Widening BDDs

Jacob M. Howe and Andy King

Technical Report No: 5-01
Date: May 2001

Abstract

Boolean functions are often represented as binary decision diagrams (BDDs). BDDs are potentially of exponential size in the number of variables of the function. Boolean functions drawn from *Pos* (the class of positive Boolean functions) and *Def* (the class of definite Boolean functions) are often used to describe the groundness of, and grounding dependencies between, program variables in (constraint) logic programs. *Pos*-based analyses are often implemented using BDDs which are sometimes problematically large. Since the complexities of the most frequently used domain operations are quadratic in the size of the input BDDs, widening BDDs for space is also a widening for time, hence is important for scalability. Two algorithms for widening BDDs for space are presented and are discussed (with relation to groundness analysis).

1 Introduction

Groundness analysis is an important theme of logic programming and abstract interpretation. Groundness analyses identify those program variables bound to terms that contain no variables (ground terms). Groundness information is typically inferred by tracking dependencies among program variables. These dependencies are commonly expressed as Boolean functions. For example, the function $x \wedge (y \leftarrow z)$ describes a state in which x is definitely ground, and there exists a grounding dependency such that whenever z becomes ground then so does y.

Groundness analyses usually track dependencies using either *Pos* [1, 2, 6, 10, 15, 18, 19], the class of positive Boolean functions, or *Def* [1, 11, 13, 14], the class of definite positive functions. *Pos* is more expressive than *Def*, but *Def* analysers can be faster [1, 13] and, in practice, the loss of precision for goal-dependent groundness analysis is usually small [13].

A cautious compiler vendor is unlikely to adopt an analysis unless it comes with scalability guarantees. For an analysis to be practical, both its speed and its memory consumption need remain within reasonable bounds, even for large programs. The time required to analyse a program depends primarily on the cost of each domain operation and the number of times these operations are applied. The number of times the domain operations are applied relates to the number of iterations that are required to reach the fixpoint. This, in turn, depends on the chain length of the underlying domain.

The cost of each domain operation required in groundness analysis depends critically on the way dependencies are represented. Prolog, C and SML based *Pos* and *Def* analysers have been constructed around a number of representations: (1) Armstrong *et al* [1] discuss Dual Blake Canonical Form (DBCF) for representing Boolean functions. (2) Howe and King [13] argue that a non-ground (non-orthogonal [1]) clausal representation is well suited to *Def*. (3) Codish and Demoen [6] use a set of possibly non-ground atoms over the alphabet $\{true, false\}$ to represent the truth table of a *Pos* function. (4) Finally, many authors [1, 2, 10, 18, 19] use binary decision diagrams (BDDs) and their variants, such as reduced, ordered binary decision diagrams, for *Pos*.

The speed of analysis is related to the compactness of its representation. BDDs give a dense representation for *Pos*, hence their popularity. However, even BDDs can get large, impacting on time as well as space. Codish [5] gives a series of programs which generate BDDs with size exponential in the size of the input program. This motivates widening BDDs for size, that is, trading some precision for a smaller representation. Fecht [10] suggests one such widening. This widening takes as input a BDD for a *Pos* formula and outputs a BDD that only records which variables are definitely ground. This paper describes two less aggressive widenings for BDDs. Both algorithms are quadratic in the size of the input BDD. The two algorithms are compared and it is shown that widening BDDs for space is not, in general, enough to bound the number of iterations of a *Pos* analysis.

The rest of the paper is structured as follows: Section 2 details the necessary preliminaries; Section 3 investigates widening for space BDD representations of Boolean functions; Section 4 discusses related work and Section 5 concludes.

2 Preliminaries

A Boolean function is a function $f : Bool^n \to Bool$ where $n \geq 0$. A Boolean function can be represented by a propositional formula over a set of variables X where $|X| = n$. The set of propositional formulae over X is denoted by $Bool_X$. Throughout this paper, Boolean functions and propositional formulae interchangeably without worrying about the distinction [1]. The convention of identifying a truth assignment with the set of variables M that it maps to *true* is also followed. Specifically, a map $\psi_X(M) : \mathcal{P}(X) \to Bool_X$ is introduced defined by: $\psi_X(M) = (\wedge M) \wedge (\neg \vee X \backslash M)$. In addition, the formula $\wedge Y$ is often abbreviated as Y.

