invariants. We used `Loops` to evaluate the effect of the widening operations in both implementations.

All experiments have been carried out on a 2.1GHz AMD Opteron processor 6172 with 32 cores and 64GB on a Ubuntu 18.04 Linux machine. From those 32 cores, we used 16 cores to run multiple instances of Crab in parallel, but each instance was executed sequentially.

For `rdd-boxes`, we statically order variables according to the order in which they first appear in the program. For `ldd-boxes`, we used two orderings: the same static ordering used by `rdd-boxes`, and the dynamic ordering used by Gurfinkel and Chaki [11], based on the Cudd library's `CUDD_REORDER_GROUP_SIFT` option.

It is important to note that the `ldd-boxes` library[11] does not provide support for arbitrary linear or tree expressions as other libraries such as Apron [14] do. `ldd-boxes` only supports assignments of the form $x \leftarrow (k_1 \times y) + [k_2, k_3]$ and linear constraints of the form $(k_1 \times x)$ relop k_2 where k_1, k_2, k_3 are integers, relop are the standard relational operators $\{\leq, \geq, <, >, =, \neq\}$, and x and y are variables.

[9] Available at `https://github.com/seahorn/crab`.
[10] Available at `https://github.com/seahorn/clam`.
[11] Available at `https://github.com/seahorn/ldd`.

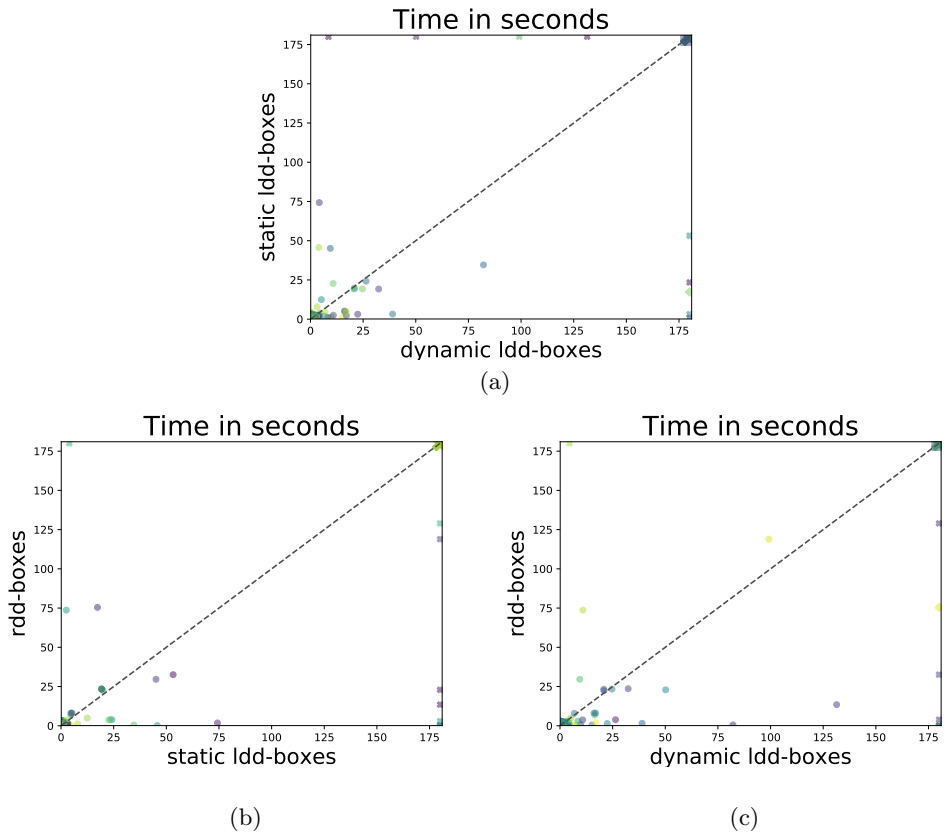

Fig. 15. Three graphs to compare analysis time in seconds on 190 Control Flow and Loops programs with timeout of 180 seconds and memory limit of 8GB. The marker ● represents domains finished before exhausting resources, ✖ represents timeout, and ◆ memory-out. The size of a marker reflects the number of scatter points at that location.

For all other cases, the abstract interpreter `Crab` simplifies arbitrary expressions by extracting interval constraints until the simplified expression can be supported by the `ldd-boxes` library. The current Crab implementation safely ignores arithmetic operations with non-unit coefficients. For our benchmarks, LLVM did not generate any instruction with non-unit coefficients.

