Symbolic Pointer Analysis Revisited [*]

Jianwen Zhu

Silvian Calman

Department of Electrical and Computer Engineering
University of Toronto, Toronto, Ontario, Canada
{jzhu, calman}@eecg.toronto.edu

ABSTRACT

Pointer analysis is a critical problem in optimizing compiler, parallelizing compiler, software engineering and most recently, hardware synthesis. While recent efforts have suggested symbolic method, which uses Bryant's Binary Decision Diagram as an alternative to capture the point-to relation, no speed advantage has been demonstrated for context-insensitive analysis, and results for context-sensitive analysis are only preliminary.

In this paper, we refine the concept of symbolic transfer function proposed earlier and establish a common framework for both context-insensitive and context-sensitive pointer analysis. With this framework, our transfer function of a procedure can abstract away the impact of its callers and callees, and represent its point-to information completely, compactly and canonically. In addition, we propose a symbolic representation of the invocation graph, which can otherwise be exponentially large. In contrast to the classical frameworks where context-sensitive point-to information of a procedure has to be obtained by the application of its transfer function exponentially many times, our method can obtain point-to information of all contexts in a single application. Our experimental evaluation on a wide range of C benchmarks indicates that our context-sensitive pointer analysis can be made almost as fast as its context-insensitive counterpart.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors – compilers, optimization

General Terms

Languages, Experimentation

Keywords

Pointer analysis, call graph construction, binary decision diagrams

[*]This work is supported by NSERC and University of Toronto.

1. INTRODUCTION

As an abstraction of memory addresses, a pointer is one of the most powerful yet problematic constructs in modern imperative programming languages. Pointers are often the sources of nontrivial software bugs, such as freed memory accesses and memory leaks, which reveal symptoms much later than their causes. Pointers significantly reduce the effectiveness of compiler optimizations, since they cause memory alias problems that obscure the data dependency. In addition, the presence of function pointers and virtual functions complicates the complete construction of call graphs and therefore interprocedural optimization. Pointers also limit the practical application of classical behavioral synthesis, which strives to compile an ordinary program directly into digital hardware, since the hardware semantics of pointers are not yet clear. Pointer analysis, which determines the *point-to information*, or the runtime values of program pointers at compile time, is therefore a subject of intensive research for the last two decades in the broad areas of programming languages, optimizing and parallelizing compiler, software engineering and more recently, computer-aided design (CAD) for integrated circuits.

The reported analysis algorithms vary with different accuracy-speed tradeoff and can be categorized according to two criteria: *flow sensitivity* and *context sensitivity*. A flow-insensitive (FI) algorithm ignores the order of statements when it calculates pointer information, whereas a flow-sensitive (FS) algorithm takes control flow within a procedure into account. A context-insensitive (CI) algorithm does not distinguish the different calling contexts of a procedure, whereas a context-sensitive (CS) does. Fast polynomial algorithms, such as derivatives of Steensgaard's [36] and Andersen's [3], have been developed for context-insensitive analysis. It has been shown that state-of-the-art implementation of CI analysis can analyze million-line code [19]. On the other hand, most context-sensitive analysis algorithms reported in the literature suffer from a worst-case exponential time complexity. The best efforts today to address the scalability problem seem to be the use of partial context sensitivity [27], or the use of a polymorphic, constraint-based analysis engine [17, 16].

Representing a wide departure from the traditional methods that use explicit point-to graph to capture the program state, in [41] we proposed the use of Binary Decision Diagram (BDD) to capture the point-to relation implicitly as Boolean functions and demonstrated positive, yet preliminary runtime result for context-sensitive analysis. In [5] Berndl et. al. demonstrated the space efficiency of a similar method applied to the context-insensitive analysis of Java programs. In this paper, we extend the previous two efforts and make the following contributions.

- **Symbolic transfer function**. We extend our original proposal of symbolic transfer function in [41], which uses a

Boolean function represented by BDD to capture the program state of a procedure as a function of its caller program state. Our extension allows the additional parameterization of callee program state, which enables the capture of transfer functions in a single pass.

- **Common CI/CS symbolic analysis framework**. We establish a common, efficient framework for both context-sensitive and context-insensitive analysis. This not only enables the leverage of transfer function for the first time to speed up CI analysis, but also enables the study of speed-accuracy trade-off among a spectrum of symbolic analysis methods with different context-sensitivity. To the best of our knowledge, such frameworks useful in many studies [22, 21, 23, 17] have not been reported for BDD-based pointer analysis.
- **Symbolic invocation graph**. Most previous methods [14, 39] for context sensitive analysis, including our own [41], requires the construction of an invocation graph, which can be exponentially large. To avoid this problem, we propose the use of BDDs to annotate the call graph edges with Boolean functions to implicitly capture the corresponding invocation edges. Such representation of the invocation graph leads to the exponential reduction of memory size in practice.
- **State superposition**. In contrast to the previous efforts where program states of a procedure under different calling contexts have to be evaluated separately by the application of transfer functions, we devise a scheme where the the symbolic invocation graph is leveraged to collectively compute a superposition of all states of a procedure under different contexts. This leads to an exponential reduction of analysis runtime in practice.

The implementation of the aforementioned ideas yielded interesting new results on a comprehensive set of C benchmarks. First, we show that the speed of the symbolic method can be made comparable to the traditional methods for context-insensitive analysis [23]. Second, we show that context-sensitive analysis can have a runtime in the same order of magnitude as its context-insensitive counterpart.

The rest of the paper is organized as follows. In Section 2, we describe the construction of our symbolic transfer function. In Section 3, we describe our analysis framework as well as the core algorithms involved. In Section 4, we describe the symbolic invocation graph and show how it can be used to compute the state superposition. In Section 5, we complement the theory with a discussion on various engineering issues. In Section 6, we present experimental results. We discuss the related work in Section 7.

2. SYMBOLIC TRANSFER FUNCTION

2.1 Symbolic Program State

The goal of pointer analysis is to statically estimate the runtime program state, or the set of values each program location can hold. To trade accuracy for analysis speed, we often collapse related program locations together, thereby forming a **block**. Locations within a block are not distinguished. The blocks can be global variables, or local variables, or procedure parameters, or dynamically allocated memory blocks. The values of interest are only the addresses of the blocks.

EXAMPLE 1. *Consider the C program in Figure 1 (a), which is modified from [27]. The program contains global blocks g, a, local blocks p, q, r, t, f, and a dynamic block m allocated at S3.*

The program state is often abstracted as a *point-to graph* $\langle V, E\rangle$, whose vertices V represent the set of blocks, and an edge $\langle u, v\rangle \in E$ from block u to block v indicates that it is possible that the content of block u is the address of block v. The set of all edges defines the point-to relation.

EXAMPLE 2. *Figure 2 shows a point-to graph capturing the program state after the completion of the main procedure in Example 1.*

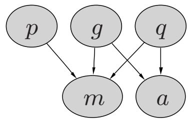


Figure 2: Program state on the completion of Example 1.

In [41], we proposed an alternative way to capture the point-to relation, which associates each block u with two Boolean functions, called its domain (denoted by u^*) and range (denoted by u for convenience). The set of domain functions of all blocks form an orthogonal function set in the Boolean space, called the domain space, spanned by the set of Boolean variables $\overrightarrow{x}^* = \{x_0^*, ..., x_{n-1}^*\}$, such that $\forall u \neq v, u \times v = 0$. Similarly, the set of range functions of all blocks form a orthogonal function set in the companion Boolean space, called the range space, spanned by the set of Boolean variables $\overrightarrow{x} = \{x_0, ..., x_{n-1}\}$.

The domain and range functions of blocks are most conveniently selected as disjoint **minterms** in the Boolean spaces [18]. For each block $u \in V$, we denote its corresponding minterms as X_u^*, X_u, or simply u^*, u[1]. Example 3 shows the assignment of minterms to the blocks. It is important to note that with this encoding scheme, the dimension of the Boolean spaces $\overrightarrow{x}$ and $\overrightarrow{x}^*$ is an exponential reduction of the number of minterms, or the number of blocks. This fact directly contributes to the efficiency of our representation, as will be explained in Section 2.3.

