Boolean equation solving as graph traversal

Brian Herlihy **Peter Schachte**[*] **Harald Søndergaard**

Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
Email: {btherl,schachte,harald}@csse.unimelb.edu.au

Abstract

We present a new method for finding closed forms of recursive Boolean function definitions. Traditionally, these closed forms are found by iteratively approximating until a fixed point is reached. Conceptually, our new method replaces each k-ary function by 2^k Boolean variables defined by mutual recursion. The introduction of an exponential number of variables is mitigated by the simplicity of their definitions and by the use of a novel variant of ROBDDs to avoid repeated computation. Experimental evaluation suggests that this approach is significantly faster than Kleene iteration for examples that would require many Kleene iteration steps.

Keywords: Boolean functions, fixed points, decision diagrams

1 Introduction

The need to obtain closed forms of recursively defined functions arises in many fields of mathematics and computer science, including formal language theory, database theory, and formal semantics. The theoretical foundations for solving recursive definitions were laid a long time ago, in the fields of order and fixed point theory. Improved algorithms, however, are still being developed for special applications.

A typical case is *abstract interpretation*, which is concerned with automated inference of program properties, as needed, for example, by optimising compilers and other program analysis and transformation tools (Cousot & Cousot 1977). Problems in abstract interpretation boil down to finding extreme fixed points (usually least fixed points) in structures that can be fairly complex. The meaning of a program is assumed to be given as a fixed point characterisation: the least fixed point of some functional F defined over a domain of program states, based on the semantic domains for the programming language. Program analysis is then obtained by faithfully abstracting F to work on some *abstract domain* instead, that is, on a domain of approximate program states. Usually the process can be divided into two stages: firstly constructing a set of (mutually) recursively defined functions as a conservative approximation of the aspect of the behaviour of the components of the program one is interested in, and secondly finding a closed-form "solution" of the recursive equations.

The usual approach to finding fixed points for recursively defined equations is *Kleene iteration.* The essential idea is to transform the set of mutually recursively defined functions into a single non-recursive functional that takes a tuple of closed form tentative solutions to these equations and uses the original definitions of the functions, modified to use the tentative solutions in place of recursive calls, to compute a better approximation. Kleene iteration repeatedly applies this functional to the result of the previous iteration until a fixed point is reached. If the functional is monotone on a lattice with the ascending-chain property, and iteration begins with the least element, the process is guaranteed to terminate with the least fixed point, *i.e.*, the strongest closed form solution to the initial set of equations.

Consider, for example, the following equation:

$$\begin{aligned} f(x,y,z) = \;& (x \wedge (y \leftrightarrow z)) \vee \\ & \exists u,v.((x \leftrightarrow (u \wedge v)) \wedge f(v,u,z)) \end{aligned}$$

This function arises in the groundness analysis of a Prolog program to find the last element of a list using an accumulating parameter. To apply Kleene iteration to this, one defines a functional

$$\begin{aligned} F(f) = \lambda x,y,z.(\; & (x \wedge (y \leftrightarrow z)) \vee \\ & \exists u,v.((x \leftrightarrow (u \wedge v)) \wedge f(v,u,z))) \end{aligned}$$

Note that the call to f is no longer recursive; it now invokes the parameter to F. Kleene iteration would begin by applying F to the least possible function f, which is $\lambda x,y,z.\mathit{0}$. For brevity, henceforth we shall agree that the arguments of this function are x, y, and z, and omit the λ. Thus the result of $F(\mathit{0})$ is $(x \wedge (y \leftrightarrow z)) \vee \exists u,v.((x \leftrightarrow (u \wedge v)) \wedge \mathit{0})) = x \wedge (y \leftrightarrow z)$. Next we apply F to this and get $(x \wedge (y \leftrightarrow z)) \vee \exists u,v.((x \leftrightarrow (u \wedge v)) \wedge v \wedge (u \leftrightarrow z)) = (x \wedge (y \leftrightarrow z)) \vee (x \leftrightarrow z)$. Applying F to this, we get $(x \wedge y) \rightarrow z$. Finally, applying F to this, we get back $(x \wedge y) \rightarrow z$, indicating a fixed point has been reached.

The first contribution of this paper is to propose an alternative view of this problem. Consider again the original function f. We build a table in which each row shows a possible input to f and the value of f for that input. In each row of the table, we substitute the input for that row, and all possible combinations of values for the existentially quantified variables. Thus the table will contain no variables whatsoever; all that remains will be Boolean constants and recursive calls. Furthermore, since each recursive call has all arguments statically known, it refers to a fixed row of the table. The following table illustrates the same example f shown above using the new approach (we use subscripts for function arguments).

[*]Peter Schachte's work on this project has been supported in part by NICTA Victora Laboratories.

$$\begin{array}{ll}
f_{000} & = 0 \vee \exists u,v.(\neg(u \wedge v) \wedge f_{vu0}) \\
f_{001} & = 0 \vee \exists u,v.(\neg(u \wedge v) \wedge f_{vu1}) \\
f_{010} & = 0 \vee \exists u,v.(\neg(u \wedge v) \wedge f_{vu0}) \\
f_{011} & = 0 \vee \exists u,v.(\neg(u \wedge v) \wedge f_{vu1}) \\
f_{100} & = 1 \vee \exists u,v.(u \wedge v \wedge f_{vu0}) \\
f_{101} & = 0 \vee \exists u,v.(u \wedge v \wedge f_{vu1}) \\
f_{110} & = 0 \vee \exists u,v.(u \wedge v \wedge f_{vu0}) \\
f_{111} & = 1 \vee \exists u,v.(u \wedge v \wedge f_{vu1})
\end{array}$$

Now we view this as a set of eight mutually-recursively defined variables. Of these, f_{100} and f_{111} are unambiguously determined to be *1*. Also, f_{101} is defined to be f_{111}, which is *1*, and f_{000}, f_{001}, f_{010}, and f_{011} are all defined to be disjunctions including either f_{100} or f_{101}, both of which are now known to be *1*, so all are *1*. This leaves only f_{110}, which is defined to be f_{110}. This indicates both truth values will be fixed points for this row. Assigning it *0* yields the least fixed point $(x \wedge y) \to z$, as given by Kleene iteration. Plugging in *1* yields $f(x,y,z) = 1$ which can also be verified to be a fixed point.

In this example, only two fixed points exist. In general, there will be 2^n fixed points, where n is the number of strongly-connected components (SCCs) in the dependency graph among rows in the truth table.

