University of Alberta

SafeType: Detecting Type Violations for Type-Based Alias Analysis of C

by

Iain Ireland

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2013
Edmonton, Alberta

Abstract

A type-based alias analysis uses the types of variables in a program to assist in determining the alias relations of those variables. The C standard restricts the types of expressions that may access objects in memory, with the intent of specifying when two objects may be aliased. In practice, however, many C programs do not conform to these restrictions, making type-based alias analysis unsound for those programs. As a result, type-based alias analysis is frequently disabled.

This thesis presents SafeType, a sound approach for compile-time detection of violations of the C standard's type-based restrictions on memory access; describes an implementation of SafeType in the IBM XL C compiler, with flow- and context-sensitive queries to handle variables with type `void *`; evaluates that implementation, showing that it scales to programs with hundreds of thousands of lines of code; and identifies a previously unreported violation in the `470.lbm` benchmark in SPEC CPU2006.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Alias analysis is a static program analysis that is used at compile time to determine whether expressions may refer, at runtime, to the same memory location. Alias analysis is essential to enable many program transformations, and to widen the scope of many others. Without aliasing information (or its close relative, points-to information), the effects of indirect memory operations must be conservatively approximated, leading to missed opportunities for transformation. Thus, accurate aliasing information improves the precision of later analyses and transformations, which in turn improves the quality of the compiled code.

One form of alias analysis is type-based alias analysis. Type-based alias analysis relies on the idea that memory references with different types should not alias. For example, consider the code in Listing 1.1. An optimizing compiler may wish to eliminate a load by rewriting the call to `bar` in line 5 as `bar(1)`, using the constant value 1 assigned to `a` in line 3. This transformation is only safe if the assignment in line 4 does not affect the value of `a`. Can `b` point to `a`? Intuitively, if the type of `b` is `int *`, then `b` can point to `a`. If the type of `b` is some other type, such as `double *`, that is not compatible with `int`, then `b` cannot point to `a`, and it is safe to rewrite the call to `bar`.

Listing 1.1: Type-Based Alias Analysis

```
1 int a;
2 void foo(___ * b) {
3   a = 1;
4   *b = 2;
5   bar(a);
6 }
```

To formalize this intuition, the C standard (in section 6.5p7) imposes type-based restrictions on memory access [17]. Objects in memory have types; if an object is accessed using an expression with a type that does not conform to the restrictions, the behaviour of the program is undefined. These restrictions are explicitly included in the C standard to enable type-based alias analysis. The C standard imposes no requirements on undefined behaviour. A compiler is therefore permitted to ignore accesses to memory that violate

these type-based restrictions, and to assume that such accesses do not exist. In this way, the C standard's type-based restrictions on memory access enable the compiler to use a type-based alias analysis to prove that expressions of different types do not alias.

Unfortunately, in practice, many C programs do not conform to the C standard with respect to type-based restrictions on memory access, and still expect a compiler to generate functional code. In some cases, the existence and identity of violations is known. For example, in the SPEC CPU2006 benchmark suite, the list of known portability issues for the `400.perlbench` benchmark includes the following warning:

> There are some known aliasing issues. The internal data structures that represent Perl's variables are accessed in such as a way as to violate ANSI aliasing rules. Compilation with optimizations that rely on strict compliance to ANSI C aliasing rules will most likely produce binaries that will not validate.

Similarly, prior to Python 3, the reference implementation of Python used a representation of Python objects that violated aliasing rules [27]. In both of these cases, the existence of violations was known. However, there are other cases in which the presence of violations is uncertain. Violations may occur due to programmer error, or ignorance of the restrictions. Violations may also occur in legacy programs written prior to the creation of the C standard. In many cases, the effort required to rewrite those programs to be compliant is prohibitive. This obstacle is particularly relevant for the largest and most important code-bases.

