Two Classes of Boolean Functions
for Dependency Analysis

Tania Armstrong, Kim Marriott,
Peter Schachte and Harald Søndergaard

Technical Report 94/10

Dept. of Computer Science
The University of Melbourne
Parkville Vic. 3052
Australia
Two Classes of Boolean Functions
for Dependency Analysis

Tania Armstrong*       Kim Marriott*
Peter Schachte†    Harald Søndergaard†

Abstract

Many static analyses for declarative programming/database languages use Boolean functions to express dependencies among variables or argument positions. Examples include groundness analysis, arguably the most important analysis for logic programs, finiteness analysis and functional dependency analysis for databases. We identify two classes of Boolean functions that have been used: positive and definite functions, and we systematically investigate these classes and their efficient implementation for dependency analyses. On the theoretical side we provide syntactic characterizations and study the expressiveness and algebraic properties of the classes. In particular, we show that both are closed under existential quantification. On the practical side we investigate various representations for the classes based on reduced ordered binary decision diagrams (ROBDDs), disjunctive normal form, conjunctive normal form, Blake canonical form, dual Blake canonical form, and two forms specific to definite functions. We compare the resulting implementations of groundness analyzers based on the representations for precision and efficiency.

Categories and Subject Descriptors: D.1.6 [Programming Languages]: Logic Programming;
D.3.4 [Programming Languages]: Processors—compilers, optimization; F.3.1 [Logics and
Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—assertions,
invariants, logics of programs

General Terms: Languages, Theory

Additional Key Words and Phrases: Abstract interpretation, Boolean algebra, code optimization,
decision diagrams, deductive databases, dependency analysis, finiteness analysis, groundness analysis, propositional logic, static analysis.

*Dept. of Computer Science, Monash University, Clayton 3168, Australia.
†Dept. of Computer Science, The University of Melbourne, Parkville 3052, Australia.
1 Introduction

Many dataflow analyses use Boolean functions to represent “dependencies” among variables or predicate arguments. The idea in a dependency based analysis is to let the statement “program variable $x$ has property $p$” be represented by the propositional variable $x_p$. A dependency such as “whenever $y$ has property $q$, $x$ has property $p$” may then be represented by a Boolean function, in this case the function denoted by $y_q \rightarrow x_p$. Important applications are groundness analysis for (constraint) logic programs, finiteness analysis for deductive databases, suspension analysis for concurrent (constraint) logic programs, and functional dependency analysis for relational and deductive databases, as well as for logic programs. Two main subclasses of Boolean functions, the positive functions and the definite functions, have been suggested for dependency analyses. The main aim of this paper is to systematically study and compare these two subclasses.

Our contributions are twofold: First we provide simple syntactic characterizations for positive and definite functions and study their algebraic properties. We give a variety of closure results for the classes; in particular both classes are closed under existential quantification.

Our second contribution is to suggest a number of different representations and implementations for these classes. Although many different representations of Boolean functions have been widely studied for other purposes, there are special properties of the functions used in dependency analyses which suggest that their representation warrants a special study. Dependency analysis requires a representation which compactly represents functions built from implications and bi-implications between conjunctions of variables and for which the join, meet, restriction and renaming operations are fast. Typically a dependency formula will involve few variables, and testing for equivalence of formulas will be infrequent. Here we investigate representations for positive and definite functions which are based on reduced ordered binary decision diagrams (ROBDDs), disjunctive normal form, conjunctive normal form, Blake canonical form, dual Blake canonical form, and two forms specific to definite functions. We compare implementations of groundness analysis based on the different representations for speed and precision.

The plan of this paper is as follows. In Section 2 we outline how to use Boolean functions for groundness, finiteness, and suspension analysis. In Section 3 we discuss in more detail two classes of Boolean functions that lend themselves naturally to this. In Section 4 we consider a variety of ways to represent Boolean functions so that their manipulation can be made efficient. In Section 5 we report our experience from experimenting with the various representations for groundness analysis. Section 6 discusses related work, and Section 7 contains a concluding discussion.

A shorter version of this paper, not including proofs and many examples was presented at SAS'94 [1].

2 Dependency Analysis Using Boolean Functions

We motivate our study of Boolean functions by sketching how they can be used to give very precise groundness, finiteness, and suspension analysis.
2.1 Groundness Analysis

Groundness analysis is arguably the most important dataflow analysis for logic programs and constraint logic programs. The question: “At a given program point, does variable $x$ always have a unique value?” is not only important for an optimizing compiler attempting to speed up unification or constraint solving, but for all programming tools that apply some kind of dataflow analysis. The reason is that most other analyses, such as independence analysis (whether constraining $x$ indirectly constrains other variables) or occur-check analysis (whether unification can safely be performed without the occur-check) are extremely inaccurate unless they also employ groundness analysis. For example, if $x$ is ground (a terminological abuse we consistently use for “bound to a unique value”), then $x$ cannot possibly share with other variables, and this is useful information for independence, occur-check, and many other dataflow analyses. If we use Boolean functions as approximations to runtime states, then abstract interpretation gives a natural way of specifying a very precise groundness analysis.

Let us illustrate the use of Boolean functions for groundness analysis. The central idea is to use implication to capture groundness dependencies. The reading of a function such as $x \rightarrow y$ is: "if the program variable $x$ is (becomes) ground, so is (does) program variable $y." In this way program variables are replaced by propositional variables. Consider the following Prolog program for sorting using difference pairs.

```prolog
quicksort(Xs, Ys) :-
    dquicksort(Xs, Ys, []). dquicksort([], Ys, Ys).
dquicksort([X|Ys], Ys, Zs) :-
    partition(Xs, X, Lows, Highs),
    dquicksort(Lows, Ys, [X|Us]),
    dquicksort(Highs, Us, Zs).

partition([], E, [], []). partition([X|Xs], E, [X|Lows], Highs) :-
    X =< E,
    partition(Xs, E, Lows, Highs).
partition([X|Xs], E, Lows, [X|Highs]) :-
    X > E,
    partition(Xs, E, Lows, Highs).
```

Given a list of numbers as a first argument and a variable as a second argument, `quicksort` will terminate and bind the variable to the sorted permutation of the list. Given a variable as first argument and a list of numbers as second argument, `quicksort` may or may not terminate, but whenever it succeeds, the variable will be bound to a list of numbers. This behavior is captured by the function $zs \leftarrow ys$. One consequence which can be read out of this formula is: “whenever `quicksort` succeeds given one of its arguments is ground, the other argument has been made ground.”
This information can be obtained automatically as follows [25]. As a first step we translate the
program to its Clark completion [10]. Since we will need to manipulate rather complex formulas
involving predicate and variable names, we deviate from Prolog conventions and use lower case
for variables, and nil and ‘:’ for list construction. This yields

\[
q(xs, ys) \leftarrow
d(xs, ys, nil)
d(xs, ys, zs) \leftarrow
\begin{align*}
& (xs = nil \land ys = zs) \\
& \lor \exists x, xs', lows, highs, us, us'. [xs = x : xs' \land p(xs', x, lows, highs) \\
& \land us' = x : us \land d(lows, ys, us') \land d(highs, us, zs)]
\end{align*}
p(xs, e, lows, highs) \leftarrow
\begin{align*}
& (xs = nil \land lows = nil \land highs = nil) \\
& \lor \exists x, xs', lows'. [xs = x : xs' \land lows = x : lows' \land x \leq e \land p(xs', e, lows', highs)] \\
& \lor \exists x, xs', highs'. [xs = x : xs' \land highs = x : highs' \land x > e \land p(xs', e, lows, highs')].
\end{align*}
\]

The second step consists of translating this into a definition of three Boolean functions in such a
way that the functions correctly describe the groundness dependencies amongst the variables of
the respective predicates. We obtain the following translation:

\[
q(xs, ys) =
d(xs, ys, true) 
d(xs, ys, zs) =
\begin{align*}
& (xs \land (ys \leftarrow zs)) \\
& \lor \exists x, xs', lows, highs, us, us'. [(xs \leftarrow (x \land xs')) \land p(xs', x, lows, highs) \\
& \land (us' \leftarrow (x \land us)) \land d(lows, ys, us') \land d(highs, us, zs)]
\end{align*}
p(xs, e, lows, highs) =
\begin{align*}
& (xs \land lows \land highs) \\
& \lor \exists x, xs', lows'. \\
& \quad [(xs \leftarrow (x \land xs')) \land (lows \leftarrow (x \land lows')) \land x \land e \land p(xs', e, lows', highs)] \\
& \lor \exists x, xs', highs'. \\
& \quad [(xs \leftarrow (x \land xs')) \land (highs \leftarrow (x \land highs')) \land x \land e \land p(xs', e, lows, highs')].
\end{align*}
\]

There are several points to notice here. The translation of the constraint \(xs = nil \land lows = nil \land
highs = nil\) is the Boolean function \(x \land lows \land highs\), which expresses that all three variables
become ground if the first clause is selected. The translation of \(“xs = x : xs”\) is slightly more
complex. The function \(xs \leftarrow (x \land xs')\) expresses the groundness dependencies amongst
the three variables, namely “if \(xs\) is (or later becomes) ground, so are (do) both of \(x\) and \(xs'\), and vice
versa.” The translation of a builtin such as “\(x > e\)” is in accordance with the builtin’s behavior
when it succeeds: For \(x > e\) to succeed, both variables must be ground, hence the translation
\(x \land e\). (We are here assuming that the Prolog system does not employ a “delay” mechanism.)

In the Clark completed program, existential quantification was used to project a formula onto
the subspace spanned by its “interesting” variables—those that are not local to a clause body.
The same applies in the translation. It may not be obvious why existential quantification over
a propositional variable is the correct counterpart to existential quantification over a program
variable or why conjunction and disjunction should correspond. The reader is referred to [26] for a justification.

Notice that the equations could be simplified at this point, by utilizing Schröder’s Elimination Principle\(^1\)

\[ \exists x. F = F[x \mapsto false] \lor F[x \mapsto true]. \]

We may, for example, simplify the definition of \( p \) to

\[ p(xs, e, lows, highs) = (xs \land lows \land highs) \lor (e \land p(xs, e, lows, highs)). \]

The last step in the analysis is to solve the set of recursive Boolean equations. The quicksort program has the call graph shown in Figure 1. We can use the call graph to find the most economic order of processing the three predicates, which in this case is the order partition, quicksort, quicksort. In general, we “stratify” the set of predicates by computing the strongly connected components (SCCs) of the call graph and sorting these topologically according to the “reachable from” ordering given by the graph.

So we first solve for \( p \). The relevant solution is the smallest fixpoint with respect to the ordering \( \models \), that is, logical consequence. We therefore compute the corresponding Kleene sequence, starting at \( false \):

\[
\begin{align*}
    p_0(xs, e, lows, highs) &= false \\
    p_1(xs, e, lows, highs) &= xs \land lows \land highs \\
    p_2(xs, e, lows, highs) &= (xs \land lows \land highs) \lor (e \land xs \land lows \land highs) \\
                      &= xs \land lows \land highs
\end{align*}
\]

so \( p_1 \) is a fixpoint. This tells us that whenever partition succeeds, it grounds three of its variables, \( \text{Xs}, \text{Lows}, \) and \( \text{Highs} \). This information makes it easy to solve the equation for \( d \):

\[
\begin{align*}
    d_0(xs, ys, zs) &= false \\
    d_1(xs, ys, zs) &= xs \land (ys \mapsto zs) \\
    d_2(xs, ys, zs) &= (xs \land (ys \mapsto zs)) \lor (\neg ys \land \neg zs) \lor (\neg xs \land \neg ys \land zs) \\
                      &= ys \mapsto (xs \land zs)
\end{align*}
\]

This turns out to be the required fixpoint and it immediately leads to the solution for \( q \):

---

\(^1\)The first explicit statement of the principle appears to be by Schröder ([30] page 22), who derived it from Boole’s principle of “development”: \( F = (F[x \mapsto false] \land \neg x) \lor (F[x \mapsto true] \land x) \) (sometimes referred to as Boole’s Expansion Theorem, or “Shannon expansion”). Boole considered disjunction to be exclusive, so the elimination principle would have made little sense to him.
\[ q(xs, ys) = d(xs, ys, true) \]
\[ = xs \leftarrow ys. \]

In other words, if one of the arguments given to quicksort is ground, the other will become ground as well.

