Positive Boolean Functions as Multiheaded Clauses

Jacob M. Howe*[1] and Andy King[2]

[1]Department of Computing, City University, London, UK
[2]Computing Laboratory, University of Kent, Canterbury, UK
{j.m.howe, a.m.king}@ukc.ac.uk

Abstract. Boolean functions are ubiquitous in the analysis of (constraint) logic programs. The domain of positive Boolean functions, Pos, has been used for expressing, for example, groundness, finiteness and sharing dependencies. The performance of an analyser based on Boolean functions is critically dependent on the way in which the functions are represented. This paper discusses multiheaded clauses as a representation of positive Boolean functions. The domain operations for multiheaded clauses are conceptually simple and can be implemented straightforwardly in Prolog. Moreover these operations generalise those for the less algorithmically complex operations of propositional Horn clauses, leading to naturally stratified algorithms. The multiheaded clause representation is used to build a Pos-based groundness analyser. The analyser performs surprisingly well and scales smoothly, not requiring widening to analyse any program in the benchmark suite.
Keywords. Abstract interpretation, (constraint) logic programs, Boolean functions, groundness analysis.

1 Introduction

Boolean functions play an important role in the practice of static analysis. Many analyses are couched in terms of Boolean functions, and manipulation of these functions is crucial to the performance of any implementation. In particular, positive Boolean functions have been applied to the analysis of logic programs for properties such as groundness, rigidity [15], finiteness [3] and sharing [8]. This paper advocates representing positive Boolean functions as multiheaded clauses and argues that Prolog is well suited to their manipulation.

The choice of abstract domain for a particular application involves the striking of a balance between efficiency and precision. The various properties tracked using positive Boolean functions give rise in practice to different forms of Boolean function. Hence, in some applications, restricting to a more computationally tractable subclass of Pos can have a significant impact on precision (for example, goal-independent analysis of library code), whilst in others little precision is lost (for example, goal-dependent groundness analysis). Elsewhere, the authors

* Work supported by EPSRC Grant GR/MO8769.

have discussed various subclasses of Pos and their computational properties [17, 19]. Here, with an eye to a wider range of applications, the authors adapt techniques from these subclasses to Pos.

Traditionally, Boolean function manipulation has been performed using binary decision diagrams (BDDs). Groundness analysis is one of the most important topics in the static analysis of (constraint) logic programs and from a logic programming point of view this analysis is the most practical test of Boolean function manipulation. BDD-based analysers have consistently outperformed those based on other representations of Boolean functions [1, 2, 10, 24] for groundness analysis, but there has been a continuous stream of work on representations amenable to Prolog implementation [7], in particular for the subclass of definite positive functions, Def [12, 13, 19]. The majority of these implementations, included those based on BDDs, require widening to analyse large benchmarks.

The Def-based groundness analyser described in [19] does not require widening and was designed so that the most frequently called domain operations are the most lightweight. The same design methodology suggests that a Pos-based analyser should represent Boolean functions as conjunctions of multiheaded clauses. In fact, in [1] (reduced) conjunctive normal form, (R)CNF, was investigated, and "performed reasonably well", but was ultimately rejected since BDDs performed 40% faster and, in C (their implementation language), conjunctive normal form is no easier to code than BDDs. Surprisingly, conjunctive normal forms have not been considered since. This paper revisits clausal representations of Pos since, in Prolog, clausal representations are much easier to code than BDDs and following the methodology of [19] the clausal representation lends itself to efficient implementation based on entailment checking.

The importance of the choice of representation is clearly illustrated by the subtle difference between multiheaded clauses and RCNF. The RCNF representation is reduced in the sense that no clause subsumes another. This reduction makes meet for RCNF quadratic in the size of the representation. The multiheaded clause representation may contain redundant clauses, enabling meet to be constant time. This is an important issue for performance since meet is by far the most frequently applied operation. Neither multiheaded clause nor RCNF representations are in a canonical form, therefore equivalence cannot be detected by straightforward syntactic identity. In [1] equivalence for RCNF is determined by computing the dual Blake canonical form of the formulae and then testing for syntactic identity. The dual Blake canonical form may be exponentially larger than the RCNF representation and must always be completely computed. Therefore the method is not amenable to filtering through lower complexity algorithms. Logical entailment, rather than syntactic equivalence, is more flexible. In practice, entailment of formulae can often be detected using an incomplete low complexity algorithm. Using such a check, many calls to the worst case algorithm can be filtered out. It is this stratified use of entailment checking that enables an analyser based on multiheaded clauses to scale surprisingly well. Speed is achieved by exploiting Prolog technology – by using a nonground representation entailment checking can be implemented efficiently using renaming and block declarations, whilst meet reduces to list concatenation implemented using difference lists. The major themes and contributions of this work are:

