Two classes of Boolean functions for dependency analysis

Tania Armstrong\textsuperscript{a}, Kim Marriott\textsuperscript{a,\*}, Peter Schachte\textsuperscript{b}, Harald Søndergaard\textsuperscript{b}

\textsuperscript{a} Department of Computer Science, Monash University, Clayton 3168, Australia
\textsuperscript{b} Department of Computer Science, The University of Melbourne, Parkville 3052, Australia

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 a form specific to definite functions. We compare the resulting implementations of groundness analyzers based on the representations for precision and efficiency. © 1998 Elsevier Science B.V. All rights reserved.

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. The work described here extends what was presented at SAS'94 [1].

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, we establish that 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. For most analyses a dependency formula will typically 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 a form 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.

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 ground-ness 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|Xs], 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, whenever quicksort succeeds, the variable will be bound to a list of numbers. This behavior is captured by the function \( xs \leftarrow ys \). 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 [31]. As a first step we translate the program to its Clark completion [12]. 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

\[
q(x_s, y_s) \leftarrow \\
\quad d(x_s, y_s, \text{nil}) \\
\quad d(x_s, y_s, z_s) \leftarrow \\
\quad (x_s = \text{nil} \land y_s = z_s) \\
\quad \lor \exists x, x'_s, lows, highs, u_s, u'_s. [x_s = x \land p(x'_s, x, lowns, highs) \\
\quad \land u'_s = x : us \land d(\text{lows}, y_s, u'_s) \land d(\text{highs}, u_s, z_s)]
\]

\[
p(x_s, e, lows, highs) \leftarrow \\
\quad (x_s = \text{nil} \land lowns = \text{nil} \land highs = \text{nil}) \\
\quad \lor \exists x, x'_s, lowns'. [x_s = x \land lowns = x : lowns' \land \\
\quad x \leq e \land p(x'_s, e, lowns', highs)] \\
\quad \lor \exists x, x'_s, highs'. [x_s = x \land highs = x : highs' \land \\
\quad x > e \land p(x'_s, e, lowns, highs')].
\]

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:

\[
q(x_s, y_s) = \\
\quad d(x_s, y_s, \text{true}) \\
\quad d(x_s, y_s, z_s) = \\
\quad (x_s \land (y_s \rightarrow z_s)) \\
\quad \lor \exists x, x'_s, lowns, highs, u_s, u'_s. [(x_s \leftrightarrow (x \land x'_s)) \land p(x'_s, x, lowns, highs) \\
\quad \land (u'_s \leftrightarrow (x \land u_s)) \land d(\text{lows}, y_s, u'_s) \land d(\text{highs}, u_s, z_s)]
\]

\[
p(x_s, e, lowns, highs) = \\
\quad (x_s \land lowns \land highs) \\
\quad \lor \exists x, x'_s, lowns'. \\
\quad [(x_s \leftrightarrow (x \land x'_s)) \land (lowns \leftrightarrow (x \land lowns')) \land x \land e \land p(x'_s, e, lowns', highs)] \\
\quad \lor \exists x, x'_s, highs'. \\
\quad [(x_s \leftrightarrow (x \land x'_s)) \land (highs \leftrightarrow (x \land highs')) \land x \land e \land p(x'_s, e, lowns, highs')].
\]

There are several points to notice here. The translation of the constraint \(x_s = \text{nil} \land lowns = \text{nil} \land highs = \text{nil}\) is the Boolean function \(x \land lowns \land highs\), which expresses that all three variables become ground if the first clause is selected. The translation of \(\text{x}'_s = x : x'_s\) is slightly more complex. The function \(x_s \leftrightarrow (x \land x'_s)\) expresses the groundness dependencies amongst the three variables, namely "if \(x_s\) 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 [32] for a justification.

Notice that the equations could be simplified at this point, by utilizing Schröder's Elimination Principle

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

We may, for example, simplify the definition of p as follows:

\[
p(xs, e, lows, highs) = (xs \land lows \land highs) \\
\lor \exists xs', lows',[(xs \leftrightarrow xs') \land (lows \leftrightarrow lows')] \\
\land e \land p(xs', e, lows', highs)] \\
\lor \exists xs', highs',[(xs \leftrightarrow xs') \land (highs \leftrightarrow highs')] \\
\land e \land p(xs', e, lows, highs')
\]

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

Notice how existential quantification worked smoothly, even though the formula contained a recursive call to \( p \); in general, we may not be able to eliminate quantifiers without introducing the constants false and true.

The last step in the analysis is to solve the set of recursive Boolean equations. The quicksort program has the call graph shown in Fig. 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, dquicksort, 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:

\[ 1 \text{The first explicit statement of the principle appears to be by Schröder [40, p. 22], who derived it from Boole's principle of "development": } F = (F[x \leftarrow false] \land \neg x) \lor (F[x \leftarrow true] \land x) \text{ (sometimes referred to as Boole's Expansion Theorem, or "Shannon expansion"). Boole considered disjunction to be exclusive, so the elimination principle would have made little sense to him.} \]
Fig. 1. Call graph for the quicksort program.

\[ p_0(xs, e, lows, highs) = \text{false} \]
\[ p_1(xs, e, lows, highs) = xs \land lows \land highs \]
\[ p_2(xs, e, lows, highs) = (xs \land lows \land highs) \lor (e \land xs \land lows \land highs) \]

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

\[ d_0(xs, ys, zs) = \text{false} \]
\[ d_1(xs, ys, zs) = xs \land (ys \leftrightarrow zs) \]
\[ d_2(xs, ys, zs) = (xs \land (ys \leftrightarrow zs)) \lor (\neg ys \land \neg zs) \lor (\neg xs \land \neg ys \land zs) \]
\[ = ys \leftrightarrow (xs \land zs). \]

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

\[ q(xs, ys) = d(xs, ys, true) \]
\[ = xs \leftrightarrow ys. \]

In other words, if one of the arguments given to \texttt{quicksort} is ground, and \texttt{quicksort} succeeds, 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. We can approximate this information, and again the idea is to mimic the execution of a given query, replacing resolution and constraint solving with corresponding operations on the Boolean functions. The simplest way to do this is to evaluate the Boolean recurrences we created earlier, recording the calls rather than the results.

Assume that we are interested in the call patterns that could possibly occur as a consequence of calling \texttt{quicksort} with a ground first argument. By a call pattern we mean a pair \((A, \phi)\), 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.

One way of computing call patterns uses a variant (due to M. Codish) of the so-called magic set transformation [13]. The idea is to extend the original program with clauses that express the relations among calls that can take place when the program is executed. This program transformation works in such a way that a bottom up analysis
of the extended program will provide call pattern information. Codish's transformation adds the following clauses to the example program:

```prolog
call_d(Xs, Ys, Zs) :-
    Zs = nil,
    call_q(Xs, Ys).

call_d(Lows, Ys, Us') :-
    Xs' = [X|Xs],
    call_d(Xs', Ys, Zs),
    Us' = [X|Us],
    p(Xs, X, Lows, Highs).

call_d(Highs, Us, Zs) :-
    Xs' = [X|Xs],
    call_d(Xs', Ys, Zs),
    p(Xs, X, Lows, Highs),
    Us' = [X|Us],
    d(Lows, Ys, Us').

call_p(Xs, X, Lows, Highs) :-
    Xs' = [X|Xs],
    call_d(Xs', Ys, Zs).

call_p(Xs, E, Lows, Highs) :-
    Xs' = [X|Xs],
    Lows' = [X|Lows],
    call_p(Xs', E, Lows', Highs),
    X =< E.

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

The first clause, for example, says that if there is a call quicksort(Xs,Ys) (or an instance thereof) then there will be a call dquicksort(Xs,Ys,nil) (or the corresponding instance thereof). This clause was generated from the first clause of the original program. Similarly, the original clause

dquicksort([X|Xs], Ys, Zs) :-
    partition(Xs, X, Lows, Highs),
    dquicksort(Lows, Ys, [X|Us]),
    dquicksort(Highs, Us, Zs).
```
gives rise to the second, third, and fourth clauses above. For example, the second clause says that if there is a call to \texttt{dquicksort([X|Xs],Ys,Zs)}, and if furthermore \texttt{partition(Xs,E,Lows,Highs)} succeeds, then there is a call to \texttt{dquicksort(Lows, Ys, [X|Us])}, and so on. In general, each clause

\[ A_0 := A_1, \ldots, A_n. \]

whose body contains \( n \) atoms, gives rise to \( n \) new clauses

\[
\begin{align*}
call.A_1 & := call.A_0. \\
call.A_2 & := call.A_0, A_1. \\
& \vdots \\
call.A_n & := call.A_0, A_1, \ldots, A_{n-1}.
\end{align*}
\]