In general, in a groundness analysis we are not only interested in what happens when a predicate succeeds, but also in the collection of calls that are made during execution, including the calls that lead to failure (backtracking). The reason is that an optimizing compiler needs this information for a variety of code improvements. In principle it is not difficult to obtain this kind of information. The idea is to mimic the execution of a given query, replacing resolution and constraint solving with a corresponding operation on the Boolean functions.

Assume that we are interested in the call patterns that could possibly occur as a consequence of calling quicksort with a ground first argument. By a call pattern we mean a pair \( \langle A, \phi \rangle \), where \( A \) is an atom that appears in the query or in a clause body, and \( \phi \) is an approximation of the contents of the constraint store restricted to the variables in \( A \) just before \( A \) is processed. For example, our query follows the call pattern \( \langle q(xs, ys), xs \rangle \). From the clause for quicksort it is apparent that one call pattern is \( \langle d(xs, ys, zs), zs \land zs \rangle \). From the recursive clause for dquicksort we then get the call pattern \( \langle p(xs, z, lows, highs), zs \land x \rangle \). To find the groundness information that pertains to the program point just before the first recursive call, we conjoin the “current” approximation \( xs \land x \) with the information previously calculated for \( p \) and get \( zs \land x \land lows \land highs \). Of these, however, only \( lows \) and \( x \) appear in the call, so we record the pattern as \( \langle d(lows, ys, x : us), lows \land x \rangle \). Similarly we get the final call pattern \( \langle d(highs, us, zs), highs \rangle \). This allows us to conclude that dquicksort is always called with a ground first argument, and that partition is always called with its first two arguments ground, information that a compiler can utilize.

This way of computing call patterns is slightly simpler than traditional abstract interpretation based methods. It relies on “condensation” [22]. For the positive functions it is as precise as more traditional approaches [25].

### 2.2 Finiteness Analysis

Finiteness analysis is one of the most important dataflow analyses for deductive databases as it is used to identify possibly non-terminating queries.

In a finiteness analysis, the description \( x \rightarrow y \) for a predicate \( p(x, y) \) is read as “for any finite assignment of values to the first argument of \( p \) there are only finitely many assignments to the second argument which satisfy the relation assigned to \( p \).” As an example consider the Datalog program

\[
\text{member}(X, Xs) :- \text{cons}(X, Ys, Xs).
\]
\[
\text{member}(X, Xs) :- \text{cons}(Y, Ys, Xs), \text{member}(X, Ys).
\]

where \( \text{cons}(x, xs, ys) \) is an infinite relation which models addition of an element to a list. It satisfies the integrity constraint \( \text{cons}(x, xs, ys) : ys \leftarrow (x \land xs) \). A finiteness analysis for this program proceeds similarly to a groundness analysis. First, we compute the Clark completion.
Next we translate the completion into a set of Boolean equations which capture the finiteness dependencies between the predicates. In this example we obtain:

\[
\begin{align*}
    \text{member}(x, xs) &= \exists ys . \text{cons}(x, ys, xs) \\
                        &\quad \lor \exists y, ys . [\text{cons}(y, ys, xs) \land \text{member}(x, ys)] \\
    \text{cons}(x, xs, ys) &= ys \leftrightarrow (x \land xs).
\end{align*}
\]

We now solve these equations to find an explicit solution. However, as finiteness is not a property that admits fixpoint induction, the approach is usually to find the greatest fixpoint for the set of Boolean equations [4]. In this case we obtain the solution by approximating from the top, that is, initially assuming the solution \textit{true}. We get:

\[
\begin{align*}
    m_0(x, xs) &= \text{true} \\
    m_1(x, xs) &= \exists ys . [\text{cons}(x, ys, xs)] \lor \exists y, ys . [\text{cons}(y, ys, xs) \land \text{true}] \\
                &= (xs \to x) \lor \text{true} \\
                &= xs \to x \\
    m_2(x, xs) &= \exists ys . [\text{cons}(x, ys, xs)] \lor \exists y, ys . [\text{cons}(y, ys, xs) \land (ys \to x)] \\
                &= xs \to x
\end{align*}
\]

In other words, the solution is

\[\text{member}(x, xs) = xs \to x\]

which indicates that for a given ground value of \(xs\) there are only finitely many answers to \textit{member}(\(x, xs\)), assuming the relation assigned to \textit{cons} satisfies its integrity constraint.

### 2.3 Suspension Analysis

Our third example of the use of Boolean functions is for suspension analysis of concurrent logic programming languages [31] and concurrent constraint programming languages [29]. Concurrent (constraint) logic languages can be viewed as specifying reactive systems consisting of collections of communicating processes. If the computation of a program reaches a state in which it requires input from the environment in order to continue, the computation and the program are said to \textit{suspend}. The presence of unintended suspended computations is a common programming error which is difficult to detect using standard debugging and testing techniques. Boolean functions can be used to give an analysis which succeeds if a program is definitely suspension free [19]. We exemplify this for a typical concurrent logic language, FCP(·) [32].

FCP(·) programs consist of finite sets of \textit{guarded clauses} which specify rules for \textit{reducing} states. The basic notions of concurrency — processes, communication, synchronization and non-determinism — are realized in concurrent logic languages by viewing each atom in a state as a separate process. Communication is achieved using logical variables. Messages are sent between processes by instantiating shared variables; synchronization is based on the general principle that the reduction of an atom with a clause is delayed until the atom’s arguments are sufficiently instantiated. Computation in FCP(·) starts with an initial state and proceeds by repeatedly rewriting states into other states. A state is a tuple containing the current goal and equation set.

A state can be rewritten into another state whenever an atom in the current goal can be reduced by a matching clause. Reduction using the clause \(H : \neg \text{Ask} : \text{Tell} \mid B\) can occur if the
current equation set implies the ask equations Ask of the clause, and is consistent with the tell equations, Tell. Reducing an atom by a clause means that the atom is replaced by the atoms in the body of the clause, B, and that the equations in the clause are added to the current equation set.

Consider the following FCP($) program [12]:

\[
p(X) :- \text{true} : x = [a|X1] \mid p(X1). \\
p(X) :- \text{true} : X = [ ] \mid \text{true}.
\]

\[
c(X) :- x = [a|X1] : \text{true} \mid c(X1). \\
c(X) :- x = [ ] : \text{true} \mid \text{true}.
\]

The first two clauses specify a producer of a stream of atoms ‘a’; while the last two clauses specify a consumer of a similar stream. Consider the initial state \( \langle p(x_1), c(x_2) \{ x_1 = x_2 \} \rangle \) executed using the above program. The equation \( x_1 = x_2 \) specifies that \( c(x_2) \) is the consumer of the stream produced by \( p(x_1) \).

The idea behind the suspension analysis is to approximate the behavior of a program and initial state by a set of recursively defined propositional formulas which capture groundness information about process arguments, as well as information about definite non-suspension. For the above program the recursive equations are:

\[
s(ns) = \exists x_1, x_2, ns_c, ns_p . p(x_1, ns_p) \land c(x_2, ns_c) \land x_1 \leftrightarrow x_2 \land ns \leftrightarrow (ns_p \land ns_c)
\]

\[
p(x, ns) = \exists x_1, ns_1 . \text{true} \rightarrow (p(x_1, ns_1) \land x \leftrightarrow x_1 \land ns \leftrightarrow ns_1)
\]

\[
c(x, ns) = \exists x_1, ns_1 . x \rightarrow (p(x_1, ns_1) \land x \leftrightarrow x_1 \land ns \leftrightarrow ns_1)
\]

\[
\land x \rightarrow (x \land ns).
\]

For instance, the first equation says that the initial state \( s \) is definitely non-suspending if the processes \( p \) and \( c \) are definitely non-suspending. If we compute the least fixpoint of these equations we obtain:

\[
p(x, ns) = x \land ns
\]

\[
c(x, ns) = x \rightarrow ns
\]

\[
s(ns) = ns.
\]

Thus we know that the original state will definitely not suspend.

## 3 Two Classes of Boolean Functions and Their Properties

We have seen that Boolean functions provide very natural descriptions of dependencies between variables and argument positions. The smallest class of Boolean functions which we shall consider consists of definite functions. Informally these allow us to use conjunction and implication and give rise to very precise analyses. However, one may obtain even more precise analyses by allowing disjunctive information as well. We call the resulting class of functions positive. The precise definitions of both classes will be given shortly and their relative expressiveness will be made clear.
**Definition.** A *Boolean function* is a function $F : \text{Bool}^n \rightarrow \text{Bool}$. We call the set of all Boolean functions $Bfun$ and let it be ordered by logical consequence ($\models$).

We assume that a fixed finite (but non-empty) set $\text{Var}$ of variables is given. We sometimes use propositional formulas over $\text{Var}$ as representations of Boolean functions without worrying about the distinction. Thus we may speak of a formula as if it were a function and in any case denote it by $F$. We shall also use the common convention of identifying a truth assignment (or *model*) with the set of variables it maps to $\text{true}$.

**Definition.** The function $F$ is *positive* iff $\text{Var} \models F$, that is, $F(\text{true}, \ldots, \text{true}) = \text{true}$. We let $\text{Pos}$ denote the set of positive Boolean functions, $\text{Def}$ the set of functions in $\text{Pos}$ whose models are closed under intersection, and $\text{Mon}$ the set of monotonic Boolean functions. Functions in $\text{Def}$ are called *definite*.

For example, the Boolean function $\neg x$ is not in $\text{Pos}$. The functions $x \rightarrow y$ and $x \wedge y$ are in $\text{Pos}$. The function $x \rightarrow y$ is in $\text{Def}$ but not in $\text{Mon}$, and $x \wedge y$ is in $\text{Mon}$ but not in $\text{Def}$. To see that $x \rightarrow y$ is in $\text{Def}$, consider its models (as subsets of $\{x, y\}$). The set of models is $\{\emptyset, \{y\}, \{x, y\}\}$, a set which is closed under intersection. On the other hand, the set of models for $x \wedge y$ is $\{\{x\}, \{y\}, \{x, y\}\}$, and this set is not closed under intersection.

Clearly $\text{Def}$ and $\text{Mon} \setminus \{\text{false}\}$ are proper subsets of $\text{Pos}$. Here we will need $\text{Mon}$ only as an aid to understanding $\text{Def}$. The Hasse diagrams in Figure 2 show the ordering of the formulas in $\text{Pos}$ and $\text{Def}$ for $\text{Var} = \{x, y\}$.

Syntactically the classes have interesting characterizations. We follow Cortesi et al. [15] in using the notation $\mathcal{F} = \Omega S$ to indicate that the set $S$ of connectives is functionally complete for the class $\mathcal{F}$ of Boolean functions. That is, connectives from the set $S$ suffice, together with variables, to represent every function in $\mathcal{F}$, and no function outside $\mathcal{F}$ can be so represented. It is well-known that $\text{Mon} = \Omega\{\wedge, \vee, \text{true}, \text{false}\}$, where $\text{true}$ and $\text{false}$ are the (overloaded) constant functions returning $\text{true}$ and $\text{false}$ respectively for all input. For $\text{Pos}$, the following strengthens a result by Cortesi et al. [15].
Theorem 3.1 \( Pos = \Omega\{\land, \to\} = \Omega\{\land, \leftrightarrow\} = \Omega\{\to, \to\} \).

Proof: Firstly, from Cortesi et al. [15] we know that \( Pos = \Omega\{\land, \lor, \to\} \). Alternatively, \( Pos = \Omega\{\land, \lor, \to\} \), as \( \land \to \) can be obtained from \( \land \) and \( \to \), while \( \to \) can be obtained from \( \land \) and \( \leftrightarrow \): \( F \to F' = F \to (F \land F') \).

