

A PRECISE SAT-BASED POINTER ANALYSIS FRAMEWORK  
FOR THE C LANGUAGE

A THESIS  
SUBMITTED TO THE COMPUTER SCIENCE DEPARTMENT  
OF STANFORD UNIVERSITY  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
COMPUTER SCIENCE WITH HONORS

Thomas Dillig  
May 2006

© Copyright by Thomas Dillig 2006  
All Rights Reserved

I certify that I have read this thesis and that, in my opinion, it is fully adequate in scope and quality as an undergraduate honors thesis.

---

Alex Aiken Principal Adviser

Approved for the University Committee on Undergraduate Studies.

# Acknowledgments

I would like to thank my adviser Alex Aiken, Yichen Xie, and all the members of the SATURN project group for their valuable contributions to this this work. In addition, I would like to thank Mary McDevitt from the Technical Communications Program for proof-reading this thesis.

# Contents

<b>Acknowledgments</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Overview and Motivation</b>	<b>2</b>
<b>3 Intra-procedural Analysis</b>	<b>6</b>
3.1 Defining an abstract name space . . . . .	6
3.2 Memory as Abstract Cells . . . . .	7
3.2.1 Properties of abstract memory cells . . . . .	9
3.2.2 Guarded Cell Sets . . . . .	11
3.3 Deriving guarded cell sets . . . . .	15
3.3.1 Types of memory cells . . . . .	15
3.3.2 Variables as guarded cell sets . . . . .	17
3.3.3 Initialization . . . . .	20
3.3.4 Deriving guarded cell sets from expressions . . . . .	20
3.3.5 Assigning a guarded cell set . . . . .	23
3.4 Analyzing a function body . . . . .	24
<b>4 Inter-procedural analysis</b>	<b>25</b>
4.1 Function summaries . . . . .	25
4.2 Using function summaries . . . . .	28
<b>Bibliography</b>	<b>29</b>

# Chapter 1

## Introduction

With the incredible pervasiveness of computers in almost all aspects of our life, software failures are an increasingly harmful, and potentially dangerous problem. As software increases in sophistication and complexity, finding and removing bugs has become harder than ever before and the “classical” techniques are increasingly insufficient to guarantee even a reasonable level of stability in software. Although extensive testing and dynamic analysis techniques help find many errors, they are very expensive and cannot achieve complete coverage of the state space of any program.

The idea of symbolic program execution to verify program properties has been pursued for a long time, but until recently most of this work was limited to relatively small and simple finite-state systems such as hardware or network protocols. Over the last few years, an increasing number of research groups have decided to focus on “bug finding” using static analysis [14, 12, 13, 3, 15]. In order to analyze programs accurately for errors, it is very helpful to have a *memory model*, i.e., a symbolic description of the memory contents at every program point. In addition to a memory model, many bug finding tools also benefit from an *alias analysis* that can identify aliased memory locations at function exit. This thesis develops a memory model and alias analysis that allows for an efficient and accurate modeling of memory of C programs that can be utilized by bug finding tools, such as null dereference or memory leak analyses.

# Chapter 2

## Overview and Motivation

To accurately identify sources of memory errors in C, we must model both the structure and content of memory. More specifically, we need the ability to identify memory locations accessible through more than one variable (i.e. *aliased locations*) and to retrieve any points-to relationships of interest to client analyses. While successful bug finders have been built without modeling memory this accurately, the ubiquitous use of both aliasing and pointers in real C programs makes bug finding tools built on an accurate memory model much more precise. Consider, for example, the following code fragment:

```
void null_error (int *a, int *b)
{
(1)  a = NULL;
(2)  b = a;
(3)  *b = 6
}
```

To detect the NULL dereference in function `null_error` the memory model must capture that `b` aliases `a` in line (3). While this example is contrived, real application code shows similar patterns. Furthermore, aliasing tends to confuse human programmers as well, leading to an increase in bug frequency when aliasing is used [4].

Furthermore, any successful pointer analysis must model fields of structs and nested pointers accurately without giving up on soundness and precision. C code is littered with structs, often with recursive fields, and nested pointer dereferences. For example, consider the code below:

```
void potential_leak (struct state *a)
{
    a->next = malloc (sizeof (struct state));
    struct state *temp = a->next;
    ...
    free (temp);
}
```

To detect that `potential_leak` does not in fact leak memory, our memory model must track that `temp` is aliasing the field `next` of `a`. While the example presented above is very straightforward, such nesting can (and does) occur to any depth in real programs.

