A Reactive Implementation of *Pos* Using ROBDDs

Roberto Bagnara

Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy
E-mail: `bagnara@di.unipi.it`.

Abstract. The subject of groundness analysis for (constraint) logic programs has been widely studied, and interesting domains have been proposed. *Pos* has been recognized as the most suitable domain for capturing the kind of dependencies arising in groundness analysis. Its (by now standard) implementation is based on *reduced ordered binary-decision diagrams* (ROBDDs), a well-known symbolic representation for Boolean functions. Even though several authors have reported positive experiences using ROBDDs for groundness analysis, in the literature there is no reference to the problem of the efficient detection of those variable which are deemed to be ground in the context of a ROBDD. This is not surprising, since most currently implemented analyzers need to derive this information only *at the end* of the analysis and only for presentation purposes. Things are much different when this information is required *during* the analysis. This need arises when dealing with languages which employ some sort of *delay mechanism*, which are typically based on groundness conditions. In these cases, the *naïf* approaches are too inefficient, since the abstract interpreter must quickly (and often) decide whether a constraint is delayed or not. Fast access to ground variables is also necessary when aliasing analysis is performed using a domain not keeping track of ground dependencies. In this paper we introduce and study the problem, proposing two possible solutions. The second one, besides making possible the quick detection of ground variables, has also the effect of keeping the ROBDDs as small as possible, improving the efficiency of groundness analysis in itself.

1 Introduction

The task of *groundness analysis* (or *definiteness* analysis as it is also referred to) is to derive, for all the program points of interest, whether a certain variable is bound to a unique value (or *ground*). This kind of information is very important: it allows substantial optimizations to be performed at compile-time, and is also crucial for most semantics-based program manipulation tools. Moreover, many other analysis are made much more precise by the availability of groundness information. For these reasons, the subject of groundness analysis for (constraint) logic programs has been widely studied. After the early attempts, some classes of Boolean functions have been recognized as constituting good abstract domains for groundness analysis [7, 16, 1, 19]. In particular, the set of *positive Boolean functions*, (namely, those functions which assume the true value under the valuation assigning true to all variables), which is denoted by *Pos*, has been recognized as the most precise domain for capturing the kind of dependencies arising in groundness analysis.

The standard implementation of *Pos* is based on *reduced ordered binary-decision diagrams* (ROBDDs), a well-known symbolic representation for Boolean functions. Indeed, ROBDDs are general enough to represent *all* Boolean functions. However, nobody has succeeded, to date, in exploiting the (seemingly very small) peculiarities of positive functions in order to obtain a more efficient implementation. Several authors have reported positive experiences using ROBDDs for groundness analysis (see, e.g., [16, 1]). However, in the literature there is no reference to the problem of detecting, as efficiently as possible, those variables which are deemed to be ground in the context of a ROBDD. This is not surprising, since most currently implemented analyzers need to derive this information only *at the end* of the analysis and only for presentation purposes. In these cases efficiency is not a problem and the simple approaches are good enough. Things are much different when this information is required *during* the analysis. This need arises when dealing with languages which employ some sort of *delay mechanism*, which are typically based on groundness conditions. One of these languages is CLP($\mathcal{R}$) [14], where non-linear constraints are delayed until they become linear; only then they are sent to the constraint solver. In the context of our work on data-flow analysis for CLP($\mathcal{R}$) we were thus facing the following problem: in programs with many non-linear constraints, the abstract interpreter was spending a lot of time deciding whether a constraint is delayed or not. In the early implementations of the China analyzer this kind of information (which is needed quite often) was derived using the ROBDD package itself (see Sect. 5). This had the advantage of making possible the use of untouched, readily-available ROBDD software, while having the big disadvantage of inefficiency.

In this paper we introduce and study the problem of quick detection of ground variables using ROBDDs. We first propose an easy, even though not completely satisfactory, solution. We then take a different approach where we represent *Pos* functions in a hybrid way: ground variables are represented explicitly, while ROBDDs come into play only for dependency and disjunctive information. This solution uses the more efficient representation for each kind of information: "surely ground variables" are best represented by means of sets (bit-vectors, at the implementation level), whereas ROBDDs are used only for "conditional" and "disjunctive" information. In such a way, besides making the information about ground variables readily available, we can keep the ROBDDs generated during the analysis as small as possible. This promises to be a win, given that most real programs (together with their typical call-patterns) exhibit a high percentage of variables which are ground at the program points of interest. Notice that Boolean functions are used in the more general context of *dependency analysis*, including *finiteness analysis* for deductive database languages and *suspension analysis* for concurrent (constraint) logic programming languages [1]. The techniques we propose might be useful also in these contexts. However, this is something we have not studied yet. In Sect. 2 we briefly review the usage of Boolean functions for groundness analysis of (constraint) logic programs (even though we assume familiarity on this subject). Sect. 3 presents the main motivations of this work. Binary-decision trees and diagrams, and the problem of extracting *sure groundness information* from them are introduced in Sects. 4 and 5. In Sect. 6 we show a first non-trivial solution to the problem, while Sect. 7 introduces the hybrid domain. The results of the experimental evaluation are reported in Sect. 8. Sect. 9 concludes with some final remarks.

2 Boolean Functions for Groundness Analysis

After the early approaches to groundness analysis [17, 15], which suffered from serious precision drawbacks, using Boolean functions has become customary in the field. The reason is that Boolean functions allow to capture in a very precise way the *groundness dependencies* which are implicit in unification constraints such as $z = f(g(x), y)$: the corresponding Boolean function is $(x \wedge y) \leftrightarrow z$, meaning that z is ground if and only if x and y are so. They also capture dependencies arising from other constraint domains: for instance, $x + 2y + z = 4$ can be abstracted as $((x \wedge y) \to z) \wedge ((x \wedge z) \to y) \wedge ((y \wedge z) \to x)$. We now introduce Boolean valuations and functions in a way which is suitable for what follows. *Vars* is a fixed denumerable set of variable's symbols.