It is possible, for the domain of Boolean functions of arity n, to give a recursive definition for which the ascending Kleene sequence has maximal possible length: $2^n + 1$. In fact this can be achieved in several different ways. The following definition exemplifies this for a function of arity 4. What is defined in this cumbersome manner is the constant function *1*, but it takes 16 Kleene iteration steps to determine this.

$$\begin{array}{l}
p(v_1,v_2,v_3,v_4) = \\
\quad \exists v_5.(v_4 \wedge p(v_1,v_2,v_3,v_5)) \\
\vee \quad (v_3 \wedge p(v_1,v_2,v_4,1) \\
\vee \quad (v_2 \wedge (v_3 \leftrightarrow v_4) \wedge p(v_1,v_3,1,1)) \\
\vee \quad (v_1 \wedge (v_2 \leftrightarrow v_3) \wedge (v_3 \leftrightarrow v_4) \wedge p(v_2,1,1,1)) \\
\vee \quad ((v_1 \leftrightarrow v_2) \wedge (v_2 \leftrightarrow v_3) \wedge (v_3 \leftrightarrow v_4))
\end{array}$$

Again, this is a definition that arises in the groundness analysis of a certain Prolog program, although the program would not appear in the typical Prolog programmer's collection. It comes from one of several families suggested by Codish (1999) and Genaim, Howe & Codish (2001) as particularly challenging for analysis, because of their heavily iterative nature. In fact, the challenge that these programs have posed to our existing analysis tools (which make use of straight-forward Kleene iteration) has been the primary motivation for the work reported here.

The remainder of this paper will proceed as follows. In Section 2 we recall some basic concepts from fixed point theory and decision diagrams. Section 3 introduces a data structure, a variant of ROBDDs, and an algorithm to efficiently solve recursive definitions of Boolean functions. Section 4 reports on the experimental evaluation of the algorithm, Section 5 discusses related work, and Section 6 concludes.

2 Preliminaries

2.1 Lattices and fixed points

A *partial ordering* is a binary relation that is reflexive, transitive, and antisymmetric. A set equipped with a partial ordering is a *poset*. Let $(X, \leq)$ be a poset. A (possibly empty) subset Y of X is a *chain* iff for all $y, y' \in Y, y \leq y' \vee y' \leq y$.

Let $(X, \leq)$ be a poset. An element $x \in X$ is an *upper bound* for $Y \subseteq X$ iff $y \leq x$ for all $y \in Y$. Dually we may define a *lower bound* for Y. An upper bound x for Y is the *least upper bound* for Y iff, for every upper bound x' for Y, $x \leq x'$, and when it exists, we denote it by $\bigsqcup Y$. Dually we may define the *greatest lower bound* $\bigsqcap Y$ for Y.

A poset X for which every subset possesses a least upper bound and a greatest lower bound is a *complete lattice*. We denote the least element $\bigsqcap X$ of X by $\bot_X$ (or usually just $\bot$). X is *ascending chain finite* iff every ascending chain in X is finite.

Let $(X, \leq)$ and $(Z, \preceq)$ be posets. $F : X \to Z$ is *monotone* iff $x \leq x' \to F(x) \preceq F(x')$ for all $x, x' \in X$. A *fixed point* for $F : X \to X$ is an element $x \in X$ such that $x = F(x)$. If X is a complete lattice, then the set of fixed points for a monotone $F : X \to X$ is itself a complete lattice. The least element of this lattice is the *least fixed point* for F, and we denote it by *lfp*(F).

2.2 ROBDDs

Let $\mathbb{B} = \{1, 0\}$. The set of Boolean functions is $\mathsf{Bool} = \bigcup_{n \in \mathbb{N}} \mathbb{B}^n \to \mathbb{B}$. Let the set $\mathcal{V}$ of propositional variables be equipped with a total ordering $\prec$.

Binary decision diagrams (BDDs) are defined inductively as follows:

- 0 is a BDD.
- 1 is a BDD.
- If $x \in \mathcal{V}$ and R_1 and R_2 are BDDs then $\mathsf{ite}(x, R_1, R_2)$ is a BDD.

The meaning of a BDD is given as follows:

$$\begin{array}{ll}
[\![0]\!] & = 0 \\
[\![1]\!] & = 1 \\
[\![\mathsf{ite}(x, R_1, R_2)]\!] & = (x \wedge [\![R_1]\!]) \vee (\neg x \wedge [\![R_2]\!])
\end{array}$$

Let $R = \mathsf{ite}(x, R_1, R_2)$. A BDD R' *appears in* R iff $R' = R$ or R' appears in R_1 or R_2. We define $\mathsf{vars}(R) = \{v \mid \mathsf{ite}(v, \_, \_) \text{ appears in } R\}$.

A BDD is an *OBDD* iff it is 0 or 1 or if it is $\mathsf{ite}(x, R_1, R_2)$ where R_1 and R_2 are OBDDs, and $\forall x' \in \mathsf{vars}(R_1) \cup \mathsf{vars}(R_2) : x \prec x'$.

An OBDD R is an *ROBDD* (Reduced Ordered Binary Decision Diagram (Bryant 1992)) iff for all BDDs R_1 and R_2 appearing in R, $R_1 = R_2$ when $[\![R_1]\!] = [\![R_2]\!]$. Practical implementations (Brace, Rudell & Bryant 1990) use a function $\mathsf{mknd}(x, R_1, R_2)$ to create all ROBDD nodes as follows:

1. If $R_1 = R_2$, return R_1 instead of a new node, as $[\![\mathsf{ite}(x, R_1, R_2)]\!] = [\![R_1]\!]$.
2. If an identical ROBDD was previously built, return that one instead of a new one; this is accomplished by keeping a hash table, called the *unique table*, of all previously created nodes.
3. Otherwise, return $\mathsf{ite}(x, R_1, R_2)$.

This ensures that ROBDDs are strongly canonical: a shallow equality test is sufficient to determine whether or not two ROBDDs represent the same Boolean function.

Figure 1 gives a visual presentation of a simple ROBDD. For all diagrams in this paper, the leftmost of the two outgoing edges from any non-terminal node is the 1-edge and the rightmost is the 0-edge.

There are 2^{2^n} different Boolean functions of arity n, and it has been shown (Liaw & Lin 1992) that an ROBDD for an n-input Boolean function requires

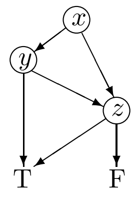


Figure 1: An ROBDD for the function $(x \wedge y) \vee z$.

at most $(2^n/n)$ nodes, so the worst-case complexity is exponential. However, the size of ROBDDs can vary enormously, not only with the ordering of variables, but also with the type of functions that are represented. For a great variety of Boolean functions, the ROBDD representation has polynomial complexity, which makes ROBDDs very attractive for a wide variety of applications.

Many important properties of Boolean functions can be easily tested from the ROBDD form of a function. Testing for unsatisfiable or tautological functions can be done in constant time: an unsatisfiable function is represented by the 0-terminal and a tautological function is represented by the 1-terminal. If the satisfiability of a function requires a particular variable to have a value v, then all nodes labelled with that variable will have their $(\neg v)$-edge point directly to the 0-terminal. Finally, the ROBDD representation of a function that is independent of a particular variable will have no nodes labelled by that variable.

3 Solving as graph manipulation

Before we embark on describing the new algorithm, we describe an algorithm that served as inspiration.

3.1 A depth-first approach

The algorithm we describe first is due to Le Charlier & Van Hentenryck (1992). It is rather more general than ours, indeed it is presented as a "universal top-down fixed point algorithm". It is well described (and appraised) by Fecht & Seidl (1999). Figure 2 is essentially Fecht and Seidl's presentation of algorithm **TD**, as they name it.[1]

The idea behind the algorithm is to maintain a "partial table" σ that maps variables to their current (approximate) values. In Figure 2, σ is a globally accessible table.