Definition 2.1 **($model_X$)** The (bijective) map $model_X : Bool_X \to \mathcal{P}(\mathcal{P}(X))$ is defined by: $model_X(f) = \{M \subseteq X \mid \psi_X(M) \models f\}$.

Observe that $model_X(f)$ is the set of models of f, whilst $\mathcal{P}(X) \setminus model_X(f)$ is the set of countermodels of f.

Example 2.1 If $X = \{x, y\}$, then the function $\{\langle true, true\rangle \mapsto true,\ \langle true, false\rangle \mapsto false,\ \langle false, true\rangle \mapsto false,\ \langle false, false\rangle \mapsto false\}$ can be represented by the formula $x \wedge y$. Also, $model_X(x \wedge y) = \{\{x, y\}\}$ and $model_X(x \vee y) = \{\{x\}, \{y\}, \{x, y\}\}$.

Definition 2.2 Pos_X is the set of positive Boolean functions over X. A function f is positive iff $X \in model_X(f)$. Def_X is the set of positive functions over X that are definite. A function f is definite iff for all $M, M' \in model_X(f)$, $M \cap M' \in model_X(f)$.

Hasse diagrams for dyadic *Pos* and *Def* can be seen in Fig. 1. Note that $Def_X \subseteq Pos_X$. One useful representational property of Def_X is that each $f \in Def_X$ can be described as a conjunction of definite (propositional) clauses, that is, $f = \wedge_{i=1}^{n}(y_i \leftarrow Y_i)$ [9].

Example 2.2 Suppose $X = \{x, y, z\}$ and consider the following table, which states, for some Boolean functions, whether they are in Def_X or Pos_X and also gives $model_X$.

f	Def_X	Pos_X	$model_X(f)$
$false$			$\emptyset$
$x \wedge y$	●	●	$\{\{x,y\}, \{x,y,z\}\}$
$x \vee y$		●	$\{\{x\}, \{y\}, \{x,y\}, \{x,z\}, \{y,z\}, \{x,y,z\}\}$
$x \leftarrow y$	●	●	$\{\emptyset, \{x\}, \{z\}, \{x,y\}, \{x,z\}, \{x,y,z\}\}$
$x \leftarrow (y \leftarrow z)$		●	$\{\{x\}, \{z\}, \{x,y\}, \{x,z\}, \{x,y,z\}\}$
$true$	●	●	$\{\emptyset, \{x\}, \{y\}, \{z\}, \{x,y\}, \{x,z\}, \{y,z\}, \{x,y,z\}\}$

Note, in particular, that $x \vee y \notin Def_X$ (since its set of models is not closed under intersection) and that $false$ is in neither Pos_X nor Def_X.

Defining $f_1 \dot{\vee} f_2 = \wedge\{f \in Def_X \mid f_1 \models f \wedge f_2 \models f\}$, the 4-tuple $\langle Def_X, \models, \wedge, \dot{\vee}\rangle$ is a finite lattice, where *true* and X are the top and bottom elements. Existential quantification is defined by Schröder's Elimination Principle, that is, $\exists x.f = f[x \mapsto true] \vee f[x \mapsto false]$. Note that if $f \in Def_X$ then $\exists x.f \in Def_X$ [1].

Example 2.3 If $X = \{x, y\}$ then $x \dot{\vee} (x \leftrightarrow y) = \wedge\{(x \leftarrow y), true\} = (x \leftarrow y)$, as can be seen in the Hasse diagram for dyadic Def_X (Fig. 1). Note also that $x \dot{\vee} y = \wedge\{true\} = true \neq (x \vee y)$.

The set of (free) variables in a syntactic object o is denoted $var(o)$. Also, $\exists\{y_1, \ldots, y_n\}.f$ (project out) abbreviates $\exists y_1. \ldots .\exists y_n.f$ and $\overline{\exists} Y.f$ (project onto) denotes $\exists var(f) \setminus Y.f$.