Definition 1. (Boolean valuations.) *The set of* Boolean valuations over *Vars is given by* $\mathcal{A} \stackrel{\text{def}}{=} \mathit{Vars} \to \{0,1\}$. *For each* $a \in \mathcal{A}$, *each* $x \in \mathit{Vars}$, *and each* $c \in \{0,1\}$ *the valuation* $a[c/x] \in \mathcal{A}$ *is given, for each* $y \in \mathit{Vars}$, *by*

$$a[c/x](y) \stackrel{\text{def}}{=} \begin{cases} c, & \text{if } x = y; \\ a(y), & \text{otherwise.} \end{cases}$$

For $X = \{x_1, x_2, \dots\} \subseteq \mathit{Vars}$, *we write* $a[c/X]$ *for* $a[c/x_1][c/x_2]\cdots$.

Definition 2. (Boolean functions.) *The set of* Boolean function over *Vars is* $\mathcal{F} \stackrel{\text{def}}{=} \mathcal{A} \to \{0,1\}$. *The distinguished elements* $\top, \bot \in \mathcal{F}$ *are the functions defined by* $\top \stackrel{\text{def}}{=} \lambda a \in \mathcal{A} \mathrel{.} 1$ *and* $\bot \stackrel{\text{def}}{=} \lambda a \in \mathcal{A} \mathrel{.} 0$. *For* $f \in \mathcal{F}$, $x \in \mathit{Vars}$, *and* $c \in \{0,1\}$, *the function* $f[c/x] \in \mathcal{F}$ *is given, for each* $a \in \mathcal{A}$, *by* $f[c/x](a) \stackrel{\text{def}}{=} f\bigl(a[c/x]\bigr)$. *When* $X \subseteq \mathit{Vars}$, $f[c/X]$ *is defined in the obvious way.*

The question whether a Boolean function f forces a particular variable x to be true (which is what, in the context of groundness analysis, we call *sure groundness information*) is equivalent to the question whether $f \to x$ is a tautology (namely, $f \to x = \top$). In the sequel we will also need the notion of *dependent variables* of a function.

Definition 3. (Dependent and true variables.) *For* $f \in \mathcal{F}$, *the set of* variables on which f depends *and the set of* variables necessarily true for f *are given, respectively, by*

$$\begin{aligned} \mathit{vars}(f) &\stackrel{\text{def}}{=} \bigl\{\, x \in \mathit{Vars} \bigm| \exists a \in \mathcal{A} \mathrel{.} f\bigl(a[0/x]\bigr) \neq f\bigl(a[1/x]\bigr) \,\bigr\}, \\ \mathit{true}(f) &\stackrel{\text{def}}{=} \bigl\{\, x \in \mathit{vars}(f) \bigm| \forall a \in \mathcal{A} : f(a) = 1 \implies a(x) = 1 \,\bigr\}. \end{aligned}$$

Two classes of Boolean functions which are suitable for groundness analysis are known under the names of *Def* and *Pos* (see [1] for details). *Pos* consists precisely of those functions assuming the true value under the *everything-is-true* assignment (i.e., $f \in Pos$ iff $f \in \mathcal{F}$ and $f[1/Vars] = \top$). *Pos* is strictly more precise than *Def* for groundness analysis [1]. The reason is that the elements of *Pos* allow to maintain disjunctive information which is, instead, lost in *Def*.

3 Combination of Domains and Reactivity

It is well known that different data-flow analyses can be combined together. In the framework of abstract interpretation this can be achieved by means of standard constructions such as reduced product and down-set completion [10, 11]. The key point is that the combined analysis can be more precise than each of the component ones for they can mutually improve each other. However, the degree of cross-fertilization is highly dependent on the degree and quality of interaction taking place among the component domains. For the limited purpose of this paper, when we talk about *combination of domains* we refer to the following situation: we have several distinct (both conceptually and at the implementation level) analysis' domains and, for the sake of ensuring correctness or improving precision, there must be a flow of information between them. This can be formalized in different ways. A methodology for the combination of abstract domains has been proposed in [9]. It is based onto low level actions such as *tests* and *queries*. Basically, the component domains have the ability of querying other domains for some kind of information. Of course, they must also be able to respond to queries from other domains. For instance, the operations of a domain for numerical information might ask a domain for groundness whether a certain variable is guaranteed to be ground or not. Another way of describing this kind of interaction is the one proposed in [3]. Here the interaction among domains is asynchronous in that it can occur at any time, or, in other words, it is not synchronized with the domain's operations. This is achieved by considering so called *ask-and-tell constraint systems* built over *product constraint systems*. These constraint systems allow to express communication among domains in a very simple way. They also inherit all the semantic elegance of concurrent constraint programming languages [18], which provide the basis for their construction. We will now see, staying on an intuitive level and following the approach of [3] for simplicity, examples of how these combinations look like.