Suppose we have initiated solving for x and in the process find that we need the value of y. The approach is to eagerly turn to solving for y, that is, to find the next approximation to y's final value. However, there are two situations in which eagerness is abandoned and the current approximation $\sigma(y)$ to y's value is used instead. One is where we are already in an iteration initiated for y. A set *Called* is thus maintained to keep track of variables for which an iteration has been started but not completed. The other situation is where solving for y is deemed useless because no variable that influences y has changed its value. A set *Stable* is thus maintained to keep track of stable variables. A procedure call *destabilize*(x) makes sure that the implications of changing the value of x are tracked, so that *Stable* remains reliable.

[1] Fecht and Seidl's version is incorrect, probably as a result of a typographic error. The line marked with (*) in Figure 2 is missing in Fecht and Seidl's presentation. As a result, the closed form of, say, $f(x, y) = x \vee f(y, x)$ comes out incorrectly as $f(x, y) = x$.

```
**procedure main**
  $\sigma := \varnothing$; $Stable := \varnothing$; $Called := \varnothing$; $infl := \varnothing$;
  **foreach** $x \in V$ **do** $solve(x)$ **od**
**end**

**procedure** $solve(x : V)$
  **if** $x \notin Stable$ and $x \notin Called$ **then**
    **if** $x \notin dom(\sigma)$ **then**
      $\sigma(x) := \bot$; $infl(x) := \varnothing$
    **fi**;
    $Called := Called \cup \{x\}$;
    **do**
      $Stable := Stable \cup \{x\}$;
      $old := \sigma(x)$;
      $\sigma(x) := evalrhs(x, \lambda y.eval(x, y))$;
      **if** $\sigma(x) \neq old$ **then**
        $Stable := Stable \setminus \{x\}$;                    (*)
        $destabilize(x)$
      **fi**;
    **until** $x \in Stable$;
    $Called := Called \setminus \{x\}$
  **fi**
**end**

**function** $eval(x, y : V) : D$
  $solve(y)$;
  $infl(y) := infl(y) \cup \{x\}$;
  **return** $\sigma(y)$
**end**

**procedure** $destabilize(x : V)$
  $temp := infl(x)$; $infl(x) := \varnothing$;
  **foreach** $y \in temp$ **do**
    $Stable := Stable \setminus \{y\}$;
    $destabilize(y)$
  **od**
**end**
```

Figure 2: Algorithm **TD** (after Fecht and Seidl)

The role of $evalrhs(x, \cdot)$ is to evaluate the right-hand side of x's definition. This evaluation has access to (the global) σ. However, to evaluate eagerly, and to (dynamically) keep track of variable dependencies, $\lambda y.eval(x, y)$ is used instead of σ. The table *infl* is used for the bookkeeping—*infl*(y) is the set of variables that may depend on y. Essentially, $eval(x, y)$ provides for the solving for y in a context of solving for x, updating *infl* to track the dependency.

3.2 A data structure for equation systems

The algorithm just presented is general. We now turn to the special case of finding closed forms for (mutually) recursively defined Boolean functions and the second contribution of this paper: a new data structure and algorithm for this special case.

To be more formal, assume we are given a set $\mathbb{F}$ of Boolean function names. As a convenience, we also fix a set $\mathcal{V}$ of variables, with its total ordering $\prec$. We let the smallest n variables serve as the sequence of formal parameters for all Boolean functions, with all larger variables serving as local variables in definition bodies. Then we can define the set of Boolean function bodies and definitions as:

$$\begin{aligned}\mathsf{Bod} = \; & \mathbb{B} \cup \mathcal{V} \cup (\mathbb{F} \times \mathcal{V}^n) \cup \{x \wedge y \mid x, y \in \mathsf{Bod}\} \cup \\ & \{x \vee y \mid x, y \in \mathsf{Bod}\} \cup \{\neg x \mid x \in \mathcal{V}\} \\ \mathsf{Def} = \; & \mathbb{F} \to \mathsf{Bod}\end{aligned}$$

Here the logical connectives have their usual interpretation, and local variables are implicitly existentially quantified over the definition body.

We define the set of closed form Boolean function definitions to omit function applications:

$$\begin{aligned}\mathsf{Bod_c} = \; & \mathbb{B} \cup \mathcal{V} \cup \{x \wedge y \mid x, y \in \mathsf{Bod_c}\} \cup \\ & \{x \vee y \mid x, y \in \mathsf{Bod_c}\} \cup \{\neg x \mid x \in \mathcal{V}\}\end{aligned}$$

$$\mathsf{Def_c} = \mathbb{F} \to \mathsf{Bod_c}$$

Note that $\mathsf{Def_c} \subseteq \mathsf{Def}$. Furthermore, because $\{\wedge, \vee, \neg\}$ are functionally complete for Bool, and by de Morgan's laws, $\mathsf{Bod_c}$ is equivalent to $\mathbb{B}^n \to \mathbb{B}$, and thus is a subset of Bool. Let $\mathsf{Bod_c}$ be ordered by entailment.

Now we define a functional $\mathsf{F} : \mathsf{Def} \to \mathsf{Def_c} \to \mathsf{Def_c}$ in terms of $\mathsf{E} : \mathsf{Def_c} \to \mathsf{Bod_c} \to \mathsf{Bod_c}$ as follows:

$$\mathsf{F}\; D\; C = (\mathsf{E}\; C) \circ D$$

$$\begin{aligned}\mathsf{E}\; C\; t &= t \quad \textbf{where} \quad t \in \mathbb{B} \\ \mathsf{E}\; C\; v &= v \quad \textbf{where} \quad v \in \mathcal{V} \\ \mathsf{E}\; C\; (f\; v_1 \ldots v_n) &= C\; f\; v_1 \ldots v_n \\ \mathsf{E}\; C\; (c_1 \wedge c_2) &= (\mathsf{E}\; C\; c_1) \wedge (\mathsf{E}\; C\; c_2) \\ \mathsf{E}\; C\; (c_1 \vee c_2) &= (\mathsf{E}\; C\; c_1) \vee (\mathsf{E}\; C\; c_2) \\ \mathsf{E}\; C\; (\neg v) &= \neg v\end{aligned}$$

We say $C \in \mathsf{Def_c}$ is a *closed form* for equation system $D \in \mathsf{Def}$ iff C is a fixed point of $\mathsf{F}\; D$, that is, if $C = \mathsf{F}\; D\; C$. Note that due to the way negation is included in our definition of Bod, F is monotone, ensuring that $\mathsf{F}\; D$ has fixed points. In what follows we shall take the closed form of $D \in \mathsf{Def}$ to be $\mathit{lfp}(\mathsf{F}\; D)$, although any fixed point will do, and our algorithm can be modified to find all fixed points.

