Factorizing Equivalent Variable Pairs in ROBDD-Based Implementations of *Pos*

Roberto Bagnara and Peter Schachte

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, and *Reduced Ordered Binary Decision Diagrams* (ROBDDs) are generally accepted to be the most efficient representation for *Pos*. Unfortunately, the size of an ROBDDs is, in the worst case, exponential in the number of variables it depends upon. Earlier work [2] has shown that a hybrid representation that separates the definite information from the dependency information is considerably more efficient than keeping the two together. The aim of the present paper is to push this idea further, also separating out certain dependency information, in particular all pairs of variables that are always either both ground or neither ground. We find that this new hybrid representation is a significant improvement over previous work.

1 Introduction

The aim of *groundness analysis* (sometimes called *definiteness* analysis) is to derive statically, for all the program points of interest, which variables are bound to unique values (or *ground*). This kind of information is very important: it allows substantial optimizations to be performed at compile-time, and is also crucial to most semantics-based program manipulation tools. Moreover, many other analyses are made 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 [10, 12]. In particular, the set of *positive Boolean functions*, (namely, those functions that assume the *true* value under the valuation assigning *true* to all variables), which is denoted by *Pos*, allows to express Boolean properties of program variables where the property of one variable may depend on that property of other variables. For groundness analysis, since variables can be bound to terms containing other variables, the groundness of one variable may depend on the groundness of other

Roberto Bagnara is with the Dipartimento di Matematica, Università di Parma, Via M. D'Azeglio 85/A, Parma, Italy. E-mail: `bagnara@prmat.math.unipr.it`. Much of his work was supported by EPSRC grant GR/L19515.

Peter Schachte is with the Department of Computer Science, The University of Melbourne, Parkville, Victoria 3052, Australia. E-mail: `pets@cs.mu.oz.au`

variables. *Pos* has been recognized as the most precise domain for capturing the kind of dependencies arising in groundness analysis.

This ability to express dependencies makes analysis based on *Pos* very precise, but also makes it relatively expensive, as many operations on Boolean formulae have exponential worst case complexity. Armstrong et al. [1] analyzed many representations of positive Boolean formulae for abstract interpretation, and found *Reduced Ordered Binary Decision Diagrams* (ROBDDs) [6] to give the best performance.

ROBDDs generated during program analysis often contain many variables that are definitely true. In the context of groundness analysis, this means that the corresponding program variable must be ground at that point in the program. It is shown in [2] that a hybrid representation for Boolean functions that keeps these definite variables separate is more efficient than ROBDDs alone. However, ROBDDs generated during program analysis also contain many pairs of variables that are equivalent. In terms of groundness, this means that either both variables are ground, or neither is. Such equivalent variables of course appear for a program goal of the form `X = Y`, but they also frequently appear naturally during the analysis process. For example, for a goal `X = [Y|Z]`, where it can be established that `Y` is ground, the analyzer will deduce that `X` and `Z` are equivalent. Such equivalent pairs can greatly increase the size of ROBDDs, which in turn makes ROBDD operations much more expensive. For example, the ROBDD for the Boolean function z comprises one node (not counting the **1** and **0** terminal nodes), while $(x \leftrightarrow y) \wedge z$ comprises 4 or 5 (usually 5). However, since $x \leftrightarrow y$ simply means that x and y are equivalent, we may remove y from the Boolean function altogether, leaving us again with a single node, and replace y by x in the formulae being analyzed. Since the time complexity of most ROBDD algorithms is at best quadratic in the sizes of the graphs involved, this can significantly speed up analysis.

In this paper we present a hybrid representation for Boolean functions that uses a set to represent definite variables, a set of pairs of equivalent variables to represent equivalences, and an ROBDD to represent more complex dependencies. This hybrid representation proves to be significantly more efficient overall than that of [2].

Notice that Boolean functions are used in the more general context of *dependency analysis*, including *finiteness analysis* for deductive database languages [5] *suspension analysis* for concurrent (constraint) logic programming languages [11], and functional dependency (or determinacy) analysis [14]. The hybrid representation we propose might be useful also in these contexts, although we have not studied this yet.

The balance of this paper proceeds as follows. In Sect. 2 we briefly review the usage of Boolean functions for groundness analysis of (constraint) logic programs (even though we assume familiarity with this subject) and we discuss the representation we use for Boolean functions. Section 3 presents our hybrid representation. Experimental results are presented in Sect. 5, and Sect. 6 concludes with some final remarks.

2 Preliminaries

Let U be a set. The set of all subsets of U will be denoted by $\wp(U)$. The set of all *finite* subsets of U will be denoted by $\wp_{\mathrm{f}}(U)$. The notation $S \subseteq_{\mathrm{f}} T$ stands for $S \in \wp_{\mathrm{f}}(T)$.

2.1 Boolean Functions for Groundness Analysis

Boolean functions allow to capture in a very precise way the *groundness dependencies* that 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, under CLP($\mathcal{R}$) $x + 2y + z = 4$ can be abstracted as $\bigl((x \wedge y) \to z\bigr) \wedge \bigl((x \wedge z) \to y\bigr) \wedge \bigl((y \wedge z) \to x\bigr)$, indicating that determining any two variables is sufficient to determine the third.

We now introduce Boolean valuations and functions in a way that is suitable for what follows. *Vars* is a fixed denumerable set of variable symbols. The variables are ordered by the total order relation $\prec$. For convenience we sometimes use $y \succ x$ as an alternative for $x \prec y$. We also use $x \preceq y$ and $y \succeq x$ to mean that either $x \prec y$ or $x = y$. We call the least variable α, that is, $\forall v \in \mathit{Vars} : \alpha \preceq v$.