- Pos functions can be naturally expressed as multiheaded clauses, which are particularly straightforward to understand, manipulate and code.
- The entailment checking algorithm (which is potentially exponential in the number of variables) is stratified so that checks for naturally occurring subclasses of formulae take quadratic time (in the size of the formulae); in particular the forward chaining algorithm for propositional Horn clauses is subsumed.
- The domain operations for multiheaded clauses may be coded succinctly and efficiently in Prolog, resulting in fast Pos-based goal-dependent and goal-independent groundness analysers which do not require widening for any program in the benchmark suite.
- If widening is required, the representation may be simply and naturally widened to Def or to the simpler domain EPos.
- The analysers again demonstrate the value of a principled approach to the design of a static analysis.
- An experimental evaluation of the analysers is given illustrating that a clausal representation of Pos coded in Prolog gives performances comparable to BDD representations coded in C.

The rest of this paper is structured as follows: Section 2 introduces the necessary technical background material. Section 3 details multiheaded clauses. Section 4 gives algorithms for the abstract operations of Pos represented as multiheaded clauses. Section 5 describes Pos-based groundness analysers implemented with Boolean functions represented as multiheaded clauses. Section 6 gives an experimental evaluation of these analysers. Section 7 reviews related work and Section 8 concludes.

2 Preliminaries

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

Definition 1. The map $model_X : \mathsf{Bool}_X \to \wp(\wp(X))$ is defined by: $model_X(f) = \{M \subseteq X \mid \psi_X(M) \models f\}$. Also, $countermodel_X : \mathsf{Bool}_X \to \wp(\wp(X))$ is defined by: $countermodel_X(f) = \wp(\wp(X)) \setminus model_X(f)$. Observe that $model_X$ is bijective, hence $model_X^{-1} : \wp(\wp(X)) \to \mathsf{Bool}_X$ is well defined.

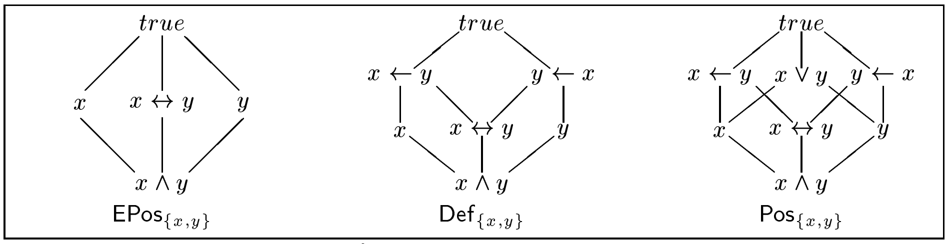

Fig. 1: Hasse diagrams

Example 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 $model_X(x \vee y) = \{\{x\}, \{y\}, \{x, y\}\}$.

The focus of this paper is on the use of subclasses of Bool_X in tracing dependencies. These subclasses are defined below:

Definition 2. A function f is positive iff $X \in model_X(f)$. Pos_X is the set of positive Boolean functions over X. A function f is definite iff $M \cap M' \in model_X(f)$ for all $M, M' \in model_X(f)$. Def_X is the set of positive functions over X that are definite. A function f is GE iff f is definite positive and for all $M, M' \in model_{var(f)}(f)$, $|M \backslash M'| \neq 1$. EPos_X is the set of GE functions over X.

Note that $\mathsf{EPos}_X \subseteq \mathsf{Def}_X \subseteq \mathsf{Pos}_X$. Also notice that $\mathsf{EPos}_X = \{\wedge F \mid F \subseteq X \cup E_X\}$, where $E_X = \{x \leftrightarrow y \mid x, y \in X\}$.

Example 2. Suppose $X = \{x, y, z\}$ and consider the following table, which states, for some Boolean functions, whether they are in EPos_X, Def_X or Pos_X and also gives $model_X$.

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

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

The 4-tuple $\langle \mathsf{Pos}_X, \models, \wedge, \vee\rangle$ is a finite lattice, where $true$ is the top element and $\wedge X$ is the bottom element. The set of (free) variables in a syntactic object o is denoted by $\mathrm{var}(o)$. Existential quantification is defined by Schröder's Elimination Principle, that is, $\exists x.f = f[x \mapsto true] \vee f[x \mapsto false]$. 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 $\exists var(f) \setminus Y.f$. Two functions f, f' are equivalent, $f \equiv f'$ if and only if $f \models f'$ and $f' \models f$. Finally, for any $f \in \mathsf{Bool}_X$, $coneg(f) = model_X^{-1}(\{X \setminus M \mid M \in model_X(f)\})$.