When applied to programs that violate the C standard's type-based restrictions, type-based alias analysis is unsafe. Its use may unintentionally alter program semantics, leading to the introduction of difficult-to-diagnose bugs. One example, reported by Reinig, comes from the `176.gcc` benchmark in SPEC CPU1995. In one function, `176.gcc` clears the contents of a structure by casting the structure to an array of `int` and assigning 0 to each element of the array. Because an array of `int` cannot be legally aliased to an arbitrary structure, the DEC C/C++ compiler undertook a series of seemingly valid transformations that left uninitialized data in the structure[28].

Bugs introduced by unsafe type-based alias analysis can be difficult to diagnose and correct. An incorrect alias analysis is not harmful on its own; a bug is only introduced when incorrect alias information is used to enable a code transformation that alters program semantics. Thus, the set of bugs that may be introduced by unsafe type-based alias analysis is highly sensitive to small changes in a program, or in the set of transformations applied to that program. As a result, type-based alias analysis is frequently turned off. This situation is unfortunate. Even in non-standard-compliant programs where type-based alias analysis is unsafe, there remain many aliasing situations where type-based alias could provide useful information.

It is therefore useful to be able to automatically identify code that may violate the C

standard's type-based restrictions. If code that violates the standard can be isolated, it becomes possible to make informed decisions about when to use type-based alias analysis. Furthermore, it also becomes possible to identify the code changes that must be made to ensure that type-based alias analysis can be safely applied to a program.

Chapter 2 gives background information on alias analysis. Chapter 3 presents a formal definition of the problem of identifying violations, and introduces a static analysis, SafeType, that soundly identifies such violations. Chapter 4 describes a prototype implementation of SafeType, including the flow-sensitive and context-sensitive extensions necessary to safely and precisely analyze variables of type `void *`. Chapter 5 evaluates a prototype implementation of SafeType, demonstrates that it scales to programs of hundreds of thousands of lines of code, and identifies a previously unreported violation of the C standard's type-based restrictions on memory access in the `470.lbm` benchmark from SPEC CPU2006. Chapter 6 describes related work on points-to analysis and safety analysis for C. Chapter 7 presents the conclusions of the thesis.

Chapter 2

Background

This chapter presents background information. Section 2.1 presents the history of type-based alias analysis. Section 2.2 describes fundamental concepts underlying alias analysis in C. Section 2.3 introduces operational semantics.

2.1 Type-Based Alias Analysis

Although the idea of type-based alias analysis has been mentioned in passing for decades [4], the first publication to present an algorithm or evaluate the benefits of type-based alias analysis was an implementation for Modula by Diwan *et al.* [7]. They present three versions of the analysis. The first version is purely type-based: two memory references may alias if and only if they have the same declared type, or if the declared type of one is a subtype of the declared type of the other. The second version improves the precision of the analysis using additional high-level information: for example, whether the address of a variable is ever taken. The third version uses a flow-insensitive algorithm to exclude aliases between a type T and a subtype S of T unless a statement assigns a reference of subtype S to a reference of type T. Each of the three versions is evaluated as the only alias analysis in a Modula compiler. All three algorithms are flow-insensitive, context-insensitive, and field-sensitive.

The first implementation of type-based alias analysis for C to be described in the literature is by Reinig for the DEC C and DIGITAL C++ compilers [28], The DEC implementation operates on each function independently and creates a set of `effects classes` and associated `effects signatures`. Each `effects class` represents a set of memory locations or a type. Every memory access in the analyzed function is assigned to an `effects class`. The `effects signature` for an effects class is a set of other `effects classes`. The `effects signatures` are constructed such that if two memory assignments may refer to the same location in memory, then the intersection of the `effects signatures` of their respective `effects classes` must be non-null. This approach allows the DEC compiler to efficiently represent both type-based alias analysis and structural aliasing. Like Diwan's

analysis, this analysis is flow-insensitive, context-insensitive, and field-sensitive.

Although no paper was published, gcc introduced type-based alias analysis in the same time period as the DEC compiler [10]. In gcc's implementation, alias information produced by the C front end based on the type assigned to an object by the programmer was propagated into the optimization passes. This information was used in conjunction with a pre-existing "base address" alias analysis, which eliminated a different set of impossible alias pairs. The combination of the two analyses was more precise than either analysis independently.