Several different approaches have been presented over the last several years to automatically extract aliasing information and points-to relationships from C programs. We first define *may-alias*( $a, b$ ) for two expressions `a` and `b` iff  $a=b$  under some execution path in the program. The first approach focuses on inferring a sound set of these may-alias relationships using a *field-insensitive* (or store-based) analysis[1, 9, 10]. In this approach, all fields within one struct as well as all elements of an array are collapsed into one abstract location. While this approach preserves soundness, it leads to a significant loss of precision in the analysis that makes it inadequate for many static bug finders, because it leads to a large number of false alarms. Furthermore, by collapsing all fields of structures and nested pointers, this method also introduces type mismatches, potentially violating useful and important invariants assumed by the client analysis. For example, a store of 7 into the struct field `num` (`s->next->num = 7`) of type `int` stores an integer in an abstract location modeling a pointer to a

struct.

A second, widely-used approach uses *k-limiting* to limit the depth of any recursive data structure [5, 6]. While this mitigates the loss of precision for the first few fields of structures and pointers, it still suffers from the same limitations as store-based techniques once the *k*th element in a pointer chain is reached. Furthermore, memory models based on *k-limiting* are very parameter sensitive: A small change in the source code may cause an important field of a struct to fall below the *k-horizon* and cause many spurious alias relationships to be inferred by the tool. While techniques have been proposed to circumvent these limitations [2, 8, 11], none of these techniques have scaled to millions of lines of source code. Also, all afore-mentioned techniques are *path-insensitive*. While this level of precision is sufficient for inferring may-alias relationships, it unfortunately yields unacceptably high false positive rates when employed in static bug finding on very large applications. Almost all properties of interest tend to be path sensitive in static bug finding and a path-insensitive approximation performs noticeably poorer when analyzing very large systems. Consider the following example:

```
void foo (int *a, int *b, int flag)
{
(1)  if(flag) a = b = malloc (sizeof (int));
(2)  ...
(3)  if(flag) free(b);
}
```

A flow-sensitive, but path-insensitive analysis would conclude that in `foo()` it is possible to leak memory on the path calling `malloc()` in line (1) and not calling `free()` in line (3). Path-sensitivity is important for eliminating false positives, and any accurate memory model needs therefore be path-sensitive if it is not to report large numbers of false positives. The approach presented in this thesis models memory fully path- and field-sensitively and allows for an arbitrary depth on all modeled pointer

and field accesses. All information collected by our analysis is precise. However, unlike shape analysis we do not attempt to infer general invariants; we lazily only model accessed memory locations.

In principle, there are two different techniques for analyzing source programs. The first approach is to inline all function calls and analyze the resulting aggregate function. This approach is relatively simple to implement, but fails in the presence of recursion and does not scale to large complex code bases. The second approach attempts to summarize relevant information of program parts and reuses these summaries whenever a program part is reached again in the analysis. The natural abstraction boundaries used by our analysis are functions; every function is analyzed by itself and its summary is used when analyzing all its callers.

Computing function summaries allows us then to be fully *context-sensitive* when analyzing source programs without having to reanalyze functions in every possible calling context. This both preserves precision and allows our analysis to scale. The most obvious drawback of this approach is that most call graphs cannot be topologically sorted, making it impossible to have summaries for all callees of a function precomputed when analyzing a function. While this issue is relevant and a real concern, it can be solved by computing fix-points; however, we do not implement fix-point computation in our framework.

# Chapter 3

## Intra-procedural Analysis

In this section, we discuss how we model memory within one function. Section 4 extends the approach presented here to function summaries and inter-procedural analysis. Our goal is to model memory fully path-sensitively and to be demand-driven. i.e., to only create representations for memory objects lazily when they are used.

### 3.1 Defining an abstract name space

In order to refer to memory locations used within one function with a canonical name, we introduce a name space mapping every concrete memory location to an abstract name. For this, we first define the *root* of a location as follows:

$$\begin{aligned} \text{root} := & \text{Const}(c) \\ & | \text{Local}(\text{name}) \\ & | \text{Param}(\text{number}) \\ & | \text{Global}(\text{name}) \\ & | \text{Unknown} \end{aligned}$$

In words, every reachable memory location can either originate from a constant with value  $c$ , a local variable, a parameter, or a global. In addition, the root of a memory location may be *Unknown* if that memory location is accessed through

operations that we do not model precisely, such as pointer arithmetic. We define an *interface object* as a root whose originating from outside of the current calling context.

$$interface\_object \in \{Param(number), Global(name)\}$$

The relation of every abstract memory location to its root is described by its *access path*. An access path is defined as follows:

$$\begin{aligned} access\_path := & \text{root} \\ & | Deref(access\_path) \\ & | AddrOf(access\_path) \\ & | FieldOf(access\_path, name) \end{aligned}$$

Every distinct memory location can now be described by a unique access path<sup>1</sup>, encoding the origin of this location from any of the possible roots.