3 Pos as Multiheaded Clauses

A Boolean function is positive if and only if every clause in its conjunctive normal form representation contains at least one positive literal. A clause is described as multiheaded if it contains one or more positive literals. In this paper, multiheaded clauses are written as implications with the body a conjunction of variables and the head a disjunction of variables. That is, a multiheaded clause has the form:

$$y_1 \wedge ... \wedge y_n \rightarrow x_1 \vee .. \vee x_m$$

Observe that the y_i and the x_j are distinct variables, otherwise the clause is equivalent to *true*. Let $f \in \mathrm{MHC}$ denote that f is represented as a conjunction of multiheaded clauses.

Proposition 1. For every $f \in \mathsf{Pos}$ there is $f' \in \mathrm{MHC}$ such that $f \equiv f'$.

Proof. It is well known that any Boolean formula is equivalent to another in conjunctive normal form. Suppose $f \equiv f'$, where f' is in conjunctive normal form. Since f is positive, every clause of f' must contain at least one positive literal, hence $f' \in \mathrm{MHC}$. ∎

In the case that $m = 1$ the multiheaded clause is simply a propositional Horn clause. This suggests that the algorithms to calculate the domain operations might perform well if they naturally specialise to efficient propositional Horn clause algorithms. This will be the case for entailment checking. Moreover, the multiheaded clauses representation is particularly amenable to widening. If widening is required, the representation may be restricted in linear time so that clauses with more than, say n, heads are discarded. If $n = 1$, this widening corresponds to restricting to Def.

4 Domain Operations for Multiheaded Clauses

This section gives algorithms for the domain operations of Pos represented as multiheaded clauses. Meet ($\wedge$) is simply conjunction of clauses and is constant time; the other domain operations described are join ($\vee$), relative pseudo-complement ($\rightarrow$), entailment checking ($\models$) and projection out ($\exists$). The algorithms form the basis of the groundness analyser whose implementation is described in the next section.

4.1 Join

Consider $f = f_1 \vee f_2$, where $f_1, f_2 \in \mathrm{MHC}$. Suppose $f_1 = c_1 \wedge ... \wedge c_n$ and $f_2 = d_1 \wedge ... \wedge d_m$. Then, distributing, $f \equiv f' = \wedge_{i=1}^{n}(\wedge_{j=1}^{m}(c_i \vee d_j))$. Suppose $c_i = y_1 \wedge ... \wedge y_k \rightarrow x_1 \vee ... \vee x_l$ and $d_i = u_1 \wedge ... \wedge u_p \rightarrow v_1 \vee ... \vee v_q$. Then $c_i \vee d_i \equiv y_1 \wedge ... \wedge y_k \wedge u_1 \wedge ... \wedge u_p \rightarrow x_1 \vee ... \vee x_l \vee v_1 \vee ... \vee v_q \in \mathrm{MHC}$. Hence $f' \in \mathrm{MHC}$. Since the above involves a quadratic blowup in the size of the representation, join is quadratic in the size of the input formulae.

4.2 Relative Pseudo-Complement

Relative pseudo-complement has recently been used to support backward reasoning. In particular to trace control flow backward (right to left) to infer moding properties of initial queries [20].

Consider $f = f_1 \to f_2$, where $f_1, f_2 \in$ MHC. Suppose $f_1 = c_1 \wedge ... \wedge c_n$ and $f_2 = d_1 \wedge ... \wedge d_m$. Then, $f \equiv f' = \wedge_{j=1}^{m}(\vee_{i=1}^{n}(c_i \to d_j))$. Suppose $c_i = y_1 \wedge ... \wedge y_k \to x_1 \vee ... \vee x_l$ and $d_i = u_1 \wedge ... \wedge u_p \to v_1 \vee ... \vee v_q$. Then

$$c_i \to d_i = \begin{cases} \wedge_{i=1}^{l}(x_i \wedge u_1 \wedge ... \wedge u_p \to v_1 \vee ... \vee v_q) \\ \wedge \wedge_{j=1}^{k}(u_1 \wedge ... \wedge u_p \to y_j \vee v_1 \vee ... \vee v_q) \end{cases}$$

Hence $f' \in$ MHC. Given that the size of f' is exponential in the size of f_1, the operation is exponential. However, it should be noted that many analyses using positive Boolean functions (including groundness) do not require this operation to be calculated. In such cases the cost of this operation is not a drawback.

4.3 Entailment Checking

Entailment checking for positive Boolean functions represented in conjunctive normal form is *co-NP* complete [1]. However, as exploited in SAT solving, many of the Boolean functions that arise in practice can be checked for satisfiability with low complexity algorithms. This observation is exploited by the two algorithms detailed below. The first, *entailslite*, is incomplete and takes quadratic time in the size of the input. The second, *entailsheavy*, adds case splitting to the first algorithm to obtain completeness (which is required to guarantee termination in the fixpoint engine). This stratified algorithm usually only requires *entailslite* to be invoked once.