Type-based alias analysis has gone on to become a common feature of production C compilers. For example, it is present in IBM's XL C compiler, the Intel Itanium compiler [11], and, as of April 2011, LLVM's Clang compiler [23]. Type-based alias analysis is also used for other languages: for example, it is incorporated into the state-of-the-art DOOP framework for Java [2].

The most important evaluation of the benefits of type-based alias analysis comes from Ghiya [11]. In the context of the Intel Itanium compiler, Ghiya evaluates seven complementary strategies for memory disambiguation: disambiguation of direct references; a simple base-offset analysis; an array data-dependence analysis; an intraprocedural version of Andersen's points-to analysis modified to be field-based; a global address-taken analysis; an interprocedural version of Andersen's analysis; and a type-based alias analysis. Each of these techniques is applied to the twelve C/C++ benchmarks from the SPEC CINT2000 benchmark suite. Ghiya determines that type-based alias analysis "pays off" for five of the twelve benchmarks, eliminating alias pairs that no combination of the other six methods can disambiguate. In particular, type-based alias analysis is important for disambiguating pointers against scalar objects. Ghiya concludes that a suite of disambiguation techniques – including type-based alias analysis – is the optimal approach.

2.2 Alias Analysis

The C standard defines an *object* as a region of data storage in the execution environment. The contents of an object represent a *value* when interpreted as having a particular type. An *lvalue* is an expression that refers to such an object. An *access* is an execution-time action reading or modifying the value of an object in memory. A *modification* is an access that assigns a new value to an object (including cases where the new value being stored is the same as the old value). A *read* is an access that uses the current value of an object.

An expression in a program is a memory reference if its execution accesses memory. In an assignment `x = y`, there are two memory references: one that reads `y`, and one that modifies `x`. Two memory references are *aliased* if they may refer to the same memory location.

A program analysis is an automatic analysis that proves a particular property about the

behaviour of a program. Program analyses can be divided into two categories. A *dynamic analysis* is performed by executing a program and examining aspects of its execution. A *static analysis* is performed without executing a program, by examining its source code or an intermediate representation of its source code.

Frequently, properties examined using static analysis cannot be proven precisely in the general case. As a result, the output of a static analysis is typically an approximation of a program's actual behaviour. A static analysis may deduce the presence or absence of a particular behaviour, or it may be unable to conclude one way or the other. An analysis that may deduce the absence of a property that is present during the execution of a program, or vice versa, is `unsafe`. An analysis that only deduces true facts is `safe`.

Definition 1. An *alias analysis* is a static program analysis that determines which memory references may be aliased in a program. For a program P, the outcome of an alias analysis A is represented as an output set $O(P)$ of unordered pairs of the form $\langle x, y \rangle$, where x and y are memory references in P, such that $\langle x, y \rangle \in O(P)$ if A determines that x and y may alias.

Definition 2. An alias analysis A is *safe* if, for every pair of memory references x, y in a program P such that x and y alias in some execution of P, $\langle x, y \rangle \in O(P)$. If a safe alias analysis cannot prove that a pair of memory references do not alias, it must conservatively include them in its output set. Thus, a more precise analysis will have a smaller output set. Given two safe alias analyses A_1 and A_2 with output sets $O_1(P)$ and $O_2(P)$, the set $O_3(P) = O_1(P) \wedge O_2(P)$ is also safe, and will be a more precise result.

An analysis is *flow-sensitive* if it considers the order of statements in a program, and computes information for each program point. An analysis that computes one set of information for the entire program is *flow-insensitive*. An analysis is *context-sensitive* if it considers the calling context of a function while computing information for that function. An analysis that does not distinguish between calling contexts is *context-insensitive*. An alias analysis is *field-sensitive* if it considers each instance of a field in an aggregate (in C, a `struct`) as a unique variable. An analysis that considers each aggregate as a unique variable, but does not distinguish between fields within that variable, is *field-insensitive*. An analysis that considers each field as a unique variable, but does not distinguish between the aggregates that contain that field, is *field-based* [25].