Now for the two-connexive characterizations:

1. \( \lor \) can be defined from \( \to \): \( F \lor F' = (F \to F') \to F' \).

2. We already saw that \( \to \) can be obtained from \( \land \) and \( \leftrightarrow \).

3. \( \land \) can be obtained from \( \to \) and \( \leftrightarrow \): \( F \land F' = F \leftrightarrow (F \to F') \).

It is well known that \( Bfun \) is a Boolean lattice, with meet and join given by conjunction and disjunction, respectively. Furthermore, \( Bfun \) is closed under existential quantification, by Schröder’s Elimination Principle. Regarding the closure properties of \( Pos \), we have the following result, from which it follows that \( Pos \) is a Boolean sublattice of \( Bfun \).

Theorem 3.2 Let \( F, G \in Pos \). The following are all positive: \( F \land G, F \lor G, F \to G, F \leftrightarrow G \), and \( \exists x . F \).

Proof: The first four claims follow from Theorem 3.1. For the last claim, notice that if \( F \) is a positive function then \( F[x \leftrightarrow true] \) is positive. Hence \( \exists x . F = F[x \leftrightarrow false] \lor F[x \leftrightarrow true] \) is positive. 

We now turn to \( Def \). Let a clause be a disjunction of literals. A definite clause is a clause with one positive literal or the empty clause. We shall usually write definite clauses using implication. For instance, \( \land \neg y_1 \lor \ldots \lor \neg y_n \) is regarded as \( x \leftarrow y_1 \land \ldots \land y_n \). In such a formula, \( x \) is referred to as the head. A definite sentence is a conjunction of definite clauses. The following is a reformulation of Dart’s [17] Proposition 3.1.

Theorem 3.3 The function \( F \) in \( Def \) iff \( F \) can be represented as a definite sentence. 

Sometimes it is useful to represent a function in \( Def \), not as a definite sentence, but in a closely related conjunctive normal form where each variable \( x \) occurs exactly once as a head. For example, the function denoted by the definite sentence \( (x \leftarrow y) \land (x \leftarrow z) \) is written as \( x \leftarrow (y \lor z) \land y \leftarrow false \land z \leftarrow false \).

Definition. A formula

\[
F = \bigwedge_{x \in \text{Var}} (x \leftarrow M_x)
\]

is in monotonic body form (MBF) iff each \( M_x \) is monotonic. If, furthermore, for no \( x, x \in M_x \), then \( F \) is in reduced MBF (RMBF). 

\[\text{That } \land \text{ and } \to \text{ form a functionally complete set of connectives for } Pos \text{ was discovered during a conversation between W. Winsborough and H. Søndergaard in August 1992 but the demonstration was more complex than this proof.}\]
Dart [17] makes the following observation.

**Theorem 3.4** A function $F$ is definite iff $F$ can be written in RMBF. ■

MBF is no more expressive than RMBF, as $x \leftarrow M$ is logically equivalent to $x \leftarrow M[x \leftrightarrow \text{false}]$, as the reader can easily verify. This means that translation from MBF to RMBF is easy. With right-hand sides in, say, disjunctive normal form, that is,

$$F = \bigwedge_{x \in \text{Var}} (x \leftarrow (F_1 \lor \ldots \lor F_n)),$$

one can simply delete each disjunct $F_i$ containing $x$ as a conjunct. For example,

$$(x \leftarrow ((u \land x) \lor (v \land w \land x) \lor (w \land z)) \land (y \leftarrow (x \land y))$$

is equivalent to $(x \leftarrow (w \land z)) \land (y \leftarrow \text{false})$.

$\text{Def}$ does not inherit all the closure properties of $\text{Pos}$. However, the following follows immediately from the definition of $\text{Def}$.

**Theorem 3.5** $\text{Def}$ is closed under conjunction. ■

**Theorem 3.6** If $F$ is definite, so is $x \rightarrow F$.

*Proof:* Let $F$ be definite. Then $x \rightarrow F$ is positive and so has at least one model ($\text{Var}$). Let $\phi$ and $\psi$ be models of $x \rightarrow F$. If one (or both) does not contain $x$ then $(\phi \cap \psi) \models x \rightarrow F$ trivially. Otherwise $\phi \models F$ and $\psi \models F$, and so, as $F$ is definite, $(\phi \cap \psi) \models F$. But then $(\phi \cap \psi) \models x \rightarrow F$. ■

Note that $F \rightarrow x$ is not necessarily definite, even when $F$ is. For example, take $F = y \rightarrow x$. Then $F \rightarrow x$ is equivalent to $x \lor y$, which is not definite. Also, a definite $x \rightarrow F$ does not imply a definite $F$ (take $F = x \lor y$). Finally, the non-definite $(x \rightarrow y) \leftrightarrow y$ shows that $\text{Def}$ is not closed under $\leftrightarrow$.

Given that $\text{Def}$ is not closed under disjunction, it is somewhat surprising that $\text{Def}$ is closed under existential quantification:

**Theorem 3.7** If $F$ is definite, so is $\exists x . F$.

*Proof:* Let $F$ be definite. We have that $\exists x . F = F[x \rightarrow \text{false}] \lor F[x \rightarrow \text{true}]$. Since $F[x \rightarrow \text{true}]$ is positive, so is $\exists x . F$, and so it has one or more models. Let $\phi$ and $\psi$ be models of $\exists x . F$. We consider three cases and show that in each case $\phi \cap \psi$ is also a model.

1. Assume $\phi \models F[x \leftrightarrow \text{true}]$ and $\psi \models F[x \leftrightarrow \text{true}]$. Then $\phi \cup \{x\}$ and $\psi \cup \{x\}$ both satisfy $F$. As $F$ is definite, $(\phi \cup \{x\}) \cap (\psi \cup \{x\}) \models F$, so $(\phi \cap \psi) \models F[x \leftrightarrow \text{true}]$.

2. Assume $\phi \models F[x \leftrightarrow \text{false}]$ and $\psi \models F[x \leftrightarrow \text{false}]$. Then $\phi \setminus \{x\}$ and $\psi \setminus \{x\}$ both satisfy $F$, and hence, so does $(\phi \cap \psi) \setminus \{x\}$. It follows that $(\phi \cap \psi) \models F[x \leftrightarrow \text{false}]$.

10
3. For reasons of symmetry we can assume $\phi \models F[x \rightarrow false]$ and $\psi \models F[x \rightarrow true]$. Then $\phi \setminus \{x\} \models F$ and $\psi \cup \\{x\} \models F$, so $(\phi \cap \psi) \setminus \{x\} \models F$. It follows that $(\phi \cap \psi) \models F[x \rightarrow false]$.

In all cases, $(\phi \cap \psi) \models \exists x . F$. ■

**Theorem 3.8** Def is a lattice.

**Proof:** Def has a largest element, true, so the theorem follows immediately from Theorem 3.5. ■

The join on Def—let us denote it by $\hat{\vee}$—must be different from that on Pos, that is, it is not classical disjunction. Dart [17] notes that the meet can be calculated from two definite formulas (exactly as for full propositional logic) as follows: Let

$$F = \bigwedge_{x \in \text{Var}} (x \leftarrow M_x) \text{ and } F' = \bigwedge_{x \in \text{Var}} (x \leftarrow M'_x)$$

where the $M_x$ and $M'_x$ are monotonic formulas. Then $F \wedge F'$ is

$$\bigwedge_{x \in \text{Var}} (x \leftarrow (M_x \vee M'_x)).$$

However, Dart does not indicate how $\hat{\vee}$ can be computed. One might hope that by duality the join would be given by

$$F'' = \bigwedge_{x \in \text{Var}} (x \leftarrow (M_x \wedge M'_x)).$$

To see that this is not the case, consider

\begin{align*}
F & = (x \leftarrow y) \wedge (y \leftarrow u) \wedge (z \leftarrow false) \wedge (u \leftarrow false) \\
F' & = (x \leftarrow z) \wedge (y \leftarrow false) \wedge (z \leftarrow u) \wedge (u \leftarrow false).
\end{align*}

In this case we get

$$F'' = (x \leftarrow (y \wedge z)) \wedge (y \leftarrow false) \wedge (z \leftarrow false) \wedge (u \leftarrow false)$$

but

$$F''' = (x \leftarrow ((y \wedge z) \vee u)) \wedge (y \leftarrow false) \wedge (z \leftarrow false) \wedge (u \leftarrow false)$$

is also an upper bound for $F$ and $F'$, and $F''' \models F''$, and $F''$ has a model which does not satisfy $F'''$, namely $\{u\}$. (We later show that $F''' = F \hat{\vee} F'$.) This justifies the following definition.

**Definition.** Let the formula $F = \bigwedge_{x \in \text{Var}} \{x \leftarrow M_x\}$ be in MBF. Then $F$ is in orthogonal form iff, for every set $S$ of propositional variables, $F \wedge \bigwedge S \models x$ iff $\bigwedge S \models M_x \lor x$. ■

The intuition is that in every component $x \leftarrow T$ of $F$, the right-hand side $T$ must be a consequence of every implicant of $x$ (in $F$). In Section 4.2 we show that an orthogonal form always exists, a fact we use for the following result:
Theorem 3.9 If $F = \bigwedge_{x \in \text{Var}} (x \leftarrow M_x)$ and $F' = \bigwedge_{x \in \text{Var}} (x \leftarrow M'_x)$ are in orthogonal RMBF then
\[
F \vee F' = \bigwedge_{x \in \text{Var}} (x \leftarrow (M_x \land M'_x)).
\]

Proof: Let $F'' = F \vee F'$. We need to show that for every model $\phi$ for $F''$ there are models $\phi_F$ for $F$ and $\phi_{F'}$ for $F'$ such that $\phi = \phi_F \cap \phi_{F'}$. Let
\[
\phi_F = \{ x \mid \phi \models M_x \} \cup \phi \quad \text{and} \quad \phi_{F'} = \{ x \mid \phi \models M'_x \} \cup \phi.
\]
We first show that $\phi = \phi_F \cap \phi_{F'}$. Clearly $\phi \subseteq \phi_F \cap \phi_{F'}$. Let $x \in \phi_F \cap \phi_{F'}$ and assume that $x \not\in \phi$. Then $\phi \models M_x \land M'_x$. But $x \leftarrow (M_x \land M'_x)$ is a term in $F''$. As $\phi$ satisfies $F''$, $x \in \phi$. But this contradicts the assumption. So $\phi = \phi_F \cap \phi_{F'}$.

We now show that $\phi_F$ satisfies $F$. Since $F$ is definite, there is a least model $\hat{\phi}$ for $F$ with the property that $\phi_F \subseteq \hat{\phi}$ (namely the intersection of all such models). We show that $\hat{\phi} \subseteq \phi_F$.

\[
x \in \hat{\phi} \implies \bigwedge \phi \models F \leftarrow x
\]
\[
\implies (F \land \bigwedge \phi) \models x
\]
\[
\implies \bigwedge \phi \models M_x \lor x \quad \text{(by definition of orthogonality)}
\]
\[
\implies \phi \models M_x \lor \phi \models x
\]
\[
\implies x \in \phi_F.
\]

The proof that $\phi_{F'}$ satisfies $F'$ is similar. \[\square\]

Exactly how an orthogonal form is derived depends on the representation used for $\text{Def}$. We return to this point in Section 5.2.