The *entailslite* algorithm (seen Figure 2) is an incomplete test that a multiheaded clause is entailed by a conjunction of multiheaded clauses: $\wedge_{i=1}^{l} B_i \to H_i \models B \to H$, where $B = y_1 \wedge ... \wedge y_n$ and $H = x_1 \vee ... \vee x_m$. It works by propagating deterministic bindings in an attempt to detect contradiction. The algorithm terminates either when a contradiction is found or when no more bindings can be propagated: then $Flag$ is returned. Notice that this algorithm contains forward chaining for propositional Horn clauses as a special case. Also notice that the variables are assigned values only once. The auxiliary rename produces a syntactic variant of a term which does not share any variables with the original term.

The algorithm *entailsheavy* (see Figure 3) applies case splitting if *entailslite* does not detect entailment. The number of cases is potentially exponential in the number of variables left unbound by *entailslite*. However, propagation occurs after each binding, therefore deep case splitting is rarely required. A more intelligent splitting strategy (as in SAT solving) could be applied, but the naïve strategy performs more than adequately.

Proposition 2. The algorithm *entailslite* is sound, but not complete for entailment checking. The algorithm *entailsheavy* is both sound and complete.

process *entailslite*$(\wedge_{i=1}^{l} B_i \rightarrow H_i,\ B \rightarrow H)$
    $Flag := false$;
    for $i = 1$ **to** m **do** $x_i := false$;
    for $j = 1$ **to** n **do** $y_j := true$;
    for $k = 1$ **to** l **do**
        spawn *forward*$(B_k,\ H_k,\ Flag)$;
        spawn *backward*$(B_k,\ H_k,\ Flag)$;
    return $Flag$.

process *forward*$(B,\ H,\ Flag)$
    block until every $x \in B$ bound
        if $\wedge B \equiv true$ **then spawn** *maketrue*$(H,\ Flag)$
        else stop.

process *backward*$(B,\ H,\ Flag)$
    block until every $y \in H$ bound
        if $\vee H \equiv false$ **then spawn** *makefalse*$(B,\ Flag)$
        else stop.
process *maketrue*$(H = \{y_1, ..., y_m\},\ Flag)$
    block until $y_i \in H$ changes for some $i \in \{1, ..., m\}$
        if $\vee H \equiv false$ **then** $Flag := true$; **stop**
        else if $\vee H \equiv true$ **then stop**
        else if $\vee H \equiv y_i$ for some $i \in \{1, ..., m\}$ **then** $y_i := true$; **stop**
        else suspend.

process *makefalse*$(B = \{x_1, ..., x_n\},\ Flag)$
    block until $x_i \in B$ changes for some $i \in \{1, ..., n\}$
        if $\wedge B \equiv true$ **then** $Flag := true$; **stop**
        else if $\wedge B \equiv false$ **then stop**
        else if $\wedge B \equiv x_i$ for some $i \in \{1, ..., n\}$ **then** $x_i := false$; **stop**
        else suspend.

Fig. 2: The *entailslite* Algorithm

process *entailsheavy*$(F,\ f)$
    $Flag :=$*entailslite*$(F,\ f)$;
    if $Flag = true$ **then return** $true$
    else $V :=$ var(F)=$\{x_1, ..., x_n\}$;
    if $V = \emptyset$ **then return** $false$
    else do
        rename$(F \wedge f)$=$(F' \wedge f')$;
        $Flag' :=$ *entailsheavy*$(\{x'_1 \mapsto true\}F,\ \{x'_1 \mapsto true\}f)$;
        if $Flag' = true$ **then**
            rename$(F \wedge f)$=$(F'' \wedge f'')$;
            return *entailsheavy*$(\{x''_1 \mapsto false\}F'',\ \{x''_1 \mapsto false\}f'')$
        else return $false$.

Fig. 3: The *entailsheavy* Algorithm

4.4 Projection

As in [19], projection is calculated using a Fourier-Motzkin style algorithm. The projection of a single variable out of a pair of clauses, one of which contains the variable in the body and the other in its head is performed by syllogising as follows:

$$\exists z. \begin{pmatrix} y_1 \wedge ... \wedge y_p \to z \vee x_1 \vee ... \vee x_q \\ \wedge\ z \wedge y_{p+1} \wedge ... \wedge y_n \to x_{q+1} \vee ... \vee x_m \end{pmatrix} = y_1 \wedge ... \wedge y_n \to x_1 \vee ... \vee x_m$$

The correctness and completeness of this is easily confirmed using Schröder elimination, hence the algorithm below is also correct and complete. In general, each variable is eliminated in turn, as follows. Suppose z is to be projected out of f.

1. All those clauses with z in the head are found, giving $\{C_i \mid i \in I\}$ where I is a (possibly empty) index set.
2. All those clauses with z in the body are found, giving $\{D_j \mid j \in J\}$ where J is a (possibly empty) index set.
3. These clauses of f are replaced by $\{\exists z.C_i \wedge D_j | i \in I, j \in J\}$
4. A compact representation is maintained by eliminating redundant clauses (absorption).