Returning to the example, the new clauses in turn add the following to the Clark completion:

\[
\begin{align*}
cd(x_1,x_2,x_3) & \leftrightarrow \\
x_3 & = \text{nil} \land \text{cq}(x_1,x_2) \\
& \lor \exists y_1,\ldots,y_6. [y_1 = y_2 : y_3 \land cd(y_1,x_2,y_4) \land x_3 = y_2 : y_5 \land p(y_3,y_2,x_1,y_6)] \\
& \lor \exists y_1,\ldots,y_6. [y_1 = y_2 : y_3 \land cd(y_1,y_4,x_3) \land p(y_3,y_2,y_5,x_1) \\
& \land y_6 = y_2 : x_2 \land d(y_5,y_4,y_6)]
\end{align*}
\]

\[
\begin{align*}
\text{cp}(x_1,x_2,x_3,x_4) & \leftrightarrow \\
& \exists y_1, y_2, y_3. [y_1 = x_2 : x_1 \land \text{cd}(y_1,y_2,y_3)] \\
& \lor \exists y_1, y_2, y_3. [y_1 = y_2 : x_1 \land y_3 = y_2 : x_3 \land \text{cp}(y_1,x_2,y_3,x_4) \land y_2 \leq x_2] \\
& \lor \exists y_1, y_2, y_3. [y_1 = y_2 : x_1 \land y_3 = y_2 : x_4 \land \text{cp}(y_1,x_2,x_3,y_3) \land y_2 > x_2].
\end{align*}
\]

We now translate these formulas to recursive definitions of Boolean functions, in exactly the same way as before. However, at this stage, \( p \) and \( d \) are known, so we may as well replace the references to those two functions as we go:

\[
\begin{align*}
cd(x_1,x_2,x_3) & = \\
x_3 & \land \text{cq}(x_1,x_2) \\
& \lor \exists y_1,\ldots,y_6. [(y_1 \leftrightarrow (y_2 \land y_3)) \land cd(y_1,x_2,y_4) \\
& \land (x_3 \leftrightarrow (y_2 \land y_5)) \land y_3 \land x_1 \land y_6] \\
& \lor \exists y_1,\ldots,y_6. [(y_1 \leftrightarrow (y_2 \land y_3)) \land cd(y_1,y_4,x_3) \land y_3 \land y_5 \land x_1 \\
& \land (y_6 \leftrightarrow (y_2 \land x_2)) \land (y_4 \leftrightarrow (y_5 \land y_6))] \\
\text{cp}(x_1,x_2,x_3,x_4) & = \\
& \exists y_1, y_2, y_3. [(y_1 \leftrightarrow (x_2 \land x_1)) \land cd(y_1,y_2,y_3)]
\end{align*}
\]
\[ \forall \exists y_1, y_2, y_3, \left[ (y_1 \leftrightarrow (y_2 \land x_1)) \land (y_3 \leftrightarrow (y_2 \land x_3)) \right] \\
\land cp(y_1, x_2, y_3, x_4) \land y_2 \land x_2 \] \\
\forall \exists y_1, y_2, y_3, \left[ (y_1 \leftrightarrow (y_2 \land x_1)) \land (y_3 \leftrightarrow (y_2 \land x_4)) \right] \\
\land cp(y_1, x_2, x_3, y_3) \land y_2 \land x_2 \].

Straightforward simplifications now justify replacing these two equations by

\[
\begin{align*}
cd(x_1, x_2, x_3) &= x_3 \land cg(x_1, x_2) \\
&\quad \forall \exists y_1, y_2, y_3, [cd(y_1, x_2, y_4) \land (x_3 \rightarrow y_1) \land x_1] \\
&\quad \forall \exists y_1, y_2, y_3, [cd(y_1, x_2, x_3) \land x_1 \land (y_4 \leftrightarrow (y_1 \land x_2))] \\
cp(x_1, x_2, x_3, x_4) &= \exists y_1, y_2, y_3, [(y_1 \leftrightarrow (x_2 \land x_1)) \land cd(y_1, y_2, y_3)] \\
&\quad \forall (cp(x_1, x_2, x_3, x_4) \land x_2). 
\end{align*}
\]

The assumption that quicksort is called with the first argument ground translates to

\[ cg(x_1, x_2) = x_1 \]

and we can now solve first for \( cd \), then for \( cp \). It is a good exercise to verify that the solution is

\[
\begin{align*}
cd(x_1, x_2, x_3) &= x_1 \\
cp(x_1, x_2, x_3, x_4) &= x_1 \land x_2. 
\end{align*}
\]

We conclude that every call to quicksort will have the first argument ground, while every call to partition will have the first two arguments ground.

The sequence of transformations and simplifications may seem a bit daunting at first, just as the manipulations required to solve the recursive equations. But the important point is that all of these manipulations can be made entirely automatic and, as we shall see, performed efficiently by a computer.

2.2. Finiteness and functional dependencies

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 \( p \)". For details on this kind of analysis see Bigot et al. [5].

A special case of finiteness dependency is the so-called functional dependency. Here the description \( x \rightarrow y \) for a predicate \( p(x, y) \) is interpreted as follows. Let \( R \) be the relation defined by \( p \), and let \( \pi_1 \) and \( \pi_2 \) be the projection functions on pairs. Then, for all \( r, r' \in R, \pi_2(r) = \pi_2(r') \) whenever \( \pi_1(r) = \pi_1(r') \). In other words, for each \( (x, y) \in R, \)
the value $x$ determines the value $y$ uniquely. More generally, $(x_1 \land \ldots \land x_k) \rightarrow y$ indicates that any choice of $(x_1, \ldots, x_k)$ determines $y$ uniquely.

As an example consider the program

$$\begin{align*}
\text{app}(Xs, Ys, Zs) & : - Xs = [], Ys = Zs, \\
\text{app}(Xs, Ys, Zs) & : - Xs = [U|Us], Zs = [U|Vs], \text{app}(Us, Ys, Vs).
\end{align*}$$

A functional dependency analysis should be able to produce the result

$$((xs \land ys) \rightarrow zs) \land ((ys \land zs) \rightarrow xs) \land ((zs \land xs) \rightarrow ys)$$

for $\text{app}(xs, ys, zs)$ [45]. Functional dependency information is useful for many kinds of program optimizations, including goal reordering and the elimination of unnecessary choice points.

2.3. Suspension analysis

Our third example of the use of Boolean functions is for suspension analysis of concurrent logic programming languages [41] and concurrent constraint programming languages [39]. 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 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 [25]. We exemplify this for a typical concurrent logic language, FCP(:) [43].

FCP(:) programs consist of finite sets of guarded clauses which specify rules for 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 $H : - \text{Ask} : \text{Tell} | B$ can occur if the current equation set implies the ask equations $\text{Ask}$ of the clause, and is consistent with the tell equations, $\text{Tell}$. 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 [16]:

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

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

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 \( p(x_1), c(x_2); \{x_1=x_2\} \) 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 developed in [25] 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. The analysis assumes that type information about call patterns has already been computed or provided by the programmer. In this case the type information is that \( c \) is always called with a possibly incomplete list of \( a \)'s. For the above program the recursive equations are

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

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

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

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

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

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

\[
s(ns) = \neg ns.
\]

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 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 \( \text{Man} \) 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 \( x \lor y \) are in \( \text{Pos} \). The function \( x \rightarrow y \) is in \( \text{Def} \) but not in \( \text{Mon} \), and \( x \lor y \) 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 \( x \lor y \) is \( \{\{x\}, \{y\}, \{x, y\}\} \), and this set is not closed under intersection.

Clearly, \( \text{Def} \) and \( \text{Mon}\backslash \{\text{false}\} \) are proper subsets of \( \text{Pos} \). Here we will need \( \text{Mon} \) only as an aid to understanding \( \text{Def} \). The Hasse diagrams in Fig. 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. [20] 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 \( \text{Mon} = \Omega\{\land, \lor, \text{true}, \text{false}\} \), where \text{true} and \text{false} are the (overloaded) constant functions returning true and false, respectively, for all input. For \( \text{Pos} \), the following strengthens a result by Cortesi et al. [20].

**Theorem 3.1.** \( \text{Pos} = \Omega\{\land, \rightarrow \} = \Omega\{\land, \leftrightarrow \} = \Omega\{\lor, \leftrightarrow \} = \Omega\{\leftrightarrow, \rightarrow \} \).
**Proof.** First, from Cortesi et al. [20] we know that \( \text{Pos} = \Omega\{\land, \lor, \leftrightarrow\} \). Alternatively, \( \text{Pos} = \Omega\{\land, \lor, \to\} \), as \( \to \) can be obtained from \( \land \) and \( \lor \), while \( \lor \) can be obtained from \( \land \) and \( \to \) → : \( F \to F' = F \leftrightarrow (F \land F') \).

Now for the two-connective characterizations:
1. \( \lor \) can be defined from \( \to \) → : \( F \lor F' = (F \to F') \to F' \).
2. We already saw that \( \to \) (and therefore \( \lor \)) can be obtained from \( \land \) and \( \to \).
3. \( \land \) can be obtained from \( \lor \) and \( \leftrightarrow \) → : \( F \land F' = (F \lor F') \leftrightarrow (F \leftrightarrow F') \).
4. \( \land \) can be obtained from \( \rightarrow \) and \( \leftrightarrow \) → : \( F \land F' = F \leftrightarrow (F \to F') \).