We now introduce Boolean functions based on the notion of Boolean valuation.

Definition 1 (Boolean valuations.) *The set of* Boolean valuations over *Vars is given by* $\mathcal{A} \stackrel{\mathrm{def}}{=} \mathit{Vars} \to \{0, 1\}$. *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*

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

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

Definition 2 (Boolean functions.) *The set of* Boolean functions over *Vars is given by* $\mathcal{F} \stackrel{\mathrm{def}}{=} \mathcal{A} \to \{0, 1\}$. *The distinguished elements* $\top, \bot \in \mathcal{F}$ *are the functions defined by* $\top \stackrel{\mathrm{def}}{=} \lambda a \in \mathcal{A} \,.\, 1$ *and* $\bot \stackrel{\mathrm{def}}{=} \lambda a \in \mathcal{A} \,.\, 0$. *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* $f[c/x](a) \stackrel{\mathrm{def}}{=} f\bigl(a[c/x]\bigr)$. *When* $X \subseteq \mathit{Vars}$, $f[c/X]$ *is defined in the obvious way. If* $f \in \mathcal{F}$ *and* $x, y \in \mathit{Vars}$ *the function* $f[y/x] \in \mathcal{F}$ *is given, for each* $a \in \mathcal{A}$, *by* $f[y/x](a) \stackrel{\mathrm{def}}{=} f\bigl(a\bigl[a(y)/x\bigr]\bigr)$. *Boolean functions are constructed from the elementary functions corresponding to variables, and by means of the usual logical connectives. Thus* x *denotes the Boolean function* f *such that, for each* $a \in \mathcal{A}$, $f(a) = 1$ *if and only if* $a(x) = 1$. *For* $f_1, f_2 \in \mathcal{F}$, *we write* $f_1 \wedge f_2$ *to denote the function* g *such that, for each* $a \in \mathcal{A}$, $g(a) = 1$ *if and only if both* $f_1(a) = 1$ *and* $f_2(a) = 1$. *The other Boolean connectives and quantifiers are handled similarly.*

The question of whether a Boolean function f entails particular variable x (which is what, in the context of groundness analysis, we call *definite groundness information*) is equivalent to the question whether $f \to x$ is a tautology (namely, $f \to x = \top$). In what follows we will also need the notion of *dependent variables* of a function, as well as disentailed, or definitely false, variables.

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

$$\begin{aligned}
vars(f) &\stackrel{\text{def}}{=} \bigl\{\, x \in Vars \bigm| \exists a \in \mathcal{A} \,.\, f\bigl(a[0/x]\bigr) \neq f\bigl(a[1/x]\bigr) \,\bigr\},\\
true(f) &\stackrel{\text{def}}{=} \bigl\{\, x \in Vars \bigm| \forall a \in \mathcal{A} : f(a) = 1 \implies a(x) = 1 \,\bigr\},\\
false(f) &\stackrel{\text{def}}{=} \bigl\{\, x \in Vars \bigm| \forall a \in \mathcal{A} : f(a) = 1 \implies a(x) = 0 \,\bigr\},\\
equiv(f) &\stackrel{\text{def}}{=} \bigl\{\, (x, y) \in Vars^2 \bigm| x \neq y, \forall a \in \mathcal{A} : f(a) = 1 \implies a(x) = a(y) \,\bigr\}.
\end{aligned}$$

2.2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are a well-known representations of Boolean functions [6, 7]. A BDD is a rooted directed acyclic graph where each internal node is labeled with a Boolean variable and has two out edges, leading to the node's *true* and *false successors*. External (leaf) nodes are either **1** or **0**. The Boolean function represented by an BDD can be evaluated for a given truth value assignment by traversing the graph from the root node, taking the *true* edge for nodes whose label is assigned 1 and the *false* edge when the label is assigned 0. The terminal node reached in this traversal is the function value for that assignment.

When a total ordering on the variables is available, we can define *Ordered Binary Decision Diagrams* (OBDDs) as BDDs with the restriction that the label of a node is always less than the label of any internal node in its successors. *Reduced Ordered Binary Decision Diagrams* (OBDDs) are OBDDs with the additional condition that they do not contain any two distinct nodes which represent the same Boolean function. This means that the two terminal nodes must be unique, no two distinct nodes may have the same label and true and false successors, and no node may have two identical successors (because then it would represent the same Boolean function as the successors).

We now define ROBDDs formally. Although an *ROBDD* is a particular kind of rooted, directed, and acyclic graph, we prefer not to use the standard notation for graphs. Thus an ROBDD is identified with the set of its *nodes*, one of which is designated as *the root*, the edges being formally part of the nodes themselves.