Let S be a set partially ordered by $\preceq$, then $C \subseteq S$ is a chain iff for all $x, y \in C$ either $x \preceq y$ or $y \preceq x$. A chain M is maximal iff for all chains $C \subseteq S$, $|C| \leq |M|$.

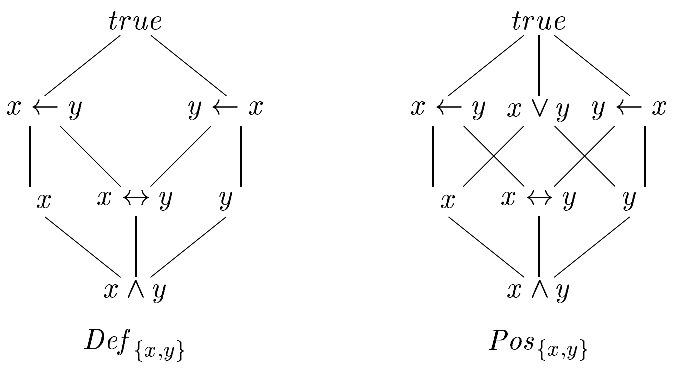

Figure 1: Hasse diagrams

3 Widening for Space and Time

Classically [8], widening is a method for enforcing termination in abstract interpretation. It consists of using a widening operator on a join semi-lattice $L(\sqsubseteq, \sqcup)$, $\nabla : L \times L \to L$, such that for all $x, y \in L$, $x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$ and for all increasing chains $x_1 \sqsubseteq x_2 \sqsubseteq \ldots$, the increasing chain defined by $y_0 = x_0, \ldots, y_{i+1} = y_i \nabla x_{i+1}$ is not strictly increasing.

Widening also can be applied to domains that satisfy the ascending chain condition in order to accelerate convergence of a fixpoint calculation. In this situation, it is usual to widen a single abstraction in isolation, rather than in the context of an increasing chain. This is because the tractability of an analysis depends, in part, on keeping all the intermediate abstractions small (not just those abstractions that occur, for instance, as call and answer patterns [17]). Intermediate abstractions, by definition, are not recorded in a database, thus the previous abstractions are not available to a widening to aid extrapolation. In this section, this widening in isolation approach is applied to BDDs.

A binary decision diagram (BDD) is a rooted, directed acyclic graph. Terminal nodes are labelled 0 or 1 and non-terminal nodes are labelled by a variable and have arcs directed towards two child nodes. In the following, BDDs have the additional properties that: 1) each path from the root to a node respects a given ordering on the variables, 2) a variable cannot occur multiply in a path, 3) no subBDD occurs multiply. Such BDDs are known as reduced, ordered binary decision diagrams and give a unique representation for every Boolean function.

The size of a BDD representing a Pos_X formula is potentially $2^{|X|}$. Since the most frequently used BDD operations are quadratic in the size of the BDD, widening the BDD for size is also a widening for time. Codish [5] gives a series of programs which generate BDDs with an exponential (in the size of the program) number of distinct nodes; this example coupled with the experimental work of Fecht [10] motivates widening BDDs for space.

The most promising representation of *Pos* using BDDs is the GER factorisation of Bagnara and Schachte [2, 18]. This hybrid representation consists of three components: a set of ground variables (G), a set of equivalent variables (E), and a BDD for more complex dependencies (R). This significantly reduces the size of the representation. A simple widening for *Pos* is to replace the R component with the logical constant 1. This corresponds to widening to a subdomain of Pos_X, namely $EPos_X$ [12], whose chain length is $|X| + 1$. This is an attractive technique, since the widening is independent of the variable ordering of the BDD. Notice that a more precise widening is likely to depend on the variable ordering, since this impacts on the size of a BDD. Also note that widening a BDD representing a *Def* function to another *Def* function is problematic, as BDDs are not closely related to functions closed under model intersection. Thus BDDs appear unsuitable for implementing Def.