We define:

$$\rho(access\_path) := \text{root of } access\_path$$

In other words, the  $\rho$  operator retrieves the origin of any root access path.

## 3.2 Memory as Abstract Cells

As hinted above, when analyzing a function our framework maps the (potentially) infinite set of concrete memory cells to a finite set of *abstract memory cells*. This mapping inherently causes an unavoidable loss of precision. An abstract memory cell represents the set of concrete locations that can be accessed through one access path. To make this mapping precise, we first give the definition of *location\_paths* for a subset of the C language which we consider the concrete domain in this chapter.

---

<sup>1</sup>This assumes no entry-aliasing when the current function is called

This definition is only missing function calls and various compiler extensions <sup>2</sup>; the treatment of function calls is detailed in Chapter 4 and any compiler extensions are currently unsoundly ignored in our implementation.

$$\begin{aligned} \text{location\_path}_c := & \text{root}_c \\ & | \text{Deref}_c(\text{location\_path}, \text{offset}) \\ & | \text{AddrOf}_c(\text{location\_path}) \\ & | \text{FieldOf}(\text{location\_path}, \text{name}) \end{aligned}$$

$$\begin{aligned} \text{root}_c := & \text{Const}_c(c) \\ & | \text{Local}_c(\text{name}) \\ & | \text{Param}_c(\text{number}) \\ & | \text{Global}_c(\text{name}) \end{aligned}$$

The only difference between the set of abstract access paths and concrete location paths is that location paths include pointer offsets such as arrays or pointer arithmetic while in our abstract domain all pointer offsets, i.e. array fields and pointer arithmetic, are collapsed into one cell. We also define the concrete location  $l$  for any expression as:

$$\lambda(\text{exp}, \sigma) = \{l \mid \text{exp can evaluate to } l \text{ at program point } \sigma\}$$

We define the abstraction function for a location path  $l$  as follows:

$$\alpha(l) = a(\text{car}(l)).\alpha(\text{cdr}(l))$$

With  $a()$  defined as follows:

---

<sup>2</sup>Such as GCC's `asm { }` statements

$$\begin{aligned}
a(\mathit{root}_c) &= \mathit{root} \\
a(\mathit{Deref}_c(\mathit{path}, \mathit{offset})) &= \mathit{Deref}(\mathit{path}) \\
a(\mathit{AddrOf}_c(\mathit{path})) &= \mathit{AddrOf}(\mathit{path}) \\
a(\mathit{FieldOf}_c(\mathit{path}, \mathit{name})) &= \mathit{FieldOf}(\mathit{path}, \mathit{name})
\end{aligned}$$

In the definition presented above, the case for  $\mathit{Deref}$  bears some elaboration. The offset in our definition of  $\mathit{Deref}$  is used to describe pointer arithmetic, such as accessing array elements. In converting concrete memory locations to abstract memory locations, our abstraction function ignores pointer arithmetic and collapses all concrete locations accessed through `var+offset` into one abstract location for `var`.

We can now define the concretization function  $\gamma$ :

$$\gamma(\mathit{access\_path}) = \{l \mid \alpha(l) = \mathit{access\_path}\}$$

Even though we have defined  $\alpha$  and  $\gamma$  in terms of one memory location, our definitions extend naturally to the concrete and abstract set of all memory locations at every execution step of a program. Also, our abstraction (considering the subset of C described above) is a sound over-approximation of the concrete domain.

### 3.2.1 Properties of abstract memory cells

In our abstract domain, we guarantee two invariants that always hold for any abstract memory cell:

<i>Invariant 1:</i> Every abstract memory cell is immutable.
<i>Invariant 2:</i> Every abstract memory cell is uniquely named by one access path.

In other words, every abstract memory location can never change its value and is always uniquely identified by its origin. It is important to note that constants are treated simply as another abstract memory location with the access path  $\mathit{Root}(\mathit{Const}(c))$ .

Every concrete program expression  $e$  can be mapped to a set of abstract memory cells; or equivalently, a set of access paths by first mapping  $e$  to a set of concrete locations and then abstracting those. We define:

$$\pi(e) = \alpha(\lambda(e))$$

In Chapter 3.3.4 we present an algorithm capable of computing this function.