4.2 Performance

Fig. 15 compares efficiency of the three implementations on the set of SV-COMP benchmarks. The top part (a) shows the result of static ordering plotted against that of dynamic ordering for `ldd-boxes`. Different orderings can have a significant impact on performance, and there is no clear winner. In the bottom part, we compare `rdd-boxes` with `ldd-boxes` using static (b) and dynamic (c) ordering.

Table 1. Comparing the precision of LDD-boxes with static ordering, LDD-boxes with dynamic reordering, and two variants of RDD-boxes on SV-COMP programs for which all the domains terminated with timeout of 180 seconds and memory limit of 8GB.

Implementation	Programs	Total Assertions	Proven Assertions
static `ldd-boxes`	168	628	497
dynamic `ldd-boxes`	168	628	494
linear `rdd-boxes`	168	628	504
`rdd-boxes`	168	628	510

Independently of variable ordering, the `rdd-boxes` analysis tends to be faster. With a time limit of 180 seconds, `rdd-boxes` timed out for 7 programs, while static and dynamic `ldd-boxes` timed out for 13 and 12 programs, respectively.

To understand the causes of the performance differences, we manually inspected several programs. We hypothesize that the main reason why `ldd-boxes` and `rdd-boxes` differ significantly in performance is the above-mentioned process of interval extraction that takes place in the Crab analyzer. This interval extraction for `ldd-boxes` is quite expensive, which sometimes makes `rdd-boxes` significantly faster. On the other hand, it may equally make `rdd-boxes` slower since `rdd-boxes` performs optimal transfer functions (which may introduce more disjunctive splits), while `ldd-boxes` does not, owing to its limited API.

4.3 Precision

Table 1 compares the precision of the two `ldd-boxes` implementations, together with two variants of `rdd-boxes` for the SV-COMP test suite: the version used for our performance evaluation which precisely supports both linear and non-linear operations, and a variant that only supports linear operations (linear `rdd-boxes`) and relies on the Crab analyzer to *linearize* [22] non-linear expressions.

Column Programs is the total number of programs for which all three implementations finished without exhausting resources. Total Assertions is the total number of assertions checked by the analysis, and Proven Assertions is the total number of proven assertions.

The variable ordering can affect the precision of widening. We believe this can explain the differences between the `ldd-boxes` implementations and linear `rdd-boxes`. Note that linear `rdd-boxes` is more precise than the `ldd-boxes` implementations, because of a more precise modelling of linear operations. The precise modelling of non-linear operations in `rdd-boxes` further improves the number of proven assertions (510 vs 504)—a relatively small, but, to an end user potentially significant, gain.

As a baseline comparison, using a traditional convex interval analysis, we were able to prove 392 of the 628 assertions, that is, 62%. Disjunctive information is, as expected, critical in the context of program verification.

5 Related work

Early examples of disjunctive analysis were primarily found in work on abstract interpretation of *declarative* programming languages [15, 17] (the "Pos" domain for groundness dependency analysis of logic programs [17] is a rare example of an abstract domain for *relational* analysis that is closed under conjunction *and* disjunction). The abstract domain refinement known as "disjunctive completion" was introduced by Cousot and Cousot [5]. Giacobazzi and Ranzato [9] explored the fact that different domain "bases" may induce identical disjunctive completions, leading to the concept of a least (or most abstract) disjunctive basis [9].

Decision diagrams for disjunctive domains are of interest also in symbolic model checking, for example for analysis of timed automata. In that context, Strehl and Thiele [25] have made use of "function graphs" which, in their "reduced" form, correspond to RDDs. Strehl and Thiele capture transition relations through an additional concept of "interval mapping functions". Implementation details are somewhat sparse, but it appears that only simple (linear) transformations are considered. Join and widening are not of relevance in model checking applications.

Clock decision diagrams (CDDs) [16] generalise Strehl and Thiele's function graphs, by allowing nodes that represent either single (clock) variables X or *differences* $X - Y$. That way, not only bounds can be expressed; it is possible to use CDDs to express difference constraints such as $X = Y$ and $X - Z \in [0, 3]$, so that CDDs support a limited form of *relational* analysis. CDDs are not canonical, and the abstract operations that would be required for program analysis (as opposed to the clock operations considered in [16]) would seem quite difficult to implement. For a program analysis tool to achieve the added expressiveness that is offered by CDDs, it would probably make better sense to use a product domain that incorporates a standard implementation of Zones [20]. Other BDD variants have been proposed that have constraints as nodes, such as difference decision diagrams (DDDs) [23] and EQ-BDDs [10].