While $\text{Pos}$ is a Boolean lattice, $\text{Def}$ is neither complemented nor distributive. An element in $\text{Def}$ which has no complement is $x \leftarrow y$. To see that $\text{Def}$ is not distributive, note that
\[
(x \leftarrow y) \land (x \leftarrow y) = x \leftarrow y \quad \text{but} \quad ((x \leftarrow y) \land x) \leftarrow ((x \leftarrow y) \land y) = x \land y.
\]
As a practical consequence, a groundness dependency analysis using $\text{Def}$ may be sensitive to unfolding, even when we unfold an atom that contains no constants or function symbols. Consider

\[
\begin{align*}
q(X, Y) & \leftarrow p(X, Y), r(X, Y) \\
p(X, X). \\
r(a, Y). \\
r(X, a).
\end{align*}
\]

For this program and the unconstrained query $q(X, Y)$, $\text{Def}$ yields $x \leftarrow y$. However, $\text{Def}$ gives the stronger $x \land y$ if we unfold $r(X, Y)$:

\[
\begin{align*}
q(a, Y) & \leftarrow p(a, Y). \\
q(X, a) & \leftarrow p(X, a). \\
p(X, X).
\end{align*}
\]

We now exemplify the relative accuracy of positive and definite functions. The following Prolog clauses could be part of a package for digital circuit design.

12
or(X, Y, Z) :-
    and(X, Y, U),
    xor(X, Y, V),
    xor(U, V, Z).

and(true, Y, Y).
and(X, true, X).
and(false, false, false).

xor(X, X, false).
xor(true, false, true).
xor(false, true, true).

The Boolean functions representing the groundness dependencies of and and xor are easily computed using the techniques from Section 2. With Pos we get

\[
and(x, y, z) = (x \land (y \leftrightarrow z)) \lor (y \land (x \leftrightarrow z))
\]
\[
xor(x, y, z) = (x \leftrightarrow y) \land z
\]
\[
or(x, y, z) = \exists u, v. [(x \land (y \leftrightarrow u)) \lor (y \land (x \leftrightarrow u))] \land (x \leftrightarrow y) \land v \land (u \leftrightarrow v) \land z
\]
\[
= x \land y \land z
\]

With Def we get

\[
and(x, y, z) = (x \land (y \leftrightarrow z)) \lor (y \land (x \leftrightarrow z))
\]
\[
= (x \land y) \leftrightarrow z
\]
\[
xor(x, y, z) = (x \leftrightarrow y) \land z
\]
\[
or(x, y, z) = \exists u, v. [(x \land y) \leftrightarrow u) \land (x \leftrightarrow y) \land v \land (u \leftrightarrow v) \land z]
\]
\[
= x \land y \land z
\]

Notice that even though Def in this example yields less precise groundness information for and, this turns out to have no effect on the result for or.

It is common to use a variant of Pos, namely Pos⊥ = Pos ∪ {false} for groundness analysis.3 The reason for this is as follows. A dataflow analysis is concerned with describing the sets of constraints that may apply at the various program points. The Boolean functions in Pos are adequate for this: F describes the set E iff F describes every e ∈ E. However, it also makes sense to include the non-positive function false, with the natural interpretation that false describes an empty set of constraints. Similarly one may use Def⊥ = Def ∪ {false}, ordered by logical consequence.

The function false does not really contribute anything in terms of groundness detection, but it does extend and improve the analysis with a reachability analysis. If false is the final approximation at a given program point, it means that control will never reach that point. Notice that there is no need for false in the finiteness analysis we sketched in 2.2, as that analysis was expressed in terms of a greatest fixpoint.

---

3In fact we did so in Section 2.
4 Representations for Pos and Def

In this section we investigate various representations for Pos and Def which provide for efficient implementation of the various operations used in dependency analyses. The examples in Section 2 indicated that we need to perform the following five operations:

- Test for equivalence so as to determine if a fixpoint has been reached.
- Compute the join of the Boolean functions corresponding to the different clauses defining an atom.
- Compute the meet of the Boolean functions corresponding to the different constraints and atoms in a clause body.
- Restrict a Boolean function, that is, existentially quantify over a local variable.
- Rename a Boolean function corresponding to an atom in a clause body so that there are no conflicts with the other variable names in the clause body.

For Boolean expressions in general, equivalence is intractable, assuming that \( \mathcal{P} \neq \mathcal{NP} \). Unfortunately, this result continues to hold for both Pos and Def, for standard representations:

**Theorem 4.1** Determining equivalence of two RMBF formulas is co-NP complete.

**Proof:** Given a truth assignment \( \phi \) and a Boolean expression \( F \), the evaluation of \( \phi F \) can be done in polynomial time. It follows that non-equivalence of two Boolean expressions is in \( \mathcal{NP} \), and so the equivalence problem for Def (and also for Pos) is in co-NP.

We prove NP-hardness by reduction from the equivalence problem for monotonic formulas, which is known to be co-NP complete [5]. Consider any two monotonic formulas \( M \) and \( M' \) and let \( x \) be a variable that does not occur in \( M \) or \( M' \). Let \( D \) and \( D' \) be the formulas \( x \leftarrow M \) and \( x \leftarrow M' \), respectively. (Clearly \( D \) and \( D' \) can be generated in polynomial time, and \( |D| + |D'| = \Theta(|M| + |M'|) \).) Then \( D \) and \( D' \) are in RMBF and logically equivalent iff \( M \) and \( M' \) are.

One well-known symbolic representation for a general Boolean function is as a formula in disjunctive normal form. More formally, a term is a conjunction of literals, with true the empty conjunction. A disjunctive normal form (DNF) formula is a disjunction of terms with false the empty disjunction. Another well-known symbolic representation for a general Boolean function is as a formula in conjunctive normal form. More precisely, a conjunctive normal form (CNF) formula is a conjunction of clauses, with true the empty conjunction.

**Theorem 4.2** Determining equivalence of two CNF formulas which represent functions in Pos is co-NP complete. This is also true if the formulas are in DNF.

**Proof:** It follows from an identical argument to the previous proof that these problems are in co-NP.
We prove NP-hardness by reduction from the satisfiability problem for Boolean formulas in CNF [13]. Consider a CNF formula $F$ and let $x$ be a variable which does not occur in $F$. Let $G$ be the formula obtained by adding $x$ to every clause in $F$ and let $G'$ be $x$. (Clearly $G$ and $G'$ can be generated in polynomial time.) Then $G$ and $G'$ are in CNF, represent positive formulas and are logically equivalent iff $F$ is not satisfiable.

The proof for NP-hardness of the case when the formulas are in DNF is by reduction from the case when they are in CNF. Consider CNF formulas $F$ and $F'$ which represent positive functions. Let $G$ and $G'$ be obtained from $F$ and $F'$ by “negating” each connective in the formula, that is $\lor$ is replaced by $\land$, $\land$ is replaced by $\lor$, and each literal $L$ is replaced by $\neg L$. We have that $(G \leftrightarrow F)$ and $(G' \leftrightarrow F')$ are valid. Now consider the formula $H = G \lor x$ and $H' = G' \lor x$ where $x$ is a variable which does not occur in $G$ or $G'$. Clearly $H$ and $H'$ can be generated in polynomial time. Then $H$ and $H'$ are in DNF, represent positive formulas and are logically equivalent iff $F$ and $F'$ are.

Given these two results, it is notable that equivalence of definite sentences (a type of CNF) is tractable. This rests on the fact that satisfiability of propositional Horn sentences has a linear time algorithm [18].

**Theorem 4.3** Equivalence of definite sentences can be decided in quadratic time.

**Proof:** Given a definite sentence $F$, it is possible to determine in linear time whether $F \rightarrow x$ is valid, by deciding whether the Horn sentence $F \land \neg x$ is satisfiable. It follows that it is possible to determine in linear time whether $F \rightarrow (x \leftarrow \bigwedge_{i=1}^{m} y_i)$ is valid, as this holds iff $(F \land \bigwedge_{i=1}^{m} y_i) \rightarrow x$. Notice that the left-hand side can be reduced in linear time, by replacing each $y_i$ in $F$ by true. Consequently, it is possible to determine equivalence of definite sentences in quadratic time. ■

It follows from Theorem 4.1 and Theorem 4.2 that, assuming $\mathcal{P} \neq \mathcal{NP}$, for any representation we choose for positive functions, either the conversion from a Boolean formula in DNF or CNF to the representation has worst-case exponential cost or else the test for equivalence between two representations has worst-case exponential cost. Similarly, for any representation we choose for definite functions, either the conversion from RMBF to the representation has worst-case exponential cost or else the test for equivalence between two representations has worst-case exponential cost. However, knowledge about an application may allow one to develop a representation which in practice gives good performance. In our application, program analysis, we know that:

- Tests for equivalence will be less common than the other operations and involve fewer variables.
- The functions will be over a relatively small number of variables as a clause in a logic program typically contains less than 30 variables.
- The base functions represent dependency information of the form $x \leftarrow (\bigwedge_{i=1}^{m} y_i)$ or of the form $x \leftarrow (\bigvee_{i=1}^{m} y_i)$. The other functions encountered in the analysis are constructed by joining, meeting and restricting these base functions.

We now briefly describe the representations we have considered and the cost of the various operations with these representations.
4.1 General Representations

The first five representations are for arbitrary Boolean functions and will be used to represent Pos. Our first representation, ROBDD, acts as a yardstick as it has been used for representing positive functions for groundness analysis in other studies [3, 14, 23].

4.1.1 ROBDD: Reduced Ordered Binary Decision Diagrams

A ROBDD is a well-known symbolic representation for Boolean functions [8]. Intuitively, a ROBDD is constructed by creating a decision tree from a truth table and then turning the tree into a dag by identifying and collapsing identical sub-trees. The value of the function for particular values of the variables can be found by following the branch corresponding to the truth value of the variable. Given a fixed variable ordering used to construct the decision tree, the ROBDD for a function is unique. Figure 3 shows the ROBDD for \((x \land y) \leftrightarrow z\) with variables ordered lexicographically. Solid arrows indicate the path to take if the variable in the source node is true, and dashed lines indicate the path if the variable is false.

Since a ROBDD is canonical, testing for equivalence takes at worst linear time. However, in practice a global unique table is kept, which means that testing equivalence has constant time [6]. Having a global unique table also saves a great deal of space, as there will never be multiple copies of identical nodes. The other operations—meet, join, restrict, and rename—have a worst case time complexity which is quadratic in the size of the ROBDDs involved. However, in the worst case the size of the ROBDD can grow exponentially with the number of variables. In practice, for the right choice of variable ordering, many Boolean functions have polynomial size ROBDDs. In particular, a formula of the form \(x \leftrightarrow (\bigwedge_{i=1}^{m} y_i)\) has a linear size representation for any variable ordering.
4.1.2 RDNF: Reduced Disjunctive Normal Form

Our second representation is the “reduced” DNF formulas. A DNF formula is reduced if no term in the formula implies another term in the formula. We let RDNF denote a formula which is in reduced DNF form. For example, \((x \land y) \leftrightarrow z\) could be represented as
\[
(x \land y \land z) \lor (\neg x \land \neg z) \lor (\neg y \land \neg z)
\]
or as
\[
(x \land y \land z) \lor (\neg x \land y \land z) \lor (\neg z \land y \land \neg z) \lor (x \land \neg y \land \neg z).
\]
Computing a reduced form for a DNF formula can be done by iteratively removing terms implied by other terms in the formula. This has, in the worst case, quadratic complexity, as each pair of terms may have to be examined.

Renaming of a RDNF formula takes linear time. By the following result, if \(F\) is in RDNF, it takes time linear in the size of \(F\) to produce a DNF representation of \(\exists x. F\).

**Proposition 4.4** [7] Let \(F\) be the DNF formula \(\bigvee_{i=1}^{n} t_i\) and let \(\text{restrict}(t, x)\) denote the term obtained by replacing occurrences of both \(x\) and \(\neg x\) in \(t\) by \(\text{true}\). Then \(\bigvee_{i=1}^{n} \text{restrict}(t_i, x)\) is a DNF representation of \(\exists x. F\).

**Example 4.1** Eliminating \(x\) from \((x \land y \land z) \lor (\neg x \land \neg z) \lor (\neg y \land \neg z)\) we get \((y \land z) \lor (\neg z) \lor (\neg y \land \neg z)\). This has reduced form \((y \land z) \lor (\neg z)\) which indeed represents \(z \rightarrow y\).