2.3 Inference Rules and Operational Semantics

Determining the safety of a program analysis frequently requires the use of formal proofs. One common form of notation for expressing these proofs, used in this thesis, is the use of `inference rules`. These rules express deductions in terms of premises and conclusions.

$$\text{(SIMPLE)} \quad \frac{\textit{premise}}{\textit{conclusion}}$$

$$\text{(COMPLEX)} \quad \frac{\textit{premise} \qquad \textit{premise}}{\textit{conclusion} \qquad \textit{conclusion}}$$

$$\text{(AXIOM)} \quad \textit{axiom}$$

Figure 2.1: Inference Rules

They are written as shown in Figure 2.1. Premises are written above a line. The conclusions that can be drawn if those premises are true are written below the line. Axioms, statements that are true without any premise, omit the line. To make it simple to refer to specific rules, they are given names, typically written to the left. As can be seen in (COMPLEX), rules may have more than one premise, in which case the inference rule requires a logical conjunction of the premises. Rules may also have more than one conclusion, in which case each conclusion holds when the premises are true.

Inference rules are commonly used to represent structural operational semantics. A structural operational semantics describes a formal systems in terms of states, and transitions between those states [26]. For example, the behaviour of a program in a given programming language can be formally represented using a structural operational semantics $\mathcal{O}$. This representation is accomplished in two parts. In the first part, a representation is introduced for the state of the program at a single point. In the second part, a set of inference rules are introduced which express the computational steps that are taken to move from one state to another. If $\mathcal{S}$, $\mathcal{S}'$ range over program states, and s ranges over statements, these rules take the following general form:

$$\text{(TRANSITION)} \quad \frac{\textit{premise}*}{\langle \mathcal{S}, s \rangle \rightarrow \mathcal{S}'}$$

This rule says that, if each of the premises is true, and the program execution is currently in some state $\mathcal{S}$, then the execution of the statement s updates the state to a new state $\mathcal{S}'$. Each type of statement s in the language being formalized corresponds to its own rule or set of rules. The execution of a program is expressed as the result of a series of transitions from one state to the next. This representation can be used to construct formal proofs about program behaviour.

Chapter 3

SafeType: An analysis to enable type-based alias analysis

This chapter presents a formal definition for the problem of identifying code that makes type-based alias analysis unsound, and introduces SafeType, a static analysis that soundly identifies such violations. Section 3.1 describes the assumptions made during the design of SafeType. Section 3.2 establishes notation for describing the behaviour of a C program. Section 3.3 uses that notation to formally define the scope of the problem. Section 3.4 introduces SafeType and shows that SafeType can be used to identify code that makes type-based alias analysis unsound.

This chapter deliberately omits discussion about how to handle variables of type `void *`. Chapter 4 describes the steps necessary to handle such variables correctly.

3.1 Assumptions

It is important to highlight the assumptions that underpin the analysis. The goal of SafeType is to identify code that causes type-based alias analysis to be unsound, but that is otherwise safe. To this end, SafeType assumes the absence of illegal memory operations: dereferences of undefined pointers, dereferences of freed memory, and array-bounds violations. As discussed in Section 6.2, many techniques already exist to detect these errors, and programmers have experience identifying and fixing them. By considering only those type-unsafe accesses that are otherwise defined behaviour, this new approach is simpler, more tractable, and more precisely focused on the problem it aims to solve.

3.2 Preliminaries

In what follows, τ ranges over types, P ranges over programs, F ranges over functions, x ranges over variables, e ranges over expressions, a ranges over addresses, s ranges over statements, v ranges over values, $\mathcal{S}$ ranges over states, $\mathcal{M}$ ranges over stores, and Γ ranges

over type environments.