Given a size bound, l, widening a BDD representing a function $f \in Pos_X$ results in a function g, whose size does not exceed l, such that $f \models g$. Notice that $X \models g$, since $X \models f$, thus $g \in Pos_X$. The loss of precision that results from widening BDDs can be quantified in terms of the number of extra models of the widened function. Moreover, suppose $f \models g_1$ and $f \models g_2$, where $|model_X(g_1)| \leq |model_X(g_2)|$ and the sizes of g_1 and g_2 do not exceed l, then the widening should be biased towards selecting g_1. Two algorithms (one sample based, one heuristic) that follow this tactic are described below. In the following, let $|g|$ denote the number of nodes in the BDD g and let $\| g \|$ be a measure of the number of countermodels, defined as follows: $\| g \| = (\| g_t \| + \| g_f \|)/2$, where g_t and g_f are the subBDDs rooted at the children of g and $\| 0 \| = 1$, $\| 1 \| = 0$.

3.1 Sample Based Widening

The sample based widening is an iterative algorithm that, at each stage, removes at least one node from a BDD g. The algorithm is parameterised by a constant size limit l and proceeds as follows. Calculate $|g|$ and choose $k \geq 1$ nodes $n_1, \ldots, n_k$ of g at random. If n_i has a child whose size does not exceed l, then let h_i denote the join of the subBDDs rooted at the children of n_i. Otherwise let h_i denote the constant 1. Construct g_i from g by replacing the subBDD at n_i with h_i. If $|g_i| \geq |g|$ then reassign h_i to 1 and re-compute $|g_i|$. Observe that for all $1 \leq i \leq k$, $|g_i| < |g|$. Compute $r^s_i = \| g_i \| / |g_i|$ for each n_i and let $r^s_{max} = \max\{r^s_i \mid 1 \leq i \leq k\}$. If $|g_{max}| \leq l$, stop and return g_{max} as the result of the widening. Otherwise reapply the procedure with g replaced with g_{max}.

This widening is $O(m^2)$ in both space and time in the number of nodes, m, of the input g. To see this, observe that computing $|g|$ is $O(m)$. Note that given $|g|$, the test $|g_i| \geq |g|$ is $O(m)$ since at most $m + 1$ nodes of g_i need to be considered. Each join operates on (at least) one subBDD whose size does not exceed the constant l. Thus each join has complexity $O(m)$. Replacing the subBDD at n_i with h_i is also $O(m)$ as is computing $\| g_i \|$ and $|g_i|$. The number of iterations of the loop is at most $m - l$ and hence the widening is $O(m^2)$.

Notice that the reliability of the widening depends primarily on the size of the sample (rather than on $|g|$). For example, with a sample of 32 nodes, there is (at least) a 97% probability that n_{max} is in the top 10% of the all the nodes of g according to the r^s_i ranking.

3.2 Heuristic Widening

The sample based widening will lose precision if the sampling is unfortunate. This motivates a widening based on a heuristic. Ideally, a widening will remove many nodes whilst introducing few extra models. The algorithm will proceed by ranking the nodes of the BDD by their suitability for removal and replacing the most suitable node with 1. The nodes of the new BDD are ranked and the procedure is repeated until the resulting BDD has fewer than l nodes. For each node n_i of the BDD g (with m nodes) consider h_i, the subBDD rooted at n_i. For each h_i the ratio $r^h_i = (\| h_i \| . \Sigma_{p \in P_i} 2^{-|p|}) / \lfloor h_i \rfloor$ is calculated, where P_i is the set of paths from the root of g to n_i, $|p|$ the length of p, $\lfloor h_i \rfloor \leq |g| - |g'|$ and g' is the result of replacing h_i with 1 in g. $\lfloor h_i \rfloor$ is calculated by counting the number of nodes is the subBDD rooted at n_i which have only one parent in g. Let $r^h_{min} = \min\{r^h_i \mid 1 \leq i \leq m\}$ and replace the subBDD rooted at n_{min} by 1. If the resulting BDD has less than l nodes, then stop, otherwise the procedure is reapplied.