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

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

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

We now turn to \( \text{Def} \). 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, \( x \lor \neg y_1 \lor \cdots \lor \neg y_n \) is regarded as \( x \leftarrow y_1 \land \cdots \land y_n \). 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 [23] Proposition 3.1.

**Theorem 3.3.** The function \( F \) is in \( \text{Def} \) iff \( F \) can be represented as a definite sentence.

Sometimes it is useful to represent a function in \( \text{Def} \), 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 \( x \leftarrow (y \lor z) \land y \leftarrow \text{false} \land z \leftarrow \text{false} \).

**Definition.** A formula

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

\(^2\)That \( \land \) and \( \to \) form a functionally complete set of connectives for \( \text{Pos} \) was discovered during a conversation between W. Winsborough and H. Sondergaard in August 1992 but the demonstration was more complex than this proof.
is in monotonic body form (MBF) iff each $M_x$ is monotonic. If, furthermore, for no $x$, $x \subseteq M_x$, then $F$ is in reduced MBF (RMBF).

Dart [23] 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, since $x \leftarrow M$ is logically equivalent to $x \leftarrow M[x \leftarrow \text{false}]$, 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,

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

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

**Def** does not inherit all the closure properties of **Pos**. However, the following follows immediately from the definition of **Def**.

**Theorem 3.5.** **Def** is closed under conjunction.

A further immediate consequence is that **Def** is a lattice. Notice that this follows from the (semantic) definition of **Def** — it has nothing to do with its representation.

**Theorem 3.6.** **Def** is a lattice.

**Proof.** **Def** has a largest element, *true*, and a meet operation (by Theorem 3.5). The theorem follows from a standard result in lattice theory. □

**Theorem 3.7.** 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 (*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 $(\phi \cap \phi) \models x \rightarrow F$. □

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 **Def** is not closed under $\leftrightarrow$ either.

The following result is both important and surprising. It is important because 'restricting attention to interesting variables' is an important operation, and this operation is nothing but existential quantification. The result is surprising, considering
existential quantification’s affinity with disjunction, together with the fact that \( \text{Def} \) is not closed under disjunction.

**Theorem 3.8.** 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 \( \phi \models F[x \rightarrow \text{true}] \) and \( \psi \models F[x \rightarrow \text{true}] \). 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 \( (\phi \cap \psi) \models F[x \rightarrow \text{true}] \).

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

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

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

The join on \( \text{Def} \) – let us denote it by \( \lor \) – must be different from that on \( \text{Pos} \), that is, it is not classical disjunction. Dart [23] notes that the meet can be calculated from two definite formulas (exactly as for full propositional logic) as follows: Let

\[
F = \bigwedge_{x \in \text{Var}} (x \leftarrow M_x) \quad \text{and} \quad F' = \bigwedge_{x \in \text{Var}} (x \leftarrow M'_x),
\]

where the \( M_x \) and \( M'_x \) are monotonic formulas. Then \( F \land F' \) is

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

However, Dart does not indicate how \( \lor \) can be computed. One might hope that by duality the join would be given by

\[
F'' = \bigwedge_{x \in \text{Var}} (x \leftarrow (M_x \land M'_x)).
\]

To see that this is not the case, consider

\[
F = (x \leftarrow y) \land (y \leftarrow u) \land (z \leftarrow \text{false}) \land (u \leftarrow \text{false})
\]

\[
F' = (x \leftarrow z) \land (y \leftarrow \text{false}) \land (z \leftarrow u) \land (u \leftarrow \text{false}).
\]

In this case we get

\[
F'' = (x \leftarrow (y \land z)) \land (y \leftarrow \text{false}) \land (z \leftarrow \text{false}) \land (u \leftarrow \text{false})
\]
but

\[ F''' = (x \leftarrow ((y \land z) \lor u)) \land (y \leftarrow \text{false}) \land (z \leftarrow \text{false}) \land (u \leftarrow \text{false}) \]

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 \( F''' = F \lor F' \).) 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, \( F \land \bigwedge S \models x \iff \bigwedge S \models M_x \lor x \).

The intuition is that in every component \( x \leftarrow T \) of \( F \), the right-hand side \( T \) must be a consequence of every conjunction of literals that entail \( x \) (in \( F \)). Every definite formula has an equivalent orthogonal formula. In Section 4.2 we show that an orthogonal form always exists, a fact we use for the following lemma and theorem (the proof of existence does not depend on the following results).

**Lemma 3.9.** Let \( F = \bigwedge_{x \in \text{Var}} \{x \leftarrow M_x\} \) and \( F' = \bigwedge_{x \in \text{Var}} \{x \leftarrow M'_x\} \) be orthogonal RMBF and let \( \phi \) be a model for \( F'' = \bigwedge_{x \in \text{Var}} \{x \leftarrow (M_x \land M'_x)\} \).

Then there are models \( \phi_F \) for \( F \) and \( \phi_{F'} \) for \( F' \) such that \( \phi = \phi_F \cap \phi_{F'} \).

**Proof.** 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 \( x \notin \phi \). 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} \Rightarrow (F \land \bigwedge \phi) \models x \quad \text{(by construction of \( \hat{\phi} \))} \]
\[ \Rightarrow (F \land \bigwedge \phi) \models x \quad \text{(by properties of implication)} \]
\[ \Rightarrow \bigwedge \phi \models M_x \lor x \quad \text{(by definition of orthogonality)} \]
\[ \Rightarrow \phi \models M_x \lor x \quad \text{(by properties of disjunction)} \]
\[ \Rightarrow \phi \models M_x \quad \text{or} \quad \phi \models x \quad \text{(by definition of \( \phi_F \)).} \]

Thus, \( \phi_F = \hat{\phi} \) and so \( \phi_F \) is a model of \( F \). The proof that \( \phi_{F'} \) satisfies \( F' \) is symmetric. \( \Box \)

**Theorem 3.10.** 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
Define $\text{RMBF}$ then

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

**Proof.** Call the right-hand side of the equation $F''$. Clearly $F \models F''$ and $F' \models F''$, so $F \lor F' \models F''$. We must now show that $F'' \models F \lor F'$. Let $\phi$ be a model of $F''$. By Lemma 3.9 there are models $\phi_F$ for $F$ and $\phi_{F'}$ for $F'$ such that $\phi = \phi_F \cap \phi_{F'}$. As $F \lor F'$ is an upper bound for $F$ and $F'$, $\phi_F$ and $\phi_{F'}$ are models of $F \lor F'$. Since $F \lor F'$ is definite, $\phi = \phi_F \cap \phi_{F'}$ is also a model of $F \lor F'$. \qed

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 $x \leftrightarrow y$. To see that $\text{Def}$ is not distributive, note that $(x \leftrightarrow y) \land (x \lor y) = x \leftrightarrow y$ but $((x \leftrightarrow y) \land x) \lor ((x \leftrightarrow y) \land y) = x \land y$. 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) & : = 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 $x \leftrightarrow y$. However, $\text{Def}$ gives the more precise result $x \land y$ if we unfold $r(X,Y)$:

\begin{align*}
q(a, Y) & : = p(a, Y). \\
r(a, Y) & : = p(a, Y). \\
p(X, X).
\end{align*}

$\text{Pos}$ does not have this kind of problem, since it is a Boolean lattice.

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:

\begin{align*}
\text{or}(X, Y, Z) : & - \\
\text{and}(X, Y, U), \\
xor(X, Y, V), \\
xor(U, V, Z).
\end{align*}

\begin{align*}
\text{and}(\text{true}, Y, Y). \\
\text{and}(X, \text{true}, X). \\
\text{and}(\text{false}, \text{false}, \text{false}).
\end{align*}

\begin{align*}
xor(X, X, \text{false}). \\
xor(\text{true}, \text{false}, \text{true}). \\
xor(\text{false}, \text{true}, \text{true}).
\end{align*}
The Boolean functions representing the groundness dependencies of and and xor are easily computed using the techniques from Section 2. With Pos we get

\[
\begin{align*}
\text{and}(x, y, z) &= (x \land (y \leftrightarrow z)) \lor (y \land (x \leftrightarrow z)) \\
\text{xor}(x, y, z) &= (x \leftrightarrow y) \land z \\
\text{or}(x, y, z) &= \exists u, v. [((x \land (y \leftrightarrow u)) \lor (y \land (x \leftrightarrow u))) \land (x \leftrightarrow y) \land v \land (u \leftrightarrow v) \land z] \\
&= x \land y \land z
\end{align*}
\]

With Def we get

\[
\begin{align*}
\text{and}(x, y, z) &= (x \land (y \leftrightarrow z)) \lor (y \land (x \leftrightarrow z)) \\
&= (x \land y) \leftrightarrow z \\
\text{xor}(x, y, z) &= (x \leftrightarrow y) \land z \\
\text{or}(x, y, z) &= \exists u, v. [((x \land y) \leftrightarrow u) \land (x \leftrightarrow y) \land v \land (u \leftrightarrow v) \land z] \\
&= x \land y \land z
\end{align*}
\]

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 \( \text{Pos}_\bot = \text{Pos} \cup \{\text{false}\} \) 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 \( E \) if \( F \) describes every \( e \in E \). 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 \( \text{Def}_\bot = \text{Def} \cup \{\text{false}\} \), 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 Section 2.2, as that analysis was expressed in terms of a greatest fixpoint.