Dominant sources of imprecision in classical abstract interpretation are *join points*—program points that represent a confluence of flow from several points. If an analysis is able to distinguish program states based on different execution traces that lead to a given program point, then imprecise joins can be avoided or delayed, resulting in greater precision of analysis. Approaches to introduction of such (limited) disjunctive information include loop unrolling and (flow- or value-based) trace partitioning [12, 19]. The idea that decision tree structures can represent control flow is reflected in various abstract domains or functors, based on decision diagrams. Examples include the "segmented decision trees" [6] designed to support analysis of array processing programs, and the decision structures used in the FuncTion analyzer [26] for proving termination based on synthesised ranking functions. Jeannet's BDDAPRON library [13] provides a broad framework for the implementation of "logico-numeric" abstract domains, supporting analysis of programs with a mixture of finite-type and numeric variables.

Regarding interval analysis, the DisInterval abstract domain used in the Clousot analysis tool [8] allows for a limited amount of disjunctive information; while it can express monadic constraints such as $|x| > 5$, it cannot express a set such as the one depicted in Fig. 1(b). For fully disjunctive interval analysis, the most important data structure so far has been the linear decision diagram (LDD), introduced by Chaki, Gurfinkel and Strichman [3]. The Crab implementation of `Boxes` that we use as a baseline corresponds to the proposal by Gurfinkel and Chaki [11], that is, it uses a restricted form of LDDs, in which nodes can only be bounds constraints.

For program analysis, Typed Decisions Graphs [18] give an alternate, more concise representation of BDDs. They might be usable as a direct replacement for LDDs, but how to extend them to handle RDDs is far from obvious since they rely on representing a Boolean function to get good compression (by negating arcs).

6 Conclusion

We have demonstrated the importance of well-chosen data structures for disjunctive interval analysis. Our focus has been on the case of variables with integer types, but an extension to rationals or floating point numbers is not difficult (the main added complication is the need to identify split points as left- or right-included, that is, to distinguish whether a range bound is included or not).

For simplicity, we have also assumed the use of integers of unlimited precision. It would not be difficult to adapt the algorithms to the case of fixed-width integers, as the RDD representation is agnostic about the underlying representation of intervals.

The use of a dedicated data structure (RDDs) for interval sets has led us to a disjunctive interval analysis that is more efficient than the current LDD-based alternative. The use of RDDs offers a more precise analysis of non-linear arithmetic expressions, and it frees us from any dependence on a theory solver. These advantages explain why we see gains in both precision and speed.

A next natural step is to explore the combination of `Boxes` with weakly relational abstract domains. We hypothesize that, in practice, this provides an avenue to obtain considerably greater expressiveness while still keeping analysis tractable. For example, a product that includes the `Zones` abstract domain should produce an efficient program analysis with the expressiveness of clock decision diagrams [16].

Acknowledgements

We thank the three anonymous reviewers for their careful reading of an earlier version of the paper, and their constructive suggestions for how to improve it. Jorge Navas has been supported by the National Science Foundation under grant number 1816936.