EXAMPLE 3. *The following table shows how the blocks in Example 1 are mapped to minterms in the Boolean spaces $\overrightarrow{x}$ and $\overrightarrow{x}^*$. Note that the dimension (number of Boolean variables) of both spaces is 4.*

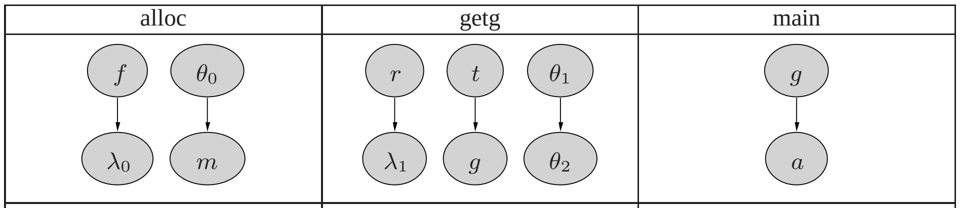
domain	range
$a^* = X_0 = \bar{x}_0^*\bar{x}_1^*\bar{x}_2^*\bar{x}_3^*$	$a = X_0^* = \bar{x}_0\bar{x}_1\bar{x}_2\bar{x}_3$
$g^* = X_1 = \bar{x}_0^*\bar{x}_1^*\bar{x}_2^*x_3^*$	$g = X_1^* = \bar{x}_0\bar{x}_1\bar{x}_2x_3$
$p^* = X_2 = \bar{x}_0^*\bar{x}_1^*x_2^*\bar{x}_3^*$	$p = X_2^* = \bar{x}_0\bar{x}_1x_2\bar{x}_3$
$q^* = X_3 = \bar{x}_0^*\bar{x}_1^*x_2^*x_3^*$	$q = X_3^* = \bar{x}_0\bar{x}_1x_2x_3$
$t^* = X_4 = \bar{x}_0^*x_1^*\bar{x}_2^*\bar{x}_3^*$	$t = X_4^* = \bar{x}_0x_1\bar{x}_2\bar{x}_3$
$r^* = X_5 = \bar{x}_0^*x_1^*\bar{x}_2^*x_3^*$	$r = X_5^* = \bar{x}_0x_1\bar{x}_2x_3$
$f^* = X_6 = \bar{x}_0^*x_1^*x_2^*\bar{x}_3^*$	$f = X_6^* = \bar{x}_0x_1x_2\bar{x}_3$
$m^* = X_7 = \bar{x}_0^*x_1^*x_2^*x_3^*$	$m = X_7^* = \bar{x}_0x_1x_2x_3$

We can now capture the point-to relation by mapping each edge $\langle u, v\rangle$ in the point-to graph by a Boolean product u^*v, where u^* represents the domain minterm of u, and v represents the range minterm of v. In other words, given a program state represented by E, the point-to relation can be represented by a Boolean function $\Sigma_{\langle u,v\rangle \in E} u^*v$.

[1]Hereafter we use the convention that subscripted lower case letters denote Boolean space variables, whereas subscripted upper case letters denote minterms in the Boolean space.

```
char *g, a;                    1
void main() {                  2
   char   *p, *q;              3
S1: alloc( &p );               4
   getg( &q );                 5
   g = &a;                     6
   }                           7
                               8
void getg( char** r ) {        9
   char  **t = &g;            10
   if( g == NULL )            11
S2:      alloc( t );          12
   *r = *t;                   13
   }                          14
                              15
void alloc( char** f ) {      16
S3: *f = malloc(1);           17
   }                          18
```

(a) C source code



$$\begin{aligned} &\mathcal{T}^{alloc} \\ =\; & f^*\lambda_0 + \theta_0^* m \\ =\; & \bar{x}_0^* x_1^* x_2^* \bar{x}_3^* x_0 \bar{x}_1 \bar{x}_2 \bar{x}_3 \\ +\; & x_0^* \bar{x}_1^* \bar{x}_2^* x_3 \bar{x}_0 x_1 x_2 x_3 \end{aligned}$$

$$\begin{aligned} &\mathcal{T}^{getg} \\ =\; & r^*\lambda_1 + t^* g + \theta_1^* \theta_2 \\ =\; & \bar{x}_0^* x_1^* \bar{x}_2^* x_3^* x_0 \bar{x}_1 x_2 \bar{x}_3 \\ +\; & \bar{x}_0^* x_1^* \bar{x}_2^* \bar{x}_3^* \bar{x}_0 \bar{x}_1 \bar{x}_2 x_3 \\ +\; & x_0^* \bar{x}_1^* x_2^* x_3^* x_0 x_1 \bar{x}_2 \bar{x}_3 \end{aligned}$$

$$\begin{aligned} &\mathcal{T}^{main} \\ =\; & g^* a \\ =\; & \bar{x}_0^* \bar{x}_1^* \bar{x}_2^* x_3^* \bar{x}_0 \bar{x}_1 \bar{x}_2 \bar{x}_3 \end{aligned}$$

(b) Transfer functions

Figure 1: A walk-through example.

EXAMPLE 4. *The program state in Example 2 can be represented by a Boolean function:*

$$\begin{aligned} & p^*m + g^*m + g^*a + q^*m + q^*a \\ =\; & \bar{x}_0^* \bar{x}_1^* x_2^* \bar{x}_3^* \bar{x}_0 x_1 x_2 x_3 + \bar{x}_0^* \bar{x}_1^* \bar{x}_2^* x_3^* \bar{x}_0 x_1 x_2 x_3 \\ +\; & \bar{x}_0^* \bar{x}_1^* \bar{x}_2^* x_3^* \bar{x}_0 \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_0^* \bar{x}_1^* x_2^* x_3^* \bar{x}_0 x_1 x_2 x_3 \\ +\; & \bar{x}_0^* \bar{x}_1^* x_2^* x_3^* \bar{x}_0 \bar{x}_1 \bar{x}_2 \bar{x}_3 \end{aligned}$$

2.2 Symbolic Transfer Function

The point-to graph can be used to represent the program state in a procedure only when the program state before the procedure is called is known. To safely capture the point-to relation under all circumstances, the concept of transfer function, which can be intuitively considered as point-to relation parameterized over different calling contexts, has been widely used [39, 10, 11]. The parameters of the transfer function do not necessarily correspond to the parameters of the procedure. In fact, any *memory dereferences*, including parameter, local and global dereferences within the procedure can be a transfer function parameter, since their values are not known until the state of its caller is known. A memory dereference can be characterized by the notion of *access path* $\langle b, l \rangle$, where b is the root memory block, and l is the level of dereferences. An access path with the form $\langle b, 0 \rangle$ is trivial and always resolve to the constant address value b, whereas an access path with the form $\langle b, 1 \rangle$ represents the address values stored in b. After the transfer functions of procedures are derived, they can be *applied* at their corresponding call sites by substituting the parameters, or the unknowns, with the known program state.

Many analysis techniques, especially those that are context-insensitive, do not use transfer functions [5]. While the overhead of transfer function application can be avoided, these techniques may have to re-analyze the procedures, which is often the case during fixed-point iteration. This not only implies redundant computation, but also implies that the program information has to be kept in memory during analysis. This potential scalability problem leads us to the decision of using the transfer function approach even for context-insensitive analysis.

In [41] we introduce the notion of **initial state blocks**, each of which corresponds to the set of possible values of a memory dereference before *entering* the procedure. An initial state block is treated as if it was a separate memory block.

One problem with only using initial blocks as transfer function parameters is that the transfer function of a procedure depends very much on the transfer functions of its callees. To make sure that the point-to information of a procedure is evaluated as late as possible, in this paper we introduce **final state blocks**, which represent possible values of a memory dereference before *leaving* the procedure. Again, we use disjoint minterms in Boolean spaces $\overrightarrow{x}^*$ and $\overrightarrow{x}$ to encode initial and final state blocks. We follow the convention that the minterms λ_k and θ_k represents the initial and final state block for memory dereference k respectively.

EXAMPLE 5. *Consider the procedure alloc in Example 1, where the parameter f is dereferenced. Since the value of f is unknown, we cannot determine the memory blocks to be updated. With the introduction of the initial state block λ_0, and the final state block θ_0, the procedure can be summarized with a transfer function as shown in the point-to graph of Figure 1 (b). Similarly, we can obtain the transfer function of procedure getg in Example 1 in Figure 1 (b) where memory dereference 1 corresponds to *r and memory dereference 2 corresponds to **t*[2]*. The introduced initial and final blocks can be encoded as minterms in $\overrightarrow{x}$ and $\overrightarrow{x}^*$ in the following table.*

domain	*range*	*deref*
$\lambda_0^* = X_8^* = x_0^* \bar{x}_1^* \bar{x}_2^* \bar{x}_3^*$	$\lambda_0 = X_8 = x_0 \bar{x}_1 \bar{x}_2 \bar{x}_3$	**f* = $\langle f, 1 \rangle$
$\theta_0^* = X_9^* = x_0^* \bar{x}_1^* \bar{x}_2^* x_3$	$\theta_0 = X_9 = x_0 \bar{x}_1 \bar{x}_2 x_3$	**f* = $\langle f, 1 \rangle$
$\lambda_1^* = X_{10}^* = x_0^* \bar{x}_1^* x_2^* \bar{x}_3^*$	$\lambda_1 = X_{10} = x_0 \bar{x}_1 x_2 \bar{x}_3$	**r* = $\langle r, 1 \rangle$
$\theta_1^* = X_{11}^* = x_0^* \bar{x}_1^* x_2^* x_3^*$	$\theta_1 = X_{11} = x_0 \bar{x}_1 x_2 x_3$	**r* = $\langle r, 1 \rangle$
$\theta_2^* = X_{12}^* = x_0^* x_1^* \bar{x}_2^* \bar{x}_3^*$	$\theta_2 = X_{12} = x_0 x_1 \bar{x}_2 \bar{x}_3$	***t* = $\langle t, 2 \rangle$

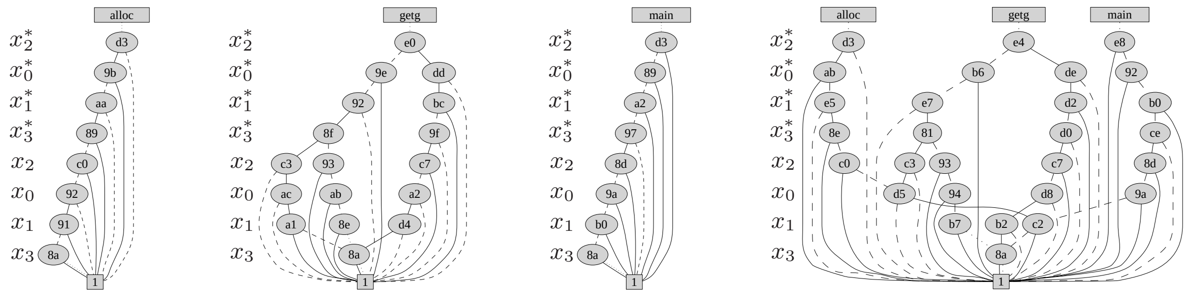
In addition, we introduce final blocks corresponding to actual parameter values passed to procedures at Line 4, 5 and 12 respectively. Note that while they do not appear in transfer functions, they will be used in the future for transfer function application.