For convenience we shall restrict the syntax of Bod to a specialised disjunctive normal form that separates closed parts of function definitions as follows:

$$\begin{aligned}\mathsf{Bod} &= \{f \vee d \mid f \in \mathsf{Bod_c} \wedge d \in \mathsf{Dis}\} \\ \mathsf{Dis} &= \mathsf{Con} \cup \{d_1 \vee d_2 \mid d_1, d_2 \in \mathsf{Dis}\} \\ \mathsf{Con} &= \{f \wedge c \mid f \in \mathsf{Bod_c} \wedge c \in \mathsf{Ap}\} \\ \mathsf{Ap} &= (\mathbb{F} \times \mathcal{V}^n) \cup \{c_1 \wedge c_2 \mid c_1, c_2 \in \mathsf{Ap}\}\end{aligned}$$

By distributivity, commutativity, and associativity of conjunction and disjunction, this new definition has equivalent expressiveness to the previous one. However, it makes our algorithms simpler.

3.3 A tabular view of equation solving

Consider the Boolean function definitions:

$$\begin{aligned}f(x, y) &= (x \leftrightarrow y) \\ g(x, y) &= (x \leftrightarrow y) \vee g(y, x) \\ h(x) &= g(\mathit{0}, x)\end{aligned}$$

Although these definitions do not fit the specified syntax, it is easy to rewrite them as equivalent definitions that do:

$$\begin{aligned}f &: (v_1 \wedge v_2) \vee (\neg v_1 \wedge \neg v_2) \\ g &: (v_1 \wedge v_2) \vee (\neg v_1 \wedge \neg v_2) \vee g(v_2, v_1) \\ h &: \neg v_{n+1} \wedge g(v_{n+1}, v_1)\end{aligned}$$

v_1	v_2	$g(v_1, \ldots, v_n)$
0	*0*	*1*
0	*1*	$g(\mathit{1}, \mathit{0})$
1	*0*	$g(\mathit{0}, \mathit{1})$
1	*1*	*1*

v_1	$h(v_1, \ldots, v_n)$
0	$g(\mathit{0}, \mathit{0})$
1	$g(\mathit{0}, \mathit{1})$

Table 1: Tabular view of example

			iteration		
v_1	v_2	$g(v_1, v_2)$	0	1	2
0	*0*	*1*	*0*	*1*	*1*
0	*1*	$g(\mathit{1}, \mathit{0})$	*0*	*0*	*0*
1	*0*	$g(\mathit{0}, \mathit{1})$	*0*	*0*	*0*
1	*1*	*1*	*0*	*1*	*1*

Table 2: Tabular Kleene iteration

Note that v_1 and v_2 are the first two formal parameters, and v_{n+1} is the smallest local variable, implicitly existentially quantified over the function body.

Table 1 presents the values of these functions for each combination of inputs. We have omitted f for brevity, as it is defined without reference to any functions, so its table is an ordinary truth table. Since g only involves parameter variables v_1 and v_2, we need only consider 4 cases, and h only requires 2 cases. To handle the unconstrained variable v_{n+1} in the definition of h, we use the disjunction of the body with v_{n+1} given value *1*, and with it given value *0*.

These tables do not show a final solution to these equations, as they are not in closed form. However, we can use Kleene iteration to find a closed form, as shown in Table 2. Assuming we wish to find the least fixed point, we begin in iteration 0 by assigning the value *0* to all rows. In iteration 1, we compute the value for each row, using the values assigned in iteration 0 wherever the definition of a row refers to a another row (or the same row). Iteration 2 repeats the process, using the values from row 1. In this example, iteration 1 is a fixed point, as confirmed in iteration 2.

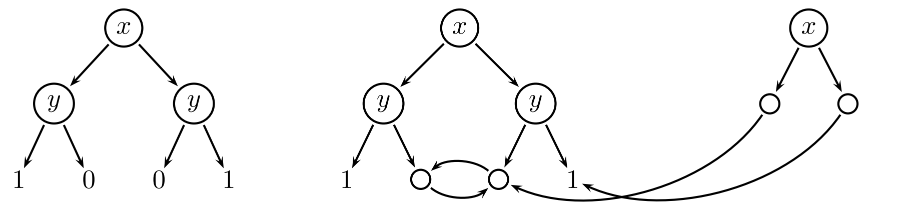
3.4 Towards ROBDDs

The tabular approach taken above will work well for functions of low arity. However, for functions with dozens of arguments, it quickly becomes unworkable. In this subsection we shall reformulate the tabular approach to work on an ROBDD-like structure.

Our revision of the ROBDD structure is similar to our relaxation of the truth table: we allow Boolean function invocations, as well as 0 and 1, as sinks of the structure. Just as we did for the tabular approach, we take advantage of the fact that, at the sinks, the values of all relevant formal parameter variables are known. Thus the "formulae" we allow for sinks are in fact just references to other sinks in the structure. Figure 3 shows the example of Section 3.3 in this view.

The structure is an ordered binary decision tree down to its leaf nodes; however, leaf nodes may refer to one another, even cyclically. It is also ordered in the same sense as an OBDD. It is not reduced, as the destinations of links from a leaf node depend upon that node's position in the tree. We refer to this structure as a tree because of its underlying structure, although it is, strictly speaking, a directed graph.

In the style of an ROBDD we would like to be able to evaluate the function for a given input by following the arcs until we reach a terminal node. If we



$f(x,y) = x \leftrightarrow y$ $\quad$ $g(x,y) = (x \leftrightarrow y) \vee g(y,x)$ $\quad$ $h(x) = g(\mathit{0},x)$

Figure 3: ROBDD-like views of recursive definitions

wish to evaluate the structure in Figure 3 for $g(\mathit{0},\mathit{0})$ or $g(\mathit{1},\mathit{1})$ then we have no problems. But when we attempt evaluation of $g(\mathit{0},\mathit{1})$ we are referred to the value of $g(\mathit{1},\mathit{0})$, and vice versa. Solving this problem is the focus of the remainder of this section. Before we can tackle that, however, we must generalise our data structure.

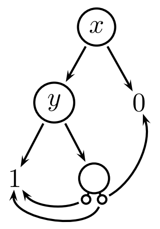
The example of Figure 3 does not show the full generality of function bodies permitted by Bod. In particular, it does not illustrate disjunctions of conjunctions of function invocations, nor does it make clear how to handle existentially quantified variables in function calls.

To handle disjunctions of conjunctions of function invocations, we generalise our definition of the DDS structure to specify that leaf nodes in the structure may be either 0, 1, or a disjunction of conjunctions of Boolean function invocations. We shall depict the new sink node graphically as an unlabelled node with a number of "pimples," each of which refers to a number of other sinks in the structure. The intended meaning is that the pimpled node should be interpreted as true iff at least one of the pimples are true. A pimple is considered to be true if all the nodes it refers to are true.

Consider the recursive definition:

$$\begin{aligned} f(x,y) \quad = \quad & x \wedge y \\ \vee \quad & x \wedge f(y,x) \wedge f(x,x) \\ \vee \quad & x \wedge f(x,x) \end{aligned}$$

This definition can be depicted as follows:



The pimpled node in this example represents the expression $(\mathit{1} \wedge \mathit{0}) \vee \mathit{1} = \mathit{1}$.