In the CLP($\mathcal{R}$) system [14] non-linear constraints (like $X = Y * Z$) are delayed (i.e., not treated by the constraint solver) until they become linear (e.g., until either Y or Z are constrained to take a single value). Obviously, this cannot be forgotten in abstract constraint systems intended to formalize correct data-flow analyses of CLP($\mathcal{R}$). When the abstract constraint system is able to extract information from non-linear constraints (such as the one proposed in [4]), you cannot simply let $X = Y * Z$, or better, its abstraction $\alpha(X = Y * Z)$ stand by itself. By doing this you would incur the risk of *overshooting* the concrete constraint system (thus loosing soundness), which is unable to deduce anything from non-linear constraints. The right thing to do is to combine the numeric abstract constraint system with one for groundness and using, instead of the abstraction $\alpha(X = Y * Z)$, the agent

$$A \stackrel{\text{def}}{=} \text{ask}\big(\mathit{ground}(Y); \mathit{ground}(Z)\big) \rightarrow \alpha(X = Y * Z).$$

The intuitive reading is that the abstract constraint system is not allowed to do anything with $X = Y * Z$ until Y *or* (this is the intuitive reading of the semicolon) Z are ground. In this way, all the abstractions of non-linear constraints are "disabled" until their wake-up conditions are met (in the abstract, which, given a sound groundness analysis, implies that these conditions are met also at the concrete level). The need for interaction between groundness and numerical domains does not end here. Consider again the constraint $X = Y * Z$: clearly X is definite if Y and Z are so. But we cannot conclude that the groundness of Y follows from the one of X and Z, as we need also the condition $Z \neq 0$. Similarly, we would like to conclude that X is definite if Y or Z have a zero value. Thus we need approximations of the concrete values of variables (i.e., bounds analysis), something which is not captured by common groundness analyses while being crucial when dealing with non-linear constraints. In the approach of [3] $X = Y * Z$ would be *abstractly compiled* into an agent of the form[1]

$$\begin{aligned}
A \parallel &\text{ask}\big(\mathit{ground}(Y) \wedge \mathit{ground}(Z)\big) \rightarrow \text{tell}\big(\mathit{ground}(X)\big) \\
\parallel &\text{ask}(Y = 0; Z = 0) \rightarrow \text{tell}\big(\mathit{ground}(X)\big) \\
\parallel &\text{ask}\big(\mathit{ground}(X) \wedge \mathit{ground}(Z) \wedge Z \neq 0\big) \rightarrow \text{tell}\big(\mathit{ground}(Y)\big) \\
\parallel &\text{ask}\big(\mathit{ground}(X) \wedge \mathit{ground}(Y) \wedge Y \neq 0\big) \rightarrow \text{tell}\big(\mathit{ground}(Z)\big).
\end{aligned}$$

Of course, this is much more precise than the *Pos* formula $X \leftarrow Y \wedge Z$, which is all you can say about the groundness dependencies of $X = Y * Z$ if you do not have any numerical information. It is clear from these examples that, when analyzing CLP($\mathcal{R}$) programs there is a bidirectional flow of information: groundness information is required for a correct handling of delayed constraints and thus for deriving more precise numerical patterns which, in turn, are used to provide more precise groundness information. Indeed, we are requiring a quite complicated interaction between domains.

Another application of groundness analysis with fast access to ground variables is for *aliasing analysis*. The most popular domain for this kind of analysis is *Sharing* [13]. Without going into details, its strength over the previous approaches [15, 12] comes from the fact that it keeps track of groundness dependencies. In fact, *Sharing* has, as far as groundness information is concerned, the same power of *Def*. When *Pos* is used for groundness, using *Sharing* for aliasing at the same time is a waste: *Sharing* spends time and space for keeping track of groundness, which is already done, and more precisely, by *Pos*. A possible solution is to adopt a variation of the domains proposed in [15, 12] (which are

[1] We choose this form of presentation for clarity. It is clear that this agent will be itself compiled to something different. For instance, the second agent of the parallel composition will "live" in the groundness component, if the latter is able to capture the indicated dependency.

much less computationally expensive than *Sharing*) and to combine it with *Pos*. We are currently working along this line. This, however, is beyond the scope of this paper.

Whatever conceptual methodology you follow to realize the combination of any domain with one for groundness, a key component for the efficiency is that the implementation of the latter must be *reactive*. By this we mean that: (a) it must react quickly to external queries about the groundness of variables; and, (b) it must absorb quickly groundness notifications coming from other domains.

4 Binary Decision Trees and Diagrams

Binary decision trees (BDTs) and diagrams (BDDs) are well-known abstract representations of Boolean functions [5, 6]. Binary decision trees, such as the ones presented in Fig. 1 are binary trees where non-terminal nodes are labeled

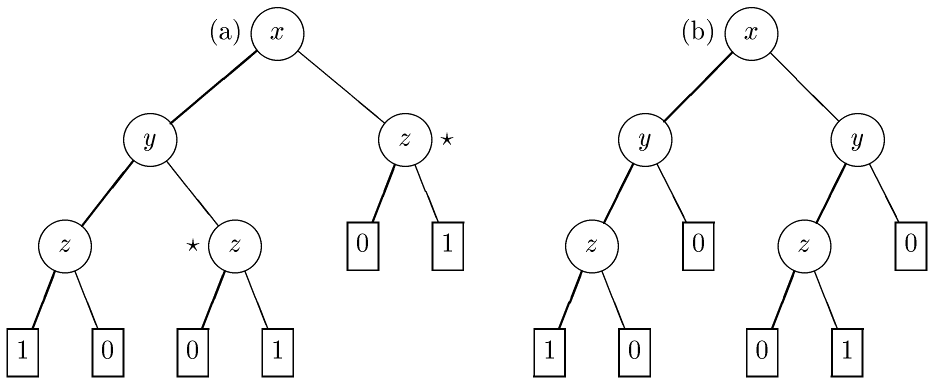

Figure 1. OBDTs for $(x \wedge y) \leftrightarrow z$ (a) and $(x \leftrightarrow z) \wedge y$ (b).