Step 4 means that the algorithm is parameterised by the compaction process. Compaction does not necessarily have to remove all redundant clauses (or indeed any), hence a tradeoff can be made between keeping the representation small and the cost of this maintenance. In projecting out a single variable, syllogising gives a quadratic blowup in the size of the representation. Thus the basic cost of projecting out a single variable is quadratic. However, the compaction step takes as its input a representation quadratic in the size of the original and the overall cost is dependent on the compaction algorithm. In the implementation, *entailslite* is used for compaction therefore the cost of projecting out a single variable is quartic. Because of the size blowup, projecting an arbitrary function onto a finite set of variables is exponential.

5 A Pos-Based Groundness Analyser

To assess the representation, two Pos-based groundness analysers built on multiheaded clauses were implemented in Prolog: one goal-dependent and one goal-independent. The analysers illustrate the ease with which the multiheaded clause representation can be used. The analysers perform surprisingly well compared with other Pos analysers (including those with BDD-based Boolean function manipulation coded in C) and compared with analysers using more computationally tractable domains. This section details the Prolog implementation.

5.1 A GEP Representation

As in [2, 19], the analyser maintains a factorised representation, that is, as a product of subdomains. The factorisation is encoded in the call and answer patterns. A call (or answer) pattern is a pair $\langle a, f \rangle$ where a is an atom and $f \in$ Pos.

Normally the arguments of a are distinct variables. The formula f is a conjunction (list) of multiheaded clauses. In a non-ground representation the arguments of a can be instantiated and aliased to express simple dependency information [17]. For example, if $a = p(x_1, ..., x_5)$, then the atom $p(x_1, true, x_1, x_4, true)$ represents a coupled with the formula $(x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5$. This enables the abstraction $\langle p(x_1, ..., x_5), f_1\rangle$ to be collapsed to $\langle p(x_1, true, x_1, x_4, true), f_2\rangle$ where $f_1 = (x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5 \wedge f_2$. This encoding leads to a more compact representation and is similar to the GER factorisation of ROBDDs proposed by Bagnara and Schachte [2]. The representation of call and answer patterns described above is called GEP (groundness, equivalences and propositional clauses) where the atom captures the first two properties and the formula the latter.

The GEP representation is advantageous since it gives a compact representation whilst incurring little overhead when the representation is non-ground. The compactness of the representation affects memory usage and the complexity of domain operations. As demonstrated in [17], many dependencies arising in groundness analysis fall into the GE component. By using the GEP representation, many calls to expensive domain operations are avoided. Note that (as in [19]) the analyser does not maintain the factorisation strictly. Dependencies that could be encoded in the GE component may exist in the P component – the advantage of this is that the implementor may choose to update the GE component only when most computationally convenient.

5.2 Domain Operations for the GEP Representation

Meet The meet of the pairs $\langle a_1, f_1\rangle$ and $\langle a_2, f_2\rangle$ can be computed by unifying a_1 and a_2 and concatenating f_1 and f_2.

Renaming The objects that require renaming are formulae and call (answer) pattern GEP pairs. If a dynamic database is used to store the pairs, then renaming is automatically applied each time a pair is looked-up in the database. Formulae can be renamed with a single call to the Prolog builtin copy_term.

Entailment Entailment checking works on three levels each called under a negation so as not produce any problematic bindings. The first entailment check operates only on the GE component (and is complete for this component). Entailment of the functions encoded in the GE component is denoted $a_1 \models a_2$. To test this, bind each distinct variable in a_1 to a distinct ground constant, resulting in a_1'. If, after this has been performed, a_1' may be unified with a_2, then $a_1 \models a_2$. Otherwise $a_1 \not\models a_2$. The second entailment check is only applied to formula in the P component. This implements the (incomplete) *entailslite* algorithm described in section 4.3. The propagating processes are realised using block declarations. A single pass over the formulae sets up the process and each clause results in two processes at any one time. The cost of suspending and resuming these processes is constant time, so propagation is achieved with very little overhead. The third entailment check implements a variant of the *entailsheavy* algorithm described in section 4.3. Copy_term produces a renamed formulae with new variables such that if any of the original variables have processes blocked on them, then the new variables will have copies of the processes blocked on them. This saves repeating work in the calls to *entailslite*.

Projection Projection is only applied to formulae in the P component. It is performed using the algorithm given in section 4.4. Clauses produced by projection that are equivalent to *true* (that is, the intersection of the head and body variables is nonempty) are immediately discarded. The compaction step is based on the *entailslite* algorithm. However, as the purpose of compaction is to prevent an explosion in the size of the representation, compaction is only performed if the representation after syllogising is larger than beforehand. Since *entailslite* is incomplete some redundant clauses may be retained, however this is more than compensated by the reduced complexity of compaction.