References

1. Apt, K.: Principles of Constraint Programming. Cambridge University Press (2003). https://doi.org/10.1017/CBO9780511615320
2. Bryant, R.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24**(3), 293–318 (1992). https://doi.org/10.1145/136035.136043
3. Chaki, S., Gurfinkel, A., Strichman, O.: Decision diagrams for linear arithmetic. In: Proceedings of the 9th Conference on Formal Methods in Computer-Aided Design (FMCAD 2009). pp. 53–60. IEEE Comp. Soc. (2009). https://doi.org/10.1109/FMCAD.2009.5351143
4. Choi, C.W., Harvey, W., Lee, J.H.M., Stuckey, P.J.: Finite domain bounds consistency revisited. In: Proceedings of the Australian Conference on Artificial Intelligence 2006. Lecture Notes in Computer Science, vol. 4304, pp. 49–58. Springer (2006). https://doi.org/10.1007/11941439_9
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the Sixth ACM Symposium on Principles of Programming Languages. pp. 269–282. ACM Press (1979). https://doi.org/10.1145/567752.567778
6. Cousot, P., Cousot, R., Mauborgne, L.: A scalable segmented decision tree abstract domain. In: Manna, Z., Peled, D.A. (eds.) Time for Verification: Essays in Memory of Amir Pnueli, Lecture Notes in Computer Science, vol. 6200, pp. 72–95. Springer (2010). https://doi.org/10.1007/978-3-642-13754-9_5
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the Fifth ACM Symposium on Principles of Programming Languages. pp. 84–97. ACM Press (1978). https://doi.org/10.1145/512760.512770
8. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: Beckert, B., Marché, C. (eds.) Formal Verification of Object-Oriented Software. Lecture Notes in Computer Science, vol. 6528, pp. 10–30. Springer (2011). https://doi.org/10.1007/978-3-642-18070-5_2
9. Giacobazzi, R., Ranzato, F.: Optimal domains for disjunctive abstract interpretation. Science of Computer Programming **32**, 177–210 (1998). https://doi.org/10.1016/S0167-6423(97)00034-8
10. Groote, J.F., van de Pol, J.: Equational binary decision diagrams. In: Parigot, M., Voronkov, A. (eds.) Logic for Programming and Automated Reasoning. Lecture Notes in Artificial Intelligence, vol. 1955, pp. 161–178. Springer (2000). https://doi.org/10.1007/3-540-44404-1_11
11. Gurfinkel, A., Chaki, S.: Boxes: A symbolic abstract domain of boxes. In: Cousot, R., Martel, M. (eds.) Static Analysis: Proceedings of the 17th International Symposium. Lecture Notes in Computer Science, vol. 6337, pp. 287–303. Springer (2010). https://doi.org/10.1007/978-3-642-15769-1_18
12. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: Levi, G. (ed.) Static Analysis. Lecture Notes in Computer Science, vol. 1503, pp. 200–214. Springer (1998). https://doi.org/10.1007/3-540-49727-7_12
13. Jeannet, B.: The BddApron logico-numerical abstract domains library (2009), available at http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/bddapron/
14. Jeannet, B., Miné, A.: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. Lecture Notes in

Computer Science, vol. 5643, pp. 661–667. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_52
15. Jensen, T.P.: Disjunctive strictness analysis. In: Proceedings of the 7th Annual IEEE Symposium of Logic in Computer Science. pp. 174–185. IEEE Comp. Soc. (1992). https://doi.org/10.1109/LICS.1992.185531
16. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Clock difference diagrams. Nordic Journal of Computing **6**(3), 271–298 (1999)
17. Marriott, K., Søndergaard, H.: Precise and efficient groundness analysis for logic programs. ACM Letters on Programming Languages and Systems **2**(1–4), 181–196 (1993). https://doi.org/10.1145/176454.176519
18. Mauborgne, L.: Abstract interpretation using typed decision graphs. Science of Computer Programming **31**(1), 91–112 (1998). https://doi.org/10.1016/s0167-6423(96)00042-1
19. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) Programming Languages and Systems: Proceedings of the 14th European Symposium. Lecture Notes in Computer Science, vol. 3444, pp. 5–20. Springer (2005). https://doi.org/10.1007/978-3-540-31987-0_2
20. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) Programs as Data Objects, Lecture Notes in Computer Science, vol. 2053, pp. 155–172. Springer (2001). https://doi.org/10.1007/3-540-44978-7_10
21. Miné, A.: The Octagon abstract domain. In: Burd, E., Aiken, P., Koschke, R. (eds.) Proceedings of the Eighth Working Conference on Reverse Engineering, pp. 310–319. IEEE Comp. Soc. (2001). https://doi.org/10.1109/WCRE.2001.957836
22. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Emerson, E.A., Namjoshi, K.S. (eds.) Verification, Model Checking, and Abstract Interpretation. Lecture Notes in Computer Science, vol. 3855, pp. 348–363. Springer (2006). https://doi.org/10.1007/11609773_23
23. Møller, J., Lichtenberg, J., Andersen, H.R., Hulgaard, H.: Difference decision diagrams. In: Flum, J., Rodriguez-Artalejo, M. (eds.) Computer Science Logic. Lecture Notes in Computer Science, vol. 1683, pp. 111–125. Springer (1999). https://doi.org/10.1007/3-540-48168-0_9
24. Srinivasan, A., Kam, T., Malik, S., Brayton, R.K.: Algorithms for discrete function manipulation. In: Computer-Aided Design: Proceedings of the IEEE International Conference. pp. 92–95. IEEE Comp. Soc. (1990). https://doi.org/10.1109/ICCAD.1990.129849
25. Strehl, K., Thiele, L.: Symbolic model checking of process networks using interval diagram techniques. In: International Conference on Computer-Aided Design. pp. 686–692. ACM Press (1998). https://doi.org/10.1145/288548.289117
26. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Müller-Olm, M., Seidl, H. (eds.) Static Analysis. Lecture Notes in Computer Science, vol. 8723, pp. 302–318. Springer (2014). https://doi.org/10.1007/978-3-319-10936-7_19