Definition 3. Types in C are *basic types* or *derived types*. The basic types are the character types, the arithmetic types, and the void type. Derived types are constructed from basic or derived types. The derived types are:

- **Pointer types**: Given a type τ, a pointer to τ, represented as $*\tau$, represents an object whose value is interpreted as the address of an object of type τ.
- **Structure and union types**: Structure types represent a sequentially allocated, non-empty set of member objects. Union types represent an overlapping, non-empty set of member objects. Both structure and union types are represented as an ordered set of n pairs $M = \{(\mathtt{x}_1, \tau_1), \ldots, (\mathtt{x}_n, \tau_n)\}$. Each pair $(\mathtt{x}_i, \tau_i)$ represents a member object with name $\mathtt{x}_i$ and type τ_i.
- **Function types**: Function types are characterized by their return types and the number and types of their parameters, and represented as a pair $\langle \tau_R, \{\tau_1, \ldots, \tau_n\} \rangle$, where τ_R is the return type, and $\{\tau_1, \ldots, \tau_n\}$ is an ordered set of n parameter types.

Definition 4. The C standard defines the *effective type* of an object in memory as the declared type of that object, if it exists. Objects allocated on the heap have no declared type. For such objects, the effective type is set whenever a value is stored into that object through an lvalue with a type that is not a character type.

The abstract syntax in Figure 3.1 defines a program in C. Without loss of generality, typecasts and modifications of memory are restricted to assignment statements. Selection statements (`if` and `switch`), iteration statements (`do`, `for`, and `while`), and jump statements (`goto`, `break`, and `continue`) are relevant for a flow-sensitive analysis, but do not have any direct effect on the aliasing behaviour of a program, and are therefore omitted here. The definition for `function` includes two lists of declarations: the first represents the set of arguments to the function, and the second represents the local variables declared inside the function.

The state of a program at a given point is represented as a tuple $\mathcal{S} = (\mathcal{M}, \Gamma, \mathcal{E})$. $\mathcal{M}$ is the state of memory, represented as a map from addresses to values. Γ is the type environment, representing the types of lvalues, such that $\Gamma \vdash l : \tau$ means that the type of an lvalue l in the scope represented by Γ is τ. $\mathcal{E}$ stores the effective types of objects in memory, represented as a map from addresses to types. For an address a, $\mathcal{E}(a)$ is the effective type of the memory object stored at that address.

The contents of the store $\mathcal{M}$ are modified at assignment statements. The contents of the type environment Γ are modified at function boundaries. The contents of the effective type map $\mathcal{E}$ are modified at function boundaries for addresses corresponding to variables on

```
program          ::=  decl-list × function*

function         ::=  decl-list × decl-list × stmt

decl-list        ::=  (identifier, type)*

stmt             ::=  stmt; stmt
                   |  assignment-stmt
                   |  selection-stmt
                   |  iteration-stmt
                   |  jump-stmt

assignment-stmt  ::=  lval = [ϵ | * | (type)] expr
                   |  lval = function(expr*)

...

lval             ::=  variable
                   |  * lval
                   |  lval[expr]
                   |  lval.identifier

expr             ::=  constant-value
                   |  identifier
                   |  &lval
                   |  unary-operator expr
                   |  expr binary-operator expr
                   |  expr.identifier
                   |  expr[expr]
```

Figure 3.1: Abstract Syntax

the stack, and are modified at assignment statements for addresses corresponding to objects allocated on the heap.

Given a state $\mathcal{S}$ and an expression e, the rules describing the evaluation of e to obtain a value v of type τ, written $\lfloor e, \mathcal{S} \rfloor \Rightarrow \langle v, \tau \rangle$, are straightforward but extensive. We omit them here, noting only that by construction the evaluation of an expression can have no side-effects. The type of an expression depends only on Γ, and not on $\mathcal{M}$ or $\mathcal{E}$. In cases where only the type is required, we write $\Gamma \vdash e : \tau$ to indicate that the type of expression e is τ.

An lvalue is an expression that refers to an object in memory. In this representation, the address accessed by an lvalue l is given by the function $\rho(l)$. For a variable x, $\rho(x)$ is the address of that variable on the stack. For a dereference $*l$, $\rho(*l)$ is $\mathcal{M}(\rho(l))$. For a member reference $l.x$, $\rho(l.x)$ is $\rho(l) + \mathit{off}$, where off is the offset of member x. For an array index expression $l[e]$, if $\lfloor S, e \rfloor \Rightarrow \langle v, \tau \rangle$, then $\rho(l[e]) = \rho(l) + v * \texttt{sizeof}(l)$.