Join Calculating the join of the pairs $\langle a_1, f_1 \rangle$ and $\langle a_2, f_2 \rangle$ is complicated by the way that join interacts with renaming. Specifically, in a non-ground representation, call (answer) patterns would be typically stored in a dynamic database so that $var(a_1) \cap var(a_2) = \emptyset$. Hence $\langle a_1, f_1 \rangle$ (or equivalently $\langle a_2, f_2 \rangle$) have to be appropriately renamed before the join is calculated. This is achieved as follows. Plotkin's anti-unification algorithm [22] is used to compute the most specific atom a that generalises a_1 and a_2. (But observe that if $a_1 \models a_2$, a_2 is a most specific generalisation of the atoms.) The basic idea is to reformulate a_1 as a pair $\langle a_1', f_1' \rangle$ which satisfies two properties: a_1' is a syntactic variant of a; the pair represents the same dependency information as $\langle a_1, true \rangle$. A pair $\langle a_2', f_2' \rangle$ reformulating a_2 is likewise constructed. The atoms a, a_1' and a_2' are unified and the formula $f' = (f_1 \wedge f_1') \vee (f_2 \wedge f_2')$ is calculated. This calculation is filtered by entailment checking. If $f_1 \wedge f_1' \models f_2 \wedge f_2'$ can be detected using *entailslite*, then $f' = f_2 \wedge f_2'$ (and symmetrically). In this case the entailment check saves a call to join (and the associated projection) and the creation of a new data-structure, f'. Otherwise the join f' is computed as in section 4.1. Redundant clauses are removed from f' using *entailslite* to give f, and thereby the join $\langle a, f \rangle$.

5.3 Fixpoint Algorithms

The goal-dependent analyser is driven by an induced magic based iteration strategy, refining that used in [19]. Induced magic was introduced in [5], where a meta-interpreter for semi-naïve, goal-dependent, bottom-up evaluation is presented. Simple optimisations can significantly impact on performance. In particular, as noted in [18], evaluations resulting from new calls should be performed before those resulting from new answers, and a call to solve for one rule should finish before another call to solve for another rule starts. These optimisations have been incorporated into the induced magic framework by using an explicit redo list storing those call and answer patterns which have changed, thereby defining the clauses which need to be reevaluated. The goal-independent analyser is based on semi-naïve iteration. Neither of these analysers has exploited condensing [16, 21].

6 Experimental Evaluation

To assess the feasibility of multiheaded clauses as a representation of positive Boolean functions, the Pos-based groundness analysers were tested on a large benchmark suite.

BDD representations of Boolean functions have been popular for the implementation of Pos-based groundness analysers. For this reason an analyser using a BDD package has also been instrumented. The BDD package available does not employ a GER factorisation. However, it should be noted that turning off the GEP factorisation with the multiheaded clause analyser does not greatly affect its performance. This is a strength of clausal representations. An RCNF analyser was also implemented in Prolog to aid the assessment of MHC. The three goal-dependent analysers share the same fixpoint algorithm and therefore run in lock-step.

The analysers are coded in SICStus Prolog 3.8.6 with the exception for the domain operations for BDD-based Pos, which were written in C by Schachte [23], and compiled with O2 level of optimisation. The analysers were run on a 296MHz Sun UltraSPARC-II with 1GByte of RAM running Solaris 7. Programs are abstracted following the elegant (two program) scheme of [4] to guarantee correctness. Programs containing disjunctions are normalised to definite clauses. Timeouts were set at two minutes.

Table 1 presents the experimental results for the larger programs in the benchmark suite. The columns detail the following information, file: the program name; size: the number of abstract clauses; abs: the time require to read, parse, normalise and abstract the program. For goal-dependent analysis the fixpoint times for the MHC, RCNF and BDD analysers are given, along with count: the number of ground argument positions in the call and answer patterns found by the analyser. For goal-independent analysis, the fixpoint times for MHC are given, along with the number of ground arguments in the success patterns. Timeout is denoted by '–'. The goal-independent counts are occasional larger than the goal-dependent counts owing to the presence of code unreachable from the initial query.

Multiheaded clauses perform consistently better than RCNF for goal-dependent analysis. This is unsurprising given the cost of meet and the relative expense of equivalence checking via dual Blake canonical form, together with the filtering applied to join in MHC. MHC compares favourably with BDDs, especially considering that the BDD operations exploit memoisation and are coded in C. In terms of runtime, MHC and BDDs give similar results, although as would be expected, the different representations performed differently on different programs. For example, BDD perform well on sim.pl, whereas MHC perform well on sim_v5-2.pl. The MHC analyser appears to scale smoothly for both goal-