**Example:**

```
void memory (int *a, struct state *s)
{
(1)
    s->num = *a;
(2)
    *a = 6;
(3)
    int b = s->data;
}
```

At point (1), the two abstract memory locations introduced are:

$$\pi(a) = \{Root(Param(0))\}$$

$$\pi(s) = \{Root(Param(1))\}$$

At position (2), we now have:

$$\pi(s) = \{Root(Param(1))\}$$

$$\pi(s \rightarrow num) = \{Deref(Root(Param(0)))\}$$

At (3) we added:

$$\pi(*a) = \{Root(Const(6))\}$$

At (4) we have:

$$\pi(b) = \{FieldOf(Deref(Root(Param(1))), data)\}$$

It is important to understand that while any expression may be assigned to any other value, the abstract value described by any access path is immutable. In the example presented above, even though at position (3), the abstract location that the expression `*a` was equal to at (2) still exists, it is not reachable from any program expression after position (2). The immutability of abstract memory cells is a central property of our approach.

Also, every abstract memory cell is uniquely described by an access path. This very useful invariant allows us to use access path and abstract memory cell interchangeably in the following chapters and follows directly from the immutability of abstract locations discussed above.

### 3.2.2 Guarded Cell Sets

To facilitate a concise presentation we have until now avoided programs involving conditionals. To deal with conditions path-sensitively, we build on the SATURN framework, which provides boolean statement guards for every program point.

A *statement guard* is the guard under which a statement in the source program is executed. Statement guards are obtained by a forward analysis of a function's control flow graph. We say a *split* occurs in a control flow graph if control goes from a program point to two (or more) points under different conditions. Similarly, we say a *merge* occurs if control goes from two (or more) different program points to one program point. In terms of the statement guard, every split in the CFG ANDs an additional condition to the set of conditionals; every merge in the CFG ORs two (or more) sets

of conditionals. We represent statement guards as boolean formulas. We achieve full precision by modeling each location as a vector of boolean variables, one for every bit of the location. For example, a C integer (on the x86-32 bit architecture) is modeled as a vector of 32 boolean variables. Most primitives such as addition, subtraction, bit-shift, etc. can be encoded precisely by a boolean formula. This technique is fully described in [14, 12, 13].

In our discussion, we assume the following two invariants about guards in split/merge operations:

<i>Invariant 1 (disjoint split):</i>	At every split, for the outgoing guards $G_1, \dots, G_n$ : $\forall i, j \ i \neq j (G_i \wedge G_j = \text{false})$
<i>Invariant 2 (complete merge):</i>	At every merge, for all incoming guards $G_1, \dots, G_n$ and the outgoing guard $G$ the following holds: $\forall_{G_i} G_i = G$

To achieve path-sensitivity, we define in our abstract domain:

<i>Definition:</i>	A <i>guarded cell set</i> of an expression $e$ is a set of pairs of the form $(\text{access\_path}, \text{guard})$ encoding which access path $e$ evaluates to under which guard.
--------------------	---

We define  $\Pi()$  as a natural extension of  $\pi()$  defined earlier:

$$\Pi(e) = \{(\alpha(\lambda(e)), g) \mid e \text{ evaluates to } \alpha(\lambda(e)) \text{ under guard } g\}$$

In this way we encode a precise conditional mapping from a concrete program expression to a set of abstract memory locations uniquely identified by access paths. We also define:

$$\Pi^{-1}(g) = \{(\alpha(\lambda(e)), g) \mid e \text{ evaluates to } \alpha(\lambda(e)) \text{ under guard } g\}$$

We maintain the following invariants about any guarded cell set:

- (1)  $(\forall(p_i, g_i) \in \Pi(e)) \Rightarrow (\bigvee_j g_j = true)$
- (2)  $(\forall(p_i, g_i) \in \Pi(e)) \wedge (\forall(p_j, g_j) \in \Pi(e)) \wedge (i \neq j) \Rightarrow (p_i \neq p_j)$
- (3)  $(\forall(p_i, g_i) \in \Pi(e)) \wedge (\forall(p_j, g_j) \in \Pi(e)) \wedge (i \neq j) \Rightarrow (g_i \wedge g_j = false)$

In other words, the disjunction of all guards is true, i.e., every expression must evaluate to some abstract location under any guard<sup>3</sup>. Furthermore, every access path is contained at most once in a guarded cell set and any two guards associated with two different access paths must be mutually exclusive (no “uncertainty” in expression evaluation is allowed).

It is important to note that we have so far only given a definition of guarded cell sets, but we have not presented an algorithm to compute them. In the remainder of this section we give an example of guarded cell sets in a small C function. In Section 3.3 we present a framework and an algorithm to compute guarded cell sets in C code.