Definition 4 (ROBDD) *If N is the set of nodes of an ROBDD then N satisfies*

$$N \subseteq \{\mathbf{0}, \mathbf{1}\} \cup Vars \times N \times N.$$

The nodes **0** *and* **1** *are called* terminal nodes. *All the other nodes in N are called* non-terminal nodes. *For each non-terminal node $n \in N$, $n_{\mathrm{var}} \in Vars$ denotes the variable associated with n, $n_{\mathrm{false}} \in N$ denotes the false successor of n, and $n_{\mathrm{true}} \in N$ denotes the true successor of n. With this notation, N must also satisfy the irredundancy and the ordering conditions: for each non-terminal node $n \in N$ we have $n_{\mathrm{false}} \neq n_{\mathrm{true}}$ and $\bigl(m = n_{\mathrm{false}}$ or $m = n_{\mathrm{true}}\bigr) \implies \bigl(m \in \{\mathbf{0}, \mathbf{1}\}$ or $n_{\mathrm{var}} \prec m_{\mathrm{var}}\bigr)$. Moreover, N is* rooted *and* connected*, that is, there exists $r \in N$ (the* root*) such that*

$$\forall n \in N \setminus \{r\} : \exists m \in N \,.\, \bigl(n = m_{\mathrm{false}} \text{ or } n = m_{\mathrm{true}}\bigr).$$

A ROBDDs is a pair (r, N) that satisfies the above conditions. The set of all ROBDDs is denoted by $\mathcal{D}$.

The meaning of an ROBDD is given as follows.

Definition 5 (Semantics of ROBDDs.) *The function* $[\![\cdot]\!]_{\mathcal{D}} \colon \mathcal{D} \to \mathcal{F}$ *is given, for each* $(r, N) \in \mathcal{D}$, *by*

$$[\![(r,N)]\!]_{\mathcal{D}} \stackrel{\text{def}}{=} \begin{cases} \bot, & \textit{if } r = \mathbf{0}; \\ \top, & \textit{if } r = \mathbf{1}; \\ \Bigl(r_{\text{var}} \wedge [\![(r_{\text{true}}, N)]\!]_{\mathcal{D}}\Bigr) \vee \Bigl(\neg r_{\text{var}} \wedge [\![(r_{\text{false}}, N)]\!]_{\mathcal{D}}\Bigr), & \textit{otherwise.} \end{cases}$$

We will identify an ROBDD with the ROBDD node that constitutes its root, since the set of all the nodes can be recovered by any traversal that starts from the root.

In the implementation, a new ROBDD node is created, given a label variable v and true and false successors n and m respectively, by the make_node(v, n, m) function. This is defined such that, if $n = m$, n will be returned. Furthermore, if an identical call to make_node has previously been made, the result of that call will be returned. This guarantees that if n and m are reduced, then so is the resulting node. Note that it is an error if $v \succeq n_{\text{var}}$ or $v \succeq m_{\text{var}}$.

ROBDDs have one very important property: they are *canonical*. This means that, for each fixed variable ordering, two ROBDDs represent the same function if and only if they are identical [6]. In fact, the definition of make_node is such that two ROBDDs are identical if and only if they are stored at the same memory address. This is important to the efficiency of many ROBDD operations.

We will often confuse ROBDDs with the Boolean functions they represent. For instance, for $n \in \mathcal{D}$, when we write $\mathit{vars}(n)$ or $\mathit{true}(n)$ what we really mean is $\mathit{vars}([\![n]\!]_{\mathcal{D}})$ or $\mathit{true}([\![n]\!]_{\mathcal{D}})$. This convention of referring to the semantics simplifies the presentation and should not cause problems.

3 A New Representation for *Pos*

We introduce a new representation for *Pos*. It is made up of three components: a set of *ground* variables, a set of *equivalent* variables, and an *ROBDD*, whence the name *GER representation*.[1] A set of ground variables is trivially an element of $\mathcal{V} \stackrel{\text{def}}{=} \wp_{\text{f}}(\mathit{Vars})$. For $G \in \mathcal{V}$ we define $[\![G]\!]_{\mathcal{V}} \stackrel{\text{def}}{=} \bigwedge(G)$, where $\bigwedge\{x_1, \ldots, x_n\} \stackrel{\text{def}}{=} x_1 \wedge \cdots \wedge x_n$ and $\bigwedge \varnothing \stackrel{\text{def}}{=} \top$.

The set of equivalent variables is simply given by a transitively closed set of ordered pairs of variables.

Definition 6 (A representation for equivalent variables.) *Sets of equivalent variables are represented by means of elements of* $\mathcal{L} \subseteq \wp_{\text{f}}(\mathit{Vars} \times \mathit{Vars})$ *such that*

1. $\forall L \in \mathcal{L} : \forall x, y \in \mathit{Vars} : (x,y) \in L \implies x \prec y$;
2. $\forall L \in \mathcal{L} : \forall x, y, z \in \mathit{Vars} : (x,y), (y,z) \in L \implies (x,z) \in L$.

For $L \in \mathcal{L}$ *we use the following notation:*

$$L|1 \stackrel{\text{def}}{=} \bigl\{\, x \in \mathit{Vars} \bigm| (x,y) \in L \,\bigr\}, \qquad \mathit{vars}(L) \stackrel{\text{def}}{=} L|1 \cup L|2,$$
$$L|2 \stackrel{\text{def}}{=} \bigl\{\, y \in \mathit{Vars} \bigm| (x,y) \in L \,\bigr\}.$$

The family of functions $\lambda_L \colon \mathit{Vars} \to \mathit{Vars}$ *is defined, for each* $L \in \mathcal{L}$ *and each* $x \in \mathit{Vars}$, *by* $\lambda_L(x) \stackrel{\text{def}}{=} \min\bigl(\{x\} \cup \bigl\{\, y \in \mathit{Vars} \bigm| (y,x) \in L \,\bigr\}\bigr)$. λ_L *maps each variable to the least*

[1] In [2] we had only a set of ground variables and a ROBDD.

variable of its equivalence class, which we call its leader. $(\mathcal{L}, \supseteq)$ *is clearly a lattice. We will denote the* glb *and the* lub *over* $(\mathcal{L}, \supseteq)$ *by* $\wedge_{\mathcal{L}}$ *(transitive closure of the union) and* $\vee_{\mathcal{L}}$ *(intersection), respectively. The semantics function* $[\![\cdot]\!]_{\mathcal{L}} \colon \mathcal{L} \to \mathcal{F}$ *is given by* $[\![L]\!]_{\mathcal{L}} \stackrel{\text{def}}{=} \bigwedge\{\, x \leftrightarrow y \mid (x, y) \in L \,\}$.