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. May also be used in composing formulas.
- Compute the meet of the Boolean functions corresponding to the different constraints and atoms in a clause body. May also be used in composing formulas.

\(^3\) In fact, we did so in Section 2.
• **Restrict** a Boolean function, that is, existentially quantify over a local variable. This enables the removal of temporary variables.

• **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 $\mathcal{P} \neq \mathcal{NP}$. This means that there is an exponential worst case for a given representation to create a formula and check for equivalence. Unfortunately, this result continues to hold for both **Pos** and **Def**, for standard representations. In the following we let $|F|$ denote the size of formula $F$, in symbols.

**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-$\mathcal{NP}$.

We prove NP-hardness by reduction from the equivalence problem for monotonic formulas, which is known to be co-NP complete [6]. 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 $D$ and $D'$ are logically equivalent iff $\phi$ and $M'$ are. $\Box$

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-$\mathcal{NP}$.

We prove NP-hardness by reduction from the satisfiability problem for Boolean formulas in CNF [18]. 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.

NP-hardness in the DNF case is shown by reduction from the CNF case. Consider CNF formulas $F$ and $F'$ which represent positive functions. Let $G$ and $G'$ be obtained
from $F$ and $F'$ as follows: each $\vee$ is replaced by $\land$, $\land$ is replaced by $\lor$, and each literal $L$ is replaced by $\neg L$. We have that $(G \iff \neg F)$ and $(G' \iff \neg F')$ are valid. Now consider the formulas $H = G \lor x$ 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 [24].

**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 $F \rightarrow (x \land \bigwedge_{j=1}^{m} y_j)$ is valid, as this holds iff $(F \land \bigwedge_{j=1}^{m} y_j) \rightarrow x$. Notice that the left-hand side can be reduced in linear time, by replacing each $y_j$ in $F$ by true. Consequently, it is possible to determine equivalence of definite sentences in quadratic time. □

It follows from Theorems 4.1 and 4.2 that, assuming $P \neq \mathcal{NP}$, 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 can assume:

- Tests for equivalence will be less common than the other operations and involve fewer variables. This is because tests for equivalence do not occur in the composition of formulas, only once for each iteration. For other operations, for each iteration there will be $n$ lubs, where $n$ is the number of clauses, and a renaming and glb for each recursive clause, and the restriction of any temporary variables for each clause. This is in addition to any operations performed in the composition of the formula. The exact composition of the frequency of operations will depend on the analyzer used. In addition, the test for equivalence is performed on formulas where all temporary variables have been restricted away. Therefore the number of variables in the formula being tested will only be the arity of the predicate.
- The functions will be over a relatively small number of variables as a clause in a logic program typically contains a small bounded number of variables. (Machine-generated programs may of course violate this assumption.)
- The base functions represent dependency information of the form $x \leftrightarrow (\bigwedge_{i=1}^{m} y_i)$ or of the form $x \leftarrow (\bigwedge_{i=1}^{m} y_i)$. 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, 19, 21].

4.1.1. ROBDD: Reduced ordered binary decision diagrams

A ROBDD is a well-known symbolic representation for Boolean functions [10]. 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 subtrees. 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. Fig. 3 shows the ROBDD for \((x \land y) \rightarrow z\) 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 [8]. 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

\[ x \]
\[ y \]
\[ z \]
\[ \text{false} \]
\[ \text{true} \]

Fig. 3. ROBDD for \((x \land y) \rightarrow z\).

\(^4\)The shorter but less precise acronyms OBDD or BDD are sometimes used; here we follow Brace et al. [8].
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 \( x \leftarrow (\bigwedge_{i=1}^{m} y_i) \) 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, \( (x \land y) \leftarrow z \) could be represented as

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

or as

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

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** [9]. 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 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 \((x \land y \land z) \lor (\neg x \lor \neg z) \lor (\neg y \lor \neg z)\) 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 subterm 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, $(x \land y) \rightarrow z$ has BCF

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

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 syllogizing), and then removing redundant disjuncts (called absorption). In practice, for efficiency, these two stages are intertwined. As we would expect from Theorem 4.2, syllogizing has exponential cost in the worst case. This means that join, meet and restriction may have exponential cost.

**Example 4.2.** Consider the formula

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

Syllogizing adds the disjuncts $false$, $\neg x \land \neg z$, and $\neg y \land \neg z$. Absorption then yields

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

**Example 4.3.** To see that syllogizing 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

$\{G_1 \land \cdots \land G_n \mid G_i \text{ is } x_i \text{ or } \neg x_i\}.$

Consider the formula $F = \bigvee\{F_i \land u_i \mid 1 \leq i \leq n\}$. 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 $N$.

The functions we usually encounter have BCF of reasonable size. In particular, a function of the form $x \rightarrow \bigwedge_{j=1}^m y_j$ has a BCF which is linear in $m$, namely

$$\left(x \land \bigwedge_{j=1}^m y_j\right) \lor \bigvee_{j=1}^m (\neg x \land \neg y_j).$$

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 \(\exists x. F\).

**Proof.** By Schröder’s Elimination Principle, \(\exists x. F = F[x \rightarrow \text{true}] \lor F[x \rightarrow \text{false}]\). Thus,

\[
\exists x. F = (F_0 \land F_x \land F_{\neg x})[x \rightarrow \text{false}] \lor (F_0 \land F_x \land F_{\neg x})[x \rightarrow \text{true}]
= (F_0 \land F_x[x \rightarrow \text{false}]) \lor (F_0 \land F_{\neg x}[x \rightarrow \text{true}])
= F_0 \land \bigwedge \{c \lor c' \mid c \in F_x[x \rightarrow \text{false}], c' \in F_{\neg x}[x \rightarrow \text{true}]\}
= F_0 \land \bigwedge \{\text{resolve}(x, c, c') \mid c \in F_x, c' \in F_{\neg x}\}. \quad \square
\]

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

\[
\left( x \lor \bigvee_{j=1}^m \neg y_j \right) \land \bigwedge_{j=1}^m (y_j \lor \neg x).
\]
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 subclause 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, \((x \land y) \rightarrow z\) 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 \( \exists x. F \) is

\[\bigwedge \{c_{i} \mid x \text{ and } \neg x \text{ do not occur in } c_{i}\}\].

Proof. Let \( G = \bigwedge \{c_{i} \mid x \text{ and } \neg x \text{ do not occur in } c_{i}\} \). We first show that \( G \) is \( \exists x. F \).

By Proposition 4.5, \( \exists x. F \) is

\[F_{0} \land \bigwedge \{\text{resolve}(x, c, c') \mid c \in F_{x}, c' \in F_{\neg x}\},\]

where \( G \) is \( F_{0} \). As \( F \) is in DBCF, each clause in \( \{\text{resolve}(x, c, c') \mid c \in F_{x}, c' \in F_{\neg x}\} \) 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 \( G' \) which is 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 \( F \) must imply \( c \). 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. \[\square\]

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 \( \text{Def} \)

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, say $F'$. 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 $\text{DBCF}_{\text{Def}}$ 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

$$\bigwedge_{x \in \text{Var}} \left( x \leftarrow \bigvee \{ B \mid x \leftarrow B \text{ is in } F \text{ and } x \text{ is not in } B \} \right).$$

We also say that $F$ is the definite sentence corresponding to this RMBF formula.

For example, recall the formula $(x \land y) \leftarrow z$ has DBCF

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

Thus, the $\text{DBCF}_{\text{Def}}$ 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 $\text{DBCF}_{\text{Def}}$ 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$_{\text{Def}}$ 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$_{\text{Def}}$ is an orthogonal form.

**Theorem 4.8.** The $\text{DBCF}_{\text{Def}}$ representation of a definite function $F$ is in orthogonal form.

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

This means that we can use Theorem 3.10 to compute the join of two DBCF_{Def} 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 \( \text{BCF}(M) \) 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 \( \text{BCF}(M \land M') \) has \( O(N^2N'^2) \) cost.

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

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

**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 3.10, \( D'' \) 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 \( D \) such that

- \( C_1 = x \leftarrow T_1 \) and \( C_2 = y \leftarrow T_2 \),
- \( T''_1 = T_1 \land T'_1 \) and \( T''_2 = T_2 \land T'_2 \).

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 \( T'' \models (T_1 \land T_2) \setminus \{y\} \). By construction, \( C \) implies the resolvent, \( x \leftarrow (T_1'' \land T_2'') \setminus \{y\} \), of \( C''_1 \) and \( C''_2 \).