Thus the cost of restricting a RDNF formula of size \(N\) is \(O(N^2)\). Similarly, the worst case cost of computing a RDNF for the join of two RDNF formulas is \(O(MN)\), where the input formulas have sizes \(M\) and \(N\). This is because the disjunction of two RDNF formulas is a DNF formula. Computation of a RDNF form for the meet of two RDNF formulas, however, is more expensive. The time complexity is \(O(M^2N^2)\). This is because computing the conjunction involves “multiplying out” the two RDNF formulas to get a DNF formula which has size \(O(MN)\) and then computing a reduced form for this formula.

It is clear from the above example that a RDNF representation of a function is not canonical. To determine whether two RDNF formulas are equivalent, it is possible to compute some canonical form for the two formulas and compare. One method is to compute the Blake Canonical Form (BCF), described in the next subsection, and then compare. However, in the worst case this has exponential cost, as could be expected considering Theorem 4.2.

4.1.3 BCF: Blake Canonical Form

Like ROBDD, Blake canonical form, BCF, is widely used to represent Boolean functions. The BCF representation of function \(F\) is the disjunction of prime implicants of \(F\). More precisely, an implicant of \(F\) is a term that implies \(F\). An implicant is prime if no proper sub-term is an implicant. The BCF of a function \(F\), written \(BCF(F)\), is the disjunction of all its prime implicants. Clearly, a BCF is always in RDNF. For example, \((x \land y) \leftrightarrow z\) has BCF
\[
(x \land y \land z) \lor (\neg x \land \neg z) \lor (\neg y \land \neg z).
\]
The BCF of a function is canonical up to reordering of the implicants and variables. Thus, if the BCF formula is ordered, testing for equivalence takes linear time. Renaming of a BCF takes linear time. The BCF of a DNF formula \( F \) can be obtained by computing certain implicants of \( F \) (called sylogizing), and then removing redundant disjuncts (called absorption). In practice, for efficiency, these two stages are intertwined. As we would expect from Theorem 4.2, sylogizing has exponential cost in the worst case. This means that join, meet and restriction may have exponential cost.

**Example 4.2** Consider the formula
\[
(x \land y \land z) \lor \neg(x \land y \land \neg z) \lor (\neg x \land \neg y \land \neg z) \lor (x \land \neg y \land \neg z).
\]
Sylogizing adds the disjuncts false, \( \neg(x \land \neg z) \), and \( \neg(y \land \neg z) \). Absorption then yields
\[
(x \land y \land z) \lor (\neg x \land \neg z) \lor (\neg y \land \neg z).
\]
\[\square\]

**Example 4.3** To see that sylogizing may be exponential, consider the set of variables \( \{x_1, \ldots, x_n\} \cup \{u_1, \ldots, u_N\} \), where \( N = 2^n \). Let \( F_1, \ldots, F_N \) be the \( N \) formulas in
\[\{G_1 \land \ldots \land G_n \mid G_i \text{ is } x_i \text{ or } \neg x_i\}.
\]
Consider the formula \( F = \lor\{F_i \land u_i \mid 1 \leq i \leq n\} \). This formula is linear in \( N \). For any subset \( U \) of \( \{u_1, \ldots, u_N\} \), \( \land U \) will be generated as an implicant, that is, the number of implicants found is exponential in \( N \). \[\square\]

The functions we usually encounter have BCF of reasonable size. In particular, a function of the form \( x \leftrightarrow \bigwedge_{j=1}^m y_j \) has a BCF which is linear in \( m \), namely
\[
(x \land \bigwedge_{j=1}^m y_j) \lor \bigvee_{j=1}^m (\neg x \land \neg y_j).
\]

4.1.4 **RCNF: Reduced Conjunctive Normal Form**

One possible problem with representations based on DNF formulas is that computation of the meet is significantly more expensive than the computation of join. For this reason we have investigated two representations based on conjunctive normal form. The first of these, *Reduced Conjunctive Normal Form (RCNF)*, corresponds to RDNF but with conjunction and disjunction exchanged, and hence the relative cost of computing meet and join is exchanged.

More precisely, a CNF formula is *reduced* iff no clause in the formula implies another clause in the formula. For example, \( (x \land y) \leftrightarrow z \) can be represented by the RCNF formula
\[
(y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (x \lor \neg z).
\]
Renaming a RCNF formula has linear cost. If two RCNF formulas have sizes \( M \) and \( N \), computation of a RCNF of their meet is \( O(MN) \) and computation of a RCNF of their join is \( O(M^2N^2) \). This is dual to the case for RDNF formulas.
Restricting a RCNF formula, however, is more complex than restricting a RDNF formula. The following theorem provides a method for doing so. Essentially we must use resolution to eliminate a variable. Let \( x \) be a variable and \( c \) be the clause \( x \lor L_1 \lor \cdots \lor L_n \) and \( c' \) be the clause \( \neg x \lor L'_1 \lor \cdots \lor L'_m \). Define \( \text{resolve}(x, c, c') \) to be the clause \( L_1 \lor \cdots \lor L_n \lor L'_1 \lor \cdots \lor L'_m \).

**Proposition 4.5** Let \( F \) be a CNF formula. Let \( F_x \) be the clauses in \( F \) containing variable \( x \), \( F_{\neg x} \) be the clauses in \( F \) containing \( \neg x \) and \( F_0 \) be the remaining clauses in \( F \). Then
\[
F_0 \land \bigwedge \{ \text{resolve}(x, c, c') \mid c \in F_x, c' \in F_{\neg x} \}
\]
is a CNF representation of \( \exists x \cdot F \).

**Proof:** By Schröder’s Elimination Principle, \( \exists x \cdot F = F[x \leftrightarrow \text{true}] \lor F[x \leftrightarrow \text{false}] \). Thus,
\[
\begin{align*}
\forall x \cdot F & = (F_0 \land F_x \land F_{\neg x})[x \leftrightarrow \text{false}] \lor (F_0 \land F_x \land F_{\neg x})[x \leftrightarrow \text{true}] \\
& = (F_0 \land F_x[x \leftrightarrow \text{false}]) \lor (F_0 \land F_{\neg x}[x \leftrightarrow \text{true}]) \\
& = F_0 \land \bigwedge \{ c \lor c' \mid c \in F_x[x \leftrightarrow \text{false}], c' \in F_{\neg x}[x \leftrightarrow \text{true}] \} \\
& = F_0 \land \bigwedge \{ \text{resolve}(x, c, c') \mid c \in F_x, c' \in F_{\neg x} \}.
\end{align*}
\]

It follows that the cost of restricting a RCNF formula of size \( N \) is \( O(N^4) \).

Determining equivalence of two RCNF formulas of sizes \( M \) and \( N \) can be done by computing their Dual Blake Canonical Form (see below) and testing for identity. As we would expect from Theorem 4.2 this has exponential worst-case cost.

As usual, in the worst case the RCNF of a function has size exponential in the number of variables. However, for a function of the form \( x \leftrightarrow \bigwedge_{j=1}^m y_j \) there is a RCNF whose size is linear in \( m \). It is:
\[
(x \lor \bigvee_{j=1}^m \neg y_j) \land \bigwedge_{j=1}^m (y_j \lor \neg x).
\]

### 4.1.5 DBCF: Dual Blake Canonical Form

The second representation based on CNF we call Dual Blake Canonical Form (DBCF). It corresponds to Blake Canonical Form but with conjunction and disjunction exchanged.

A *consequent* of function \( F \) is a clause implied by \( F \). A consequent is *prime* iff no proper sub-clause is a consequent. The DBCF of a function \( F \), written \( \text{DBCF}(F) \), is the conjunction of all its prime consequents. It can be obtained by using resolution to find all consequents of a CNF formula (this may be exponential), and then deleting any implied clauses. For example, \( (x \land y) \leftrightarrow z \) has DBCF
\[
(y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (x \lor \neg z).
\]

Testing for equivalence and renaming have linear cost, the same as for BCF. Somewhat surprisingly, unlike for BCF, restriction also has linear cost because of the following result.

**Proposition 4.6** Let \( F \) be the DBCF formula \( \bigwedge_{i=1}^n c_i \). Then the DBCF of \( \exists x \cdot F \) is
\[
\bigwedge \{ c_i \mid x \text{ and } \neg x \text{ do not occur in } c_i \}.
\]
Proof: Let $G$ be $\bigwedge \{ c_i | x \text{ and } \neg x \text{ do not occur in } c_i \}$. We first show that $G$ is $\exists x . F$. By Proposition 4.5, $\exists x . F$ is

$$F_{\emptyset} \land \bigwedge \{ \text{resolve}(x, c, c') | c \in F_x, c' \in F_{\neg x} \}$$

where $G$ is $F_{\emptyset}$. As $F$ is in DBCF, each clause in $\{ \text{resolve}(x, c, c') | c \in F_x, c' \in F_{\neg x} \}$ is implied by a clause in $G$. Thus $G = \exists x . F$.

We now show that $G$ is in DBCF. Let $G'$ be $\text{DBCF}(G)$. We show that $G = G'$. Assume that there is a clause $c$ in $G'$ but not in $G$. By the definition of DBCF, $c$ is a prime consequent of $G$. Thus it must be a consequent of $F$. However, as it is not in $F$, it cannot be prime. That is, some clause in $F$ implies $c$. But, this means that some clause in $G$ implies $c$, which contradicts the assumption that $c$ is a prime consequent of $G$. Thus $G'$ is contained in $G$. Now assume that there is clause $c$ in $G$ but not in $G'$. This means that $c$ is implied by some other clause $c'$ in $G$. Thus $F$ contains two clauses one of which implies the other. This contradicts the assumption that $F$ is in DBCF. Thus $G$ and $G'$ are identical.  

A function of the form $x \leftrightarrow \bigwedge_{j=1}^{m} y_j$ has a DBCF whose size is linear in $m$. This is the RCNF representation given in the previous subsection.

4.2 Specialized Representations for Def

Representations based on CNF can be specialized for $\text{Def}$ by making use of results from Section 3, where we discussed (reduced) monotonic body form (RMBF) and definite sentences. The reason is that the RMBF provides a compact representation of a definite sentence, and a definite sentence is just a type of CNF. We look at two representations, the first based on DBCF, the second based on RCNF.

4.2.1 DBCF$_{\text{Def}}$: Dual Blake Canonical Form for Def

The DBCF of a definite function is always a definite sentence.

Theorem 4.7 $F$ is a definite function iff $\text{DBCF}(F)$ is a definite sentence.

Proof: By Theorem 3.3, $F$ is a definite function iff it has a definite sentence representation, $F'$ say. The resolvent of two definite clauses is always a definite clause. Thus the resolvents of $F'$ are always definite clauses, and so $\text{DBCF}(F')$ contains only definite clauses. 

Thus we can compactly represent the DBCF of a definite function in RMBF. The DBCF$_{\text{Def}}$ representation of a definite function $F$, written $\text{DBCF}_{\text{Def}}(F)$, is the RMBF corresponding to $\text{DBCF}(F)$. More precisely, let $F$ be a definite sentence. Then the RMBF corresponding to $F$ is the formula

$$\bigwedge_{x \in \mathcal{V}_F} (x \leftarrow \bigvee \{ B | x \leftarrow B \text{ is in } F \text{ and } x \text{ is not in } B \})$$

We also say that $F$ is the definite sentence corresponding to this RMBF formula.
For example, recall the formula \((x \land y) \rightarrow z\) has DBCF

\[(y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (x \lor \neg z).\]

Thus the DBCF_{D, f} is

\[(x \leftarrow z) \land (y \leftarrow z) \land (z \leftarrow (x \land y)).\]

Notice, however, that in general, the left-hand sides of clauses will be *disjunctions of conjunctions.*

As the DBCF_{D, f} is a syntactic variant of the DBCF, testing for equivalence, renaming, and restriction are defined in the obvious manner and, like DBCF, have linear complexity. Meet is also defined in the obvious manner and, like DBCF, may have exponential cost.