domain	*range*	*deref*
$\theta_3^* = X_{13}^* = x_0^* x_1^* \bar{x}_2^* x_3^*$	$\theta_3 = X_{13} = x_0 x_1 \bar{x}_2 x_3$	*p* = $\langle p, 0 \rangle$
$\theta_4^* = X_{14}^* = x_0^* x_1^* x_2^* \bar{x}_3^*$	$\theta_4 = X_{14} = x_0 x_1 x_2 \bar{x}_3$	*q* = $\langle q, 0 \rangle$
$\theta_5^* = X_{15}^* = x_0^* x_1^* x_2^* x_3^*$	$\theta_5 = X_{15} = x_0 x_1 x_2 x_3$	**t* = $\langle t, 1 \rangle$

2.3 Binary Decision Diagram

We have established the use of Boolean functions as an alternative to capture the point-to relation. However, other than being well

[2]Note that here we follow the convention of writing L-values, thus the R-value *t at line 14 of Example 1 is written as **t.



(a) alloc (b) getg (c) main (d) all

Figure 3: Transfer functions in BDD.

founded on the formalism of Boolean algebra, we have not yet justified its use in terms of efficiency. In this section, we introduce Bryant's Reduced Ordered Binary Decision Diagram (ROBDD or simply BDD) [6], a proven technology for the efficient manipulation of Boolean functions.

Traditional representations of Boolean functions include truth tables, Karnaugh maps, or sum-of-products [18], each suffering from an exponential size with respect to the number of variables. Bryant used a rooted, directed binary graph to represent an arbitrary Boolean function. Given a Boolean space $\overrightarrow{x} = \{x_0, x_1, x_2, ..., x_{n-1}\}$, a Boolean function f_v corresponds to a graph rooted at graph node v. Each node in the graph is characterized by an *index* i, corresponding to a Boolean variable x_i, as well as its negative *cofactor* f_{low} and positive cofactor f_{high}, each of which is by itself a Boolean function, and therefore a graph node. Logically, f_v is related to its two cofactors by Shannon expansion $f_v = x_i f_{low} + \bar{x_i} f_{high}$. Two outstanding nodes, called the *terminal nodes*, represent the constant logic value 0 and 1. The terminal nodes are assumed to have an index of infinity. By imposing two invariants on the graph, Bryant manages to keep the representation canonical. First, all variables have a fixed ordering, that is, the index of any non-terminal node must be less than the index of its cofactors. Second, all isomorphic subgraphs are reduced into one, that is, if the cofactors of two graph nodes u and v are the same, and their indices are the same, then they will be the same.

Figure 3 shows the BDD representation of symbolic transfer functions in the previous section. Note that we use BDD to represent both the transfer functions and the program states. The fact that BDD is nothing but a graph representation of a Boolean function begs the question that why we do not use the point-to graph in the first place, which seems to be much more intuitive. One primary advantage of using BDD is that point-to graphs need to be maintained for every procedure, each of which may share many common edges. In other words, there is a large amount of redundancy. In contrast, BDD enables the maximum sharing among graph nodes, and point-to information in different procedures, at different program points can be reused. As an example, the internal BDD nodes *8a, d5, c2* are shared among different transfer functions. As the program grows large, such sharing occurs in a large scale. As a result, when BDD is used to represent a point-to set, its size is not necessarily proportional to its cardinality, as in the case of point-to graph – often times it is proportional to the dimension of the Boolean space. This space efficiency will translate into speed efficiency, as will become apparent in later sections.

3. SYMBOLIC POINTER ANALYSIS FRAMEWORK

3.1 Recurrence Equations

We now describe our pointer analysis framework. In order to focus on the fundamentals, rather than the implementation details, we assume that after preprocessing, the program can be characterized by the following mathematical model. Note that in this model we do not distinguish between call graph and invocation graph, thus both context-insensitive analysis and context-sensitive analysis can be applied equally well based on this formulation. Also note that for now we assume there is no dynamic procedure calls. The call graph or invocation graph can therefore be built in advance. This assumption will be relaxed by more careful engineering discussed in Section 5.

- $I \subset [0, \infty)$ is the set of procedures. For context-insensitive analysis, they correspond to the nodes in the call graph. For context-sensitive analysis, they correspond to the nodes in the invocation graph, each of which corresponds to a calling path in the call graph. We also assume that procedure 0 corresponds to the top procedure in the whole program.

- $J \subset [0, \infty)$ is the set of memory blocks contained in the program. It includes globals, locals, parameters as well as heap objects.

- $K \subset [0, \infty), \forall i \in I$ corresponds to the set of memory dereferences.

- $\mathcal{D} : K \mapsto J \times \mathcal{Z}$ characterizes the access path of each memory dereference $k \in K$ by a tuple $\langle b, l \rangle$ where $b \in J$ is a memory block, and $l \in \mathcal{Z}$ is the level of dereferences. This representation can be extended with more complex access patterns.

- $\{\mathcal{T}^i(\overrightarrow{\lambda}, \overrightarrow{\theta}) | \forall i \in I\}$ corresponds to the set of transfer functions for each procedure i. Here $\overrightarrow{\lambda} = [\lambda_0, ...\lambda_{|K|-1}]$ corresponds to the initial state blocks, and $\overrightarrow{\theta} = [\lambda_0, ...\lambda_{|K|-1}]$ corresponds to final state blocks.

- $\mathcal{C} : I \mapsto 2^I$ corresponds to the calling relation. $\forall i \in I, \mathcal{C}(i)$ gives the set of callees of i, $\mathcal{C}^{-1}(i)$ gives the set of callers of i.

- $\mathcal{B} : I \times I \times K \mapsto K$ corresponds to parameter binding. For each call site with caller $i \in I$ and callee $j \in I$, and the formal parameter dereference $k \in K$, $\mathcal{B}(i, j, k)$ gives the dereference in caller i corresponding to the actual.

The task of pointer analysis is therefore finding program state S^i for each procedure $i \in I$. We obtain the results by solving the following recurrence equations.

$$\begin{aligned} \Lambda_k^i &= \sum_{j \in \mathcal{C}^{-1}(i)} \Theta^j_{\mathcal{B}(j,i,k)}, \forall k \in K, i \in I && (1) \\ \Theta_k^i &= \text{query}(S^i, \mathcal{D}(k)), \forall k \in K, i \in I && (2) \\ S^i &= \sum_{j \in \mathcal{C}^{-1}(i)} S^j + \sum_{j \in \mathcal{C}(i)} S^j + && (3) \\ & \quad \mathcal{T}^i(\overrightarrow{\lambda} \rightarrow \overrightarrow{\Lambda}^i, \overrightarrow{\theta}^i \rightarrow \overrightarrow{\Theta}^i), \forall i \in I \end{aligned}$$

Equation (1) computes the initial value of a formal parameter, or memory dereference k in procedure i before entering the procedure. It is computed by combining the states of the corresponding actuals from all incoming callers. The set of callers are computed by $\mathcal{C}^{-1}(i)$. Given caller j and callee i, the actual memory dereference corresponding to the formal k is given by $\mathcal{B}(j, i, k)$, whose corresponding value is given by $\Theta^j_{\mathcal{B}(j,i,k)}$. Equation (2) computes the final value of memory dereference k in procedure i before leaving the procedure. It is computed by performing a *state query* on S^i. Equation (3) computes the state S^i of procedure i. It is computed by adding the states of its callers and callees as well as new states originating from itself. The latter is computed by substituting the initial and final state blocks that appear in its transfer functions by the actual state blocks computed in Equation 1 and Equation 2. This procedure is called *transfer function application*.

The recurrence equation set can be solved by standard iterative framework that terminates at a fixed point. The initial condition for the iteration is set in the following equation, which essentially computes the sum of all parameter-independent point-to information in the transfer functions.

$$S^i = \sum_{i \in I} \mathcal{T}^i(\overrightarrow{\lambda} \rightarrow 0, \overrightarrow{\theta} \rightarrow 0) \qquad (4)$$

EXAMPLE 6. *The initial state of the program in Example 1 is $g^*a + t^*g$.*

3.2 Symbolic State Query

We now consider how to perform state query efficiently. Given a memory dereference of block b with level l, Algorithm 1 performs the state query by computing the reachable envelop of depth l on the point-to graph starting from block b. In contrast to the traditional approach where a breadth-first search has to be performed to explicitly enumerate all neighbors of a node in the point-to graph, our representation enables the use of implicit technique originally developed in the CAD community for the formal verification of digital hardware. This approach relies on the efficiency of *image computation*, which *collectively* computes the set of successors in a graph given a set of predecessors. Since in our representation, a set of memory blocks can be represented as a Boolean function, the image computation can be formulated as Boolean function manipulation, which in turn can be efficiently implemented on BDD. As shown in Line 5, the image computation is performed by multiplying the state with the Boolean function of the predecessor in the domain space, and then existentially abstracting away the Boolean variables in the domain space. Example 7 illustrate how it works. Many efforts have been invested to make this operation particularly efficient [12, 13, 7, 29].