---

<sup>3</sup>Even if the location is not reachable through any program variable, it still exists in our representation

**Example:**

```

void memory2 (int *a, struct state *s, int flag)
{
(1)
    if (flag)
        s->num = *a;
(2)
    if (!flag)
        *a = 6;
(3)
    else *a = 3;
(4)
}

```

Here, at (1) we have:

$$\Pi(a) = \{(Root(Param(0)), true)\}$$

At program point (2) our guarded cell set encodes the fact that `s->num` can now refer to two different abstract locations under different guards at this program point.

$$\Pi(s \rightarrow num) = \{Deref((Root(Param(0))), flag), \\ Deref(FieldOf(Root(Param(1)), num), !flag)\}$$

At program point (3) we observe that `*a` can still refer to its old abstract cell under then negation of the the assignment guard `flag`.

$$\Pi(*a) = \{(Root(Const(6)), flag), (Deref(Root(Param(0))), flag)\}$$

Finally, at program point (4) we have:

$$\Pi(*a) = \{(Root(Const(6)), !flag), (Root(Const(3)), flag)\}$$

It should be noted that under any statement guard a concrete expression  $e$  can evaluate only to a subset of access paths. We define:

$$Filter(\Pi(e), G) = \{(p_i, g_i) \in \Pi(e) \mid g \wedge G \neq false\}$$

We note that by guarded cell set invariant (1)  $Filter()$  is always guaranteed to return a non-empty subset of  $\Pi(e)$ . Using this mechanism we can accurately determine which abstract locations an expression can evaluate to under a statement guard and therefore achieve path-sensitivity for all modeled memory locations.

### 3.3 Deriving guarded cell sets

To derive guarded cell sets for all expressions at any program point, we model memory with a points-to graph that accurately describes the structure of our abstract memory at every program point and is built by successively applying the derivation and assignment rules presented later in this chapter in a forward analysis through one function body.

#### 3.3.1 Types of memory cells

In our points-to graph, we distinguish three different types of abstract memory cells: *scalar cells*, *pointer cells* and *struct cells*.

$$cell := cell_{scalar} \mid cell_{pointer} \mid cell_{struct}$$

Each cell is structured as follows:

$$\begin{aligned}
 cell_{scalar} & := \\
 & \{ \\
 & \quad path : access\_path \\
 & \} \\
 \\
 cell_{pointer} & := \\
 & \{ \\
 & \quad path : access\_path \\
 & \quad pointees : guarded\_cell\_set \\
 & \} \\
 \\
 cell_{struct} & := \\
 & \{ \\
 & \quad path : access\_path \\
 & \quad fields : (field\_name \rightarrow guarded\_cell\_set)map \\
 & \}
 \end{aligned}$$

These cells fully capture then structure of our abstract memory. The scalar cell is of interest only if there exists a pointer within the function that points to the scalar at some program point. The access path of a scalar tells us whether its value is coming from the outside of the function or is equal to a constant<sup>4</sup>

The pointer cell (whose origin is also fully described by its access path) includes a guarded cell set of pointees, which models the memory locations a given pointer can point to under a certain guard.

---

<sup>4</sup>In which case the constant assignment must have taken place in the current function

Structure cells are defined by an access path and a set of fields that belong to the structure. We represent struct fields as guarded cell sets where each cell represents the possible contents of this field, which accurately models the pointers reachable from pointers in nested `structs`. It is worthwhile noting that the structure cell adds precision by differentiating between different fields of a `struct`, but is not strictly necessary for a sound modeling of pointers. However, the added precision is substantial and justifies the extra overhead of including this cell type.

### 3.3.2 Variables as guarded cell sets

While the three described cell types are able to fully model abstract memory within one function, a graph of memory cells does not contain sufficient information to derive the guarded cell set for every expression. Even though every memory location is represented, we need to know which set of locations each concrete variable in the program can evaluate to at every program point in order to derive the guarded cell set for any expression from this information (the root of valid C expression must be a variable or constant). Therefore, we maintain a guarded cell set for every program variable specifying which set of abstract cells the concrete variable corresponds to at each program point. Before we formally detail the complete set of rules for building the points-to graph and extracting the desired information, we give a small, informal example highlighting the intuition behind our approach.