file	size	abs	goal-dep. MHC	RCNF	BDD	count	goal-indep. MHC	count
bridge.clpr	68	0.09	0.00	0.12	0.03	24	0.08	34
conman.pl	76	0.05	0.00	0.00	0.03	6	0.01	6
unify.pl	77	0.05	0.07	0.29	0.08	70	0.09	19
kalah.pl	78	0.05	0.02	0.11	0.04	199	0.02	42
nbody.pl	85	0.07	0.05	0.13	0.06	113	0.04	57
peep.pl	85	0.12	0.03	0.08	0.04	10	0.02	8
sdda.pl	89	0.06	0.04	0.07	0.05	17	0.02	4
bryant.pl	94	0.07	0.32	2.38	0.15	99	0.28	9
boyer.pl	95	0.08	0.05	0.07	0.04	3	0.02	5
read.pl	101	0.09	0.05	0.23	0.08	99	0.03	37
qplan.pl	108	0.09	0.03	0.25	0.07	216	0.05	27
trs.pl	108	0.13	0.10	2.28	0.26	13	0.04	7
press.pl	109	0.09	0.11	0.27	0.12	53	0.04	32
reducer.pl	113	0.07	0.08	0.17	0.09	41	0.05	21
parser_dcg.pl	122	0.09	0.09	0.29	0.08	43	0.04	24
simple_analyzer.pl	140	0.10	0.16	0.48	0.13	89	0.10	31
dbqas.pl	143	0.09	0.03	0.04	0.04	18	0.03	24
ann.pl	146	0.11	0.16	0.43	0.10	71	0.09	12
asm.pl	160	0.17	0.05	0.19	0.09	90	0.14	16
nand.pl	179	0.14	0.05	1.46	0.14	402	0.68	16
lnprolog.pl	220	0.10	0.08	0.19	0.12	143	0.07	31
ili.pl	221	0.15	0.55	1.63	0.13	4	0.15	5
strips.pl	240	0.22	0.03	0.07	0.08	142	0.06	36
sim.pl	244	0.22	1.09	24.78	0.25	100	0.62	33
rubik.pl	255	0.21	0.22	25.32	0.20	158	0.16	51
chat_parser.pl	281	0.36	0.29	1.75	0.26	505	0.30	128
sim_v5-2.pl	288	0.23	0.07	0.33	0.16	457	0.10	37
peval.pl	332	0.18	0.64	4.62	0.16	27	1.30	17
aircraft.pl	395	0.54	0.15	0.70	0.41	687	0.12	196
essln.pl	595	0.48	0.19	20.72	0.37	162	0.30	75
chat_80.pl	883	1.43	0.88	4.28	0.84	855	0.64	339
aqua_c.pl	3928	3.55	7.68	67.04	–	1285	6.59	458

Table 1. Timing and Precision Results

dependent and goal-independent analysis. Of course, any Pos-based analyser can be broken using the schema from [6, 14]; the analyser can deal with the arity 14 case of [6] before timeout (that is, a single predicate requiring 16384 iterations).

The major cost in entailment checking is incurred through case splitting in *entailsheavy*. Instrumentation has revealed that the total number of times *entailslite* is invoked in checking $F \models f$ almost never exceeds $|\text{var}(F)|$. Therefore in practice *entailsheavy* exhibits cubic behaviour in the size of the input formulae. Further instrumentation has shown that the maximum number of heads observed in a clause is four. These maxima occur infrequently. Since most clauses have few heads, typically only a small number of bindings have to be made before propagation binds sufficient variables to return the $Flag$. The calls to *entailsheavy* typically do not detect entailment, as the vast majority of entailments are detected using *entailslite*. As disentailment is demonstrated by the discovery of a single countermodel, the binding of a small number of variables to their value in a countermodel is often enough to generate the rest of this countermodel via propagation. This helps to explain the success of the stratified entailment check.

7 Related Work

The efficiency of groundness analysis depends on the way dependencies are represented and implemented. The representation decides the algorithmic complexity of the domain operations but the implementation can introduce a prohibitive constant factor or even push the complexity into a higher class if there is not a good match between the representation and the implementation language. Efficient BDD-based Pos analysis are usually implemented in languages with mutable data-structures such as C [24] or SML [10, 11]. State-of-the-art BDD-based groundness analysers employ a GE factorisation [2] which keeps simple definite information separate from dependency information. This leads to a particularly dense representation (meant informally, a small number of nodes/clauses in the representation) and is therefore an important implementation tactic.

The density of the representation is as important to Prolog as it is to C: the density determines the size of the inputs to the domain operations, as well as impacting on memory usage. The dual Blake canonical form representation of Def functions [1, 9] is attractive as it is amenable to Prolog implementation [12] and it gives a unique representation for every Def function (up to variable ordered). However, its requirement to make transitive variable dependencies explicit can compromise density. For example, the function $(x \leftarrow y) \wedge (y \leftarrow z)$ is represented as $(x \leftarrow (y \vee z)) \wedge (y \leftarrow z)$. Because of this Howe and King [19] present a (non-orthogonal [1]) clausal representation of Def as conjunctions of propositional clauses, but do not maintain a canonical form. Therefore entailment checking is required to detect stability.