ALGORITHM 1. *State query.*

```
query( $S$, $\langle b, l \rangle$ ) {                                        1
  if( $l == 0$ ) return $X_b$ ;                                         2
  else {                                                             3
    domain = query(S, $\langle b, l-1 \rangle$)$|_{\overrightarrow{x} \rightarrow \overrightarrow{x}^*}$;          4
    return $\exists \overrightarrow{x}^*.[S \wedge domain]$ ;                          5
    }                                                                6
  }                                                                  7
```

EXAMPLE 7. *Consider the state of procedure* main *represented by the point-to graph in Figure 2, which can be represented symbolically by $S = p^*m + g^*m + g^*a + q^*m + q^*a$. To find out where g points to, we first multiply S by g^*. Since g^* is orthogonal to p^* and q^* by the property of minterms, the step yields $g^*m + g^*a = \bar{x}_0^*\bar{x}_1^*\bar{x}_2^*x_3^*\bar{x}_0x_1x_2x_3 + \bar{x}_0^*\bar{x}_1^*\bar{x}_2^*x_3^*\bar{x}_0\bar{x}_1\bar{x}_2\bar{x}_3$, in other words, all irrelevant point-to facts are filtered. We then abstract away all domain variables $\overrightarrow{x}^*$, which yields $\bar{x}_0x_1x_2x_3 + \bar{x}_0\bar{x}_1\bar{x}_2\bar{x}_3 = m + a$.*

3.3 Symbolic Transfer Function Application

We now consider how to perform transfer function application efficiently. A naive way is to find the cofactors of transfer function $\mathcal{T}^i$ with respect to each parameter to be substituted. For example, the cofactor with respect to λ_k can be found by $\exists \overrightarrow{x}.(\lambda_k \wedge \mathcal{T}^i)$. The application result can then be found by summing up all cofactors multiplied by the corresponding substituent.

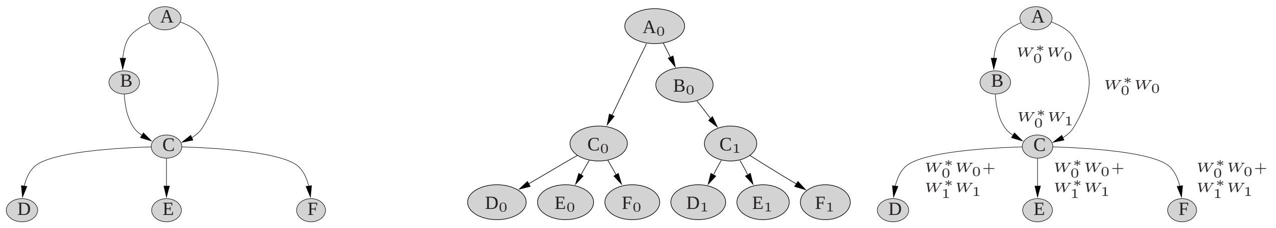
We propose a new method such that the substitutions can be performed *collectively*. This is achieved by introducing another Boolean space $\overrightarrow{y} = \{y_0, ..., y_{m-1}\}$ and its companion $\overrightarrow{y}^*$, the minterms of which are used to distinguish different memory dereferences such that dereference $k \in K$ corresponds to minterms Y_k and Y_k^* respectively. In addition, we introduce another Boolean variable pair z and z^* to distinguish the initials and finals. The Boolean variables introduced help to form **determinants** that can help to distinguish the parameters to be substituted. We can then modify each of the transfer function $\mathcal{T}^i$ into an *augmented transfer function* $\hat{\mathcal{T}}^i$ where each occurrence of parameter λ_k is multiplied by its determinant $\bar{z}Y_k$, θ_k by zY_k, and θ_k^* by $z^*Y_k^*\theta_k^*$.

EXAMPLE 8. *The augmented transfer functions of procedures in Example 1 are: $\hat{\mathcal{T}}^{alloc} = f^*\bar{z}Y_0\lambda_0 + z^*Y_0^*m$, $\hat{\mathcal{T}}^{getg} = r^*\bar{z}Y_1\lambda_1 + t^*g + z^*Y_1^*\theta_1 zY_2\theta_2$ and $\hat{\mathcal{T}}^{main} = \mathcal{T}^{main}$.*

Similarly, we can create a *binding* between all substituents and parameters by multiplying each with the corresponding determinant, that is, Λ_k^i by $\bar{z}Y_k$, Θ_k^i by zY_k. As shown in Algorithm 2, the binding can be used to multiply the augmented transfer function. Since terms with different determinants will be canceled thanks to the orthogonality of minterms, the desired result can be obtained from the multiplication result by existentially abstracting away the determinant variables.

ALGORITHM 2. *Transfer Function Application.*

```
apply( $\hat{\mathcal{T}}^i$, $\overrightarrow{\Lambda}^i$, $\overrightarrow{\Theta}^i$ ) {                                    8
  s = $\hat{\mathcal{T}}^i$;                                                          9
  binding = $\sum_{k \in K}(\bar{z}Y_k\Lambda_k^i + zY_k\Theta_k^i)$;                          10
  s = $\exists z.\exists \overrightarrow{y}.[\hat{\mathcal{T}}^i \wedge binding]$;                               11
  binding* = binding$|_{x \rightarrow x^*, \overrightarrow{y} \rightarrow \overrightarrow{y}^*, \overrightarrow{z} \rightarrow \overrightarrow{z}^*}$;           12
  return $\exists z^*.\exists \overrightarrow{y}^*.[s \wedge binding^*]$ ;                        13
  }                                                                 14
```



(a) call graph (b) invocation graph (c) symbolic invocation graph

Figure 4: Call graph and invocation graph.

4. SYMBOLIC INVOCATION GRAPH

4.1 Invocation Graph

An invocation graph is an expansion of the call graph [14]. Each node in the call graph is expanded into multiple *instances* in the invocation graph such that each node corresponds to a unique path in the call graph. The node in the invocation graph thus identifies a unique calling context. Figure 4 (a) shows the call graph of a program, whose corresponding invocation graph is shown in Figure 4 (b), where each node is labeled by an integer index representing the different instances of the procedure.

Invocation graph is essential for context-sensitive analysis since program state corresponding to each node in the invocation graph has to be computed. Unfortunately, the size of the invocation graph is exponential in relation to the call graph size. Some analysis techniques avoid the explicit construction of the invocation graph, however, the computation of context state still has to be carried out exponential number of times.

We now propose a new representation of the invocation graph whose size can be reduced exponentially. We introduce a new pair of Boolean spaces, $\overrightarrow{w}^*$ (domain) and $\overrightarrow{w}$ (range) to represent the different *instances* of a call graph node in the invocation graph. A node in the invocation graph can therefore be identified by the corresponding call graph node, as well as a minterm in the Boolean space of interest. For example, C_0 in Figure 4 (b) can be identified by C and the minterm W_0, and C_1 can be identified by C and the minterm W_1.

We define a **symbolic invocation graph** to be a call graph where each $\langle i, j\rangle$, representing a call site from procedure i to procedure j, is annotated with a Boolean function $E(i, j)$, representing the set of invocation graph edges associated with $\langle i, j\rangle$. Figure 4 (c) shows the symbolic invocation graph equivalent to Figure 4 (b). For example, the edge $\langle C, D\rangle$ is annotated with $W_0^*W_0 + W_1^*W_1$, meaning that $\langle C, D\rangle$ in the call graph can be refined into $\langle C_0, D_0\rangle$ and $\langle C_1, D_1\rangle$ in the invocation graph. Note that when $E(i, j)$ is represented by BDD, the BDD nodes can be shared among all edges in the call graph. For example, the symbolic invocation edges for $\langle C, D\rangle$, $\langle C, E\rangle$ and $\langle C, F\rangle$ in the example in Figure 4 share a common BDD node since they have exactly the same pattern.

4.2 Symbolic Invocation Graph Construction

We now present our symbolic invocation graph construction algorithm. Without loss of generality, in Algorithm 3 we only show the construction algorithm for an acyclic call graph.

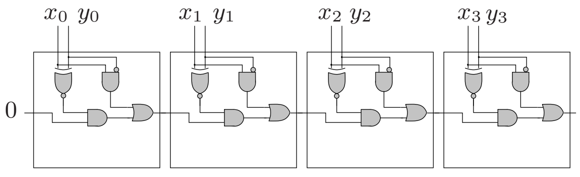
We maintain an instance count for each procedure i. Initially, the instance count of the top procedure is set to 1. We then traverse each procedure in topological order, and process each call graph edge $\langle i, j\rangle$. The symbolic invocation edge $E(i, j)$ is essentially a relation between the set of all instances of i to the set of instances of j originating from i. If we treat each instance as a *number*, then any $\langle u, v\rangle \in E(i, j)$ satisfies two conditions: (a) $u <$ count(i); (b) $u +$ offset $= v$.