2. Assume that \( y \) appears in both \( T_1 \) and \( T'_1 \). 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 \( C''_1 \) 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. \( \square \)
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 3.8, offering more insight into the operation on \( \text{Def} \). The statement was that for definite \( F \), \( \exists x . F \) is definite. By Theorem 4.7, 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 very useful criterion for 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 \( \text{Def} \)

We now consider a second representation for definite functions. The RMBF formula \( \bigwedge_{x \in \text{Var}} (x \leftarrow M_x) \) is in 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 RCNF\(_{\text{Def}}\) is in RCNF.

Not every RCNF formula which denotes a definite function corresponds to a 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 Section 4.2.1). However, every definite function has at least one RCNF\(_{\text{Def}}\) formula representing it, namely DBCF\(_{\text{Def}}\).

For RCNF\(_{\text{Def}}\) formulas, the operations renaming, meet, and restriction have the same theoretical worst case cost as for RCNF. In practice, the computation of meet is expected to be more efficient. Noting that if \( F \) is the RCNF\(_{\text{Def}}\) formula \( \bigwedge_{x \in \text{Var}} (x \leftarrow M_x) \) and \( F' \) is the RCNF\(_{\text{Def}}\) formula \( \bigwedge_{x \in \text{Var}} (x \leftarrow M'_x) \), then a RCNF\(_{\text{Def}}\) 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 RDNF\(_{\text{Def}}\) formula \( \bigwedge_{x \in \text{Var}} (x \leftarrow M_x) \), then a RCNF\(_{\text{Def}}\) function representing \( \exists x . F \) is

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

where \( F[x \leftarrow F'] \) denotes the replacement of all occurrences of the variable \( x \) in \( F \) by the formula \( F' \).

Example 4.4. The function \((x \lor y \lor z) \leftarrow (x \land y \land z)\) 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 \( y \leftarrow z \).
RCNF\textsubscript{Def} does not support efficient computation of the join on \textit{Def} as in general RCNF\textsubscript{Def} formulas are not in orthogonal form. One option is to convert to DBCF\textsubscript{Def} and then compute the join. A second option is to use

$$\bigwedge_{s \in \text{Var}} (x \leftarrow \text{RDNF}(M_s \land M'_s))$$

as an approximation to the join.

Although the theoretical worst case complexity of most operations on RCNF\textsubscript{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\textsubscript{Def} formulas in quadratic time, while testing equivalence of RCNF formulas may require exponential time.

4.3. Summary of complexities

Table 1 summarizes the complexity of each operation using the various representations. The results are given with respect to the size of the formulas, \(M\) and \(N\), rather than the number of variables, \(n\). For all the representations, the size of a formula representing \(x_1 \leftarrow (x_2 \land x_3 \land \cdots \land x_n)\) is \(O(n)\).

Note that the \(O(n)\) notation does not give an accurate idea of which operations will necessarily be the fastest in practice, as we are typically dealing with relatively small \(n\).

5. Empirical evaluation

This section contains results from an empirical investigation of the different representations and their relative cost and precision in the context of groundness analysis. The worst case complexity results from the previous section actually give little indication of the true relative efficiency. For all operations, the 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.

We first sketch our analysis framework, and then discuss the implementation of the various representations. Finally we show the results of our tests.

Table 1
Complexity of operations in terms of formulas of size \(M\) and \(N\)

<table>
<thead>
<tr>
<th>Representation</th>
<th>Join</th>
<th>Meet</th>
<th>Equiv</th>
<th>Rename</th>
<th>Restrict</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROBDD</td>
<td>(O(NM))</td>
<td>(O(NM))</td>
<td>(O(1))</td>
<td>(O(N^2))</td>
<td>(O(N^2))</td>
</tr>
<tr>
<td>RDNF</td>
<td>(O(NM))</td>
<td>(O(N^2M^2))</td>
<td>(O(2^{M+N}))</td>
<td>(O(N))</td>
<td>(O(N^2))</td>
</tr>
<tr>
<td>BCF</td>
<td>(O(2^{N+M}))</td>
<td>(O(2NM))</td>
<td>(O(M+N))</td>
<td>(O(N))</td>
<td>(O(2^N))</td>
</tr>
<tr>
<td>RCNF</td>
<td>(O(N^2M^2))</td>
<td>(O(NM))</td>
<td>(O(2^{M+N}))</td>
<td>(O(N))</td>
<td>(O(N^4))</td>
</tr>
<tr>
<td>DBCF</td>
<td>(O(2NM))</td>
<td>(O(2^{N+M}))</td>
<td>(O(M+N))</td>
<td>(O(N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>DBCF\textsubscript{Def}</td>
<td>(O(N^2M^2))</td>
<td>(O(2^{N+M}))</td>
<td>(O(M+N))</td>
<td>(O(N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>RCNF\textsubscript{Def}</td>
<td>(O(N^2M^2))</td>
<td>(O(NM))</td>
<td>(O(M+N)^2)</td>
<td>(O(N))</td>
<td>(O(N^4))</td>
</tr>
</tbody>
</table>
5.1. Analysis framework

The implementation is a hybrid: The high-level engine is written in Prolog, and the low-level operations are written in ANSI C for speed. The analysis is divided into four phases: first the input file is read and the Clark completion is collected. Second, the strongly-connected components (SCCs) in the program's call graph are collected in topological order. Third, the program is analyzed bottom up, one SCC at a time. Finally, the program is analyzed top down for call patterns, again one SCC at a time.

The bottom up analysis of each SCC is done in two stages. First, we prepare each predicate by constructing a distilled form of the code with as much of the analysis precomputed as possible. Since we are analyzing bottom up, all of the atoms in a clause, except for recursive atoms, have a fixed analysis which we need compute only once. We also find the least upper bound of the analyses of all the clauses with no recursive atoms, as these will not change during analysis. This least upper bound also becomes our first approximation of the success pattern for the predicate (we bypass the zeroth approximation of \textit{false}).

The second stage of the bottom up analysis of an SCC is the fixpoint iteration. For non-recursive predicates, for which the distillation is a single constant Boolean function, 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.

The top down phase is much the same, but in reverse. Initially, each predicate's call pattern is \textit{false}. Then we analyze the initial goal for its call pattern, followed by each SCC, proceeding top down through the program call graph.

The first stage of the top down analysis of an SCC is the fixpoint iteration. We again use the distillation we prepared for the bottom up analysis, but this time we use some extra information that was not needed earlier. While computing the distillation of a clause we annotate each call with the greatest lower bound of the success patterns of all the non-recursive atoms preceding it. For recursive atoms, we include this annotation in the distillation. Thus, for the top down analysis, we can now quickly traverse the distillation of each predicate computing the greatest lower bound of its current approximate call pattern with the success patterns of all the clauses preceding each recursive atom to find a new call pattern for that atom. We compute the join of this new call pattern with the previous approximation to find the next one.

In the second stage of the top down analysis, we propagate the call pattern information down the call graph. Thus, we traverse the code for each predicate in that SCC once, computing the meet of the call pattern for the predicate and the earlier computed greatest lower bound of the exit patterns of all the atoms preceding each atom in the clause, and combining this with the previous approximate call pattern for the predicate called by that atom.

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 contain radically differing data structures and algorithms. For all representations, a variable is represented as a positive integer. The arguments in a clause head are always numbered from one through to the predicate arity. Other variables in the clause are assigned numbers continuing this sequence. 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 is a parameter that can easily be changed.

The main operations, introduced in Section 4, used to perform the analysis are: equivalence, join, meet, restriction and renaming. The equivalence, join and meet operations are as described previously. For efficient implementation, however, the restriction and renaming operations have been modified to work on more than one variable at a time.

Restriction is performed via the operation \( \text{restrictThresh} \). This operation restricts all variables above a threshold value, rather than restricting a single variable. The renaming operation, \( \text{renameArray} \), simultaneously renames the variables 1 to \( n \), to the variables given in an array of \( n \) variables. Simultaneous renaming of variables is essential as iteratively renaming single variables will produce incorrect results when the renamings are not independent of each other. Another operation involving renaming is also required for the top down analysis. \( \text{ReverseRenameArray} \) performs a simultaneous renaming of certain variables in a formula, given in an array of \( n \) variables, to the variables 1 to \( n \). The formula is then restricted to only those variables that were renamed.

In addition, the following operations are used to create and manipulate the representations:

- \( \text{variableRep} \) – Given a variable number, create that variable in the appropriate representation.
- \( \text{implies} \) – Given the representation of two formulas \( F \) and \( G \), return the representation corresponding to \( F \rightarrow G \).
- \( \text{iff.conj} \) – Create a representation of \( x \leftrightarrow (\bigwedge_{i=1}^{m} y_i) \), given the variable numbers \( x \) and \( y_i \).
- \( \text{copy} \) – Create a copy of the representation.

All operations use destructive update when this is more efficient.

ROBDDs are implemented using the basic implementation sketched by Brace et al. [8]. The \( \text{ite constant} \) 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 name 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 $x_{32x_{n+k}}$ 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 data structure used for DBCF$_{Def}$ 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 implementation does 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 DBCF$_{Def}$ followed by comparison of bodies for corresponding heads. Presumably, the reason is that the formulas being compared have relatively few variables.