with variable names, while terminal nodes are labeled with the Boolean constants 0 or 1. The value of the represented function, for a given assignment of Boolean values to variables, can be recovered by following a particular path from the root: at any non-terminal node labeled with a variable v, the thick branch is taken if v is assigned to 1, otherwise the thin branch is taken. The terminal node reached by this walk on the tree is the function value. For a non-terminal node n, we will call the node connected to n by means of the thick (resp. thin) edge the *true* (resp. *false*) *successor of* n. A BDD is a directed acyclic graph which can be thought of as obtained from a BDT by collapsing identical subtrees. With reference to Fig. 1 (a), the subtrees marked with '$\star$' can be collapsed, as well as all the terminal nodes having the same label. The action of collapsing identical subtrees does not change the represented function. Given a total ordering on the variable symbols, an *ordered binary decision tree* (OBDT) is a BDT where the sequence of variables (associated to non-terminals) encountered in any path from the root is strictly increasing. The trees depicted in Fig. 1 are indeed OBDTs where the ordering is such that $x \prec y \prec z$. Applying the very same restriction to BDDs we obtain the notion of *ordered binary decision diagram*, or OBDD.

Definition 4. (BDTs and OBDTs.) *A* binary decision tree *is any string generated by the grammar*

$$\mathrm{BDT} ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{ite}(v, \mathrm{BDT}, \mathrm{BDT})$$

where $v \in \mathit{Vars}$. The set of all BDTs is denoted by $\mathcal{B}$. The semantics of BDTs is expressed by the function $[\![\cdot]\!] \colon \mathcal{B} \to \mathcal{F}$, defined as follows:

$$[\![\mathbf{0}]\!] \stackrel{\mathrm{def}}{=} \bot, \qquad [\![\mathbf{1}]\!] \stackrel{\mathrm{def}}{=} \top, \qquad [\![\mathbf{ite}(v, b_1, b_0)]\!] \stackrel{\mathrm{def}}{=} \mathit{ite}\bigl(v, [\![b_1]\!], [\![b_0]\!]\bigr),$$

where for each $w \in \mathit{Vars}$, $f_1, f_0 \in \mathcal{F}$, and each $a \in \mathcal{A}$,

$$\mathit{ite}(w, f_1, f_0)(a) \stackrel{\mathrm{def}}{=} \begin{cases} f_1(a), & \text{if } a(w) = 1; \\ f_0(a), & \text{if } a(w) = 0. \end{cases}$$

The subset $\mathcal{B}_o \subseteq \mathcal{B}$ of ordered BDTs (OBDTs) *is defined be the following recurrent equation:*

$$\mathcal{B}_o \stackrel{\mathrm{def}}{=} \{\mathbf{0}, \mathbf{1}\} \cup \left\{ \mathbf{ite}(v, b_1, b_0) \;\middle|\; \begin{array}{c} \forall i = 0, 1 : b_i \in \mathcal{B}_o \wedge \\ \exists w \in \mathit{Vars} \,.\, \exists b_1', b_0' \in \mathcal{B}_o \,. \\ b_i = \mathbf{ite}(w, b_1', b_0') \Rightarrow v \prec w \end{array} \right\}.$$

In the sequel we will deliberately confuse a BDT with the boolean function it represents. In particular, for $b \in \mathcal{B}$, when we write $\mathit{vars}(b)$ or $\mathit{true}(b)$ what we really mean is $\mathit{vars}([\![b]\!])$ or $\mathit{true}([\![b]\!])$. This convention of referring to the semantics simplifies the presentation and should not cause problems.

A *reduced ordered binary decision diagram*, or ROBDD, is an OBDD such that:

1. there are no duplicate terminal nodes;
2. there are no duplicate non-terminal nodes (i.e., nodes having the same label and the same true and false successors);
3. there are no redundant tests, that is each non-terminal node has distinct true and false successors.

Any OBDD can be converted into a ROBDD by repeatedly applying the reduction rules corresponding to the above properties: collapsing all the duplicate nodes into one and removing all the redundant tests, redirecting edges in the obvious way. Application of these rules does not change the represented functions. ROBDDs have one very important property: they are *canonical.* This means that, for each fixed variables' ordering, two ROBDDs represent the same function if and only if they are identical.

The nice computational features of ROBDDs make them suitable for implementing *Pos* (see, e.g., [16, 1]), even though ROBDDs are clearly able to represent any Boolean function. In this paper we deal formally only with OBDTs, since our results do not need all the properties of ROBDDs. Indeed, since every OBDT is an OBDD and the reduction rules do not change the represented Boolean function, everything we say about OBDTs is true also for ROBDDs.

5 Is x Ground?

Capturing dependency and disjunctive information is essential for precise groundness analysis. However, this kind of information is only needed for maintaining precise intermediate results *during* the analysis. Instead, the only information which is relevant for the user of the analysis' results is whether a certain variable is guaranteed to be ground at a certain point or not. When combinations of domains are considered, as explained in Sect. 3, it is vital to recover the set of ground variables quickly even *inside* the analysis. The problem of deriving this sure groundness information from ROBDDS has not been tackled in previous works [7, 16, 1]. Basically we know about five ways of doing that:

1. Given $x \in \mathit{Vars}$ and a ROBDD representation b of a boolean function f, use the ROBDD package to test whether $f \to x$ is a tautology, that is, whether $f \to x$ is equivalent to $\top$. This test can be performed in $O(|b|)$ time. The main advantage of this solution is that it does not require any change to the ROBDD package. One of the drawbacks is that the reduction of $f \to x$ causes the creation and disposal of "spurious" nodes.
2. Given $x \in \mathit{Vars}$ and a ROBDD representation b of a boolean function f, the information whether x is forced to 1 by f is obtained by visiting b. The answer is affirmative if (a) there is at least one node in b labeled with x; and, (b) each node in b labeled with x has its false branch equal to $\mathbf{0}$. This method has still linear complexity, requires the incorporation of the visit into the ROBDD package, and does not involve the creation of any node.
3. Another possibility is to visit the ROBDD representation b to derive, in one shot, the set G of *all* the variables which are forced to 1. We will see how this can be done in Sect. 6.
4. A variation of the previous method consists in avoiding visiting b, while obtaining exactly the same information, by modifying ROBDD's nodes so that every node records the set of variables which are forced to true by the boolean function it represents. Sect. 6 explains how this method of keeping explicit the information about true variables can be easily implemented.
5. The last method is based on a quite radical, though very simple, solution. Intuitively, it is based on the idea of keeping the information about true variables totally separate from dependency and disjunctive information. True variables are represented naturally by means of sets whereas only the dependency and disjunctive information is maintained by means of ROBDDs. This will be explained in Sect. 7.

6 Extracting Sure Groundness from ROBDDs

Here is the only property of OBDTs (and thus of ROBDDs) we need.

Definition 5. (Weak normal form.) *A BDT $b \in \mathcal{B}$ is said to be* in weak normal form *if and only if either $b = \mathbf{0}$ or $b = \mathbf{1}$, or there exist $b_1, b_0 \in \mathcal{B}$ such that $b = \mathbf{ite}(v, b_1, b_0)$, $v \notin \mathit{vars}(b_1) \cup \mathit{vars}(b_0)$, and both b_1 and b_0 are in weak normal form.*

Proposition 6. *Each OBDT $b \in \mathcal{B}_o$ is in weak normal form.*

This is indeed the distinctive property of "free BDDs" or "1-time branching programs", where no ordering is required but each path from the root is allowed to test a variable only once [6].

Theorem 7. *Let $b = \mathbf{ite}(v, b_1, b_0)$ be in weak normal form. Then we have*

$$true(b) = \begin{cases} true(b_0), & \text{if } b_1 = \mathbf{0}; \\ \{v\} \cup true(b_1), & \text{if } b_1 \neq \mathbf{0} \text{ and } b_0 = \mathbf{0}; \\ true(b_1) \cap true(b_0), & \text{otherwise.} \end{cases}$$

This theorem gives us at least two ways of deriving sure groundness information from ROBDDs. One is by implementing a post-order visit, collecting true variables as indicated. Another one, which is more in the spirit of a reactive implementation, is based on a modification of the node structure which is used to represent ROBDDs. In standard implementations, a non-terminal node n has one field $n.V$ which holds the test variable, plus two fields $n.T$ and $n.F$ which are references to the nodes which are the roots of the true and false branch, respectively. All the nodes are created by means of a function $create(v, @n_1, @n_0)$, taking a variable's symbol and two references to (already created) nodes, and returning a reference to the newly created node. We can modify this state of things by adding to the node structure a field $n.G$, containing the set of true variables for the function represented by the ROBDD rooted at n, and by modifying the creation function to initialize $n.G$ as indicated by Theorem 7.

7 A New, Hybrid Implementation for *Pos*

The observation of many constraint logic programs shows that the percentage of variables which are found to be ground during the analysis, for typical invocations, is as high as 80%. This suggests that representing *Pos* elements simply by means of ROBDDs, as in [16, 1], is probably not the best thing we can do. Here we propose a hybrid implementation where each *Pos* element is represented by a pair: the first component is the set of true variables (just as in the domain used in early groundness analyzers [17, 15]); the second component is a ROBDD. In each element of this new representation there is no redundancy: the ROBDD component does not contain any information about true variables. In fact, as we will see, the hybrid representation has the property that ROBDDs are used only in what they are good for: keeping track of dependencies and disjunctive information. True variables, instead are more efficiently represented by means of sets. The hybrid representation has two major advantages: (a) it is *reactive* in the sense of Sect. 3; and, (b) it allows for keeping the ROBDDs small, during the analysis, when many variables come out to be true, as it is often the case. Consider Fig. 1 (b): the information about y being a true variable (besides not being readily available) requires two nodes. In more involved cases, the information about trueness of a variable coming late in the ordering can be scattered over a large number of nodes. Notice that, while having many true variables, in

a straight ROBDD implementation, means that the *final* ROBDDs will be very similar to a linear chain of nodes, the intermediate steps still require the creation (and disposal) of complex (and costly) ROBDDs. This phenomenon is avoided as much as possible in the hybrid implementation.
(By $\wp_f(\mathit{Vars})$ we denote the set of all *finite* subsets of Vars.)

Definition 8. (Hybrid repr.) *The* hybrid representation *for Pos is*

$$\mathcal{G} \stackrel{\text{def}}{=} \bigl\{ \langle G, b\rangle \bigm| G \in \wp_f(\mathit{Vars}), b \in \mathcal{B}_o, \mathit{vars}(b) \cap G = \mathit{true}(b) = \emptyset \bigr\}.$$

The meaning of $\mathcal{G}$'s elements is given by the overloading $[\![\cdot]\!] \colon \mathcal{G} \to \mathcal{F}$:

$$[\![\langle G, b\rangle]\!] \stackrel{\text{def}}{=} \bigwedge(G) \wedge [\![b]\!],$$

where $\bigwedge\{x_1, \ldots, x_n\} \stackrel{\text{def}}{=} x_1 \wedge \cdots \wedge x_n$ and $\bigwedge \emptyset \stackrel{\text{def}}{=} \top$.