Condition (a) can be generalized over any instance count number into a relation $R_<(x, y)$. This relation can be easily pre-constructed using BDD in a way that mimics the construction of the hardware comparator [18] for "less than", as shown in Figure 5 (a). Similarly, condition (b) can be generalized over any offset number into a relation $R_+(x, y, z)$. This relation can be easily pre-constructed using BDD in a way that mimics the hardware adder [18] concatenated with a hardware comparator for equality, as shown in Figure 5 (b). Computing $E(i, j)$ then amounts to plugging in the constant values of instance count and offset into the pre-constructed relations and then finding their conjunction. After a call graph edge is processed, the offset value is updated accordingly. After all call graph edges originating from a procedure are processed, its instance count is updated accordingly.

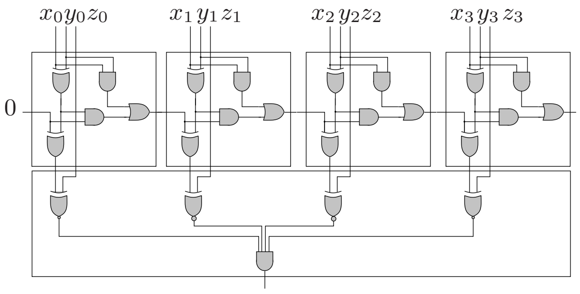
We now show that both the space complexity of symbolic invocation graph representation, and the time complexity of its construction algorithm are polynomial with respect to the number of call graph nodes. It is important to note that while the number of contexts, or the number of call graph node instances, are exponential in relation to $|I|$, the number of BDD variables used to encode the contexts is logarithmic to the number of contexts. Therefore, $|\overrightarrow{w}|$ and $|\overrightarrow{w}^*|$ is of $O(|I|)$. On the other hand, it is well-known that the BDD representations of both the adder and comparator circuits are linear with respect to the number of BDD variables, we can therefore conclude that the size of the generalized relation is $O(|I|)$. Since BDD conjunction is proportional to the size of its operands only, our conclusion follows.

ALGORITHM 3. *Symbolic Invocation Graph Construction.*

```
constructSymbolicInvocationGraph() {
  count(0) = 1;
  forall( $i \neq 0, i \in I$ ) in topological order {
    offset = 0;
    forall( $j \in C^{-1}(i)$ ) {
      $E(i, j)$ = $R_+(\overrightarrow{w}^*$, offset, $\overrightarrow{w}) \wedge R_<(\overrightarrow{w}^*, count(j))$;
      offset = offset + count(j);
    }
    count(i) = offset;
  }
}
```



(a) $R_<(x,y) : x < y$



(b) $R_+(x,y,z) : x + y = z$

Figure 5: Construction of helper symbolic relations.

4.3 Symbolic State Superposition

We now demonstrate that the space efficiency achieved by the symbolic invocation graph representation can be exploited to achieve an exponential reduction of analysis runtime in practice as well. The key idea is to compute the state of an invocation graph node associated with a common procedure *collectively*. Note that this does not mean we will collapse all states together, as is done in context-insensitive analysis. Instead, we compute what we call a **state superposition**, defined as the sum of all invocation graph node states associated with a common procedure multiplied by the corresponding minterms in the instance space $\overrightarrow{w}$. Note that the state of an individual invocation graph node can be retrieved from the state superposition easily by multiplying the corresponding minterm and then abstracting away the instance variables.

EXAMPLE 9. *Consider procedure $alloc$ in Example 1, which contains two invocation graph node instances $alloc_0$ and $alloc_1$, where the formal corresponds to the calling path $main \rightarrow alloc$ and the latter corresponds to the calling path $main \rightarrow getg \rightarrow alloc$. The state for $alloc_0$ is p^*m. The state for $alloc_1$ is g^*m. The state superposition for $alloc$ is $W_0p^*m+W_1g^*m$. The state of $alloc_0$ can be retrieved from the state superposition by $\exists\overrightarrow{w}.[W_0 \wedge (W_0p^*m + W_1g^*m)] = p^*m$.*

The recurrence equations (5), (6) and (7) are modified from equations (1), (2) and (3) to carry out context-sensitive analysis. Note that the procedure of state query and transfer function application remains unchanged, except that they now operate on state superposition.

All modified components concern propagating states, or point-to facts from caller to callee and vise versa. The challenge stems from the fact that in order to be context-sensitive, states need to be *translated* from the instance space of procedure i to a different instance space of procedure j. Such translation can be achieved by exploiting the symbolic invocation edges $E(i,j)$. For example, the modified Equation (5), which is responsible for parameter binding, first mirrors the actual $\Theta^j_{\mathcal{B}(j,i,k)}$ for parameter k into the $\overrightarrow{w}^*$ space, and then multiply it by $E(j,i)$. This way, the state information from caller j will not corrupt the state information of other contexts originating from a different caller of i. The desired state values can be obtained by further abstracting away the $\overrightarrow{w}^*$ variables. It is important to note that $E(i,j)$ may capture thousands of actual invocation edges, therefore the symbolic procedure described above is very efficient. Similarly, in (7), such instance space translation between callers and callees can be computed symbolically. Note that when propagating point-to information from callee to caller, irrelevant information, such as the state of callee formal parameters needed not to be propagated. Such pruning can be computed efficiently using symbolic method [41].

$$\begin{aligned}
\Lambda^i_k &= \sum_{j\in\mathcal{C}^{-1}(i)} \exists\overrightarrow{w}^*.[\Theta^j_{\mathcal{B}(j,i,k)}|_{\overrightarrow{w}\rightarrow\overrightarrow{w}^*} \wedge E(j,i)], &(5)\\
&\quad \forall k \in K, i \in I\\
\Theta^i_k &= \text{query}(S^i, \mathcal{D}(k)), \forall k \in K, i \in I &(6)\\
S^i &= \sum_{j\in\mathcal{C}^{-1}(i)} \exists\overrightarrow{w}^*.(S^j|_{\overrightarrow{w}\rightarrow\overrightarrow{w}^*} \wedge E(j,i)) + &(7)\\
&\quad \sum_{j\in\mathcal{C}(i)} \exists\overrightarrow{w}.[\text{prune}(S^j) \wedge E(i,j)]|_{\overrightarrow{w}^*\rightarrow\overrightarrow{w}} +\\
&\quad \text{apply}(\hat{\mathcal{T}}^i, \overrightarrow{\Lambda}^i, \overrightarrow{\Theta}^i), \forall i \in I
\end{aligned}$$

EXAMPLE 10. *The complete illustration of solving the above equations for Example 1 can be found in Appendix A.*

5. ENGINEERING ISSUES

We have left out several engineering issues in the theoretical discussion earlier. They are nevertheless important factors that contribute to the overall efficiency of the proposed methods. Some issues are common to all pointer analysis frameworks, some are unique to the proposed symbolic analysis framework.

The presence of *recursion* in the program can make the invocation graph infinitely large. We use Tarjan's algorithm to detect nested strongly connected components [37] in the call graph. The acyclic symbolic call graph construction algorithm presented earlier is then applied hierarchically in a bottom up fashion. The presence of *function pointers* prevents the complete pre-construction of the call graph. In our analysis, new call graph edges will be dynamically added as new point-to information related to function pointers are discovered. The affected symbolic edges will also be dynamically constructed.

It is well known that *variable ordering* has a large impact on the size of BDD and dynamic variable reordering is often the strategy of choice in many BDD-based algorithms. As later shown in Section 6, we have found that for pointer analysis, the variable order has a rather small impact on BDD size. As a result, dynamic variable reordering adversely impacts the analysis speed. While we do not perform variable reordering during the analysis, we do apply one important constraint to the variable order. By making the corresponding variables in the domain and range spaces adjacent to each other, we can keep the mirroring operation, which substitutes the range variables in a Boolean function by its domain variables, linear. As shown in Algorithm 1 Line 4, mirroring is a frequent operation.

An extremely important technique that can help speed up the analysis time is the use of *caching*. Caching keeps a hash table that stores the result of a BDD computation. The hash table is keyed by a signature consisting of the type of an BDD computation as well as its operands, which are also BDDs. Thanks to the canonical property of BDD, common BDD computation that shares the same result can be easily identified by the signature and the result can be reused on a large scale using the cache. This efficiency is in essence the same as the dynamic programming principle: if a subproblem can be uniquely identified, it should be solved only once and its result should be shared by other upper-level problems. The use of BDD allows dynamic programming to be applied at a very fine grain level, which is otherwise very hard to identify manually.

Another important technique is *lazy garbage collection*. BDD nodes are often shared by other BDD nodes. When the reference count of a BDD node goes to zero, its memory needs to be reclaimed, or garbage collected. On the other hand, there is a high chance that this BDD node maybe re-created later. We choose to garbage collect a BDD node lazily, that is, only when a threshold value of heap size is exceeded. As shown later in Section 6, this choice has a positive impact on the analysis speed.

We also apply *incremental evaluation* of the recurrence equations, meaning that we only apply the equations on the changes from the previous iterations. This greatly limits the computation involved in the later iterations of the fixed-point computation since the BDD size involved is much smaller. This technique is well known in symbolic reachability analysis and is used in [5] as well.

6. EXPERIMENTAL RESULTS

Our symbolic pointer analysis tool is implemented in C, and makes use of a compiler infrastructure to translate from several frontends (e.g. C, Java, Verilog, etc.), into an intermediate representation (IR). In the *setup pass*, the infrastructure traverses the IR generated by the frontends to produce the call graph (CG) and control flow graph (CFG). Following the setup, an *intraprocedural analysis pass* is performed on all user-defined procedures in the program, iterating over the CFG and creating a flow-insensitive transfer function for each procedure. An *interprocedural pass* is then followed, which performs either a context-insensitive analysis, or context-sensitive analysis. We use Somenzi's publicly available CUDD package [35] for BDD implementation. Our current implementation does not support non-local control transfer (*setjmp/longjmp* calls), location sets [39], and assumes no ill advised use of pointers is made (like random memory accessing via integers). Heap objects are named after the allocation site. Lastly, the C library function's transfer functions are precomputed and applied as necessary.