ROBDDs have been easy to implement as algorithms were generally available, and the code is compact since most operations use the same function (‘ite’). The largest implementation efforts have been for the disjunctive and conjunctive normal forms used for \textit{Pos}.

5.3. Test results

We have evaluated our implementations with a test suite of programs commonly used for this purpose. (Certain small programs such as append have been omitted as
their analysis is too fast to yield meaningful timing results.) In addition to standard
test programs we have used two larger programs: bryant, a Prolog implementation of
ROBDDs, and analyze, the groundness analyzer itself. Table 2 shows various statistics
about these programs. 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 3 shows the results. All times are given as the minimum of 10 runs of the
analyzer. Testing was performed on a SPARCserver 1000 with four 50 MHz TI Super-
SPARC processors (though only one was used in these tests) and 256 Mbyte of main
memory. The Prolog code was compiled with Quintus Prolog version 3.2; the C code
was compiled with GNU CC version 2.5.8, 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 remaining columns show the time
to perform both bottom up and top down analysis using the various representations of
Boolean functions.

Table 4 shows these results as a fraction of the times using ROBDDs, which have
been chosen for several implementations of Pos for groundness analysis.

Table 5 illustrates the precision of the bottom up analysis using definite functions,
relative to that using positive functions. The first two columns following the test name
present for each test case the total number of variables for which we want to determine
call patterns (that is, the total arity of the predicates in the test) and total of the heights

---

Table 2
Various statistics on the test suite

<table>
<thead>
<tr>
<th>File</th>
<th>SCCs</th>
<th>Preds</th>
<th>Clauses</th>
<th>Vars Avg</th>
<th>Vars Max</th>
<th>Args Avg</th>
<th>Args Max</th>
</tr>
</thead>
<tbody>
<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>diej</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>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>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>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>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>bp0-6</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>rdtock</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>bryant</td>
<td>25</td>
<td>32</td>
<td>57</td>
<td>8.70</td>
<td>24</td>
<td>4.81</td>
<td>9</td>
</tr>
<tr>
<td>analyze</td>
<td>299</td>
<td>328</td>
<td>881</td>
<td>2.07</td>
<td>44</td>
<td>3.72</td>
<td>19</td>
</tr>
</tbody>
</table>
Table 3
Benchmark timings (ms)

<table>
<thead>
<tr>
<th>File</th>
<th>Read</th>
<th>SCCs</th>
<th>ROBDD</th>
<th>RDNF</th>
<th>BCF</th>
<th>RCNF</th>
<th>DBCF</th>
<th>DBCF Def</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs</td>
<td>160</td>
<td>10</td>
<td>130</td>
<td>110</td>
<td>90</td>
<td>150</td>
<td>130</td>
<td>100</td>
</tr>
<tr>
<td>disj</td>
<td>140</td>
<td>10</td>
<td>80</td>
<td>60</td>
<td>100</td>
<td>110</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>gabriel</td>
<td>90</td>
<td>0</td>
<td>70</td>
<td>70</td>
<td>60</td>
<td>70</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>kalah</td>
<td>770</td>
<td>0</td>
<td>180</td>
<td>120</td>
<td>90</td>
<td>170</td>
<td>160</td>
<td>130</td>
</tr>
<tr>
<td>peep</td>
<td>520</td>
<td>10</td>
<td>220</td>
<td>230</td>
<td>580</td>
<td>590</td>
<td>260</td>
<td>370</td>
</tr>
<tr>
<td>press1</td>
<td>390</td>
<td>20</td>
<td>220</td>
<td>260</td>
<td>220</td>
<td>220</td>
<td>210</td>
<td>230</td>
</tr>
<tr>
<td>press2</td>
<td>380</td>
<td>20</td>
<td>210</td>
<td>240</td>
<td>180</td>
<td>260</td>
<td>230</td>
<td>250</td>
</tr>
<tr>
<td>read</td>
<td>380</td>
<td>20</td>
<td>340</td>
<td>360</td>
<td>310</td>
<td>330</td>
<td>340</td>
<td>290</td>
</tr>
<tr>
<td>ann</td>
<td>400</td>
<td>20</td>
<td>220</td>
<td>440</td>
<td>360</td>
<td>230</td>
<td>250</td>
<td>210</td>
</tr>
<tr>
<td>boyer</td>
<td>360</td>
<td>10</td>
<td>130</td>
<td>390</td>
<td>410</td>
<td>490</td>
<td>440</td>
<td>240</td>
</tr>
<tr>
<td>bp0-6</td>
<td>150</td>
<td>10</td>
<td>110</td>
<td>80</td>
<td>110</td>
<td>140</td>
<td>210</td>
<td>80</td>
</tr>
<tr>
<td>rdtok</td>
<td>190</td>
<td>10</td>
<td>120</td>
<td>110</td>
<td>80</td>
<td>110</td>
<td>110</td>
<td>80</td>
</tr>
<tr>
<td>semi20</td>
<td>180</td>
<td>10</td>
<td>90</td>
<td>90</td>
<td>100</td>
<td>130</td>
<td>130</td>
<td>80</td>
</tr>
<tr>
<td>bryant</td>
<td>290</td>
<td>10</td>
<td>400</td>
<td>4000</td>
<td>1530</td>
<td>270</td>
<td>430</td>
<td>150</td>
</tr>
<tr>
<td>analyze</td>
<td>4280</td>
<td>130</td>
<td>6100</td>
<td>2567490</td>
<td>226630</td>
<td>10040</td>
<td>10760</td>
<td>1320</td>
</tr>
</tbody>
</table>

Average 644 74 574 171 603 15 389 886 922 248

Table 4
Benchmark timings, as fraction of ROBDD time

<table>
<thead>
<tr>
<th>File</th>
<th>ROBDD</th>
<th>RDNF</th>
<th>BCF</th>
<th>RCNF</th>
<th>DBCF</th>
<th>DBCF Def</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs</td>
<td>1.00</td>
<td>0.85</td>
<td>0.69</td>
<td>1.15</td>
<td>1.00</td>
<td>0.77</td>
</tr>
<tr>
<td>disj</td>
<td>1.00</td>
<td>0.75</td>
<td>1.25</td>
<td>1.38</td>
<td>1.25</td>
<td>1.25</td>
</tr>
<tr>
<td>gabriel</td>
<td>1.00</td>
<td>1.00</td>
<td>0.71</td>
<td>0.86</td>
<td>1.00</td>
<td>0.86</td>
</tr>
<tr>
<td>kalah</td>
<td>1.00</td>
<td>0.67</td>
<td>0.50</td>
<td>0.94</td>
<td>0.89</td>
<td>0.72</td>
</tr>
<tr>
<td>peep</td>
<td>1.00</td>
<td>1.05</td>
<td>2.64</td>
<td>2.68</td>
<td>1.18</td>
<td>1.68</td>
</tr>
<tr>
<td>press1</td>
<td>1.00</td>
<td>1.18</td>
<td>1.00</td>
<td>1.00</td>
<td>0.95</td>
<td>1.05</td>
</tr>
<tr>
<td>press2</td>
<td>1.00</td>
<td>1.14</td>
<td>0.86</td>
<td>1.24</td>
<td>1.10</td>
<td>1.19</td>
</tr>
<tr>
<td>read</td>
<td>1.00</td>
<td>1.06</td>
<td>0.91</td>
<td>0.97</td>
<td>1.00</td>
<td>0.85</td>
</tr>
<tr>
<td>ann</td>
<td>1.00</td>
<td>2.00</td>
<td>1.64</td>
<td>1.05</td>
<td>1.14</td>
<td>0.95</td>
</tr>
<tr>
<td>boyer</td>
<td>1.00</td>
<td>3.00</td>
<td>3.15</td>
<td>3.77</td>
<td>3.38</td>
<td>1.85</td>
</tr>
<tr>
<td>bp0-6</td>
<td>1.00</td>
<td>0.73</td>
<td>1.00</td>
<td>1.27</td>
<td>1.91</td>
<td>0.73</td>
</tr>
<tr>
<td>rdtok</td>
<td>1.00</td>
<td>0.92</td>
<td>0.67</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td>semi20</td>
<td>1.00</td>
<td>1.00</td>
<td>1.11</td>
<td>1.44</td>
<td>1.44</td>
<td>0.89</td>
</tr>
<tr>
<td>bryant</td>
<td>1.00</td>
<td>10.00</td>
<td>3.83</td>
<td>0.68</td>
<td>1.07</td>
<td>0.38</td>
</tr>
<tr>
<td>analyze</td>
<td>1.00</td>
<td>420.90</td>
<td>37.15</td>
<td>1.65</td>
<td>1.76</td>
<td>0.22</td>
</tr>
</tbody>
</table>

Average 1.00 29.75 3.81 1.40 1.33 0.95

of the lattices for each predicate in that test, that is,
\[ \sum_{p \in \text{test}} 2^{\text{arity}(p)}. \]