In the GER representation, an element of *Pos* is represented by an element of $\mathcal{V} \times \mathcal{L} \times \mathcal{D}$. There are elements of *Pos* that can be represented by several such triples and, in the GER representation, we need to make a choice among those. This choice must be *canonical* and *economical*. Economy can be explained as follows: true variables are most efficiently represented in the first component (a bit-vector at the implementation level) and should not occur anywhere else in the representation. Equivalent variables are best represented in the second component of the GER representation (implemented as a vector of integers). As equivalent variables partition the space of variables into equivalence classes, only one variable per equivalence class must occur in the ROBDD constituting the third component of the representation. If we choose, say, the least variable (with respect to the $\preceq$ ordering on *Vars*) of each equivalence class as the representative of the class, we have also ensured canonicity.

Definition 7 (GER representation.) *The* GER representation for *Pos is given by the set*

$$\mathcal{G} \stackrel{\text{def}}{=} \left\{ \langle G, L, n \rangle \;\middle|\; \begin{array}{l} G \in \mathcal{V}, L \in \mathcal{L}, n \in \mathcal{D}, \\ G \cap \mathit{vars}(L) = G \cap \mathit{vars}(n) = L|2 \cap \mathit{vars}(n) = \varnothing, \\ \mathit{true}(n) = \mathit{equiv}(n) = \varnothing \end{array} \right\}.$$

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

$$\big[\!\big[\langle G, L, n \rangle\big]\!\big]_{\mathcal{G}} \stackrel{\text{def}}{=} [\![G]\!]_{\mathcal{V}} \wedge [\![L]\!]_{\mathcal{L}} \wedge [\![n]\!]_{\mathcal{D}},$$

What is required now is a normalization function mapping each element of $\mathcal{V} \times \mathcal{L} \times \mathcal{D}$ into the right representative in $\mathcal{G}$.

Definition 8 (Normalization function η.) *The function* $\eta \colon \mathcal{V} \times \mathcal{L} \times \mathcal{D} \to \mathcal{V} \times \mathcal{L} \times \mathcal{D}$ *is given by* $\eta\big(\langle G, L, n \rangle\big) \stackrel{\text{def}}{=} \langle \hat{G}, \hat{L}, \hat{n} \rangle$ *where*

$$\begin{aligned}
\hat{G} &\stackrel{\text{def}}{=} \mathit{true}\Big(\big[\!\big[\langle G, L, n \rangle\big]\!\big]_{\mathcal{G}}\Big), \\
\hat{L} &\stackrel{\text{def}}{=} \mathit{equiv}\Big(\big[\!\big[\langle G, L, n \rangle\big]\!\big]_{\mathcal{G}}\Big) \setminus \{\, (x, y) \in \hat{G}^2 \mid x \prec y \,\}, \\
\hat{n} &\stackrel{\text{def}}{=} n[1/\hat{G}]\big[\lambda_{\hat{L}}(x_1)/x_1\big] \cdots \big[\lambda_{\hat{L}}(x_n)/x_n\big], \quad \text{if } \mathit{vars}(n) \setminus \hat{G} = \{x_1, \ldots, x_n\}.
\end{aligned}$$

A very basic implementation for η is given by the normalize function depicted in Alg. 1. The need for looping can be understood by means of the following examples. Forcing a variable to true in a ROBDD can result in new entailed variables: if n represents $x \to y$ then $n[1/x]$ represents y. Renaming a ROBDD node n by means of a set of equivalent variables L can also give rise to new entailed variables. Suppose that n represents the Boolean formula $x \vee y$ and that $L = \big\{(x, y)\big\}$. Then $n\big[\lambda_L(y)/y\big]$ represents x. Renaming can also result in new equivalent variables: take n representing $x \leftrightarrow (y \wedge z)$ and $L = \big\{(y, z)\big\}$ for an example.

Require: an element $\langle G, L, n\rangle \in \mathcal{V} \times \mathcal{L} \times \mathcal{D}$