The goal of our empirical evaluation is three-fold. Our primary goal is to quantify the speed and space efficiency of the proposed symbolic method. Our second goal is to verify if a context-sensitive analysis can provide more precision than context-insensitive analysis. Our third goal is to quantify the BDD-related engineering issues discussed in Section 5.

With the common analysis framework described earlier, we report results on both context-insensitive analysis (Referred to as **CI**) and two types of context-sensitive analysis. Referred to as **CS I**, the first type does not distinguish between call sites in a procedure targeting the same callee. Note that results from [17, 16] are reported with this type of context-sensitivity. Referred to as **CS II**, the second type does make such a distinction, and it was our observation that the size of contexts involved in CS II is significantly larger than CS I.

We perform our evaluation against three benchmark suites: *prolangs* [32], the popular benchmark suite from the pointer analysis community, the integer suite in SPEC2000 [1], and finally MediaBench [25]. The *prolangs* benchmarks were utilized in evaluating the performance of many pointer analysis algorithms, and as such serves

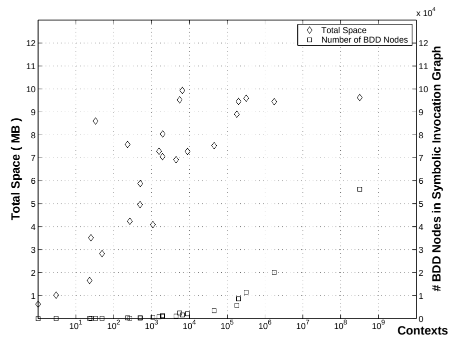


Figure 6: Memory usage versus context count.

as a valid comparison with previous work in this area. The SPEC2000 and MediaBench benchmarks, which are relatively large, are selected to help study the robustness and scalability of our algorithm. The characteristics of the reported benchmarks in this paper are shown in Table 1.

The experiment was performed on a Sun Blade 150 workstation with 550 MHz CPU and 128MB RAM, running on Solaris 8 Operating System. The executable was built using gcc-2.93 with the -O2 option.

6.1 Space Efficiency

We first demonstrate in Figure 6 that the symbolic representation in general, and symbolic invocation graph in particular is efficient in space. Here, the horizontal axis indicates the number of contexts in the evaluated benchmarks in *log* scale. It can be observed that some benchmarks may approach half a billion contexts. In the first plot, we show the total memory usage with respects to the context count. The total memory usage never exceeds 11MB. We also plot the number of BDD nodes used in the symbolic invocation graph. Compared to the corresponding context count, which is the number of invocation graph nodes if an explicit invocation graph representation is used, the BDD node count is exponentially smaller.

6.2 Runtime Efficiency

We now demonstrate the runtime efficiency of the proposed symbolic analysis algorithms. The detailed results on runtime and memory statistics for three types of analysis are given in Table 6.2. Here, the time for the setup pass is referred to as the *Setup Time*. The time it takes the intra-procedural analysis pass to derive all transfer functions is referred to as the *Intra-Time*. The time it takes for the interprocedural analysis pass to reach a fixed point is referred to as the *Inter-Time*.

We draw several observations from the runtime result. First, the runtime of our context-insensitive analysis (CI), based on a loose comparison with [23], is comparable with classical methods such as Anderson's algorithm. Second, the runtime of type 1 context-sensitive analysis (CS I) is very close to its context-insensitive counterpart. Almost all benchmark takes at most twice as much time to execute. Third, the full-context sensitive analysis (CS II), is at most 6 times slower than its context-insensitive counterpart.

Figure 7 offers more insight on the dependency of total analysis time versus context count. Again, the context count is indicated as the horizontal axis in *log* scale. Figure 7 also plots the construction time of symbolic call graph. It is clear that even for benchmark with half a billion contexts, the graph can be constructed in a few seconds.

Table 1: **Benchmark characteristics.**

prolangs				MediaBench				SPEC2000			
name	#lines	#contexts	#blocks	name	#lines	#contexts	#blocks	name	#lines	#contexts	#blocks
315	1411	49	136	gsm	5473	267	1124	bzip2	4665	495	995
TWMC	24032	6522	4613	pegwit	5503	1968	1121	gzip	8218	503	905
simulator	3558	8953	1316	pgp	28065	199551	5265	vpr	16984	179905	4318
larn	9933	1750823	6180	mpeg2dec	9823	44979	2748	crafty	19478	317378	5282
moria	25002	318675286	9446	mpeg2enc	7605	1955	2997	twolf	19756	5538	4231

Table 2: **Analysis runtime and space usage result.**

Benchmarks			Intra time (s)	Inter time (s)	Total time (s)	Memory used (MB)
prolangs	315	CI	0.04	0.03	0.07	1.397
		CS I	0.08	0.08	0.16	1.710
		CS II	0.09	0.12	0.21	2.827
	T-W-MC	CI	9.87	6.56	16.43	8.598
		CS I	10.03	8.39	18.42	8.093
		CS II	13.50	24.91	38.41	9.935
	larn	CI	5.97	16.86	22.83	8.073
		CS I	5.94	22.68	28.62	7.901
		CS II	6.65	88.79	95.44	9.444
	moria	CI	8.19	25.71	33.90	8.369
		CS I	8.20	41.53	49.73	9.790
		CS II	10.09	166.53	176.62	9.622
	simulator	CI	0.93	0.64	1.57	4.161
		CS I	0.93	1.73	2.66	5.595
		CS II	0.96	2.64	3.60	7.279
MediaBench	gsm	CI	0.80	0.20	1.00	2.259
		CS I	0.84	0.40	1.24	3.768
		CS II	0.90	0.55	1.45	4.238
	mpeg2dec	CI	1.96	1.06	3.02	4.503
		CS I	1.92	1.44	3.36	7.696
		CS II	2.38	3.84	6.22	7.532
	mpeg2enc	CI	2.07	0.60	2.67	4.413
		CS I	2.01	1.03	3.04	6.599
		CS II	2.94	3.87	6.81	7.048
	pegwit	CI	0.76	0.41	1.17	3.565
		CS I	0.78	1.27	2.05	5.589
		CS II	0.84	2.41	3.25	8.038
	pgp	CI	4.83	7.87	12.70	6.918
		CS I	4.92	15.52	20.44	7.697
		CS II	5.79	50.92	56.71	9.454
SPEC2000	bzip2	CI	0.64	0.20	0.84	3.279
		CS I	0.65	0.37	1.02	3.834
		CS II	0.70	0.70	1.40	4.962
	crafty	CI	4.97	3.25	8.22	5.551
		CS I	4.91	4.92	9.83	8.048
		CS II	6.48	26.41	32.89	9.594
	gzip	CI	0.74	0.19	0.93	3.496
		CS I	0.78	0.36	1.14	4.072
		CS II	0.89	1.14	2.03	5.880
	twolf	CI	10.83	3.59	14.42	8.503
		CS I	10.86	5.77	16.63	7.886
		CS II	12.87	13.56	26.43	9.525
	vpr	CI	5.21	2.50	7.71	5.339
		CS I	5.05	6.97	12.02	7.568
		CS II	5.80	14.49	20.29	8.899

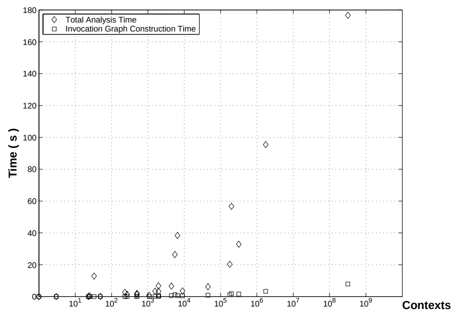


Figure 7: **Algorithm runtime versus context count.**

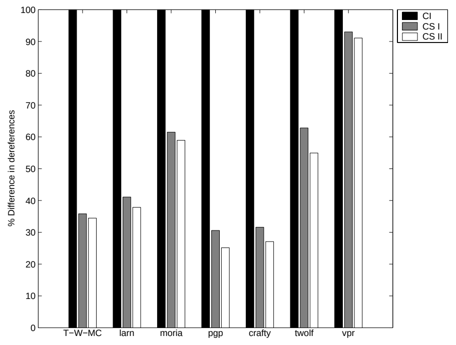


Figure 8: **Precision result.**

6.3 Precision

Many studies have been performed on the impact of context sensitivity on analysis precision [31, 17]. Since this study focuses on the runtime of symbolic analysis, other analysis dimensions, such as field sensitivity, heap naming scheme, which could significantly affect the analysis precision, are not included. Our reported results should therefore be taken as a confirmation that context-sensitivity does help improve analysis precision for some benchmarks rather than basis for a quantitative conclusion. We use the popular metric of average dereference size, defined as the average size of a point-to set for each memory load or store in the program. The dereference sizes for all three types of analysis are plotted for comparison. As in [17], we normalize the metric to the context-insensitive analysis result. It can be observed that while large improvement can sometime result with context-sensitive analysis, the difference between the two types of context-sensitive analysis is usually minor.