Now, we briefly review the operations we need over *Pos* (and thus over $\mathcal{G}$) for the purpose of groundness analysis. The constraint accumulation process requires computing the logical conjunction of two functions, the merge over different computation paths amounts to logical disjunction, whereas projection onto a designated set of variables is handled through existential quantification. Functions of the kind $x \leftrightarrow (y_1, \ldots, y_m)$, for $m \geq 0$, accommodate both abstract *mgu*s and the combination operation in domains like $\mathtt{Pat}(\mathit{Pos})$ [9]. Before introducing

$\mathcal{B}_o$ op	Complexity	Meaning	$\mathcal{G}$ op	
$b_1 \ddot{\wedge} b_2$	$O(\lvert b_1\rvert \lvert b_2\rvert)$	$[\![b_1]\!] \wedge [\![b_2]\!]$	$g_1 \otimes g_2$	
$b_1 \ddot{\vee} b_2$	$O(\lvert b_1\rvert \lvert b_2\rvert)$	$[\![b_1]\!] \vee [\![b_2]\!]$	$g_1 \oplus g_2$	
$\ddot{\exists}_{\bar{V}}\, b$	$O\bigl(\lvert b\rvert^{2^{\lvert V\rvert}}\bigr)$	$\exists_{\bar{V}} [\![b]\!]$	$\exists\!\!\exists_{\bar{V}}\, g$	Note: $V \subseteq \mathit{vars}(b)$.
$\ddot{\bigwedge}(V)$	$O(\lvert V\rvert)$	$\bigwedge(V)$		Note: $V \neq \emptyset$.
$x \ddot{\leftrightarrow} V$	$O(\lvert V\rvert)$	$x \leftrightarrow \bigwedge(V)$		Note: $x \notin V$.
$b[1/V]$	$O(\lvert b\rvert)$	$[\![b]\!][1/V]$		

Table 1. Operations defined over $\mathcal{B}_o$ and $\mathcal{G}$.

the $\mathcal{G}$'s operations we introduce, by means of Table 1, the needed operations over OBDTs and ROBDDs, their complexity and semantics, as well as the correspondent operations over $\mathcal{G}$. In the sequel we will refer to some operations on OBDTs whose meaning and complexity is specified in the table. The restriction operation $b[1/V]$ (also called *valuation* or *co-factoring*) is used for maintaining the invariant specified in Definition 8. In the definition of the abstract operators used in groundness analysis, the functions of the form $x \leftrightarrow (y_1, \ldots, y_m)$ are always *conjuncted* with some other function. For this reason we provide a family of specialized operations $(x, V) \overset{\leftrightarrow}{\otimes} \colon \mathcal{G} \to \mathcal{G}$, indexed over variables and finite sets of variables. The operation $(x, V)\overset{\leftrightarrow}{\otimes}$ builds a representation for $\bigl(x \leftrightarrow \bigwedge(V)\bigr) \land f$, given one for f.

Definition 9. (Operations over $\mathcal{G}$.) *The operation $\otimes \colon \mathcal{G} \times \mathcal{G} \to \mathcal{G}$ is defined, for each $\langle G_1, b_1\rangle, \langle G_2, b_2\rangle \in \mathcal{G}$, as follows:*

$$\langle G_1, b_1\rangle \otimes \langle G_2, b_2\rangle \stackrel{\mathrm{def}}{=} \eta\Bigl(G_1 \cup G_2, b_1\bigl[1/(G_2 \setminus G_1)\bigr] \mathbin{\ddot{\land}} b_2\bigl[1/(G_1 \setminus G_2)\bigr]\Bigr),$$

where, for each $G \in \wp_f(\mathit{Vars})$ and $b \in \mathcal{B}_o$ such that $G \cap \mathit{vars}(b) = \emptyset$,

$$\eta(G, b) \stackrel{\mathrm{def}}{=} \begin{cases} \langle G, b\rangle, & \text{if } \mathit{true}(b) = \emptyset; \\ \eta\Bigl(G \cup \mathit{true}(b), b\bigl[1/\,\mathit{true}(b)\bigr]\Bigr), & \text{otherwise.} \end{cases}$$

The join operation $\oplus \colon \mathcal{G} \times \mathcal{G} \to \mathcal{G}$ is given by

$$\langle G_1, b_1\rangle \oplus \langle G_2, b_2\rangle \stackrel{\mathrm{def}}{=} \Bigl\langle G_1 \cap G_2, \bigl(b_1 \mathbin{\ddot{\land}} \ddot{\bigwedge}(G_1 \setminus G_2)\bigr) \mathbin{\ddot{\lor}} \bigl(b_2 \mathbin{\ddot{\land}} \ddot{\bigwedge}(G_2 \setminus G_1)\bigr)\Bigr\rangle.$$

For each $\langle G, b\rangle \in \mathcal{G}$, each $V \in \wp_f(\mathit{Vars})$, and $x \in \mathit{Vars}$, the unary operations $\exists_{\bar{V}} \colon \mathcal{G} \to \mathcal{G}$ and $(x, V)\overset{\leftrightarrow}{\otimes} \colon \mathcal{G} \to \mathcal{G}$ are given by

$$\exists_{\bar{V}} \langle G, b\rangle \stackrel{\mathrm{def}}{=} \langle G \setminus V, \ddot{\exists}_{\bar{V}}\, b\rangle; \quad \text{and}$$

$$(x, V)\overset{\leftrightarrow}{\otimes}\langle G, b\rangle \stackrel{\mathrm{def}}{=} \begin{cases} \eta\Bigl(G \cup V, b\bigl[1/(V \setminus G)\bigr]\Bigr), & \text{if } x \in G; \\ \eta\bigl(G \cup \{x\}, b[1/x]\bigr), & \text{if } V \subseteq G; \\ \eta\Bigl(G, b \mathbin{\ddot{\land}} \bigl(x \mathbin{\ddot{\leftrightarrow}} (V \setminus G)\bigr)\Bigr), & \text{if } x \notin G \text{ and } V \nsubseteq G. \end{cases}$$

The following result holds almost by definition.