Computation of the join of two DBCF_{D, f} formulas, however, is quicker than the computation of the join for DBCF. In fact it has polynomial rather than exponential worst case cost. We first prove that DBCF_{D, f} is an orthogonal form.

**Theorem 4.8** The DBCF_{D, f} representation of a definite function \(F\) is in orthogonal form.

**Proof:** Let \(T\) be a conjunction of propositional variables such that DBCF_{D, f}(\(F\), \(T\)) \models x for some variable \(x\) and let DBCF_{D, f}(\(F\)) = \(\bigwedge_{x \in \text{Var}} M_x\). We must show that \(T \models M_x \lor x\). If \(T\) contains \(x\), then this is clearly true. Otherwise assume that \(x\) does not appear in \(T\). By the first assumption, DBCF_{D, f}(\(F\)) \models T \Rightarrow x and so DBCF(\(F\)) \models x \leftarrow T. But \(x \leftarrow T\) is just a clause, and so, by the definition of DBCF, there is some clause \(x \leftarrow T'\) in DBCF(\(F\)) of which \(x \leftarrow T\) is a logical consequence. That is, \(T \models T'\). By the construction of the DBCF_{D, f} representation, \(T' \models M_x\). Thus \(T \models M_x\) as required. \(\blacksquare\)

This means that we can use Theorem 3.9 to compute the join of two DBCF_{D, f} formulas. The reason why this is cheaper than the usual join operation for DBCF is that there is no need to consider resolvents when computing the DBCF form of the join.

We note that if \(M\) is a monotonic DNF formula, computation of BCF(\(M\)) takes quadratic time as it is just the reduced form of \(M\). Thus, if \(M\) and \(M'\) are monotonic RDNF formulas with sizes \(N\) and \(N'\), then computation of BCF(\(M \land M'\)) has \(O(N^2 N'^2)\) cost.

**Proposition 4.9** Let \(F\) and \(F'\) be definite functions with DBCF_{D, f} representations \(\bigwedge_{x \in \text{Var}} (x \leftarrow M_x\) and \(\bigwedge_{x \in \text{Var}} (x \leftarrow M'_x)\) respectively. Then

\[\text{DBCF}_{D, f}(F \lor F') = \bigwedge_{x \in \text{Var}} (x \leftarrow \text{BCF}(M_x \land M'_x)).\]

**Proof:** Let \(D = \bigwedge_{x \in \text{Var}} x \leftarrow M_x\), \(D' = \bigwedge_{x \in \text{Var}} x \leftarrow M'_x\) and \(D'' = \bigwedge_{x \in \text{Var}} x \leftarrow \text{BCF}(M_x \land M'_x)\). By Theorem 3.9 \(D''\) denotes \(F \lor F'\). We now show that \(D''\) is in DBCF.

We first show that any clause obtained by resolving clauses \(C_1', C_2''\) in \(D''\) is a logical consequence of some other clause in \(D''\). Let \(C_1'' = x \leftarrow T_1''\) and \(C_2'' = y \leftarrow T_2''\) where \(y\) appears in \(T_1''\). By the construction of \(D''\), there are clauses \(C_1', C_2'\) in \(D'\) and \(C_1, C_2\) in \(D\) such that

\[\bullet\ C_1' = x \leftarrow T_1'\quad\text{and}\quad C_2' = y \leftarrow T_2'.\]
• \( C_1 = x \leftarrow T_1 \) and \( C_2 = y \leftarrow T_2 \)

• \( T''_1 = T_1 \land T'_2 \) and \( T''_2 = T_2 \land T'_1 \).

The variable \( y \) must appear in either \( T_1 \) or \( T'_1 \). There are 3 cases:

1. Assume that \( y \) appears in \( T_1 \) but not in \( T'_1 \). Then \( x \leftarrow (T_1 \land T_2) \setminus \{y\} \) is a resolvent of \( D \). As \( D \) is in DBCF, this clause is a logical consequence of some clause \( x \leftarrow T \) in \( D \). That is, \( T \models (T_1 \land T_2) \setminus \{y\} \). This means that there is a clause \( C = x \leftarrow T'' \) in \( D'' \) such that \( T'' \models (T_1 \land T_2 \land T'_1) \setminus \{y\} \). But by construction, \( C \) implies the resolvent, \( x \leftarrow (T''_1 \land T''_2) \setminus \{y\} \), of \( C'' \) and \( C''_2 \).

2. Assume that \( y \) appears in both \( T_1 \) and in \( T'_1 \). Then \( x \leftarrow (T_1 \land T_2) \setminus \{y\} \) is a resolvent of \( D \) and \( x \leftarrow (T'_1 \land T'_2) \setminus \{y\} \) is a resolvent of \( D' \). Thus there is a clause \( x \leftarrow T \) in \( D \) such that \( T \models (T_1 \land T_2) \setminus \{y\} \) and there is a clause \( x \leftarrow T' \) in \( D' \) such that \( T \models (T'_1 \land T'_2) \setminus \{y\} \). This means that there is a clause \( C = x \leftarrow T'' \) in \( D'' \) such that \( T'' \models (T_1 \land T_2 \land T'_1 \land T'_2) \setminus \{y\} \). But by construction, \( C \) implies the resolvent, \( x \leftarrow (T''_1 \land T''_2) \setminus \{y\} \), of \( C'' \) and \( C''_2 \).

3. Finally assume that \( y \) appears in \( T'_1 \) but not in \( T_1 \). This is symmetric to the first case.

It follows from the definition of \( D'' \) (all bodies in BCF) that no clause in \( D'' \) can imply another clause in \( D'' \), as they would have to have the same head. Thus \( D'' \) is in DBCF.

Before considering our next representation, let us point out two interesting consequences of Theorem 4.7. First the theorem provides an alternative proof of Theorem 3.7, offering more insight into the operation on \( \text{Def} \). The statement was that for definite \( F, \exists x.f \) is definite. By Theorem 4.8, the DBCF representation of a definite function must be a definite sentence. So by Proposition 4.6 the restriction is also a definite sentence. By Theorem 3.3 this represents a definite function.

The second consequence is that it provides a way of determining whether a propositional formula denotes a definite function. This is not always obvious, for example consider the RCNF formula \( (x \lor y) \land (\neg x \lor y) \land (x \lor \neg y) \). Its DBCF is \( x \land y \) showing that it does denote a definite function.

### 4.2.2 RCNF\(_{\text{Def}}\): Reduced Conjunctive Normal Form for Def

We now consider a second representation for definite functions. The RMBF formula \( \bigwedge_{x \in V} (x \leftarrow M_x) \) is in \( \text{RCNF}_{\text{Def}} \) if for each \( x, M_x \) is in RDNF. The reason for this name is that the definite sentence corresponding to a formula in \( \text{RCNF}_{\text{Def}} \) is in RCNF.

Not every RCNF formula which denotes a definite function corresponds to a \( \text{RCNF}_{\text{Def}} \), witness \( (x \lor y) \land (\neg x \lor y) \land (x \lor \neg y) \) (here “corresponds to” is meant in the technical sense of the previous subsection). However, every definite function has at least one \( \text{RCNF}_{\text{Def}} \) formula representing it, namely \( \text{DBC}_{\text{Def}} \).

For \( \text{RCNF}_{\text{Def}} \) formulas, the operations: renaming, meet, and restriction have the same worst case cost as for RCNF. The computation of meet can be sped up in practice by noting that if
$F$ is the $\text{RCNF}_{Def}$ formula $\bigwedge_{x \in \text{Var}} (x \leftarrow M_x)$ and $F'$ is the $\text{RCNF}_{Def}$ formula $\bigwedge_{x \in \text{Var}} (x \leftarrow M'_x)$, then a $\text{RCNF}_{Def}$ function representing $F \land F'$ is

$$\bigwedge_{x \in \text{Var}} (x \leftarrow \text{RCNF}(M_x \lor M'_x)).$$

Similarly we can speed up the computation of restriction: By Proposition 4.5, if $F$ is the $\text{RDNF}_{Def}$ formula $\bigwedge_{x \in \text{Var}} (x \leftarrow M_x)$, then a $\text{RCNF}_{Def}$ function representing $\exists x. F$ is

$$(x \leftarrow \text{false}) \land \bigwedge_{y \in \text{Var}, y \neq x} (y \leftarrow \text{RDNF}(M_y[x \mapsto M_x[y \mapsto \text{false}]]))$$

where $F[x \mapsto F']$ denotes the replacement of all occurrences of the variable $x$ in $F$ by the formula $F'$.

**Example 4.4** The function $(x \lor y \lor z) \leftrightarrow (x \land y \land z)$ can be written in RMBF as for example

$$(x \leftarrow (y \land z)) \land (y \leftarrow (x \lor z)) \land (z \leftarrow y).$$

Eliminating $x$ from this formula yields

$$(x \leftarrow \text{false}) \land (y \leftarrow z) \land (z \leftarrow y)$$

which represents the function $y \leftrightarrow z$.  

RCNF$_{Def}$ does not support efficient computation of the join on Def as in general RCNF$_{Def}$ formulas are not in orthogonal form. One option is to convert to DBCF$_{Def}$ and then compute the join. The second option is to use

$$\bigwedge_{x \in \text{Var}} (x \leftarrow \text{RDNF}(M_x \land M'_x)).$$

as an approximation to the join. It can be computed relatively quickly, albeit with some loss of precision.

Although the theoretical worst case complexity of most operations on RCNF$_{Def}$ is the same as for RCNF, in practice they can be expected to be cheaper. One operation which is cheaper is testing equivalence. By Theorem 4.3, we can test equivalence of RCNF$_{Def}$ formula in quadratic time, while testing equivalence of RCNF formula may require exponential time.

## 5 Empirical Evaluation

This section contains results from an empirical investigation of the different representations and their relative cost for performing groundness analysis. The reason for an empirical investigation is that the worst case complexity results from the previous section actually give little indication of the true relative efficiency. In part this is because for all operations the worst case complexity for each representation was given in terms of the size of the operands. As this may vary from representation to representation it is hard to compare the worst case complexities directly. Although we have only evaluated the representations for groundness analysis, we believe that our findings hold for other applications using propositional dependency formulas.
5.1 Analysis Framework

We first sketch our analysis framework, and then discuss the implementation of the various representations discussed earlier. The implementation is a hybrid: The high-level engine is written in Prolog, and the low-level operations are written in C. The analysis is divided into three phases: First the input file is read and a list of ground representations of the Clark completions of the predicates is collected. Second, the strongly-connected components (SCCs) in the program’s call graph are collected in topological order. Finally, the program is analyzed bottom up, one SCC at a time.

The analysis of each SCC is done in two stages. First, we collect a list of pairs for each predicate in that SCC, one pair for each clause\(^4\). The first component of each pair, the variable part, is a list of calls to predicates in that same SCC. The second component, the fixed part, is the analysis of all explicit unifications and all calls to predicates defined outside that SCC (which will already have been analyzed). We also compute the least upper bound of the fixed parts of all the clauses for which the variable part is empty. This is our first approximation of the analysis of that predicate, because only clauses with no variable part (that is, no calls to predicates in the same SCC) can contribute to the first approximate analysis of a predicate.

The second part of the analysis of an SCC is the fixpoint iteration. As mentioned above, for non-recursive predicates, all of whose clauses will have empty variable parts, the first approximation is correct, so no further work will be done. For all other SCCs, we repeatedly analyze all predicates in that SCC until a fixpoint is reached.

Our analyzer consists of about 1500 lines (including comments) of Prolog code, with all operations on propositional formulas implemented in ANSI C for speed.

5.2 Implementation of the Different Representations

This section contains a description of the C implementation of the various representations for Boolean functions. The implementations are built around a common interface to the analyzer but have significantly different data structures and algorithms.

For all representations, a variable in a clause and its corresponding formula is represented as a positive integer. The arguments of the atom in the clause head are always numbered from one through to the predicate arity. Other variables in the clause are assigned numbers greater than the head atom’s arity.