Recently, Genaim and Codish [13] have proposed a dual representation for Def. For function f, the models of $coneg(f)$ are named and f is represented by a tuple recording for each variable of f which of these models the variable is in. For example, the models of $coneg(x \to y)$ are $\{\{x, y\}, \{x\}, \emptyset\}$. Naming the three models a, b, c respectively, f is represented by $\langle ab, a\rangle$. This representation cleverly allows ACI1 unification theory to be used for the domain operations and elegantly supports a GE factorisation. Promising experimental results are reported [13], but a widening is required to analyse the aqua_c benchmark.

Codish and Demoen [7] describe a model based Prolog implementation technique for Pos that would encode $x_1 \leftrightarrow (x_2 \wedge x_3)$ as three tuples $\langle true, true, true\rangle$, $\langle false, \_, false\rangle$, $\langle false, false, \_\rangle$. The technique performs well against BDD-based Pos analysis of its era [24] but it does not scale smoothly to the larger benchmarks. Heaton *et al.* [17] therefore propose EPos, a sub-domain of Def, that can only propagate dependencies of the form $(x_1 \leftrightarrow x_2) \wedge x_3$ across procedure boundaries. This information is precisely that contained in one of the fields of the GE factorisation. The main finding of [17] is that this sub-domain retains reasonably precision for goal-dependent analysis and possesses good scaling behaviour.

8 Conclusion

Positive Boolean functions can be naturally expressed as multiheaded clauses which are straightforward to understand, manipulate and code in Prolog. Multiheaded clauses have been used as the basis for efficient goal-dependent and goal-independent Pos-based groundness analysers. The key to the success of these analysers is their constant time meet and their use of entailment checking succinctly and efficiently coded using block declarations. Entailment checking is stratified so that many entailments are detected using a low complexity algorithm. The full exponential algorithm is only applied when necessary for detecting stability, and even then the number of case splits is typically very small. The analysers do not require widening for any of the benchmarks; however, natural widenings to Def or to EPos are available if required [6, 14]. This work illustrates the subtlety of choosing a representation and its associated operations, even for a well known domain. Minor changes to the representation can have a significant impact on performance if they affect frequently occurring operations. It also demonstrates the effectiveness of stratifying high complexity operations to avoid expensive computation whenever possible. The intelligent application of the simple entailment checking algorithm is the heart of the analyser presented in this paper.

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 *Seventh International Conference on Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 471–485. Springer-Verlag, 1999.
3. P. Bigot, S. Debray, and K. Marriott. Understanding Finiteness Analysis using Abstract Interpretation. In *Joint International Conference and Symposium on Logic Programming*, pages 735–749. MIT Press, 1992.
4. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 108–124. Springer-Verlag, 1996.
5. M. Codish. Efficient Goal Directed Bottom-up Evaluation of Logic Programs. *Journal of Logic Programming*, 38(3):355–370, 1999.
6. M. Codish. Worst-Case Groundness Analysis using Positive Boolean Functions. *Journal of Logic Programming*, 41(1):125–128, 1999.
7. 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.

8. 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.
9. P. Dart. On Derived Dependencies and Connected Databases. *Journal of Logic Programming*, 11(1–2):163–188, 1991.
10. C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung*. PhD thesis, Universität des Saarlandes, 1997.
11. C. Fecht and H. Seidl. A Faster Solver for General Systems of Equations. *Science of Computer Programming*, 35(2-3):137–162, 1999.
12. 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.
13. S. Genaim and M. Codish. The Def-inite Approach to Dependency Analysis. In *European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 417–32. Springer-Verlag, 2001.
14. S. Genaim, J. M. Howe, and M. Codish. Worst-Case Groundness Analysis using Definite Boolean Functions. *Theory and Practice of Logic Programming*, 2001. Forthcoming.
15. R. Giacobazzi, S. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 25(3):191–247, 1995.
16. R. Giacobazzi and F. Scozzari. A Logical Model for Relational Abstract Domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.
17. 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.
18. M. Hermenegildo, G. Puebla, K. Marriot, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transaction on Programming Languages and Systems*, 22(2):187–223, 2000.
19. 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. Available at http://www.cs.ukc.ac.uk/pubs/2000/949/.
20. A. King and L. Lu. A Backwards Analysis for Constraint Logic Programs. Technical Report 4-01, University of Kent, 2001.
21. 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.
22. G. Plotkin. A Note on Inductive Generalisation. *Machine Intelligence*, 5:153–163, 1970.
23. P. Schachte. *Precise and Efficient Static Analysis of Logic Programs*. PhD thesis, Department of Computer Science, The University of Melbourne, Australia, 1999.
24. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the Domain *Prop*. *Journal of Logic Programming*, 23(3):237–278, 1995.