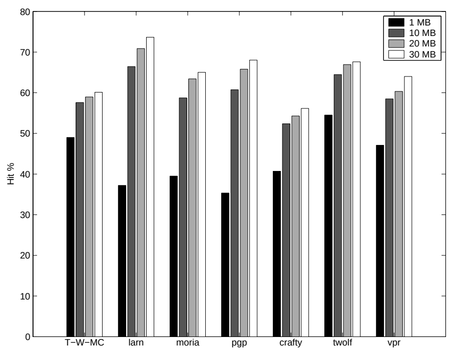


Figure 9: Cache hit rate.

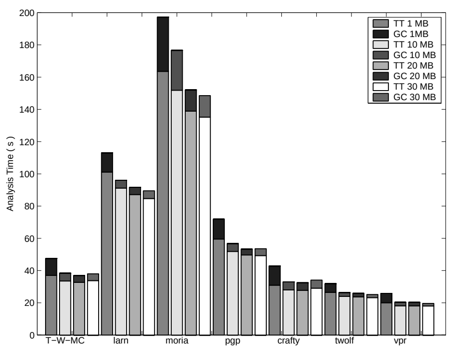


Figure 10: Time spent on garbage collection.

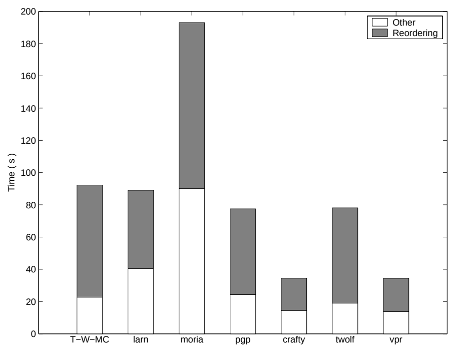


Figure 11: Time spent on variable reordering.

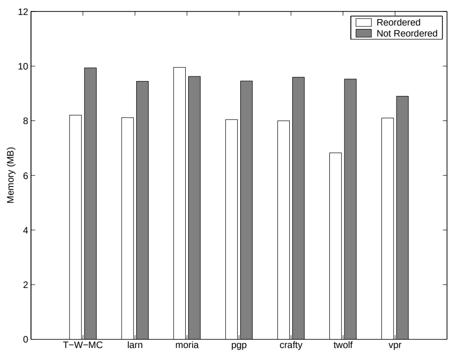


Figure 12: Memory size with/without variable reordering.

6.4 Impact of Caching

As discussed earlier, the cache is used to store the results of basic BDD operations like AND, OR, and many others. As such, a higher hit rate will translate into improved performance, since a successful cache lookup requires fewer computations. The cache hit rate usually ranges from 40% to 60%. We also observe a lower cache hit rate in the context-sensitive analysis. This can be explained by the higher memory consumption in context-sensitive analysis, which forces the BDD manager to evict nodes out of the cache.

It is obvious that the size of cache may impact the cache hit rate. Figure 9 shows the cache hit rate of selected large benchmarks under different cache size configurations. It can be observed that a large cache size in general lead to a higher hit rate. On the other hand, up to certain limit, increasing the cache size does not increase the hit rate.

6.5 Impact of Lazy Garbage Collection

To see how lazy garbage collection can affect analysis speed, we demonstrate the time spent on garbage collection versus other processing time for selected benchmarks under different threshold heap size values. It can be observed that in general a larger heap size will reduce the amount of time spent on garbage collection and therefore the overall analysis speed. On the other hand, there is almost nothing to gain if the threshold is increased beyond a certain value.

6.6 Impact of Variable Reordering

Variable reordering was attempted in order to see what improvements might have in terms of space and runtime. Sifting, commonly regarded as the best reordering algorithm, was used to dynamically reorder the BDD variables in the program. However, as can be seen from Figure 11 and Figure 12, the results are rather negative. The analysis takes longer to complete, with most of the time spent on reordering the variables. Furthermore, even if we discount the time spent on reordering the variables, only minor improvements are obtained in terms of runtime. In terms of space efficiency, no major improvements are obtained with few exceptions. These results seem to show that variable reordering, as is done by the CUDD package, offers no improvement over our static variable ordering.

7. RELATED WORK

Due to its importance, pointer analysis has been actively investigated for the past two decades. Hind gave an excellent survey on the state-of-the-art in the field [20]. According to Hind, over seventy-five papers and nine PhD thesis was published on the subject by the time [20] was published.

In the category of FICI pointer analysis, Steensgaard's work [36] stands out as the first equality-based method, which treats assignment as bidirectional and uses a union-find data structure. His analyzer is extremely fast and has analyzed million lines of industrial code. However, the precision of equality-based approach degrades very fast in general, even with later improvement [40]. Andersen's [3] popular subset-based improves precision by treating assignment as a unidirectional flow of values. However, Andersen's algorithm has a cubic runtime.

While there are many variations and improvements of Steensgaard's and Anderson's algorithm, a major advancement of analysis efficiency attributes to the use of a constraint-based solver [15, 30]. With this formulation, the point-to information is evaluated lazily – instead of modeling the state of a block as the immediate successors of the corresponding node in the point-to graph, it is modeled as all nodes reachable from the corresponding node. Point-to information of different blocks can therefore be shared via common path. As a result, the state-of-the-art implementation of this method for FICI analysis can scale to million-line code [19].

A definition of context-sensitivity has been given in [34]. The most popularly used form of context-sensitivity is the concept of call string, defined as a path of call sites on the call stack. A new form, called object-sensitivity has been proposed for object oriented programs [28]. Landi et al. [24] first performed context-sensitive pointer analysis by the use of inter-procedural control flow graph, which can be prohibitively large. Emami et al. [14] introduced the use of invocation graph, where control flow graph among different invocations of the same procedure can be shared. Wilson and Lam [39] also used invocation graph. In addition, the concept of partial transfer function was proposed in an effort to reduce the number of times a procedure has to be re-analyzed. Chatterjee et al. [10] proposed the use of summary functions to completely capture the transfer function of a procedure. A further development with the same strategy was proposed by Cheng and Hwu [11], with the addition of using access path originating from parameters and globals as transfer function parameters.

Context-sensitive analysis algorithms suffer from an exponential runtime in general. Several efforts target towards analysis scalability. One direction is to use partial sensitivity. For example, Recently Liang and Harrold [27]'s analyzer treats globals in a context-insensitive way and is able to analyze industrial programs. Another direction is to exploit the efficiency of constraint-based solver, which is extremely successful in context-insensitive analysis. Fähndrich et al. reported a polymorphic (context-sensitive) analyzer (equivalent to CS I) with a cubic runtime [16].

Even though the concept of BDD appeared much earlier [2], it was Bryant's ROBDD [6], designed to be compact and canonical, makes it successful. It was applied to a wide range of tasks, including simulation, synthesis and formal verification in the CAD community. McMillan et al. [9] and O. Coudert et al. [12] were the first to introduce BDD into the model checking of sequential circuits, which can be abstracted as finite state machines. Their pioneer work replaces the explicit state enumeration by implicit state enumeration using BDDs. This key concept, complemented by further improvements [7, 13, 8, 29], was responsible for the first application of model checking to practical problems.

Other efforts in using a Boolean framework for program analysis can be found in areas such as shape analysis [33] and predicate abstraction [4]. However, the number of Boolean variables introduced in these frameworks is proportional to the number of subjects of interest. The application of BDD technique to pointer analysis problem was first reported in [41], where memory blocks are logarithmically encoded into the Boolean domain. The concept of symbolic transfer function and the use of BDD image computation to perform program state query was proposed and its speed efficiency was demonstrated. Berndl et al. reported a context-insensitive pointer analysis algorithm using BDD in [5], where the space efficiency, and therefore better scalability than the classical methods for analyzing Java programs was demonstrated. The interest in exploiting BDD for program analysis seems to be growing: in the same proceeding Lhoták and Hendren [26] built a relational database abstraction on top of the low-level BDD manipulation to facilitate program analysis. Whaely and Lam [38] reported another method for context-sensitive analysis using BDD.

8. CONCLUSION

In this paper, we present a new formalism for pointer analysis. Based on Boolean algebra, this formalism is simple enough to be summarized in three recurrence equations. In addition, it enables the use of Binary Decision Diagram to achieve both space and speed efficiency. A common framework is established to perform both context-insensitive and context-sensitive analysis.

Based on our study, we conclude that the key concepts proposed in this paper, namely symbolic transfer function and symbolic invocation graph, can effectively reduce the runtime of the otherwise expensive context-sensitive analysis to one comparable to its context-insensitive counterpart.

In the future, we plan to leverage and extend our symbolic framework for other important issues, including flow sensitivity, distinction of record fields and array elements, as well as the generalization of the symbolic framework to other program analysis problems.

9. ACKNOWLEDGMENT

We would like to sincerely thank the anonymous reviewers for their constructive comments for the draft of this paper. We would also like to thank Dr. Hind for pointing us to the relevant benchmarks in this area.

10. REFERENCES

[1] SPEC CPU2000 benchmarks. `http://www.specbench.org/cpu2000/`.

[2] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computer*, C-27(6):509–516, June 1978.

[3] O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, 1994.

[4] T. Ball and T. Millstein. Polymorphic predicate abstraction. Technical Report MSR-TR-2001-10, Microsoft Research, June 24, 2003.