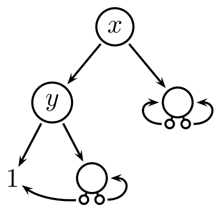
Existentially quantified variables are slightly harder to handle. A naive approach would be to treat an existentially quantified variable the same as any other, and simply extend the tree with new variables. While this approach would indeed work, it leaves the task of eliminating the existentially quantified variables.

Since existentially quantified variables are all greater (in the variable ordering) than all formal parameter variables, they are always placed on the tree's fringe. Thus eliminating these variables is a matter of disjoining all the sinks below the greatest parameter variable. Since our representation of sinks explicitly allows disjunctions, handling existentially quantified variables is simply a matter of coalescing the disjunctions that would appear anywhere under the greatest parameter variable into a single sink node.

Consider, for example, the function:

$$f(x,y) = (x \wedge y) \vee (\exists u.f(x,u))$$

We treat the $\exists u.f(x,u)$ expression as if it were (the equivalent) $f(x,\mathit{0}) \vee f(x,\mathit{1})$, leading to this:



3.5 Dendritic decision structures

We can improve on the **TD** algorithm by exploiting properties specific to our Boolean domain. Two observations are crucial here. First, as the recursively defined objects are simply truth values, living in a domain of height one, once a variable's value changes, it never needs to be re-visited. Second, as a fringe node in the structure effectively is an expression in DNF, a simple bookkeeping technique allows us to determine an appropriate time to re-evaluate a variable. Ideally we would like to re-evaluate a variable only when that evaluation would trigger a change of value. The idea is that, when the number of conjuncts in each disjunct is high, we often find that re-evaluation of one conjunct does not change the value of the conjunction. If a variable's right hand side has d disjuncts with c conjuncts in each, then it is possible for all but one conjunct in each disjunct to become true without making the whole equation true. This means it may take $(c-1)d+1$ evaluations before the variable becomes true.

It would naturally be more efficient if we could cache the work we did during the first evaluation and re-use it, instead of repeating it. The third contribution of this paper is to propose a very simple data structure and corresponding algorithm that does this very efficiently.

We consider three possible states for a variable: Undetermined ($\perp$), definitely true ($\mathit{1}$), and definitely false ($\mathit{0}$). Evaluation follows the rules of Kleene's logic:

$\neg$	
$\perp$	$\perp$
0	*1*
1	*0*

$\wedge$	$\perp$	*0*	*1*
$\perp$	$\perp$	*0*	$\perp$
0	*0*	*0*	*0*
1	$\perp$	*0*	*1*

$\vee$	$\perp$	*0*	*1*
$\perp$	$\perp$	$\perp$	*1*
0	$\perp$	*0*	*1*
1	*1*	*1*	*1*

To implement these rules we introduce an efficient structure to represent a partially evaluated disjunction. A "pimple" off a node v becomes a *receptor* which keeps count of the number of nodes that must be true before v becomes true. Dependency arrows become reversed, turning into *axons* pointing from nodes to receptors. We call the resulting structures *dendritic decision structures*, or simply DDSs. Their purpose is to speed up processing of cyclic dependencies discovered during evaluation.

Take the situation where a depends on b and b depends on a. If we start evaluation at a we discover a cycle when we request a's value during evaluation of b. In the standard recursive algorithm we would record the information that b depends on a, and use that to remember that b must be re-evaluated if a's value ever changes. In our new approach we create an axon at a and a receptor at b. The receptor has a value of 1, indicating that there is a conjunction in b with one and only one axon targeting it. Note that we only create a receptor for a node if the value of that variable depends upon as yet undetermined variables.

If the value of a changes to *1* then we follow the axon to the receptor, which is decremented. If and when a receptor reaches 0, all values in the conjunction in b represented by the receptor have become *1*. In that case, b has been sufficiently stimulated to change and is given value *1*.

3.6 The algorithm

Figure 4 gives the algorithm that uses DDSs.

The method uses a principle of aggressive propagation of truth. For each node x, a set of axons point to the nodes that depend on the value of x. More specifically, the axons point to receptors that represent conjunctions containing x. We use the notation $u \mapsto (v, c)$ for an axon that emanates from node u and points to the receptor c associated with node v. The procedure *fire* shows what happens when a node u has been determined to be *1*: Every receptor (v, c) pointed to is decremented, and if and when it reaches 0, v is deemed to have value *1* as well. Solving is then complete as far as v is concerned, and the immediate consequences of v taking value *1* are pursued: v activates *its* axons.

As in the previous algorithm, a set *Called* keeps track of nodes for which an iteration has been started but not completed. Another set, *Evaluated*, keeps track of nodes that no longer need be considered. This, however, does not just mean nodes that have a final value *0* or *1*. Once a node has been examined, it is not "solved for" again; it is considered "evaluated". However, it may still change its value from $\perp$, through appropriate activation of axons that point to the node.

3.7 An example

Figure 5 gives an example to illustrate the benefits of axons and receptors. If we begin evaluation at a and evaluate right-hand sides from left to right then we immediately find we must evaluate b, which sends us to c and then to d, which refers back to a. At this point a receptor is associated with d, starting with the value 1, and an axon is created at a pointing to that receptor. If a ever changes value later, we can immediately determine that d depends on it and update its value directly.

Continuing with evaluating d we reach the reference to b, also in the same conjunction. b is also under evaluation, so we create an axon on b pointing back to the same receptor and increase the value of the receptor to 2. If both a and b take the value *1* then the receptor will reach 0 and d will be set to *1*.

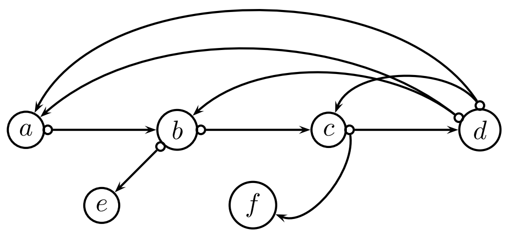
```
procedure main
  $\sigma := \varnothing$; $Called := \varnothing$; $Evaluated := \varnothing$;
  foreach $x \in V$ do $solve(x)$ od
end

function $solve(x : V) : D$
/* Solve for variable $x$ */
  if $x \in Evaluated$ then
    /* $x$ has final value or is $\perp$ and has receptors */
    return $\sigma(x)$
  elseif $x \in Called$ then
    /* $x$ is currently under evaluation */
    return $\perp$
  else
    /* $x$ has not been examined before */
    $Called := Called \cup \{x\}$;
    $\sigma(x) := evalrhs(x)$;
    $Called := Called \setminus \{x\}$;
    $Evaluated := Evaluated \cup \{x\}$;
    if $\sigma(x) = 1$ then $fire(x)$ fi;
    return $\sigma(x)$
  fi
end

function $evalrhs(x : V) : D$
/* Evaluate right hand side of $x$'s definition */
  foreach disjunct $d$ in $rhs(x)$ do
    $Uncertain := \varnothing$;
    foreach conjunct $c$ in $d$ do
      if $solve(c) = \perp$ then
        $Uncertain := Uncertain \cup \{c\}$
      fi;
    od;
    if $Uncertain \neq \varnothing$ then
      Create receptor $(x, c)$ with value $|Uncertain|$;
      foreach $u \in Uncertain$ do
        Create an axon $u \mapsto (x, c)$
      od
    fi
  od;
  if all conjuncts of any disjunct have value $1$ then
    return $1$
  elseif $x$ has receptors then
    return $\perp$
  fi;
  return $0$
end

procedure $fire(u : V)$
/* Activate all axons from node $u$ */
  foreach axon $u \mapsto (v, c)$ do
    Decrement $c$;
    if $c = 0$ then
      $\sigma(v) := 1$;
      $Evaluated := Evaluated \cup \{v\}$;
      $fire(v)$
    fi
  od
end
```