Note that $\lfloor h_i \rfloor$ is less than or equal to the number of the nodes removed by replacing the subBDD rooted at n_i by 1 for two reasons. Firstly, nodes with more than one parent, but whose parents all have n_i as an ancestor, are not counted. In fact these will be removed. Secondly, there may be a subBDD which occurs in both the old and the new BDD that has extra parents in the

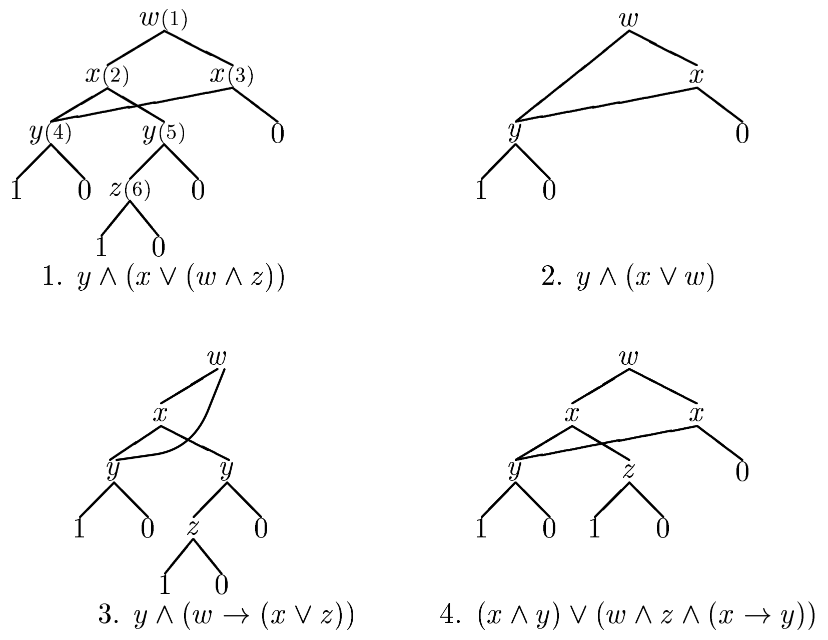
w(1)
x(2)
x(3)
y(4)
y(5)
z(6)
1. $y \wedge (x \vee (w \wedge z))$
2. $y \wedge (x \vee w)$
3. $y \wedge (w \to (x \vee z))$
4. $(x \wedge y) \vee (w \wedge z \wedge (x \to y))$

Figure 2: BDDs for Example 3.1

new BDD.

This widening is also $O(m^2)$ in both space and time in the number of nodes, m, of the input BDD g. To see this, observe that counting the number of parents (references) to each node n_i of g, and computing $\lfloor h_i \rfloor$ for each n_i, can all be computed in a single pass over g in $O(m)$ time. The sets P_i can be computed in a single pass of g in $O(m)$. Replacing the subBDD at n_i with 1 is also $O(m)$, as is computing $\| h_i \|$. The number of iterations of the loop is at most $m - l$, since each iteration must remove at least one node. Hence the widening is $O(m^2)$.

Example 3.1 This example illustrates the application of the two widenings to the BDD for $y \wedge (x \vee (w \wedge z))$, which is 1. in Fig. 2 (where the left branch is the true branch and the right branch is the false branch). The variable ordering is alphabetical, and the constant size limit l is 4. Observe that the size of BDD 1. is 6. Following the sample based widening, nodes (2), (3) and (5) were chosen at random (using a die). The subBDDs located at (2), (3) and (5) all have a child of size less than l. The construction of the g_i for the 3 nodes result in the BDDs 2., 3. and 4., respectively. The r_i^s for 2., 3. and 4. are (5/8)/3=5/24, (9/16)/5=9/80 and (5/8)/5=1/8, respectively. Hence BDD 2. is the result of the widening (as its size is less than l).

Following the heuristic widening, r_i^h is calculated for each node, giving $r_1^h = ((11/16).1)/5{=}11/80$, $r_2^h{=}((5/8).(1/2))/3{=}5/48$, $r_3^h = ((3/4).(1/2))/1{=}3/8$, $r_4^h = ((1/2).(1/2))/0 = \infty$, $r_5^h = ((3/4).(1/4))/2{=}3/32$, $r_6^h = ((1/2).(1/8))/1{=}1/16$. Hence node 6 is replaced by 1, to give BDD 2. as the result of the widening (as its size is less than l). Notice that both widenings result in the same BDD and this includes just one extra model.