[5] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Point-to analysis using BDD. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, June 2003.

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

[7] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, Edinburgh, Scotland, 1991.

[8] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, (13), 1994.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, Washington, DC, 1990.

[10] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *Proceedings of Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[11] B.-C. Cheng and W.-M. W. Hwu. Modular interprocedural pointer analysis using access paths: Design implementation and evaluation. In *Proceedings of SIGPLAN Conference on*

Programming Language Design and Implementation, pages 57–69, Vancouver, British Columbia, Canada, June 2000.

[12] O. Coudert, C. Berthet, and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 126–129, November 1990.

[13] O. Coudert and J. C. Madre. Symbolic computation of the valid states of a sequential machine: Algorithms and discussion. In *ACM Workshop on Formal Methods in VLSI Design*, 1991.

[14] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[15] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. *ACM SIGPLAN Notices*, 33(5):85–96, 1998.

[16] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–263, Vancouver, British Columbia, Canada, June 2000.

[17] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of Static Analysis Symposium*, pages 175–198, June 2000.

[18] D. Gajski. *Principles of Digital Design*. Prentice Hall, 1997.

[19] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[20] M. Hind. Pointer analysis: Haven't we solved this problem yet. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, June 2001.

[21] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[22] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Proceedings of Static Analysis Symposium*, pages 57–81, 1998.

[23] M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[24] W. Landi and B. Ryder. A safe approximate algorithm for inter-procedural pointer aliasing. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[25] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Micro 30*, 1997.

[26] O. Lhoták and L. Hendren. Jedd: A BDD-based relational extension of Java. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[27] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Proceedings of Static Analysis Symposium*, pages 279–298, 2001.

[28] A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analysis for Java. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 1–12, 2000.

[29] I.-H. Moon, J. H. Kukula, K. Ravi, and F. Somenzi. To split or to conjoin: the question in image computation. In *Design Automation Conference*, pages 23–28, 2000.

[30] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. *ACM SIGPLAN Notices*, 35(5):47–56, 2000.

[31] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–22, La Jolla, California, June 1995.

[32] B. Ryder. Prolangs analysis framework. `http://www.prolangs.rutgers.edu`.

[33] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[34] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[35] F. Somenzi. CUDD: Binary decision diagram package release. `http://vlsi.Colorado.EDU/~fabio/CUDD/cuddIntro.html`, 1998.

[36] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[37] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[38] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[39] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[40] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Foundations of Software Engineering*, pages 81–92, 1996.

[41] J. Zhu. Symbolic pointer analysis. In *Proceedings of the International Conference in Computer Aided Design*, San Jose, November 2002.

APPENDIX

A. SOLUTION PROCESS ILLUSTRATION

Here we show the complete fixed-point iteration process of Example 1 for solving recurrence equations (1), (2) (3) for context-insensitive analysis; and recurrence equations (5), (6), (7) for context-sensitive analysis. For the economy of space, those values that are unchanged during the iterations are listed separately in the row marked as "Unchanged values". For fast convergence, the procedure states are evaluated in a bottom-up fashion along the call graph. For presentation clarity, the augmented transfer functions are not used. In addition the pruning process is applied for both analysis and therefore formal parameter states are not propagated to callers.

$\mathcal{T}^0 = \mathcal{T}^{main} = g^*a$	$\mathcal{C}(0) = \{1,2\}$	$E(0,1) = W_0^*W_0$	$\mathcal{B}(0,1,1) = 4$
$\mathcal{T}^1 = \mathcal{T}^{getg} = r^*\lambda_1 + \theta_1^*\theta_2 + t^*g$	$\mathcal{C}(1) = \{2\}$	$E(0,2) = W_0^*W_0$	$\mathcal{B}(0,2,0) = 3$
$\mathcal{T}^2 = \mathcal{T}^{alloc} = f^*\lambda_0 + \theta_0^*m$	$\mathcal{C}(2) = \emptyset$	$E(1,2) = W_0^*W_1$	$\mathcal{B}(1,2,0) = 5$

Iteration	Context-insensitive	Context-sensitive
Initial values	$\begin{aligned} S^0 &= S^1 = S^2 = g^*a + t^*g \\ \Theta_0 &= \Theta_1 = 0 \\ \Theta_2 &= \text{query}(S^1, \langle t, 2\rangle) = a \end{aligned}$	$\begin{aligned} S^0 &= S^1 = S^2 = g^*a + t^*g \\ \Theta_0 &= \Theta_1 = 0 \\ \Theta_2 &= \text{query}(S^1, \langle t, 2\rangle) = a \end{aligned}$
Unchanged values	$\begin{aligned} \Theta_3 &= \text{query}(S^0, \langle p, 0\rangle) = p \\ \Theta_4 &= \text{query}(S^0, \langle q, 0\rangle) = q \\ \Theta_5 &= \text{query}(S^1, \langle t, 1\rangle) = g \\ \Lambda_0 &= \Theta_{\mathcal{B}(0,2,0)} + \Theta_{\mathcal{B}(1,2,0)} = p + g \\ \Lambda_1 &= \Theta_{\mathcal{B}(0,1,1)} = q \end{aligned}$	$\begin{aligned} \Theta_3 &= \text{query}(S^0, \langle p, 0\rangle) = p \\ \Theta_4 &= \text{query}(S^0, \langle q, 0\rangle) = q \\ \Theta_5 &= \text{query}(S^1, \langle t, 1\rangle) = g \\ \Lambda_0 &= W_0p + W_1g \\ \Lambda_1 &= W_0q \end{aligned}$
1	$\begin{aligned} S^2 &= g^*a + t^*g + f^*p + f^*g \\ S^1 &= g^*a + t^*g + r^*q \\ S^0 &= g^*a + t^*g \\ \Theta_0 &= p + g \\ \Theta_1 &= q \\ \Theta_2 &= a \end{aligned}$	$\begin{aligned} S^2 &= g^*a + t^*g + W_0f^*p + W_1f^*g \\ S^1 &= g^*a + t^*g + W_0r^*q \\ S^0 &= g^*a + t^*g \\ \Theta_0 &= W_0p + W_1g \\ \Theta_1 &= W_0q \\ \Theta_2 &= a \end{aligned}$
2	$\begin{aligned} S^2 &= g^*a + t^*g + f^*p + f^*g \\ &+ p^*m + g^*m \\ S^1 &= g^*a + t^*g + r^*q \\ &+ p^*m + g^*m + q^*a \\ S^0 &= g^*a + t^*g \\ &+ p^*m + g^*m + q^*a \\ \Theta_0 &= p + g \\ \Theta_1 &= q \\ \Theta_2 &= a + m \end{aligned}$	$\begin{aligned} S^2 &= g^*a + t^*g + W_0f^*p + W_1f^*g \\ &+ W_0p^*m + W_1g^*m \\ S^1 &= g^*a + t^*g + W_0r^*q \\ &+ W_0g^*m + W_0q^*a \\ S^0 &= g^*a + t^*g \\ &+ W_0p^*m + W_0g^*m + W_0q^*a \\ \Theta_0 &= W_0p + W_1g \\ \Theta_1 &= W_0q \\ \Theta_2 &= a + W_0m \end{aligned}$
3	$\begin{aligned} S^2 &= g^*a + t^*g + f^*p + f^*g \\ &+ p^*m + g^*m + q^*a \\ S^1 &= g^*a + t^*g + r^*q \\ &+ p^*m + g^*m + q^*a + q^*m \\ S^0 &= g^*a + t^*g \\ &+ p^*m + g^*m + q^*a + q^*m \\ \Theta_0 &= W_0p + W_1g \\ \Theta_1 &= W_0q \\ \Theta_2 &= a + W_0m \end{aligned}$	$\begin{aligned} S^2 &= g^*a + t^*g + W_0f^*p + W_1f^*g \\ &+ W_0p^*m + W_1g^*m + W_1q^*a \\ S^1 &= g^*a + t^*g + W_0r^*q + W_0g^*m \\ &+ W_0q^*a + W_0q^*m \\ S^0 &= g^*a + t^*g + W_0p^*m + W_0g^*m \\ &+ W_0q^*a + W_0q^*m \\ \Theta_0 &= W_0p + W_1g \\ \Theta_1 &= W_0q \\ \Theta_2 &= a + W_0m \end{aligned}$
4	$\begin{aligned} S^2 &= g^*a + t^*g + f^*p + f^*g \\ &+ p^*m + g^*m + q^*a + q^*m \\ S^1 &= g^*a + t^*g + r^*q \\ &+ p^*m + g^*m + q^*a + q^*m \\ S^0 &= g^*a + t^*g \\ &+ p^*m + g^*m + q^*a + q^*m \\ \Theta_0 &= p + g \\ \Theta_1 &= q \\ \Theta_2 &= a + m \end{aligned}$	$\begin{aligned} S^2 &= g^*a + t^*g + W_0f^*p + W_1f^*g \\ &+ W_0p^*m + W_1g^*m + W_1q^*a \\ &+ W_1q^*m \\ S^1 &= g^*a + t^*g + W_0r^*q + W_0g^*m \\ &+ W_0q^*a + W_0q^*m \\ S^0 &= g^*a + t^*g + W_0p^*m + W_0g^*m \\ &+ W_0q^*a + W_0q^*m \\ \Theta_0 &= W_0p + W_1g \\ \Theta_1 &= W_0q \\ \Theta_2 &= a + W_0m \end{aligned}$