Figure 4: The algorithm based on DDSs

We do the same with the second conjunction in d's right-hand side, creating a second axon at a and an axon at c, both pointing to another receptor with value 2. This concludes the evaluation of d. We will never evaluate d again, although we may visit it later as the result of an axon firing. The configuration of

$$\begin{aligned} a &= b \\ b &= c \vee e \\ c &= d \wedge f \\ d &= (a \wedge b) \vee (a \wedge c) \\ e &= \mathit{1} \\ f &= \mathit{0} \end{aligned}$$

Dependencies:



Axons and receptors:

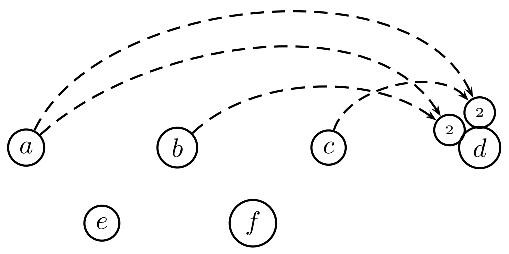
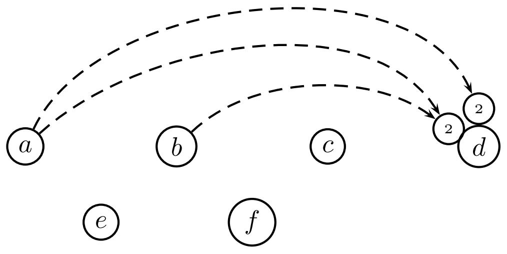


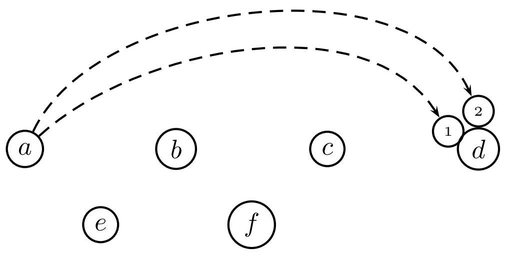
Figure 5: Use of axons and receptors

receptors and axons after processing of d is shown in figure 5. If we had started from a different definition, the configuration could be different. Note that in this figure and the following diagrams we omit the tree structure above the leaf nodes, as they do not contribute to the discussion.

Evaluation now returns to c. The value of d remains $\perp$ for now. c requires f, which immediately evaluates to $\mathit{0}$, so c has value $\mathit{0}$ also. Receptors and axons are now:



Evaluation of c has finished, so we return to evaluating b. This requires e which immediately evaluates to $\mathit{1}$. As the two variables that b depends on are disjoined, the value of b is $\mathit{1}$. Now the axon emanating from b is fired, reducing the value of its target receptor to 1. No action is taken at d, as all receptors are still non-zero:



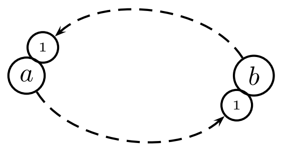
Finally, we return to evaluation of a. It depends only on b which now has value $\mathit{1}$, so a is $\mathit{1}$. This causes two axons to fire. The order of firing is not important, so let us consider the axons in the order in which they were created. The first axon's target receptor is decremented to 1. The second axon's target receptor is decremented to 0. This receptor represents a conjunction in which all conjuncts are $\mathit{1}$, so we can set the value of d to $\mathit{1}$. At this point we should activate d's axons as well, but there are no axons for this node. Evaluation of a has finished, and the system is solved completely. The solution is:

$$\begin{aligned} a = b = d = e &= \mathit{1} \\ c = f &= \mathit{0} \end{aligned}$$

3.8 Unresolved axons

It is possible to be left with a cycle of axons and receptors when we finish evaluation of the initial variable. The simplest example of this involves just two variables.

$$\begin{aligned} a &= b \\ b &= a \end{aligned}$$



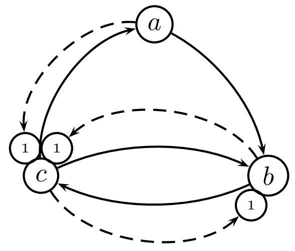
Clearly, this system has two solutions. In general, for n sets of variables which are linked by unresolved axons there are 2^n solutions to the system of equations. The sets of variables are SCCs, considering axons as edges. We obtain the least solution by setting all variables in all SCCs to $\mathit{0}$, but *any* solution can be generated easily at this point.

3.9 Fine-tuning the algorithm

We use shortcut Boolean evaluation within *evalrhs* for a considerable gain in efficiency. Once we have determined that a conjunct is $\mathit{0}$, there is no need to continue evaluation of other conjuncts within that disjunct. Likewise, once any disjunct is determined to be $\mathit{1}$, we can finish evaluation of the variable altogether.

If we have decided in advance that we want to find the *least* solution then we can avoid creating some axons and receptors. By incorporating Tarjan's SCC detection algorithm (Tarjan 1972) into our algorithm we can determine when we are at the "head" of an SCC—the first variable of the SCC we discovered (also the variable we finish evaluation of last). If a variable evaluating to $\perp$ is known to be the head of an SCC then we may safely set it to $\mathit{0}$, as we will be doing that in the final phase in any case. It is also unnecessary to place any receptors on the head of an SCC as it always takes a final value. If we are lucky this will resolve all variables in the SCC to a final value. If we are unlucky we may get a case such as the following:

$$
\begin{aligned}
a &= b \\
b &= c \\
c &= a \vee b
\end{aligned}
$$



The diagram shows the state when a finishes evaluation. As a is the head of an SCC and its value is $\perp$, we set it to *0*. This disables the receptor it points to, but cannot resolve the value of c. Likewise, b does not receive a final value.

To avoid the final phase of finding unresolved axons and setting them to *0* then we can keep a record of all variables which were given receptors within each SCC. When we finish with the head of the SCC we can go through this list and set any variables which still have the value $\perp$ to *0*. This helps to reduce the amount of space needed by the algorithm, as fewer axons and receptors are in existence at any one time.