function normalize$\bigl(\langle G, L, n\rangle\bigr)$

$G_{\mathrm{new}} := G$; $L_{\mathrm{new}} := L$; $n_{\mathrm{new}} := n$;
repeat
  $G_{\mathrm{old}} := G_{\mathrm{new}}$; $L_{\mathrm{old}} := L_{\mathrm{new}}$; $n_{\mathrm{old}} := n_{\mathrm{new}}$;
  $G_{\mathrm{new}} := G_{\mathrm{new}} \cup \bigl\{\, x, y \bigm| (x, y) \in L_{\mathrm{new}}, \{x, y\} \cap G_{\mathrm{new}} \neq \varnothing \,\bigr\}$
  $L_{\mathrm{new}} := L_{\mathrm{new}} \setminus \bigl\{\, (x, y) \in G_{\mathrm{new}}^2 \bigm| x \prec y \,\bigr\}$
  $n_{\mathrm{new}} := n_{\mathrm{new}}[1/G_{\mathrm{new}}]$;
  $G_{\mathrm{new}} := G_{\mathrm{new}} \cup \mathit{true}(n_{\mathrm{new}})$;
  $L_{\mathrm{new}} := L_{\mathrm{new}} \wedge_{\mathcal{L}} \mathit{equiv}(n_{\mathrm{new}})$;
  $\{x_1, \ldots, x_k\} := \mathit{vars}(n_{\mathrm{new}})$;
  $n_{\mathrm{new}} := n_{\mathrm{new}}\bigl[\lambda_{L_{\mathrm{new}}}(x_1)/x_1\bigr] \cdots \bigl[\lambda_{L_{\mathrm{new}}}(x_k)/x_k\bigr]$
until $G_{\mathrm{new}} = G_{\mathrm{old}}$ and $L_{\mathrm{new}} = L_{\mathrm{old}}$ and $n_{\mathrm{new}} = n_{\mathrm{old}}$;
return $\langle G_{\mathrm{new}}, L_{\mathrm{new}}, n_{\mathrm{new}}\rangle$;

Algorithm 1: The normalize function.

Theorem 9 *We have that* $\eta \colon \mathcal{V} \times \mathcal{L} \times \mathcal{D} \to \mathcal{G}$. *Furthermore, for each triple* $\langle G, L, N\rangle \in \mathcal{V} \times \mathcal{L} \times \mathcal{D}$, *we have* $\bigl[\!\bigl[\langle G, L, N\rangle\bigr]\!\bigr]_{\mathcal{G}} = \bigl[\!\bigl[\eta(\langle G, L, N\rangle)\bigr]\!\bigr]_{\mathcal{G}}$. *Finally, the* normalize *function in* Alg. 1 *is a correct implementation of* η.

It is important to remark that in the actual implementation several specializations are used instead of Alg. 1. In other words, for every possible use of normalize, conditions can be granted so as to use a simpler algorithm instead. While space limitations do not allows us to be more precise, we just observe that roughly 50% of the times normalize would be called with the ROBDD **1**. This indicates that definitely ground variables and equivalent variables constitute a significant proportion of the dependencies that arise in practice.

3.1 Operations for the Analysis

Let us briefly review the operations we need over *Pos* for the purpose of groundness analysis. Modeling forward execution of (constraint) logic programs 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. Conjunction with functions of the form $x \leftrightarrow (y_1 \wedge \cdots \wedge y_k)$, for $k \geq 0$, accommodate both abstract *mgu*s and the *combination* operation in domains like $\mathtt{Pat}(\mathit{Pos})$ [9].

Let Ω be an operation over *Pos*. The corresponding operation over $\mathcal{G}$ can be specified, roughly speaking, as $\eta \circ \Omega \circ [\![\cdot]\!]_{\mathcal{G}}$. However, this is simply a specification: the problem is how to compute $\eta \circ \Omega \circ [\![\cdot]\!]_{\mathcal{G}}$ more efficiently exploiting the fact that both definitely ground variables and pair of equivalent variables are kept separate in the GER representation. The intuitive recipe (which has been extensively validated through experimentation) for achieving efficiency can be synthesized in the *motto* “keep the ROBDD component as small as possible”. The specification above does the contrary: it pushes all the information into the ROBDD component, performs the operation on the ROBDD, and normalizes the result. Let us take the conjunction operation $\wedge_{\mathcal{G}} \colon \mathcal{G} \times \mathcal{G} \to \mathcal{G}$ and suppose we want to compute $\langle G_1, L_1, n_1\rangle \wedge_{\mathcal{G}} \langle G_2, L_2, n_2\rangle$. A first approximation is to compute

$$\eta\bigl(\langle G_1 \cup G_2, L_1 \wedge_{\mathcal{L}} L_2, n_1 \wedge_{\mathcal{D}} n_2\rangle\bigr), \tag{1}$$

but we can do better if we reduce the ROBDDs n_1 and n_2 before computing the conjunction (whose complexity is $O(|n_1| \cdot |n_2|)$, where $|n|$ denotes the number of nodes in the ROBDD n). In order to apply the $\wedge_{\mathcal{D}}$ operator to the smallest possible ROBDD nodes we can use the alternative expression

$$\eta\Bigl(\bigl\langle G_1' \cup G_2', L_1' \wedge_{\mathcal{L}} L_2', n_1' \wedge_{\mathcal{D}} n_2'\bigr\rangle\Bigr), \tag{2}$$

where $\langle G_i', L_i', n_i'\rangle = \eta\bigl(\langle G_1 \cup G_2, L_1 \wedge_{\mathcal{L}} L_2, n_i\bigr)$, for $i = 1, 2$. For lack of space we cannot enter into details, but the current implementation uses an expression which is intermediate between (1) and (2). Indeed, the attentive reader will have noticed that there is a tradeoff in the above *motto*: keeping the ROBDDs as small as possible, as in (2), implies performing several (possibly fruitless) visits of the ROBDDs in order to collect entailed and equivalent variables.