The implementation of ROBDD does not have an upper limit on the number of variables in a clause. All the other representations have been implemented with a maximum variable number of 64, although this can be increased if required.

The main operations, introduced in Section 4, used to perform the analysis are: equivalence, meet, join, restriction and renaming. The equivalence, meet and join operations are as described previously. Restriction, however, restricts all variables above a threshold, rather than restricting a single variable. Restriction is used to restrict the Boolean function corresponding to the clause body to the arguments of the clause head. It is more efficient to perform all restrictions at

\(^4\)This is an abuse of the word clause, as we have formed the Clark completion of each predicate. By clause we mean a single disjunct in the body of the Clark completion.
once, rather than to iteratively restrict away single variables one at a time. Restriction of a single variable is also possible. The renaming operation simultaneously renames a number of variables. Simultaneous renaming of variables is essential as iteratively renaming single variables will produce incorrect results when the renamings are not independant of each other.

In addition to the above operations, several other operations have been provided to create and manipulate the representations:

- Given two formulas $F$ and $G$, return the formula corresponding to $F \rightarrow G$.
- Create a copy of a representation.
- Given a variable number, represent the variable in the appropriate representation.
- Create a representation of $x \leftarrow (\bigwedge_{i=1}^{m} y_i)$, given the variable numbers $x$ and $y_i$.
- Delete a representation.

All use destructive update when this is more efficient.

ROBDDs are implemented using the basic implementation sketched by Brace et al. [6]. The ile constant algorithm is used for testing equivalence. Renaming of an individual variable is performed by equating the old value of the variable to the new one, taking the meet with the function, and then restricting the old names away. Before renaming an atom, it is necessary to find all the strongly connected components of the renaming, introduce temporary variables when required, and then rename each variable iteratively. Restriction in ROBDD is performed by finding the first variable greater than the threshold value. If all leaves of the subtree are false, the pointer to the subtree is changed to point to false, otherwise the pointer is changed to point to true. The changes are propagated up the tree so as to retain the canonical form.

Each term in the implementation of RDNF is represented by two arrays of 32 bit integers. One array corresponds to the positive variables within the term. The other array denotes negated variables within a term. The $k$th bit of the $n$th integer is set when the variable $x_{2^k-1}$ is included in the term. Each of the terms are linked together in an unordered list. It was found that an unordered list was more efficient than an ordered list. Ordering variables reduces the complexity of operations such as equivalence and testing if the DNF formula is reduced, but keeping the list ordered increases the time complexity for the more common operations such as meet and join. Testing equivalence in RDNF is performed by converting the two formulas to be compared into BCF, and then checking that each formula absorbs the other. The syllogizing and absorption used to compute the canonical form are not performed as two separate steps, but as one combined iteration. This reduces the number of intermediate terms produced, and so speeds up the conversion. Restriction is performed by simply deleting the required literal wherever it appears. Renaming is performed by adding all the new literals to all terms where the old literals appears, and deleting the old literals.

The BCF implementation is a variant of RDNF. The RDNF and BCF data structures are identical, but each term in BCF represents a prime implicant, whereas this is not necessarily so for RDNF. The operations in BCF are performed in the same way as RDNF, except that care must be taken to preserve the canonical form. It is necessary to recompute the canonical form after the operations restrict, implies and meet.
RCNF has the same data structure as RDNF, but each term represents a disjunction of literals rather than a conjunction, and the links between terms represent conjunction. Restriction is performed by creating copies with the variable to be removed set to true and false, and joining the two formulas.

DBCF is a variant of RCNF in the same way that BCF is a variant of RDNF. Restriction is performed by deleting any clause which contains the variable to be restricted, and this retains the canonical form, unlike BCF.

The two implementations of definite-specific representations are variants of $\text{RCNF}_{D_{def}}$. The implementations differ in whether or not the join is precise. The non-precise implementation, called $\text{RCNF}_{D_{def}}^{jack}$, uses the quick imprecise join while the precise implementation $\text{RCNF}_{D_{def}}^{prec}$, converts the formula to $\text{DBCF}_{D_{def}}$ before performing the join operation. $\text{DBCF}_{D_{def}}$ has not been implemented, as the precision is the same as for $\text{RCNF}_{D_{def}}^{prec}$, but $\text{DBCF}_{D_{def}}$ incurs greater costs as it must maintain the canonical form.

The data structure for the implementations of $\text{RCNF}_{D_{def}}$ is an ordered list of formulas of the form $x \leftarrow M$. Each of these is implemented as a head, $x$, and a pointer to the body $M$. The body is represented as for RDNF except that negated variables are not needed. The implementations do not use the quadratic algorithm given in Section 4 for testing equivalence of definite sentences. This is because converting the representation into a form suitable for the algorithm, and then performing the test, was found to be slower than computing the $\text{DBCF}_{D_{def}}$ followed by comparison of bodies for corresponding heads. Presumably the reason is that the formulas being compared have relatively few variables.

The representations for $\text{Def}$ are relatively simple, and the implementations are smaller than for $\text{Pos}$. ROBDDs have been easy to implement as algorithms were generally available, and the code is compact since most operations use the same function (’ite’). Thus the implementation effort has been largest for the disjunctive and conjunctive forms for $\text{Pos}$.

5.3 Test Results

We have evaluated our implementations with a test suite including the programs used in [23]. Table 1 shows various statistics about these tests. The first column shows the number of strongly-connected components in the test, followed by the number of predicates and the total number of clauses. Next we show the average and maximum number variables in all the clauses in the test, and finally the average and maximum number of arguments (arity) of the predicates in the test.

Table 2 shows the results. All times are given as the average of 5 runs of the program. Testing was performed on a Sparcstation 2 with a local swap disk and 64 megabytes of main memory. The Prolog code was compiled with Quintus Prolog version 3.1.4; the C code was compiled with GNU CC version 2.5.7, optimized with -O2. The first column of the table shows the time to read the source program, collecting the Clark completion of each predicate; the second column shows the time to find the SCCs. Both of these are independent of the representation chosen. The succeeding columns show the time to perform the analysis using our implementations of the various representations of the Boolean functions. Note that these times include only bottom-up analysis, which determines only success patterns, not call patterns. A separate top-down pass is required to determine the call patterns.
<table>
<thead>
<tr>
<th></th>
<th>SCCs</th>
<th>Preds</th>
<th>Clauses</th>
<th>Avg Vars</th>
<th>Max Vars</th>
<th>Avg Args</th>
<th>Max Args</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4.67</td>
<td>7</td>
<td>1.50</td>
<td>3</td>
</tr>
<tr>
<td>cs</td>
<td>36</td>
<td>38</td>
<td>65</td>
<td>5.57</td>
<td>21</td>
<td>2.63</td>
<td>9</td>
</tr>
<tr>
<td>disj</td>
<td>35</td>
<td>35</td>
<td>64</td>
<td>4.50</td>
<td>13</td>
<td>1.94</td>
<td>4</td>
</tr>
<tr>
<td>gabriel</td>
<td>21</td>
<td>21</td>
<td>46</td>
<td>5.35</td>
<td>14</td>
<td>2.81</td>
<td>6</td>
</tr>
<tr>
<td>kalah</td>
<td>42</td>
<td>46</td>
<td>89</td>
<td>5.62</td>
<td>15</td>
<td>2.70</td>
<td>9</td>
</tr>
<tr>
<td>peep</td>
<td>19</td>
<td>22</td>
<td>228</td>
<td>6.43</td>
<td>11</td>
<td>3.05</td>
<td>6</td>
</tr>
<tr>
<td>pg</td>
<td>11</td>
<td>11</td>
<td>19</td>
<td>5.89</td>
<td>12</td>
<td>2.82</td>
<td>5</td>
</tr>
<tr>
<td>plan</td>
<td>17</td>
<td>17</td>
<td>30</td>
<td>4.47</td>
<td>8</td>
<td>2.24</td>
<td>4</td>
</tr>
<tr>
<td>press1</td>
<td>49</td>
<td>53</td>
<td>159</td>
<td>5.53</td>
<td>12</td>
<td>2.70</td>
<td>6</td>
</tr>
<tr>
<td>press2</td>
<td>49</td>
<td>53</td>
<td>165</td>
<td>5.55</td>
<td>12</td>
<td>2.70</td>
<td>6</td>
</tr>
<tr>
<td>qsort</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>5.57</td>
<td>9</td>
<td>2.25</td>
<td>4</td>
</tr>
<tr>
<td>queens</td>
<td>6</td>
<td>6</td>
<td>10</td>
<td>4.40</td>
<td>8</td>
<td>1.83</td>
<td>3</td>
</tr>
<tr>
<td>read</td>
<td>31</td>
<td>43</td>
<td>167</td>
<td>5.10</td>
<td>13</td>
<td>2.81</td>
<td>7</td>
</tr>
<tr>
<td>bid</td>
<td>22</td>
<td>22</td>
<td>53</td>
<td>4.57</td>
<td>9</td>
<td>2.41</td>
<td>4</td>
</tr>
<tr>
<td>boyer</td>
<td>28</td>
<td>29</td>
<td>139</td>
<td>5.13</td>
<td>10</td>
<td>2.34</td>
<td>5</td>
</tr>
<tr>
<td>boyer2</td>
<td>26</td>
<td>27</td>
<td>141</td>
<td>5.15</td>
<td>10</td>
<td>2.33</td>
<td>3</td>
</tr>
<tr>
<td>bp6</td>
<td>15</td>
<td>16</td>
<td>34</td>
<td>7.76</td>
<td>20</td>
<td>3.00</td>
<td>7</td>
</tr>
<tr>
<td>deriv</td>
<td>15</td>
<td>15</td>
<td>62</td>
<td>5.26</td>
<td>9</td>
<td>2.67</td>
<td>4</td>
</tr>
<tr>
<td>peephole</td>
<td>12</td>
<td>17</td>
<td>135</td>
<td>5.30</td>
<td>10</td>
<td>2.24</td>
<td>3</td>
</tr>
<tr>
<td>rdtok</td>
<td>12</td>
<td>18</td>
<td>55</td>
<td>5.55</td>
<td>9</td>
<td>2.89</td>
<td>5</td>
</tr>
<tr>
<td>semi20</td>
<td>21</td>
<td>21</td>
<td>47</td>
<td>6.94</td>
<td>13</td>
<td>2.90</td>
<td>7</td>
</tr>
<tr>
<td>splay</td>
<td>10</td>
<td>10</td>
<td>23</td>
<td>9.83</td>
<td>16</td>
<td>3.30</td>
<td>6</td>
</tr>
<tr>
<td>bryant</td>
<td>24</td>
<td>31</td>
<td>57</td>
<td>8.70</td>
<td>24</td>
<td>4.97</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 1: Various statistics on the test suite

Table 3 illustrates the precision of the two representations for definite functions, relative to that for positive functions. The first column shows the number of predicates in the program. The second column shows how many of these predicates had a weaker result for $\text{RCNF}^{\text{prec}}_{D_{def}}$ than for $\text{Pos}$. The fourth column shows the same, but for the less precise $\text{RCNF}^{\text{black}}_{D_{def}}$.

We conclude that using positive functions generally achieves significantly higher precision than using definite functions, at a small extra cost. Comparing the two implementations of definite functions, it appears that forcing the formulas to be orthogonal before computing the join is inexpensive, but then it improves results only marginally. Taking implementation effort into account, we conclude that one should not force orthogonality.