4 Experimental evaluation

We have evaluated the algorithm discussed above, including the optimisations discussed in Section 3.9, for groundness analysis over the domain *Pos* for Prolog programs (Armstrong, Marriott, Schachte & Søndergaard 1998). The analysis derives the possible groundness states on success of each predicate in a program, possibly consisting of many source files. All source files are read into memory simultaneously, but predicates are only analysed simultaneously when required. We have implemented the fixed point algorithm in C and the abstracter in Prolog, using an established ROBDD manipulation library (Schachte 1996), written in C, in both.

The computing/test environment used has the following characteristics: desktop system with X server running, a timer with 1ms resolution (not the standard 10ms), Debian GNU/Linux version 3.1, Linux kernel version 2.6.8, Intel Pentium 4 3.0GHz CPU with 1 MB cache, 1 GB memory.

Here are the characteristics of the benchmarking undertaken:

- Timings include only fixed point processing, not the translation to Boolean form.
- Each test was performed sufficiently often to consume 10 minutes, up to a maximum of 1000 repetitions, and the smallest time was taken.
- The majority of programs benchmarked had results under 2 ms for all algorithms. These programs are not included in the table as the results were considered too inaccurate.
- NT stands for a single repetition which took more than 60 seconds.

4.1 Challenging examples

Table 3 looks at the performance of the new algorithm on the "challenge" examples for Kleene iteration. These examples are difficult because they require the maximum number of iteration steps to find a fixed point — a number which grows exponentially in the number of arguments.

Table 3: Challenge benchmark results (ms)

Benchmark	Kleene	DDSs
chain8	7.20	1.50
chain10	48.87	6.42
chain12	508.65	28.62
chain14	8382.47	130.52
chain16	NT	588.78
def8	8.56	0.87
def10	59.02	3.39
def12	569.65	15.10
def14	11362.96	66.41
def16	NT	295.14
challenge6	2.15	8.09
challenge8	18.84	150.27
challenge10	292.38	2767.88
challenge12	10099.08	48657.50

Table 4: Standard benchmark results (ms)

Benchmark	Kleene	DDSs
reducer	2.11	2.82
press	2.13	2.26
ann	2.42	7.49
bryant	2.46	20.83
ili	2.61	2.78
qplan	2.71	3.29
parser_dcg	3.09	4.42
neural	3.54	3.68
scc1	3.73	4.00
ga	3.98	4.81
simple_analyzer	5.29	21.10
sim_v5-2	5.73	6.15
peval	7.51	9.17
chat_parser	13.08	240.96
chat	13.36	247.43
chat_80	39.37	345.94

Each program is parametrised over an integer. The minimum value for each parameter is that for which analysis took at least 2ms for at least one algorithm. For `chain` and `def`, the value of the parameter is also the number of argument variables in each predicate. For `challenge`, the number of argument variables is twice the value of the parameter.

The `chain` class of programs are due to Codish (1999). The `def` and `challenge` classes of programs are due to Genaim et al. (2001). We already met an example recursive definition arising from the `def` family in Section 1: the function p is the one that arises from `def4`.

Examining traces of execution with Valgrind (Nethercote & Seward 2003) reveals much about the behaviour of both algorithms. Of course, the cost of Kleene iteration grows quickly as the number of arguments is increased because the number of iteration steps grows exponentially. But to make matters worse, the size of the data structures grows with the number of arguments, so the cost of each iteration step also grows. The new algorithm does create and traverse dependency chains of exponential length (this appears to be unavoidable in some cases), however it only traverses each chain once.

The relative performance of the `challenge` class of programs is interesting to observe. For all sizes of `challenge` problems considered in our testing, Kleene iteration outperforms our algorithm. However, note that for the higher arity cases, the time cost of Kleene iteration grows faster than that of our algorithm. For slightly higher arities, our algorithm will overtake Kleene iteration. In fact, for all three problem classes in Table 3, the cost of Kleene iteration grows substantially faster than the cost of our algorithm.

4.2 Standard benchmarks

Table 4 shows results for a number of small standard test cases. For these, Kleene iteration performs better or much better than our new algorithm. The new algorithm can be quite efficient, but it only takes one predicate with high arity or a large number of existentially quantified variables to make it impractical.

The examples for which our algorithm perform significantly slower merit closer examination. Analysis of the chat parser (`chat`, `chat_parser`, `chat_80`) was particularly expensive. Most of the analysis time was spent on an SCC which contains a predicate `possessive/14`. This predicate contains 8 local variables which are abstracted as existentially quantified variables. For each of the 2^{14} combinations of argument variables, there are 2^8 combinations of values for the existentially quantified variables, any of which could make the object variable take value T. Our algorithm checks each of these combinations, leading to poor performance in these cases. Clearly, for our approach to be practical for groundness analysis, it will be necessary to find ways to avoid complete exploration of all possible inputs and all possible valuations of existentially quantified variables.

5 Related work

One broad class of fixed point algorithms use a worklist as a basic structure. These algorithms usually include some method of detecting dependencies between variables. The general idea is to add variables into the worklist whenever a value they depend on has been changed. Variables are then selected from the worklist to be re-evaluated until the worklist is empty.

The simplest worklist algorithm adds *all* variables to the worklist when *any* variable changes value. This requires no knowledge of dependencies between variables and is trivially correct. However, if variables from the worklist are chosen in a poor order, the method may end up taking the maximum number of iterations. We can improve on this by making a preliminary pass over all the right hand sides, and finding which variables depend (statically) on other variables. For each variable x we want to know the set $\mathit{infl}(x)$, that is, the variables that depend on x. The algorithm of Kildall (1973) uses this idea. Many variants of this idea are possible, depending on policy for removal of worklist items, and the order in which elements are added. A good approach appears to be *eager evaluation* (Wunderwald 1995).

More efficient worklist algorithms track "dynamic" dependencies. The dependencies that can be read off a recursive definition over-approximate the actual dependencies, which may well change during evaluation. For example, raising the value of x from $x = F$ to $x = T$ in the expression $x \vee y$ removes its dependence on the value of y. "Dynamic" algorithms have such independence detection built in. Variants include the methods of Jørgensen (1994) and Vergauwen, Wauman & Lewi (1994), as well the **W** algorithm of Fecht & Seidl (1999).

A clever alternative to worklist algorithms with dynamic dependency detection is offered by Le Charlier & Van Hentenryck (1992). Their work is based on the observation that when a variable has many dependencies, it is highly likely that the value we get will change during the computation. It would be better to evaluate those variables that do not depend on others first, followed by evaluating those which depend on variables which already have their final value, and so on. The method of Le Charlier & Van Hentenryck aims at maintaining the precision provided by dynamic dependency detection while at the same time processing variables in an optimal order. For a variable which has not been evaluated yet, the worklist solver uses the initial approximation $\bot$. Le Charlier & Van Hentenryck, on the other hand, suspend evaluation of the current variable and eagerly begin evaluation of the needed variable. This leads to the recursive algorithm shown in Figure 2. A further refinement is the **WRT** algorithm of Fecht & Seidl (1999).

Our algorithm is based on Le Charlier & Van Hentenryck (1992), but is otherwise incomparable to the methods discussed here, as it exploits properties of the Boolean domain to make shortcuts not available to a general algorithm.