3.3 Comparison of the Widenings

The two widenings are in some sense dual. The sample based widening is biased towards a loss of precision in node selection, whereas the heuristic widening is biased towards a loss of precision in pruning. More exactly, on the one hand, by computing joins (if possible), the first widening retains some precision in its pruning step. On the other hand, it relies on random sampling for node selection. Conversely, by using a heuristic, the second widening is likely to locate good candidate nodes for elimination, but this elimination can lose significant precision. It is unclear which tactic is the most effective, as the precision of both techniques depends in part on the number of iterations required.

It is desirable for a widening to be linear. Although both widenings detailed above are quadratic, both could be made linear by bounding the number of iterations about the loop by a constant (say, l), returning 1 if the number of iterations exceeds this limit. This would then reduce the complexity of the widenings to $O(m)$, at the expense of precision. However, assuming that the input BDDs are not excessively large, quadratic behaviour is acceptable.

It is surprising to observe that even widening a BDD to $|X|$ nodes is not sufficient to avoid chain of exponential size. Consider the program in Example 3.2. Ordering the variables alphabetically, the size of the BDD for each of the iterates does not exceed $|X|$. Also note that the widenings above can be applied to the R component of the GER factorisation, ensuring that groundness and simple bidirectional dependencies are retained.

Example 3.2 The following program is the arity 4 instance of the schema given by Codish in [5].

chain(c, c, c, c).
chain(v, c, c, c) :- chain(_, v, v, v).
chain(w, v, c, c):- chain(w, _, v, v).
chain(w, x, v, c):- chain(w, x, _, v).
chain(w, x, y, _):- chain(w, x, y, _).

The results of *Pos*-based success pattern groundness analysis of this program are summarised by the table below, where i is the iteration number, fact$_i$ is the (single) new fact which is added in the i^{th} interpretation and f_i is the formula which describes the chain(w, x, y, z) atoms in the i^{th} interpretation.

i	fact$_i$	f_i	i	fact$_i$	f_i
1	chain(c, c, c, c)	$w \wedge x \wedge y \wedge z$	2	chain(c, c, c, _)	$w \wedge x \wedge y$
3	chain(c, c, _, c)	$w \wedge x \wedge (y \vee z)$	4	chain(c, c, _, _)	$w \wedge x$
5	chain(c, _, c, c)	$w \wedge (x \vee (y \wedge z))$	6	chain(c, _, c, _)	$w \wedge (x \vee y))$
7	chain(c, _, _, c)	$w \wedge (x \vee y \vee z)$	8	chain(c, _, _, _)	w
9	chain(_, c, c, c)	$w \vee (x \wedge y \wedge z)$	10	chain(_, c, c, _)	$w \vee (x \wedge y)$
11	chain(_, c, _, c)	$w \vee (x \wedge (y \vee z))$	12	chain(_, c, _, _)	$w \vee x$
13	chain(_, _, c, c)	$w \vee x \vee (y \wedge z)$	14	chain(_, _, c, _)	$w \vee x \vee y$
15	chain(_, _, _, c)	$w \vee x \vee y \vee z$	16	chain(_, _, _, _)	*true*

4 Related work

Mauborgne [16] shows how to perform strictness analysis of higher-order functions with typed decision graphs (TDGs) [3]. A TDG [3] is a BDD variant in which the decision node for each variable x_i is additionally tagged with a polarity. The polarity information enables the TDG for the function $f(x_i, \ldots, x_n)$ to share its nodes with the TDG for $\neg f(x_i, \ldots, x_n)$ (if both functions occur together) and thus a TDG can encode some functions more compactly than a classic BDD [4]. Nevertheless, Mauborgne [16] advocates widening TDGs for space. He proposes an operator $\nabla(l, f)$ that takes, as input, a TDG that encodes a function f and returns, as output, a TDG g with at most l nodes such that $f \models g$. The first widening he proposes is at least $O(n^4)$ in the number of nodes in the input TDG. This is because (one iteration of) the widening algorithm computes the meet of each pair of nodes in the TDG and meet is $O(n^2)$. To improve efficiency, Mauborgne suggests a second widening that consists of taking a TDG of n nodes and computing the TDGs $f_1, \ldots, f_n$ obtained by replacing node i with 1. The f_i are filtered to remove those TDGs whose size exceed $n/2$. Of the remaining f_i, an f_{max} is selected which "gives the best result". The widening is reapplied to f_{max} if its TDG contains more than l nodes. This widening appears to be $O(n^2)$ time (assuming that it takes $O(n)$ steps to assess the accuracy of each of the filtered f_i). Both of these widenings could be adapted to BDDs. However, the $O(n^4)$ algorithm appears to be too expensive to be practical. The $O(n^2)$ algorithm is more aggressive than the widenings presented in Section 5, whilst its complexity is in the same class.