Disjunction is computationally less complex than conjunction in that it does not require normalization through η. This, however, comes at the price of some extra complication in the definition.

$$\langle G_1, L_1, n_1\rangle \vee_{\mathcal{G}} \langle G_2, L_2, n_2\rangle \stackrel{\text{def}}{=} \langle G_1 \cap G_2, L', n_1' \vee_{\mathcal{D}} n_2'\rangle,$$

with $L' \stackrel{\text{def}}{=} L_1' \vee_{\mathcal{L}} L_2'$ and

$$\begin{aligned}
L_1' &\stackrel{\text{def}}{=} L_1 \wedge_{\mathcal{L}} \bigwedge\nolimits_{\mathcal{L}}^{\substack{(x,y)\in G_1\setminus G_2\\ x\prec y}} \bigl\{(x,y)\bigr\}, &\qquad L_2' &\stackrel{\text{def}}{=} L_2 \wedge_{\mathcal{L}} \bigwedge\nolimits_{\mathcal{L}}^{\substack{(x,y)\in G_2\setminus G_1\\ x\prec y}} \bigl\{(x,y)\bigr\},\\
G_1' &\stackrel{\text{def}}{=} \bigl\{\, \lambda_{L'}(x) \bigm| x \in G_1 \setminus G_2 \,\bigr\}, & G_2' &\stackrel{\text{def}}{=} \bigl\{\, \lambda_{L'}(x) \bigm| x \in G_2 \setminus G_1 \,\bigr\},\\
L_1'' &\stackrel{\text{def}}{=} (L_1' \setminus L_2') \vee_{\mathcal{L}} L_1, & L_2'' &\stackrel{\text{def}}{=} (L_2' \setminus L_1') \vee_{\mathcal{L}} L_2,\\
n_1' &\stackrel{\text{def}}{=} n_1 \wedge_{\mathcal{D}} \bigwedge\nolimits_{\mathcal{D}}^{x\in G_1'} x \wedge_{\mathcal{D}} \bigwedge\nolimits_{\mathcal{D}}^{(x,y)\in L_1''} x \leftrightarrow y, & n_2' &\stackrel{\text{def}}{=} n_2 \wedge_{\mathcal{D}} \bigwedge\nolimits_{\mathcal{D}}^{x\in G_2'} x \wedge_{\mathcal{D}} \bigwedge\nolimits_{\mathcal{D}}^{(x,y)\in L_2''} x \leftrightarrow y.
\end{aligned}$$

For $i = 1, 2$, L_i' contains the equivalent variables in L_i plus those implied by the groundness not shared by the two representations that are about to be disjoined (since $x \wedge y$ implies $x \leftrightarrow y$). Thus L' contains the common equivalent pairs. For $i = 1, 2$, G_i' contains the non-common ground variables to be restored into the ROBDD components, taking into account the common equivalences. Similarly, L_i'' contains the non-common equivalences to be restored into the respective ROBDD: notice that, for $x, y \in G_i \setminus G_{(i \bmod 2)+1}$, care is taken not to restore both $x \wedge y$ and $x \leftrightarrow y$.

For the projection operation over $\mathcal{G}$, which is indeed quite simple, we refer the reader to [4].

4 Some Specialized Algorithms

In order to implement the normalize function, its specializations, and the other operations for the analysis, we need efficient algorithms for several operations. Algorithms for finding all the variables entailed in an ROBDD have been presented in [2, 13], while the operation $n[1/V]$ (called *valuation* or *co-factoring*) can be easily implemented as described in [7].

4.1 Finding Equivalent Variables in ROBDDs

An algorithm for finding all the pairs of variables in an ROBDD that are equivalent is presented as Alg. 2. The algorithm follows directly from the following theorem.

Require: an ROBDD node n
function equiv_vars(n)
equiv_vars_aux($n, \{\langle x, y\rangle : \alpha \preceq x \prec y \preceq \max vars(n)\}$)

function equiv_vars_aux(n, U)
if $n = \mathbf{1}$ **then**
$\varnothing$
else if $n = \mathbf{0}$ **then**
U
else
$$\left\{ \langle n_{\text{var}}, v\rangle \;\middle|\; v \in \big(\text{vars\_entailed}(n_{\text{true}}) \cap \text{vars\_disentailed}(n_{\text{false}})\big) \right\}$$
$$\cup\big(\text{equiv\_vars\_aux}(n_{\text{true}}, U) \cap \text{equiv\_vars\_aux}(n_{\text{false}}, U)\big)$$

Algorithm 2: The equiv_vars function.

Theorem 10 $[\![n]\!]_{\mathcal{D}}$ *entails* $x \leftrightarrow y$ *where* $x \prec y$ *if and only if* $n = \mathbf{0}$*, or* $n_{\text{var}} = x$ *and* $[\![n_{\text{true}}]\!]_{\mathcal{D}}$ *entails* y *and* $[\![n_{\text{false}}]\!]_{\mathcal{D}}$ *disentails* y*, or* $n_{\text{var}} \prec x$ *and* $[\![n_{\text{true}}]\!]_{\mathcal{D}}$ *and* $[\![n_{\text{false}}]\!]_{\mathcal{D}}$ *both entail* $x \leftrightarrow y$*.*

We refer the reader to [2, 13] for the possible implementations of vars_entailed (and, by duality, of vars_disentailed). Observe that a crucial ingredient for the efficiency of the implementation is caching the results of the calls to equiv_vars, vars_entailed, and vars_disentailed.

4.2 Removing Equivalent Variables

Once we have identified which variables are equivalent to which others, we can significantly reduce the size of an ROBDD by removing all but one of each equivalence class of variables. Defining the leader function for an ROBDD node n as

$$\lambda_n \stackrel{\text{def}}{=} \lambda_{equiv(n)},$$

our aim is to restrict away all but the first variable in each equivalence class, that is, all variables v such that $\lambda_n(v) \neq v$. To motivate the algorithm, we begin with a simple theorem.