The following two columns show the number and percentage of predicates in each file for which DBCF_{Def} produces a weaker answer than Pos. The next two columns show
Table 5
Precision of DBCF\_Def bottom up analysis, relative to Pos

<table>
<thead>
<tr>
<th>File</th>
<th>Total arity</th>
<th>Total height</th>
<th>Difference in:</th>
<th>ground vars</th>
<th>lattice height</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs</td>
<td>100</td>
<td>1399</td>
<td>4 (10%)</td>
<td>0 (0%)</td>
<td>22 (17%)</td>
</tr>
<tr>
<td>disj</td>
<td>68</td>
<td>175</td>
<td>4 (11%)</td>
<td>1 (3%)</td>
<td>5 (5%)</td>
</tr>
<tr>
<td>kalah</td>
<td>124</td>
<td>1121</td>
<td>15 (32%)</td>
<td>4 (9%)</td>
<td>365 (129%)</td>
</tr>
<tr>
<td>peep</td>
<td>67</td>
<td>317</td>
<td>7 (31%)</td>
<td>2 (20%)</td>
<td>62 (36%)</td>
</tr>
<tr>
<td>press1</td>
<td>143</td>
<td>479</td>
<td>3 (5%)</td>
<td>1 (3%)</td>
<td>5 (2%)</td>
</tr>
<tr>
<td>press2</td>
<td>143</td>
<td>479</td>
<td>3 (5%)</td>
<td>1 (3%)</td>
<td>5 (2%)</td>
</tr>
<tr>
<td>read</td>
<td>121</td>
<td>599</td>
<td>9 (20%)</td>
<td>0 (0%)</td>
<td>120 (58%)</td>
</tr>
<tr>
<td>boyer</td>
<td>68</td>
<td>190</td>
<td>4 (13%)</td>
<td>0 (0%)</td>
<td>4 (3%)</td>
</tr>
<tr>
<td>bp0-6</td>
<td>48</td>
<td>259</td>
<td>4 (25%)</td>
<td>0 (0%)</td>
<td>8 (7%)</td>
</tr>
<tr>
<td>rtok</td>
<td>52</td>
<td>169</td>
<td>1 (5%)</td>
<td>0 (0%)</td>
<td>1 (1%)</td>
</tr>
<tr>
<td>semi20</td>
<td>61</td>
<td>331</td>
<td>6 (28%)</td>
<td>2 (18%)</td>
<td>89 (64%)</td>
</tr>
<tr>
<td>bryant</td>
<td>154</td>
<td>2817</td>
<td>13 (40%)</td>
<td>0 (0%)</td>
<td>948 (76%)</td>
</tr>
<tr>
<td>analyze</td>
<td>2262</td>
<td>1665519</td>
<td>178 (26%)</td>
<td>9 (1%)</td>
<td>47937 (156%)</td>
</tr>
</tbody>
</table>

the number and percentage of the predicate arguments which Pos was able to determine would always be ground on call but DBCF\_Def could not. The final two columns show the sum of the lattice heights gained by DBCF\_Def over Pos and the percentage of the gain over Pos. Recall that lower lattice positions provide more information.

Table 6 shows the precision of the call pattern analysis using definite functions, relative to that using positive functions. Note the considerable loss of information about definitely ground variables. The set of variables that are definitely ground at some point is arguably the most useful information provided by groundness analysis to an optimizing compiler, so the loss of precision here is probably the most important. Interestingly, the loss is much greater than was the case for the bottom-up analysis. It should be mentioned that the definite functions' loss of precision in this case partly is a result of our method together with the fact that Def is not condensing [31]. For example, with the program

\[
q(X, Y) :- p(X, Y), r(X, Y), s(X).
\]

\[
p(X, X).
\]

\[
r(a, Y).
\]

\[
r(X, a).
\]

\[
s(X).
\]

our call pattern analysis using Def will not discover that s is called with a ground argument, as the Boolean function obtained before the call to s is \((x \leftrightarrow y) \land \text{true}\). However, even using Def, a call pattern analysis could do better by analyzing the clauses for \(r\) using the knowledge expressed by \(x \leftrightarrow y\). In both cases one argument will ground the other, so both must be ground after \(r(X, Y)\). Still, as witnessed by the bottom up results, we can conclude that using positive functions generally achieves significantly higher precision than using definite functions, at a small extra cost.
Table 6
Call pattern precision of DBCF_{Def}, relative to Pos

<table>
<thead>
<tr>
<th>File</th>
<th>Total arity</th>
<th>Total height</th>
<th>Difference in:</th>
<th>Difference in:</th>
<th>Difference in:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>predicates</td>
<td>ground vars</td>
<td>lattice height</td>
</tr>
<tr>
<td>cs</td>
<td>100</td>
<td>1399</td>
<td>6 (15%)</td>
<td>6 (10%)</td>
<td>14 (17%)</td>
</tr>
<tr>
<td>disj</td>
<td>68</td>
<td>175</td>
<td>6 (17%)</td>
<td>11 (28%)</td>
<td>29 (50%)</td>
</tr>
<tr>
<td>kalah</td>
<td>124</td>
<td>1121</td>
<td>36 (78%)</td>
<td>41 (51%)</td>
<td>109 (105%)</td>
</tr>
<tr>
<td>peep</td>
<td>67</td>
<td>317</td>
<td>10 (45%)</td>
<td>18 (51%)</td>
<td>176 (262%)</td>
</tr>
<tr>
<td>press1</td>
<td>143</td>
<td>479</td>
<td>6 (11%)</td>
<td>4 (26%)</td>
<td>28 (8%)</td>
</tr>
<tr>
<td>press2</td>
<td>143</td>
<td>479</td>
<td>6 (11%)</td>
<td>4 (26%)</td>
<td>28 (8%)</td>
</tr>
<tr>
<td>read</td>
<td>121</td>
<td>599</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>boyer</td>
<td>68</td>
<td>190</td>
<td>4 (13%)</td>
<td>9 (56%)</td>
<td>16 (11%)</td>
</tr>
<tr>
<td>bp0-6</td>
<td>48</td>
<td>259</td>
<td>8 (50%)</td>
<td>12 (41%)</td>
<td>31 (77%)</td>
</tr>
<tr>
<td>rdtok</td>
<td>52</td>
<td>169</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>semi20</td>
<td>61</td>
<td>331</td>
<td>3 (14%)</td>
<td>1 (9%)</td>
<td>92 (57%)</td>
</tr>
<tr>
<td>bryant</td>
<td>154</td>
<td>2817</td>
<td>12 (37%)</td>
<td>15 (39%)</td>
<td>914 (136%)</td>
</tr>
<tr>
<td>analyze</td>
<td>2262</td>
<td>1665519</td>
<td>162 (24%)</td>
<td>51 (16%)</td>
<td>40108 (14%)</td>
</tr>
</tbody>
</table>

Let us finally consider the pattern of usage of operations. Table 7 gives information about the frequency of the different operations as they get applied in our (full call-pattern) groundness analysis. The bottom row lists the number of operations relative to equivalence, to give an idea of how many times each operation is typically done within one iteration.

6. Related work

6.1. Positive propositional logic

Positive\(^5\) Boolean functions have previously been studied, but we are not aware of practical applications outside the area covered by this paper. The class plays a role in Post's criterion for functional completeness in classical propositional logic [35]. Hilbert and Bemays [28] discussed a "positive Logik" which was intended to be the part of propositional calculus that is independent of a concept of negation. It can be extended, for example, to full classical propositional calculus or to intuitionistic propositional calculus. Rasiowa and Sikorski [38] and Rasiowa [37] have studied positive logic and several related logics in more detail. They show a strong relation between positive logic and relatively pseudo-complemented lattices.

In Hilbert and Bemays's axiomatization of positive logic, validity corresponds exactly to intuitionistic validity. Their positive logic does not include certain classical tautologies, such as \((x → y) → x\) → x (an instance of Peirce's law), as theorems. If one prefers, our groundness analyzer can be considered intuitionistic, as it is faithful

\(^5\)Some authors, including Chang and Keisler [11] and Dart [23] refer to what is commonly called a "monotonic" function as "positive". We use the more common terminology (although in [30] the elements of Pos were erroneously called "monotonic").
<table>
<thead>
<tr>
<th>Table 7: Frequency of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>equiv</strong></td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>cs</td>
</tr>
<tr>
<td>disi</td>
</tr>
<tr>
<td>kalah</td>
</tr>
<tr>
<td>peep</td>
</tr>
<tr>
<td>press1</td>
</tr>
<tr>
<td>press2</td>
</tr>
<tr>
<td>read</td>
</tr>
<tr>
<td>boyer</td>
</tr>
<tr>
<td>boyer6</td>
</tr>
<tr>
<td>rdbck</td>
</tr>
<tr>
<td>semi20</td>
</tr>
<tr>
<td>bryant</td>
</tr>
<tr>
<td>analyze</td>
</tr>
<tr>
<td><strong>Total</strong></td>
</tr>
</tbody>
</table>
to the axiomatization of Hilbert and Bernays. In particular, it never builds a formula of the form $(F \rightarrow F') \rightarrow F''$, that is, places an arrow in contravariant position.