**Example:** Below is an example function and a diagram modeling the memory structure after `memory2` executes its last line (guarded cell sets for concrete variables are written out):

```
void memory2 (int *a, struct state *s, int flag)
{
    if (flag)
        s->num = *a;
    if(!flag)
        *a = 6;
    else a = NULL;
}
```



### 3.3.3 Initialization

Before we begin analyzing any function, we initialize our points-to graph in the following way: First, every function parameter, local variable and global has a guarded cell set associated with it. Initially, each of these evaluate to exactly one cell of the correct type under guard `true`. Any other cells are only lazily created when needed later in the function. All pointees of pointer cells as well as fields of struct cells are lazily initialized to `{}`. Because we only expand these sets lazily as needed, we can achieve a fine-grained and demand-driven memory abstraction that does not suffer from the *k-horizon* problem as *k-limiting* does. The treatment of loops and recursive functions in our demand-driven abstraction is described in Section 3.4.

### 3.3.4 Deriving guarded cell sets from expressions

In this section we give formal rules deriving guarded cell sets for a subset of C expressions. For our purposes, a valid C expression is any expression valid on one side of the C assignment operator. We will only consider expressions of the following forms:

- (1) `v`
- (2) `v.field`
- (3) `*v`
- (4) `&v`
- (5) `constant`

Any other C expression can be translated into a combination of these primitives using additional temporary variables. In our implementation, we rely on a CIL extension [7] to do this translation. For any C expression not composed out of those primitives (such as arithmetic operators, bit shifts, etc.) we currently return  $\{Root(Unknown), true\}$  as the guarded cell set.

Our algorithm maintains the above-described points to graph  $G$  as well as a set of (mutable) guarded cell sets  $S$  for each concrete program variable in the current function. To facilitate a concise presentation, we first define the following helper functions:

$$\begin{aligned} \text{get\_gcs}(v) &= \text{retrieves the guarded cell set associated with } v \text{ from } S. \\ \text{merge}(gcs) &= \forall (p, g_j) \in gcs, \text{ add } (p, \vee_i g_i) \end{aligned}$$

**Example:**

$$\text{merge}(\{(p_1, a \wedge b), (p_1, a \wedge (!b)), (p_2, !a)\}) = \{(p_1, a), (p_2, !a)\}$$

In other words,  $\text{merge}()$  ensures that all paths in a guarded cell set are unique and ORs all the guards associated with one path.

$$\begin{aligned} \text{condition}(gcs, \text{guard}) &= \forall (p_i, g_i) \in gcs, \text{ add } (p_i, g_i \wedge \text{guard}) \\ \text{get\_field\_gcs}(\text{access\_path}, \text{field\_name}) &= \text{retrieves the gcs associated with} \\ &\quad \text{field\_name from the struct cell named} \\ &\quad \text{by access\_path} \\ \text{get\_pointee\_gcs}(\text{access\_path}) &= \text{retrieves the gcs associated with the} \\ &\quad \text{pointees of access\_path} \end{aligned}$$

In cases 1-3 below, we assume for conciseness that the abstract memory cells retrieved from our points-to graph exist. If we encounter not yet existing cells (such as an empty pointee gcs), we add a new abstract location and the gcs ( $\text{new\_path}, \text{true}$ ) before executing the algorithm presented below:

1.  $\Pi(v) = \text{get\_gcs}(v)$
2.  $\Pi(v.\text{field})$

$$\begin{aligned} gcs_{v.\text{field}} &= \forall (p_i, g_i) \in \text{get\_gcs}(v) \text{ condition}(\text{get\_field\_gcs}(p_i, \text{field}), g_i) \\ &\quad \text{return merge}(gcs_{v.\text{field}}) \end{aligned}$$

In words, for all access paths in  $gcs_v$ , we retrieve the gcs associated with  $field$ .

**Example:** Consider

$$\begin{aligned} gcs(v) &= \{(p_1, g_1), (p_2, g_2)\} \\ get\_field\_gcs(p_1, field) &= \{(p_3, g_3), (p_4, g_4)\} \\ get\_field\_gcs(p_2, field) &= \{(p_3, true)\} \end{aligned}$$

Then we have:

$$\begin{aligned} gcs_{v.field} &= \{(p_3, g_1 \wedge g_3), (p_4, g_1 \wedge g_4), (p_3, g_2 \wedge true)\} \\ merge(gcs_{v.field}) &= \{(p_3, (g_1 \wedge g_3) \vee g_2), (p_4, g_1 \wedge g_4)\} \end{aligned}$$

### 3. $\Pi(*v)$

$$\begin{aligned} gcs_{*v} &= \forall (p_i, g_i) \in get\_gcs(v) \ condition(get\_pointee\_gcs(p_i), g_i) \\ &\quad return \ merge(gcs_{*v}) \end{aligned}$$

### 4. $\Pi(\&v)$

Since there does not yet exist an abstract location associated with  $\&v$ , we create a new pointer cell with pointee guarded cell set equal to  $get\_gcs(v)$ . Furthermore, the new pointer cell will maintain a reference instead of a copy to the gcs of  $v$  in  $S$ . This is necessary because any change to  $*(\&v)$  also changes the guarded cell set of  $v$ . The exact rules and details of assignments are described in the next section.