Theorem 11 *Given an ROBDD rooted at* n*, and its corresponding leader function* λ_n*, for every node* $m \neq n$ *appearing in the ROBDD such that* $\lambda_n(m_{\text{var}}) = n_{\text{var}}$*, we have either* $m_{\text{true}} = \mathbf{0}$ *or* $m_{\text{false}} = \mathbf{0}$*.*

We "remove" a variable from a Boolean function using existential quantification. For an ROBDD node m, removing m_{var} leaves disjoin$(m_{\text{true}}, m_{\text{false}})$. So Theorem 11 tells us that when $\lambda_n(m_{\text{var}}) \neq m_{\text{var}}$, either m_{true} or m_{false} will be $\mathbf{0}$, making the disjunction trivial.

This suggests the algorithm shown as Algorithm 3 for removing all the "unneeded" variables in an ROBDD n given its leader function λ_n. Two obvious optimizations of this algorithm immediately suggest themselves. Firstly, we may easily compute the last variable (in the ordering) z such that $\lambda_n(z) \neq z$; we may then add the case **else if** $n_{\text{var}} > z$ **then** n immediately after the initial **if**. The second and more important optimization is to avoid recomputing the squeeze_equiv function by the usual caching technique, returning the result of an earlier call with the same arguments. Since the λ_n function is the same in all recursive calls to squeeze_equiv, we may simplify this by clearing our table of previous results whenever squeeze_equiv is called non-recursively (from outside). This allows us to use only the n argument as a parameter to this cache.

```
Require: an ROBDD node n and a leader function λ
  function squeeze_equiv(n, λ)
  if is_terminal(n) then
    n
  else if λ(n_var) = n_var then
    make_node(n_var, squeeze_equiv(n_true, λ), squeeze_equiv(n_false, λ))
  else if n_true = 0 then
    squeeze_equiv(n_false, λ)
  else
    squeeze_equiv(n_true, λ)
```

Algorithm 3: The squeeze_equiv function.

When we conjoin two Boolean functions in their GER representation, we also have the opportunity to use the variable equivalences of each argument to reduce the size of the ROBDD component of the *other* argument. In order to do this, we need an algorithm to compute, given any ROBDD m and equivalent variable set L, the ROBDD n whose semantics is

$$[\![n]\!]_{\mathcal{D}} = \exists L|2 \,.\, [\![L]\!]_{\mathcal{L}} \wedge [\![m]\!]_{\mathcal{D}}.$$

Space limitations preclude a full exposition of this algorithm, but it may be found in [4].

5 Experimental Evaluation

The ideas presented in this section have been experimentally validated in the context of the development of the China analyzer [3]. 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 C++ and Prolog. It performs bottom-up analysis deriving information about success-patterns and, optionally, call-patterns by means of program transformations and optimized fixpoint computation techniques. We have performed the analysis of a suite comprising 170 programs on the domain Pattern(*Pos*) (similar to `Pat`(*Pos*) [3]), switching off all the other domains currently supported by China,[2] and switching off the widening operations normally used to throttle the complexity of the analysis.

A selection of the experimental results is reported in Tables 1 and 2. These tables give, for each program, the analysis times and the number of ROBDD nodes allocated for the standard implementation based on ROBDDs only, but making use of the optimized algorithms described in [13] (R), for the implementation where definitely ground variables

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

	Goal independent			Goal dependent		
Program	R	GR	GER	R	GR	GER
`action.pl`	1.59	1.58	0.17	3.21	2.78	1.44
`bp0-6.pl`	0.18	0.09	0.04	0.18	0.06	0.07
`bridge.clpr`	0.3	0.33	0.11	0.1	0.02	0.02
`chat_parser.pl`	∞	∞	0.54	∞	∞	2.11
`critical.clpr`	0.18	0.17	0.03	∞	∞	0.14
`cs2.pl`	0.11	0.09	0.04	0.08	0.03	0.04
`csg.clpr`	0.11	0.11	0.01	0.06	0.04	0.02
`ime_v2-2-1.pl`	0.28	0.19	0.08	0.53	0.2	0.12
`kalah.pl`	0.23	0.1	0.05	0.24	0.09	0.12
`log_interpreter.pl`	0.51	0.43	0.17	2.95	2.56	0.6
`mixtus.pl`	∞	∞	1.28	0.01[3]	0.01	0.01
`peval.pl`	0.87	0.73	0.31	1.97	1.58	0.55
`press.pl`	0.09	0.08	0.06	0.72	0.55	0.38
`read.pl`	0.41	0.16	0.1	0.76	0.54	0.24
`reducer.pl`	0.11	0.1	0.07	0.9	0.83	0.25
`rubik.pl`	∞	∞	0.13	∞	∞	0.64
`scc.pl`	∞	∞	0.62	1.04	0.15	0.14
`sdda.pl`	0.11	0.09	0.03	1.94	1.47	0.16
`sim_v5-2.pl`	0.24	0.21	0.19	0.37	0.25	0.29
`simple_analyzer.pl`	∞	∞	0.16	∞	∞	4.2
`unify.pl`	1.39	0.66	0.14	∞	∞	0.78

Table 1: Results obtained with China: analysis time in seconds.

are factored out from the ROBDDs as explained in [2] (GR), and for the implementation based on the ideas presented in this paper (GER). The analysis has been considered impractical (and thus stopped) as soon as the amount of memory used by China exceeded 16 MB (for medium sized programs this corresponds to roughly 320.000 ROBDD nodes). This is indicated by ∞ in Table 1 and by ✠ in Table 2.