Fecht & Seidl (1999) examine the choice of which equation to re-evaluate first in a system of recursively defined equations. They demonstrate that a worklist based solver which uses timestamps as a method of dynamic SCC detection is generally more efficient than the approach of Le Charlier and Van Hentenryck. As a measure of efficiency they use the number of evaluations of right hand sides. The complexity of each right hand side is not taken into account. The improved efficiency consists of differences in handling strongly connected components. These differences do not always result in an improvement however.

The issues discussed by Fecht and Seidl are not relevant to us. Once we have found an SCC using depth first search, we treat it as a single entity and solve it at once. At the valuation level, we examine each valuation only once, revisiting it only when required by an axon. This makes a comparison based on number of evaluations of right hand sides inappropriate.

Englebert, Le Charlier, Roland & Van Hentenryck (1993) introduce two optimisations which can be applied to many fixpointing algorithms. One is clause prefix dependency, and the other is caching of operations. The first improvement avoids re-evaluating a clause prefix when no abstract value on which it depends has been updated. The second improvement consists of caching all operations on substitutions and reusing the results whenever possible. They note that the second optimisation largely subsumes the first, as the caching eliminates most of the cost of evaluating clause prefixes. Analysis time was reduced by around 28% in a C implementation by these methods. Our approach of using DDSs effectively implements this caching optimisation.

Fecht & Seidl (1998) look at limiting the calculations needed to raise a particular value in the right hand side from its previous approximation to the new approximation. Instead of recalculating the right hand side from the new value, they calculate the right hand side for the difference between the new and old values, defined as $\mathit{diff}(d_{old}, d_{new}) = d_{\mathit{diff}}$ such that $d_{old} \sqcup d_{\mathit{diff}} = d_{new}$. The result is then joined with the result of the old calculation. In our setting the domain has only two values, so this optimisation is not useful.

6 Conclusion

We have presented a novel approach to the problem of finding closed forms of recursively defined Boolean functions. Our approach centers on the idea of solving these equations one valuation at a time. The advantage of this is that the solution for a single valuation is ultimately a Boolean value. This means that, since we are climbing an ascending chain of height one, once the solution for a valuation changes from an initial value of false, it will not change again.

To capitalise on this advantage, we have introduced a new data structure, the dendritic decision structure. This data structure captures the dependency relation among valuations, leading to an algorithm for finding closed forms that very efficiently handles problems requiring many iterations when solved by Kleene iteration.

This new algorithm, like Kleene iteration, suffers from very bad worst-case performance. Unfortunately, this worst-case performance occurs in our testing domain of groundness analysis of common programs for our algorithm, but only on rare or artificial cases for Kleene iteration. Therefore we do not yet consider this algorithm suitable for practical use in groundness analysis. Further work is needed to make it competitive in general. In particular, it will be necessary to find a way to avoid building and exploring the entire DDS structure for a function whenever possible. It will also be important to avoid exploring all valuations for existentially quantified variables whenever possible.

On the positive side, it is clear that a hybrid approach would give the best of both worlds. We can perform a few steps of Kleene iteration, which is enough to solve the majority of inputs. If this does not produce a solution, then we can use the last approximation from Kleene iteration as a starting point for our algorithm. This is easily accomplished by replacing the fixed part of the definitions of all functions in the SCC being solved with the corresponding current Kleene approximation. Based on the benchmark results, such an approach would be efficient for all cases except `challenge`, which is difficult for all algorithms.

Acknowledgements

We would like to thank Martin Harris and Adrian Teo for their contributions to early versions of the algorithms and data structures presented in this paper.

References

Armstrong, T., Marriott, K., Schachte, P. & Søndergaard, H. (1998), 'Two classes of Boolean functions for dependency analysis', *Science of Computer Programming* **31**(1), 3–45.

Brace, K., Rudell, R. & Bryant, R. (1990), Efficient implementation of a BDD package, *in* 'Proc. Twenty-seventh ACM/IEEE Design Automation Conf.', pp. 40–45.

Bryant, R. (1992), 'Symbolic Boolean manipulation with ordered binary-decision diagrams', *ACM Computing Surveys* **24**(3), 293–318.

Codish, M. (1999), 'Worst-case groundness analysis using positive Boolean functions', *Journal of Logic Programming* **41**(1), 125–128.

Cousot, P. & Cousot, R. (1977), Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, *in* 'Proc. Fourth ACM Symp. Principles of Programming Languages', ACM Press, pp. 238–252.

Englebert, V., Le Charlier, B., Roland, D. & Van Hentenryck, P. (1993), 'Generic abstract interpretation algorithms for Prolog: Two optimization techniques and their experimental evaluation', *Software Practice and Experience* **23**(4), 419–459.

Fecht, C. & Seidl, H. (1998), Propagating differences: An efficient new fixpoint algorithm for distributed constraint systems, *in* C. Hankin, ed., 'Proc. ESOP'98', Vol. 1381 of *Lecture Notes in Computer Science*, Springer, pp. 90–104.

Fecht, C. & Seidl, H. (1999), 'A faster solver for general systems of equations', *Science of Computer Programming* **35**(2–3), 137–161.

Genaim, S., Howe, J. M. & Codish, M. (2001), 'Worst-case groundness analysis using definite Boolean functions', *Theory and Practice of Logic Programming* **1**, 611–615.

Jørgensen, N. (1994), Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration, *in* B. Le Charlier, ed., 'Static Analysis', Vol. 864 of *Lecture Notes in Computer Science*, Springer, pp. 329–345.

Kildall, G. A. (1973), A unified approach to global program optimization, *in* 'Proc. First Annual ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages', ACM Press, pp. 194–206.

Le Charlier, B. & Van Hentenryck, P. (1992), A universal top-down fixpoint algorithm, Technical Report CS–92–25, Brown University, Department of Computer Science.

Liaw, H.-T. & Lin, C.-S. (1992), 'On the OBDD-representation of general Boolean functions', *IEEE Transactions on Computers* **C-41**(6), 661–664.

Nethercote, N. & Seward, J. (2003), 'Valgrind: A program supervision framework', *Electronic Notes in Theoretical Computer Science* **89**(2).

Schachte, P. (1996), Efficient ROBDD operations for program analysis, *in* K. Ramamohanarao, ed., 'ACSC'96: Proceedings of the 19th Australasian Computer Science Conference', Australian Computer Science Communications, pp. 347–356.

Tarjan, R. E. (1972), 'Depth-first search and linear graph algorithms', *SIAM Journal of Computing* **1**(2), 146–170.

Vergauwen, B., Wauman, J. & Lewi, J. (1994), Efficient fixpoint computation, *in* B. Le Charlier, ed., 'Static Analysis', Vol. 864 of *Lecture Notes in Computer Science*, Springer, pp. 314–328.

Wunderwald, J. E. (1995), Memoing evaluation by source-to-source transformation, *in* M. Proietti, ed., 'Logic Program Synthesis and Transformation', Vol. 1048 of *Lecture Notes in Computer Science*, Springer, pp. 17–32.