6.2. Groundness analysis

Early "mode" analyses by Mellish [34] and Søndergaard [42] included groundness analysis without considering dependencies. Dart [22, 23] introduced definite functions, under the name of dependency formulas, and used them for groundness analysis in deductive databases. The use of positive functions was suggested for groundness analysis by Marriott and Søndergaard [30] (under the less suggestive name "Prop") and further studied by Cortesi et al. [20].

Several independent implementations have indicated that groundness analysis based on positive functions is very accurate and is perfectly practical for "real-world" programs [19, 21]. Apart from the current study, little effort has been devoted to improving implementations by investigating different representations for the Boolean functions, including positive and definite functions. Implementors generally seem to favor ROBDDs [3, 21].

The groundness analysis given by Barbuti et al. [4] uses hypergraphs to capture groundness dependencies. It is instructive to compare this hypergraph approach with the approach that uses Boolean functions — the power and elegance of the latter is striking. Incidentally, the Barbuti et al. hypergraph approach involves a "merge" operation which implies a loss of precision, compared with an analysis using Pos.

Consider the program

\[
\text{q}(X, Y) :- \text{p}(X, Y), X = Y.
\]

\[
\text{p}(a, Y).
\]

\[
\text{p}(X, a).
\]

Owing to a "merge" the groundness dependency information obtained for \(p\) is (in our notation) true, and hence for \(q\) it is \(x \leftrightarrow y\).

An approach which is closely related to ours is due to Codognet and Filé [17]. This approach has subsequently been adopted and extended in various ways by Corsini et al. [19], and by Codish and Demoen [14]. It is "syntactic" in the sense that it translates a program to be analyzed into a constraint logic program which, when run, will yield groundness information about the original program. In the case of Corsini et al. the resulting program is written in Toupie, a language designed with dataflow analysis applications in mind. Codish and Demoen also use readily available theorem proving technology by translating into Prolog.

For example, Codish and Demoen translate the above program to

\[
\text{q}(X, Y) :- \text{p}(X, Y), \text{iff}(X, Y).
\]

\[
\text{p}(X, Y) :- X = \text{true}.
\]

\[
\text{p}(X, Y) :- Y = \text{true}.
\]

\[
\text{iff}(false, false).
\]

\[
\text{iff}(true, true).
\]
Note that querying the transformed program by \( q(X, Y) \) yields the answer \( X = \text{true}, Y = \text{true,} \) which corresponds to \( x \land y \). Corsini et al., as well as Codish and Demoen in fact compute results in a bottom up manner, utilizing the fact that the program's success set must be finite. For example, Codish and Demoen generate the success set

\[
\{q(\text{true, true}), p(\text{true, true}), \text{iff}(\text{false, false}), \text{iff}(\text{true, true})\}
\]

which can then be queried. They also make use of the Prolog database to generate call pattern information, using a variant of the magic-set transformation. This syntactic approach gives surprisingly good performance [19, 141. Nevertheless, on most example programs in our suite, these other approaches are typically 20 times slower. Taking the different implementation platforms into account, our approach still appears to be an order of magnitude faster, which is not surprising, given the genericity of the previous tools.

Filé and Ranzato [26] explore a more precise approach which unfortunately seems rather expensive, but so far no experimental results are available. In the Filé–Ranzato approach, sets of positive functions replace positive functions. At first sight, one may think that this should not improve precision of an analysis, since \( \text{Pos} \) is closed under disjunction. Consider, however, the program

\[
p(X, Y, Z) :- X = \text{g}(Y, a). \\
p(X, Y, Z) :- Z = \text{g}(Y, Y).
\]

With the approach described in this paper, the result for \( p \) is \( (x \leftrightarrow y) \lor (y \leftrightarrow z) \). However, Filé and Ranzato point out that the set \( \{ (x \leftrightarrow y), (y \leftrightarrow z) \} \) contains more information. A constraint such as \( Y = \text{g}(X, Z) \) is not correctly approximated by either element of the set, and hence not by the set itself. However, it is approximated by the function \( (x \leftrightarrow y) \lor (y \leftrightarrow z) \), since \( y \leftrightarrow (x \land z) \models (x \leftrightarrow y) \lor (y \leftrightarrow z) \). Hence the set is more precise.

6.3. Other applications

Definite functions have found several types of usage in the database area. Finiteness dependencies for Datalog were introduced by Ramakrishnan et al. [36], and Zobel [44] studied finite dependencies as a type of Boolean dependencies. Armstrong and Delobel [2] introduced functional dependencies for relational databases and Zobel [44] studied functional dependency analysis for deductive databases.

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, as well as for suspension analysis for concurrent constraint logic programs. Section 2 contains references to work on suspension analysis, as well as finiteness analysis. Giacobazzi [27] shows how positive functions can be used also in automatic manipulation of the semi-linear norms used by Bossi et al. [7] for proving
termination properties for logic programs. Consider the usual append program

\[
\text{app}(Xs, Ys, Zs) :- Xs = [], Ys = Zs.
\]

\[
\text{app}(Xs, Ys, Zs) :- Xs = [U|Us], Zs = [U|Vs], \text{app}(Us, Ys, Vs).
\]

Regarding "length dependencies" for terms, the first clause gives rise to the function \( xs \land (ys \leftrightarrow zs) \). This says that the length of Xs is fixed (the term is rigid in the terminology of Bossi et al.), while the length of Ys varies with that of Zs. In fact, this is a fixpoint (we shall not show the details, but subsequent processing of the second clause returns the same result). Compare this result with that obtained in a groundness dependency analysis, that is, \((xs \land ys) \leftrightarrow zs\).

Using the same technology as for their groundness analysis (see above), Codish and Demoen \cite{15} have based a kind of type inference on different "incarnations" of Pos. Loosely, each incarnation corresponds to a "type" such as list, tree, dlist (difference-lists), or any. For example, the append program is translated to

\[
\text{app}(Xs, Ys, Zs) :- \text{is\_nil}(Xs), Ys = Zs.
\]

\[
\text{app}(Xs, Ys, Zs) :- \text{list\_dep}(Xs, Us), \text{list\_dep}(Zs, Vs), \text{app}(Us, Ys, Vs).
\]

\[
\text{is\_nil}\text{(list)}.
\]

\[
\text{list\_dep}\text{(list, list)}.
\]

\[
\text{list\_dep}\text{(any, any)}.
\]

The success set gives the appropriate "type" information for app, containing both \text{app}(list, list, list) and \text{app}(list, any, any). From the point of view of propositional logic, one can think of list as standing for false and any as standing for true, so that \text{list\_dep} simply defines biimplication, as did iff in Section 6.2. Other types use other names for false, ensuring that the various logics can be combined to a sophisticated type system. This is the sense in which Codish and Demoen use "incarnations" of Pos. Codish and Demoen also generalize their approach to handle a (non-standard) kind of parametric polymorphism.

It remains to be seen whether the use of an implicative fragment of propositional logic could improve dataflow analyses developed for functional programming languages, such as analyses for strictness, usage, and binding time. While there are approaches to, say, strictness analysis that use variants of ROBDDs — see, for example, Mauborgne \cite{33} — it is worth noting that such analyses do not handle dependencies but are confined to manipulating monotonic Boolean functions. One would expect that better tailored representations could yield better performance.

7. Conclusions

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
analysis. We have studied the algebraic properties of these subclasses and also looked at different representations and implementations.

We have taken groundness analysis as a — hopefully representative — example case, in order to compare the precision obtained using positive and definite functions. Some care should be taken when generalizing to other applications using propositional logic for dependency analysis, but we believe that our results are fairly indicative.

We have found that the precision achieved using \textit{Pos} is significantly higher than that of \textit{Def}. This is particularly conspicuous in the case of call pattern analysis, but it should be noticed that the precision lost by \textit{Def} is mainly due to the fact that \textit{Def} is not condensing [31]. A top–down analyzer using \textit{Def} would not lose as much precision as our bottom–up analyzer.

When we look at the variables definitely entailed by the analyses — in our experiment, the set of definitely ground variables in each call — we find, however, that the difference in precision between \textit{Pos} and \textit{Def} is quite small. This suggests that the information “lost” by \textit{Def} does not contribute greatly to the information sought by a compiler.

Regarding efficiency, our experiments indicate that ROBDDs are the fastest representation for \textit{Pos}. While the disjunctive forms appear quite attractive for smaller tests, they perform badly for larger tests and can clearly be dismissed. The conjunctive forms, particularly RCNF, perform reasonably well but, as they are not especially easier to implement than ROBDDs, ultimately have little to recommend them.

However, our specialized representation for \textit{Def} is clearly the fastest of the representations we studied. For larger tests it is several times faster than ROBDDs, and the performance gap widens with larger tests, suggesting that \textit{DBCDef} would scale better to large programs. \textit{DBCDef} is clearly the representation of choice for performance-conscious implementors interested only in definite variables and willing to accept a relatively small loss in precision.

References