### 5. $\Pi(constant)$

In this case we simply create a new scalar cell with value  $constant$  in our points-to graph and return  $\{(Const(constant), true)\}$ .

### 3.3.5 Assigning a guarded cell set

So far, we are able to derive guarded cell sets for any RHS-expression, but we cannot yet model updates. This section describes how we model assignments. In this section, we again restrict ourselves to describing only the following C structures; more complex nested relations can be modeled using temporary variables.

- (1) `v`
- (2) `v.field`
- (3) `*v`

To model any assignment, we first retrieve the guarded cell set for the RHS of the expression. We define:

$$gcs_{rhs} = gcs \text{ for RHS computed as specified in Section 3.3.4}$$

$$gcs_{lval} = gcs \text{ for Lval computed as specified in Section 3.3.4}$$

Since we do not take the current statement guard into account when retrieving the guarded cell set from an expression, we need to incorporate it explicitly now when modeling assignments. We assume all assignments happen under statement guard  $SG$ . We now define the following helper function:

$$assign(gcs_{rhs}, SG) = merge(\forall(p_i, g_i) \in gcs_{lval}(p_i, g_i \wedge !SG), \forall(p_j, g_j) \in gcs_{rhs}(p_j, g_j \wedge SG))$$

1.  $Lval = v$

Here, we set the gcs for  $v$  in  $S$  to  $assign(gcs_{rhs})$ .

2.  $Lval = v.field$

Here, we first retrieve the set of cells for  $v$  from  $\Pi(v)$  and then for each of those struct cells  $field$  we set them to  $assign(gcs_{rhs}, SG \wedge cell\_guard\_from\_gcs\_of\_v)$

3.  $Lval = *v$ 

Here, we first retrieve the set of cells for  $v$  from  $\Pi(v)$  and then for each of those pointer cells pointees we set them to  $assign(gcs_{rhs}, SG \wedge cell\_guard\_from\_gcs\_of\_v)$

### 3.4 Analyzing a function body

In the preceding sections, we have given algorithms for both deriving the guarded cell set of an expression and assigning a guarded cell set to an expression. These rules allow us to model the state of memory accurately when traversing a function body. Our analysis is forwards: on conditionals we proceed to analyze the conditional branches in undefined order but never continue beyond a merge point until all paths leading to the merge point have been analyzed. Since we maintain the invariant that every split is disjoint and our assignment rules always take the current statement guard into account,  $Filter(\Pi(exp))$  will return the correct set of cells that  $exp$  can be equal to under the current statement guard at any program point without explicit splitting or merging operations. This is one of the main advantages of our approach.

To guarantee termination, we unroll loops  $k$  times (with  $k = 3$  in our current implementation) and remove any back edge after this from the control flow graph. This is a source of unsoundness in our approach.

# Chapter 4

## Inter-procedural analysis

So far, our discussion has avoided procedure calls. All the algorithms and definitions presented in Chapter 3 are only applicable within one procedure. Our analysis achieves *context-sensitivity* by using a *summary-based* approach. We first sort the call graph topologically (currently, we unsoundly break cycles at arbitrary points) and then analyze all functions in bottom-up order.

We first describe how we generate function summaries, then how we apply them when analyzing a function.

### 4.1 Function summaries

Two essential properties we must summarize are which interface objects are aliased to which access paths and which interface objects are assigned constant values. We first note that our approach makes it easy to extract this information at the end of each function. To detect any side effects visible outside the function, we explore all cells reachable from visible interface objects access paths and collect all assignments visible from outside the current function <sup>1</sup>.

---

<sup>1</sup>For example, any store into a parameter pointer will be visible outside the function

More specifically, we do the following: For parameters, we note that side effects can only persist if the parameters are pointers. We therefore find the cell in the points-to graph with access path  $Root(Param(i))$  (which must be of type *pointer\_cell* or *struct\_cell* with at least one pointer field embedded in it) and look at its pointees guarded cell set. If the gcs's only entry is  $\{(Deref(Root(Param(i))), true)\}$ , we know that `*param` is unmodified and continue checking `**param` if its type allows. Otherwise, we know exactly under which guard(s) `*param` was reassigned to what other locations. We also note that the only valid reassignments are to access paths that are either contacts, globals, other parameters or the terminal *Invalid*. A reassignment to a path ending in a local variable would indicate a programmer error, e.g., the programmer set `*param` to an uninitialized local stack variable. For globals, all side effects persist.