The computation times have been taken on a Pentium II machine clocked at 233MHz, with 64 MB of RAM, and running Linux 2.0.32.

As it can be seen from the tables, the proposed technique improves the state-of-the-art of groundness analysis with Pattern(*Pos*) considerably. Programs that were out of reach for previous imlementations are now analyzable in reasonable time, while for most other programs the measured speedup is between a factor of 2 and an order of magnitude. As far as the the memory requirements of the analysis are concerned, the new representation allows for big savings, as indicated by Table 2. Comparing the results with those of [8, page 45], and scaling the timings in order to account for the difference in performance between a PentiumII at 233MHz and a Sun SparcStation 10/30, it can be seen that we have significantly pushed forward the practicality of *Pos*.

It is worth noticing that while the analyses based on Pattern(*Pos*) are computationally more complex than those simply based on *Pos* (Cortesi et al. measured a slowdown of around 20), they are also significantly more precise [8].

[3]Just the time to notify the user about the presence of an instance of `call/1` where the analyzer is unable to derive anything about the possible arguments.

Program	Goal independent R	GR	GER	Goal dependent R	GR	GER
`action.pl`	228913	228027	6301	186745	173167	20861
`bp0-6.pl`	33838	12694	2016	12162	1219	103
`bridge.clpr`	14765	14762	6324	4044	3243	2174
`chat_parser.pl`	✠	✠	17291	✠	✠	26634
`critical.clpr`	14824	14284	1893	✠	✠	7846
`cs2.pl`	16044	11359	1698	4425	214	64
`csg.clpr`	317	106	23	196	30	27
`ime_v2-2-1.pl`	42088	21210	3336	59203	20693	2634
`kalah.pl`	42008	10962	2253	8487	322	114
`log_interpreter.pl`	61249	50083	3070	213388	167080	9354
`mixtus.pl`	✠	✠	22527			
`peval.pl`	96883	75218	14256	190905	147545	20357
`press.pl`	11433	8845	2007	33080	23773	3401
`read.pl`	49710	14883	2108	55804	32764	3095
`reducer.pl`	13534	11542	2435	92485	87306	7317
`rubik.pl`	✠	✠	3825	✠	✠	5261
`scc.pl`	✠	✠	16751	82788	5762	215
`sdda.pl`	19561	14360	786	201732	157191	2798
`sim_v5-2.pl`	18600	13073	4969	5958	319	120
`simple_analyzer.pl`	✠	✠	6772	✠	✠	65125
`unify.pl`	188476	94923	9569	✠	✠	25679

Table 2: Results obtained with China: number of BDD nodes.

6 Conclusion

We have studied the problem of efficient dependency analysis, and in particular groundness analysis, of (constraint) logic programs, using the *Pos* domain. As others have concluded that ROBDDs are the most efficient representation for use in this sort of analysis, we have concentrated on improving the efficiency of the operations needed during program analysis for ROBDDs. However, since many ROBDD operations have super-linear time cost, we sought to reduce the size of the ROBDDs being manipulated by removing certain information from the ROBDDs and representing it in a way specialized to its nature. We remove definite variables as in [2], storing them in a bit vector. The main accomplishment of this work, however, has been to remove all pairs of equivalent variables, storing them as an array of variable numbers. We have shown how this new hybrid representation significantly decreases the size of the ROBDDs being manipulated. More importantly, analysis times are significantly improved beyond the significant speedup achieved in [2].

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. A reactive implementation of *Pos* using ROBDDs. In H. Kuchen and S. D. Swierstra, editors, *Programming Languages: Implementations, Logics and Programs, Proceedings of the Eighth International Symposium*, volume 1140 of *Lecture Notes in Computer Science*, pages 107–121, Aachen, Germany, 1996. Springer-Verlag, Berlin.

[3] R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages.* PhD thesis, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125 Pisa, Italy, March 1997. Printed as Report TD-1/97.

[4] R. Bagnara and P. Schachte. Efficient implementation of *Pos*. Technical Report 98/5, Department of Computer Science, The University of Melbourne, Australia, 1998.

[5] P. Bigot, S. K. Debray, and K. Marriott. Understanding finiteness analysis using abstract interpretation. In K. Apt, editor, *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 735–749, Washington, USA, 1992. The MIT Press.

[6] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[7] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[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, 1994.

[10] P. W. Dart. *Dependency Analysis and Query Interfaces for Deductive Databases.* PhD thesis, The University of Melbourne, Department of Computer Science, 1988. Printed as Technical Report 88/35.

[11] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence and concurrent constraint programming. In V. S. Alagar and M. Nivat, editors, *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology (AMAST'95)*, volume 936 of *Lecture Notes in Computer Science*, pages 531–545. Springer-Verlag, Berlin, 1995.

[12] K. Marriott and H. Søndergaard. Notes for a tutorial on abstract interpretation of logic programs. North American Conference on Logic Programming, Cleveland, Ohio, USA, 1989.

[13] P. Schachte. Efficient ROBDD operations for program analysis. In K. Ramamohanarao, editor, *ACSC'96: Proceedings of the 19th Australasian Computer Science Conference*, pages 347–356. Australian Computer Science Communications, 1996.

[14] J. Zobel. *Analysis of Logic Programs.* PhD thesis, The University of Melbourne, 1990.