Zaffanella *et al.* [20] propose several widenings for domain the *Sharing*. Since there exists an isomorphism between *Sharing* and *Pos* [7], these widenings can be reinterpreted as widenings for *Pos*. However, it is not clear that these widenings can be applied to a BDD efficiently.

5 Conclusion

This paper has proposed two widenings for space for BDDs. Since the size of the representations of Boolean functions impact on the complexity of domain operations, the widenings improve both space and time aspects of groundness analysis. and help to ensure that program analysis remains tractable and scales smoothly. However, it was also shown that widening BDDs for space does not necessarily restrain chain length.

Further experimental work will quantitatively assess the widenings. It is suspected that is difficult to better quadratic complexity for a BDD widening whilst retaining good precision and it would be insightful to formalise this intuition.

Acknowledgements

We thank Florence Benoy, Mike Codish, Pat Hill and Jan-Georg Smaus for interesting discussions. This work was supported, in part, by EPSRC grant GR/MO8769.

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming*, 31(1):3–45, 1998.

[2] R. Bagnara and P. Schachte. Factorizing Equivalent Variable Pairs in ROBDD-Based Implementations of *Pos*. In *Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 471–485. Springer-Verlag, 1999.

[3] J.-P. Billon. Perfect Normal Forms for Discrete Programs. Technical Report 87019, Bull Corporate Research Center, 1987.

[4] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Digrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[5] M. Codish. Worst-Case Groundness Analysis Using Positive Boolean Functions. *Journal of Logic Programming*, 41(1):125–128, 1999.

[6] M. Codish and B. Demoen. Analysing Logic Programs using "prop"-ositional Logic Programs and a Magic Wand. *Journal of Logic Programming*, 25(3):249–274, 1995.

[7] M. Codish, H. Søndergaard, and P. Stuckey. Sharing and Groundness Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.

[8] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[9] P. Dart. On Derived Dependencies and Connected Databases. *Journal of Logic Programming*, 11(2):163–188, 1991.

[10] C. Fecht. *Abstrakte Interpretation Logischer Programme: Theorie, Implementierung, Generierung.* PhD thesis, Universität des Saarlandes, 1997.

[11] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–614, 1996.

[12] A. Heaton, M. Abo-Zaed, M. Codish, and A. King. A Simple Polynomial Groundness Analysis for Logic Programs. *Journal of Logic Programming*, 45(1–3):143–156, 2000.

[13] J. M. Howe and A. King. Implementing Groundness Analysis with Definite Boolean Functions. In *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 200–214. Springer-Verlag, 2000.

[14] A. King, J.-G. Smaus, and P. Hill. Quotienting Share for Dependency Analysis. In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 59–73. Springer-Verlag, 1999.

[15] K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Letters on Programming Languages and Systems*, 2(4):181–196, 1993.

[16] L. Mauborgne. Abstract Interpretation Using Typed Decision Graphs. *Science of Computer Programming*, 31(1):91–112, 1998.

[17] P. Mildner. *Type Domains for Abstract Interpretation: A Critical Study.* PhD thesis, University of Uppsala, 1999. Uppsala Theses in Computer Science, 31.

[18] P. Schachte. *Precise and Efficient Static Analysis of Logic Programs.* PhD thesis, Department of Computer Science, The University of Melbourne, Melbourne, Australia, 1999.

[19] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the Domain *Prop*. *Journal of Logic Programming*, 23(3):237–278, 1995.

[20] E. Zaffanella, R. Bagnara, and P. Hill. Widening Sharing. In *Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 414–431. Springer-Verlag, 1999.