More formally, we want to obtain a list of assignments and guards accurately summarizing the aliasing and side effect behavior of the current function. We obtain this list consisting of tuples (*interface objects*, *access paths*) by the following algorithms:

```

PROCEDURE get_param_side_effect(num, expected_path)
1. result = ;
2. gcs =  $\Pi(*v)$ ;
3. FOR EACH  $(p_i, g_i) \in gcs$  with  $p \neq expected\_path$  DO
4.   result = result  $\cup$  (expected_path,  $p_i$ )
5. END;
6. IF (pointer_type (**v)) THEN
7.   result = result cup get_param_side_effect (num, Deref(expected_path))
8. END;
9. RETURN result;

PROCEDURE get_global_side_effect(v, expected_path)
1. result = ;

```

```

2.  gcs =  $\Pi(v)$ ;
3.  FOR EACH  $(p_i, g_i) \in gcs$  with  $p \neq expected\_path$  DO
4.     $result = result \cup (expected\_path, p_i)$ 
5.  END;
6.  IF (pointer_type (*v)) THEN
7.     $result = result \cup get\_global\_side\_effect(v, Deref(expected\_path))$ 
8.  END;
9.  RETURN result;

```

We can now retrieve the desired list of all possible side effects by calling `get_param_side_effects(num, Root(param(num)))` and similarly `get_global_side_effect(name, Root(Global(name)))` for all globals.

For clarity, the sudo-code presented here does not deal with structures; however, the extension to all three abstract cell types is straightforward and implemented in our framework.

In order to generate a function summary, we simply include the list of assignments returned by `get_param_side_effect` and `get_global_side_effect` in our function summary. To make our analysis scale to millions of lines of source code, we currently limit the path-sensitive information included in our summary to simple boolean variables of the form `c` or `!c`. All statements involving more complex conditionals are represented as `true`. Although this is a source of imprecision in the analysis, it is not a source of unsoundness. Empirically, we found this approximation to work well with the null dereference analysis that was built on top of the pointer analysis described here.

## 4.2 Using function summaries

In order to incorporate function summaries, we retrieve the list of side effects generated as described in Section 4.1, translate the right hand side into the current function's name space and execute the algorithm detailed in Section 3.3.5 to model assignments.

# Bibliography

- [1] Alain Deutsch. *Operational models of programming languages and representations of relations on regular languages with application to the static determination of dynamic aliasing properties of data*. PhD thesis, University Paris VI (France), 1992.
- [2] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 230–241, New York, NY, USA, 1994. ACM Press.
- [3] Dawson Engler, David Chen, Seth Hallem, and Andy Chou. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, 2001.
- [4] Brian Hackett and Alex Aiken. How is aliasing used in systems software? <http://theory.stanford.edu/~aiken/publications/new/aliasinguse.pdf>.
- [5] William Landi and Barbara Ryder. Pointer-induced aliasing: a problem taxonomy. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103, New York, NY, USA, 1991. ACM Press.
- [6] William Landi and Barbara Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Not.*, 39(4):473–489, 2004.
- [7] George Necula, Scott McPeak, Shree Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In

- Lecture Notes in Computer Science*, volume 2304, page 213. Springer Berlin, January 2002.
- [8] Thomas Reps. Shape analysis as a generalized path problem. In *PEPM '95: Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 1–11, New York, NY, USA, 1995. ACM Press.
- [9] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–293, New York, NY, USA, 1988. ACM Press.
- [10] Jan Stransky. A lattice for abstract interpretation of dynamic (lisp-like) structures. *Inf. Comput.*, 101(1):70–102, 1992.
- [11] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape analysis. In *Lecture Notes in Computer Science*, volume 1781, page 1. Springer Berlin, January 2000.
- [12] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 115–125, New York, NY, USA, 2005. ACM Press.
- [13] Yichen Xie and Alex Aiken. Saturn: A sat-based tool for bug detection. In *Lecture Notes in Computer Science*, volume 3576, pages 139–143. Springer, July 2005.
- [14] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. *SIGPLAN Not.*, 40(1):351–363, 2005.
- [15] Yichen Xie and Dawson Engler. Using redundancies to find errors. *IEEE Transactions on Software Engineering*, 29(10):915–928, 2003.