Theorem 10. *The operations of* Definition 9 *are well-defined. Moreover, for each $g, g_1, g_2 \in \mathcal{G}$, each $V \in \wp_f(\mathit{Vars})$, and $x \in \mathit{Vars}$,*

$$[\![g_1 \otimes g_2]\!] = [\![g_1]\!] \land [\![g_2]\!], \qquad [\![g_1 \oplus g_2]\!] = [\![g_1]\!] \lor [\![g_2]\!],$$

$$\bigl[\!\bigl[\exists_{\bar{V}}\, g\bigr]\!\bigr] = \exists_{\bar{V}} [\![g]\!], \qquad \bigl[\!\bigl[(x, V)\overset{\leftrightarrow}{\otimes} g\bigr]\!\bigr] = \bigl(x \leftrightarrow \bigwedge(V)\bigr) \land [\![g]\!].$$

Notice that $\mathcal{G}$ operations make use of the $\mathcal{B}_o$ (ROBDD) operations only when strictly necessary. When this happens, expensive operations like $\ddot{\land}$ and $\ddot{\lor}$ are invoked with operands of the smallest possible size. In particular, we exploit the fact that the restriction operation is relatively cheap. However, we cannot avoid searching for true variables, as the $\otimes$ and $(x, V)\overset{\leftrightarrow}{\otimes}$ operators need that. For this purpose, the procedure implicit in Theorem 7 comes in handy. In programs where many variables are ground the ROBDDs generated will be kept small, and so also the cost of searching will be diminished. As a final remark, observe that in a real implementation the operations which are executed can be further optimized. Without entering into details, the basic analysis step, for what concerns groundness and in a bottom-up framework, generates *macro-operations* of the form $(x_1, V_1) \overset{\leftrightarrow}{\otimes} \big((x_2, V_2) \overset{\leftrightarrow}{\otimes} \big(\cdots((x_n, V_n) \overset{\leftrightarrow}{\otimes} (g_1 \otimes g_2 \otimes \cdots \otimes g_m))\cdots\big)\big)$. These operations can be greatly simplified by first collecting all the true variables in the g_i's in one sweep (a bunch of set unions) and iterating through the $(x_i, V_i) \overset{\leftrightarrow}{\otimes}$ indexes for collecting further true variables. Then the ROBDDs which occur in the macro-operation are restricted using the collected true variables and, at the end of this process, the ROBDD package is invoked over the simplified arguments. Only then we search for further true variables in the resulting ROBDD.

8 Experimental Evaluation

The ideas presented in this paper have been experimentally validated in the context of the development of the CHINA analyzer [2]. CHINA is a data-flow analyzer for CLP($\mathcal{H}$, $\mathcal{N}$) languages (i.e. Prolog, CLP($\mathcal{R}$), `clp(FD)` and so forth) written in Prolog and C++. It performs bottom-up analysis deriving information on both call and success patterns by means of program transformations and optimized fixpoint computation techniques.

The assessment of the hybrid domain has been done in a quite radical way. In fact, we have compared the standard, pure ROBDD-based implementation of *Pos* against the hybrid domain on the following problem: *deriving, once for each clause's evaluation, a boolean vector indicating which variables are known to be ground and which are not.* This is a very minimal demand for each analysis requiring the knowledge about definitely ground variables *during* the analysis. We have thus performed the analysis of a number of programs on a domain similar to `Pat`(*Pos*) [9], switching off all the other domains currently supported by CHINA[2]. `Pat`($\Re$) is a generic structural domain which is parametric with respect to any abstract domain $\Re$. Roughly speaking, `Pat`($\Re$) associates to each variable the following information:

- a *pattern*, that is to say, the principal functor and subterms which are bound to the variable;
- the “properties” of the variable, which are delegated to the $\Re$ domain (the two implementations of *Pos*, in our case).

As reported in [8], `Pat`(*Pos*) is a very precise domain for groundness analysis.

The experimental results are reported in Table 2. The table gives, for each program, the analysis times and the number of ROBDD nodes allocated for the standard implementation (STD) and the hybrid one (HYB), respectively. It also shows the ratio STD/HYB for the above mentioned figures (S/H). The computation times have been taken on a 80486DX4 machine with 32 MB of RAM running Linux 1.3.64. The tested programs have become standard for the evaluation of data-flow analyzers. They are a cutting-stock program `CS`, the generate and test version of a disjunctive scheduling program `Disj`, a program to put boolean formulas in disjunctive normal form `DNF`, the `Browse` program `Gabriel` taken from Gabriel benchmark, an alpha-beta procedure `Kalah`, the

[2] Namely, numerical bounds and relations, aliasing, and polymorphic types

	Analysis time (sec)			N. of BDD nodes		
Program	STD	HYB	S/H	STD	HYB	S/H
`CS`	1.06	0.6	1.77	12387	391	31.7
`Disj`	1.06	0.6	1.77	72918	176	414.3
`DNF`	5.17	4.4	1.18	5782	111	52.1
`Gabriel`	1.13	0.74	1.53	28634	10472	2.73
`Kalah`	3.92	2.02	1.94	43522	645	67.5
`Peep`	6.13	5.52	1.11	176402	128332	1.37
`PG`	0.37	0.25	1.48	3732	86	43.4
`Plan`	0.59	0.5	1.18	1736	65	26.7

Table 2. Experimental results obtained with the CHINA analyzer.

peephole optimizer of SB-Prolog `Peep`, a program `PG` written by W. Older to solve a particular mathematical problem, and the famous planning program `Plan` by D.H.D. Warren.