6 Related Work

Definite functions, under the name of dependency formulas, were introduced by Dart and used for groundness analysis in deductive databases [16, 17]. Finiteness dependencies for Datalog were
<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>SCCs</th>
<th>ROBDD</th>
<th>RDNF</th>
<th>BCF</th>
<th>RCNF</th>
<th>DBCF</th>
<th>RCNF^{prec} _\text{Def}</th>
<th>RCNF^{exact} _\text{Def}</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cs</td>
<td>352</td>
<td>37</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>350</td>
<td>366</td>
<td>183</td>
<td>183</td>
</tr>
<tr>
<td>disj</td>
<td>266</td>
<td>31</td>
<td>167</td>
<td>167</td>
<td>250</td>
<td>233</td>
<td>216</td>
<td>166</td>
<td>150</td>
</tr>
<tr>
<td>gabriel</td>
<td>219</td>
<td>18</td>
<td>116</td>
<td>100</td>
<td>134</td>
<td>133</td>
<td>117</td>
<td>117</td>
<td>100</td>
</tr>
<tr>
<td>kalah</td>
<td>501</td>
<td>61</td>
<td>250</td>
<td>250</td>
<td>300</td>
<td>367</td>
<td>300</td>
<td>233</td>
<td>200</td>
</tr>
<tr>
<td>peep</td>
<td>1140</td>
<td>42</td>
<td>400</td>
<td>433</td>
<td>484</td>
<td>583</td>
<td>567</td>
<td>467</td>
<td>433</td>
</tr>
<tr>
<td>pg</td>
<td>109</td>
<td>7</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>66</td>
<td>33</td>
<td>50</td>
<td>33</td>
</tr>
<tr>
<td>plan</td>
<td>145</td>
<td>11</td>
<td>50</td>
<td>66</td>
<td>67</td>
<td>67</td>
<td>83</td>
<td>50</td>
<td>83</td>
</tr>
<tr>
<td>press1</td>
<td>1251</td>
<td>76</td>
<td>366</td>
<td>384</td>
<td>366</td>
<td>450</td>
<td>466</td>
<td>350</td>
<td>366</td>
</tr>
<tr>
<td>press2</td>
<td>781</td>
<td>74</td>
<td>333</td>
<td>433</td>
<td>400</td>
<td>433</td>
<td>500</td>
<td>400</td>
<td>333</td>
</tr>
<tr>
<td>qsort</td>
<td>40</td>
<td>2</td>
<td>0</td>
<td>16</td>
<td>16</td>
<td>17</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>queens</td>
<td>54</td>
<td>3</td>
<td>17</td>
<td>16</td>
<td>17</td>
<td>33</td>
<td>34</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>read</td>
<td>820</td>
<td>59</td>
<td>533</td>
<td>566</td>
<td>617</td>
<td>684</td>
<td>666</td>
<td>517</td>
<td>516</td>
</tr>
<tr>
<td>bid</td>
<td>230</td>
<td>19</td>
<td>100</td>
<td>83</td>
<td>83</td>
<td>100</td>
<td>100</td>
<td>116</td>
<td>100</td>
</tr>
<tr>
<td>boyer</td>
<td>780</td>
<td>36</td>
<td>216</td>
<td>266</td>
<td>250</td>
<td>400</td>
<td>400</td>
<td>266</td>
<td>266</td>
</tr>
<tr>
<td>boyer2</td>
<td>1252</td>
<td>33</td>
<td>200</td>
<td>234</td>
<td>217</td>
<td>283</td>
<td>266</td>
<td>234</td>
<td>266</td>
</tr>
<tr>
<td>bp0-6</td>
<td>331</td>
<td>18</td>
<td>217</td>
<td>583</td>
<td>1066</td>
<td>1017</td>
<td>1250</td>
<td>183</td>
<td>150</td>
</tr>
<tr>
<td>deriv</td>
<td>326</td>
<td>21</td>
<td>134</td>
<td>183</td>
<td>150</td>
<td>167</td>
<td>217</td>
<td>200</td>
<td>133</td>
</tr>
<tr>
<td>peephole</td>
<td>689</td>
<td>32</td>
<td>333</td>
<td>400</td>
<td>350</td>
<td>484</td>
<td>467</td>
<td>383</td>
<td>366</td>
</tr>
<tr>
<td>rdtok</td>
<td>376</td>
<td>21</td>
<td>217</td>
<td>167</td>
<td>184</td>
<td>200</td>
<td>233</td>
<td>200</td>
<td>216</td>
</tr>
<tr>
<td>semi20</td>
<td>366</td>
<td>19</td>
<td>183</td>
<td>200</td>
<td>216</td>
<td>333</td>
<td>317</td>
<td>183</td>
<td>116</td>
</tr>
<tr>
<td>splay</td>
<td>236</td>
<td>10</td>
<td>100</td>
<td>484</td>
<td>233</td>
<td>217</td>
<td>233</td>
<td>67</td>
<td>100</td>
</tr>
<tr>
<td>bryant</td>
<td>591</td>
<td>34</td>
<td>1716</td>
<td>14850</td>
<td>89033</td>
<td>1217</td>
<td>1250</td>
<td>800</td>
<td>783</td>
</tr>
</tbody>
</table>

Table 2: Benchmark timings, in milliseconds


The history of positive functions in program analysis is somewhat shorter. The use of positive functions was suggested for groundness analysis by Marriott and Søndergaard [24] (under the less suggestive name ‘Prop’) and further studied by Cortesi et al. [15]. Bigot et al. [4] give a precise finiteness analysis which is based on positive functions. Both positive and definite functions have been suggested as the basis for groundness analysis of constraint logic programs, normal logic programs, logic programs with dynamic scheduling and for suspension analysis of concurrent logic languages.

Apart from the mentioned work on analysis, there are only few studies of the positive\(^5\) func-

\(^5\)Some authors, including Chang and Keisler [9] and Dart [17] refer to what is commonly called a ‘monotonic’ function as ‘positive.’ We use the more common terminology (although in [24] the elements of Pos were erroneously called ‘monotonic’).
<table>
<thead>
<tr>
<th>Preds</th>
<th>RCNF\textsuperscript{prec}</th>
<th></th>
<th></th>
<th>RCNF\textsuperscript{slack}</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Preds</td>
<td>Weaker than Pos</td>
<td>% weaker</td>
<td>Preds</td>
<td>Weaker than Pos</td>
<td>% weaker</td>
</tr>
<tr>
<td>append</td>
<td>2</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>cs</td>
<td>38</td>
<td>5</td>
<td>13.16</td>
<td>5</td>
<td>13.16</td>
<td></td>
</tr>
<tr>
<td>disj</td>
<td>35</td>
<td>4</td>
<td>11.43</td>
<td>4</td>
<td>11.43</td>
<td></td>
</tr>
<tr>
<td>gabriel</td>
<td>21</td>
<td>4</td>
<td>19.05</td>
<td>4</td>
<td>19.05</td>
<td></td>
</tr>
<tr>
<td>kalah</td>
<td>46</td>
<td>15</td>
<td>32.61</td>
<td>15</td>
<td>32.61</td>
<td></td>
</tr>
<tr>
<td>peep</td>
<td>22</td>
<td>8</td>
<td>36.36</td>
<td>8</td>
<td>36.36</td>
<td></td>
</tr>
<tr>
<td>pg</td>
<td>11</td>
<td>0</td>
<td>0.00</td>
<td>1</td>
<td>9.09</td>
<td></td>
</tr>
<tr>
<td>plan</td>
<td>17</td>
<td>1</td>
<td>5.88</td>
<td>1</td>
<td>5.88</td>
<td></td>
</tr>
<tr>
<td>press1</td>
<td>53</td>
<td>3</td>
<td>5.66</td>
<td>6</td>
<td>11.32</td>
<td></td>
</tr>
<tr>
<td>press2</td>
<td>53</td>
<td>3</td>
<td>5.66</td>
<td>6</td>
<td>11.32</td>
<td></td>
</tr>
<tr>
<td>qsort</td>
<td>4</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>queens</td>
<td>6</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>43</td>
<td>10</td>
<td>23.26</td>
<td>13</td>
<td>30.23</td>
<td></td>
</tr>
<tr>
<td>bid</td>
<td>22</td>
<td>6</td>
<td>27.27</td>
<td>6</td>
<td>27.27</td>
<td></td>
</tr>
<tr>
<td>boyer</td>
<td>29</td>
<td>4</td>
<td>13.79</td>
<td>4</td>
<td>13.79</td>
<td></td>
</tr>
<tr>
<td>boyer2</td>
<td>27</td>
<td>2</td>
<td>7.41</td>
<td>2</td>
<td>7.41</td>
<td></td>
</tr>
<tr>
<td>bp0-6</td>
<td>16</td>
<td>4</td>
<td>25.00</td>
<td>4</td>
<td>25.00</td>
<td></td>
</tr>
<tr>
<td>deriv</td>
<td>15</td>
<td>1</td>
<td>6.67</td>
<td>1</td>
<td>6.67</td>
<td></td>
</tr>
<tr>
<td>peephole</td>
<td>17</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>rdtok</td>
<td>18</td>
<td>2</td>
<td>11.11</td>
<td>5</td>
<td>27.78</td>
<td></td>
</tr>
<tr>
<td>semi20</td>
<td>21</td>
<td>7</td>
<td>33.33</td>
<td>7</td>
<td>33.33</td>
<td></td>
</tr>
<tr>
<td>splay</td>
<td>10</td>
<td>6</td>
<td>60.00</td>
<td>6</td>
<td>60.00</td>
<td></td>
</tr>
<tr>
<td>bryant</td>
<td>31</td>
<td>24</td>
<td>77.42</td>
<td>24</td>
<td>77.42</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>557</td>
<td>109</td>
<td>19.57</td>
<td>122</td>
<td>21.90</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Precision of $\text{RCNF}_{\text{Def}}^{\text{prec}}$ and $\text{RCNF}_{\text{Def}}^{\text{slack}}$, relative to $\text{Pos}$.

tions. The class does play a role in Post’s criterion for functional completeness [27]. Also, Hilbert and Bernays [21] discuss a “positive Logik” which is intended to be the part of propositional calculus that is independent of a concept of negation. It can thus be extended to either full classical propositional calculus or intuitionistic propositional calculus, for example. The axiomatization of positive logic does not have a classical tautology such as $((x \rightarrow y) \rightarrow x) \rightarrow x$ (an instance of Peirce’s Law) as a theorem. The reader may wonder about the status of such a formula in our system, but the fact is that the analyzer will never create a formula of the form $(F \rightarrow F’) \rightarrow F’$, whatever representation is used.

Several independent implementations have recently indicated that groundness analysis based on positive functions is very accurate and is perfectly practical for “real-world” programs [11, 14, 23]. However, little work that we know of has been devoted to improving implementations by investigating different representations for the Boolean functions, including positive and definite
functions. Recent implementations seem to favor ROBDDs. Le Charlier and Van Hentenryck [23] and Baker and Søndergaard [3] represent positive functions by ROBDDs, for example. Codish and Demoen [11] use a simple representation of positive functions based on Prolog terms which appears to give good performance for small programs.

7 Conclusion

We have systematically examined two subclasses of Boolean functions, the positive functions and the definite functions. These functions are important because they naturally arise in dependency based analyses such as groundness or finiteness. We have studied the algebraic properties of these subclasses and also looked at different representations and implementations.

As one might expect, it seems possible to obtain faster analysis by using a more restrictive representation that can only handle definite functions. However, the gain appears rather limited and the cost is a significant loss of precision. However, if one does use definite functions then it seems acceptable to implement a simple version that does not maintain “orthogonality” of clauses and thus computes an imprecise join, as the loss of precision seems marginal.

Of the studied methods to implement positive functions, the most efficient ones use either the (by now standard) ROBDD representation or a (reduced) conjunctive normal form representation.

Although our tests were in the setting of groundness analysis, we believe that these findings extend to other dependency based analyses. It remains to be seen whether the use of an implicative fragment of propositional logic could improve dataflow analyses developed for functional programming languages. This would seem quite plausible. For example, the “constraint” approach to higher-order binding-time analysis [20] involves constraint systems that appear to have a good deal of affinity with implicative propositional logic.

All the representations we have used for positive functions have one thing in common: they are all more expressive than needed. All can be used for the full set of Boolean functions. It is possible that there are natural representations specialized for Pos that allow higher efficiency by trading in expressiveness.

Acknowledgements

We thank Manuel Hermenegildo and Pascal Van Hentenryck for making test suites available.

References