The results indicate that the hybrid implementation outperforms the standard one in both time and space efficiency. The systematic speed-up obtained was not expected. Indeed, we were prepared to content ourselves with a moderate slow-down which would have been recovered in the reactive combinations. The space figures show that we have achieved significant (and sometimes huge) savings in the number of allocated ROBDD nodes. With the hybrid domain we are thus able to keep the ROBDDs which are created and traversed during the analysis as small as possible. This phenomenon is responsible for the speed-up. It seems that, even for programs characterized by not-so-many ground variables, there are always enough ground variables to make the hybrid implementation competitive. This can be observed, for instance, in the case of the `Peep` program, which was analyzed with a non-ground, most-general input pattern. The following additional observations are important for a full understanding of Table 2:

1. we are not comparing against a poor standard implementation of *Pos*, as can be seen by comparing the analysis times with those of [8]. The ROBDD package we are using is fine-tuned: it employs separate caches for the main operations (with hit-rates in the range 95%–99% for almost all programs), specialized and optimized versions of the important operations over ROBDDs, as well as aggressive memory allocation strategies. Indeed, we were led to the present work by the apparent impossibility of further optimizing the standard implementation. Moreover, the hybrid implementation has room for improvement, especially for what concerns the handling of bit-vectors.
2. We are not taking into account the cost of garbage-collection for ROBDD nodes. In particular, the sizes of the relevant data-structures were chosen so that the analysis of the tested programs could run to completion without any node deallocation or reallocation.

3. The boolean vectors computed during our test analyses are what is necessary for, say, the quick handling of delayed constraints and goals, and the efficient simplification of aliasing information. However, the experiment does not take into account the inevitable gains which are a consequence of the fast access to ground variables. Furthermore, in a truly reactive combination, the set of ground variables is not needed only at the end of each clause's evaluation (this is the optimistic hypothesis under which we conducted the experimentation), but at each body-atom evaluation for each clause. In this context the hybrid implementation, due to its incrementality, is even more favored with respect to the standard one (which is not incremental at all).

9 Conclusion

We have studied the problem, given an implementation of *Pos* based on ROBDDs, of determining as efficiently as possible the set of variables which are forced to true in the abstract representation. We have explained why, for the sake of realizing reactive combinations of domains, it is important to detect these variables (which correspond to *ground* ones at the concrete level) as quickly as possible. This problem has not been treated before in the literature [16, 1, 19]. After reviewing the *naïf* approaches, we have presented a simple method of detecting all the true variables in a ROBDD representation at once. We have then proposed a novel hybrid representation for Boolean functions. This representation is designed in a way to take advantage from the observation that most programs (together with their typical queries) have a high percentage of variables which are deemed to be ground at the program points of interest. With the new representation, not only the information about true (ground) variables is always readily available (instead of being scattered all over the ROBDDs), but we are also able to keep the usage of (expensive) ROBDDs at a minimum. This is clearly important for efficiency reasons. In fact, we have presented the experimental results obtained with a prototype implementation of the hybrid domain which outperforms, from any point of view, the standard implementation based on ROBDDs only. Surprisingly enough, we have thus been able to assess the superiority of the hybrid domain even for those cases where fast access to ground variables is not important.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Sondergaard. Two classes of boolean functions for dependency analysis. Technical Report 94/211, Dept. Computer Science, Monash University, Melbourne, 1994.
2. R. Bagnara. On the detection of implicit and redundant numeric constraints in CLP programs. In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Proceedings of the "1994 Joint Conference on Declarative Programming (GULP-PRODE '94)"*, pages 312–326, Peñíscola, Spain, September 1994.
3. R. Bagnara. A hierarchy of constraint systems for data-flow analysis of constraint logic-based languages. Technical Report TR-96-10, Dipartimento di Informatica, Università di Pisa, 1996. To appear on a special issue of "Science of Computer Programming".

4. R. Bagnara, R. Giacobazzi, and G. Levi. An application of constraint propagation to data-flow analysis. In *Proceedings of "The Ninth Conference on Artificial Intelligence for Applications"*, pages 270–276, Orlando, Florida, March 1993. IEEE Computer Society Press, Los Alamitos, CA.
5. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
6. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
7. A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.
8. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Conceptual and software support for abstract domain design: Generic structural domain and open product. Technical Report CS-93-13, Brown University, Providence, RI, 1993.
9. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 227–239, Portland, Oregon, January 1994.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.
11. P. Cousot and R. Cousot. Abstract interpretation and applications to logic programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
12. S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(3):418–450, 1989.
13. D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In E. Lusk and R. Overbeek, editors, *Proc. North American Conf. on Logic Programming'89*, pages 154–165. The MIT Press, Cambridge, Mass., 1989.
14. J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.
15. N. D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood Ltd, 1987.
16. B. Le Charlier and P. Van Hentenryck. Groundness analysis for Prolog: Implementation and evaluation of the domain prop. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 99–110. ACM Press, 1993.
17. C. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2:43–66, 1985.
18. V. A. Saraswat. *Concurrent Constraint Programming.* MIT Press Cambridge, Mass., 1993.
19. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain `PROP`. *Journal of Logic Programming*, 23(3):237–278, June 1995. Extended version of [16].