$\mathcal{S}[\mathsf{element} = \mathsf{value}; \ldots]$ represents updates to elements of $\mathcal{S}$, such that $\mathcal{S}[\mathcal{M}(a) = v]$ is the state $\mathcal{S}$ where the value of $\mathcal{M}(a)$ has been updated to v. Shorthand notation $\mathcal{S}[\mathrm{def}(x, \tau)]$ represents $\mathcal{S}[\Gamma \vdash x : \tau; \mathcal{E}(\rho(x)) = \tau]$. The notation $\mathcal{S}[\mathcal{E}(a) \leftharpoonup \tau]$ indicates that the value of $\mathcal{E}(a)$ becomes τ if address a is on the heap, and stays unchanged if a is on the stack.

We define the behaviour of our abstract syntax using a structural operational semantics. Selected rules are shown in Figure 3.2. These rules are written using the standard form for inference rules as described in Section 2.3: premises are listed above the horizontal line, and the conclusions that can be derived if the premises are true are listed below the horizontal line. The transition relation $\langle \mathcal{S}, s \rangle \rightarrow \mathcal{S}'$ indicates that the execution of statement s has taken the state from $\mathcal{S}$ to $\mathcal{S}'$. Rules that do not have a direct effect on $\mathcal{S}$ — loops, conditionals, and other control flow statements — are omitted. Rule (`ASST`) describes a basic assignment: $l = e$. If expression e, evaluated in state $\mathcal{S}$, gives a value v of type τ, then the assignment $l = e$ updates state $\mathcal{S}$ such that $\mathcal{M}(\rho(l))$, the value of memory at the address a that l accesses, becomes v. Furthermore, if a is on the heap, $\mathcal{E}(\rho(l))$, the effective type of a, is updated to τ (as indicated by the use of $\leftharpoonup$). Rule (`LOAD`) describes the case in which the assigned expression dereferences a pointer: $l = *e$. In this case, the value of the expression e is an address, which is looked up in $\mathcal{M}$ to determine the value assigned to l. The operational semantics of typecasts as defined in (`CAST`) are completely standard; (`CAST`) is written as a separate rule to facilitate our later discussion of type safety. Rule (`CALL`) describes a function call. The body of the called function is evaluated in the appropriate context, and the updated state is returned. For clarity, the rule is written for a function with one argument x_a and one local variable x_1. However, the rule extends straightforwardly to functions with multiple arguments and/or multiple local variables.

$$\text{(ASST)}\quad \frac{\lfloor e, \mathcal{S} \rfloor \Rightarrow \langle v, \tau \rangle}{\langle \mathcal{S}, l = e \rangle \rightarrow \mathcal{S}[\mathcal{M}(\rho(l)) = v; \mathcal{E}(\rho(l)) \leftharpoonup \tau]}$$

$$\text{(LOAD)}\quad \frac{\lfloor e, \mathcal{S} \rfloor \Rightarrow \langle a, *\tau \rangle \qquad \mathcal{M}(a) = v}{\langle \mathcal{S}, l = *e \rangle \rightarrow \mathcal{S}[\mathcal{M}(\rho(l)) = v; \mathcal{E}(\rho(l)) \leftharpoonup \tau]}$$

$$\text{(CAST)}\quad \frac{\lfloor e, \mathcal{S} \rfloor \Rightarrow \langle v, \tau_{\text{orig}} \rangle}{\mathcal{S}' = S[\mathcal{M}(\rho(l)) = \text{cast}(\tau_c, v); \mathcal{E}(\rho(l)) \leftharpoonup \tau_c] \qquad \langle \mathcal{S}, l = (\tau)e \rangle \rightarrow \mathcal{S}'}$$

$$\text{(CALL)}\quad \frac{\begin{array}{c}\lfloor e, \mathcal{S} \rfloor \Rightarrow \langle v, \_ \rangle \\ f = \langle \{(x_a, \tau_a)\}, \{(x_l, \tau_l)\}, s \rangle \qquad \langle \mathcal{S}[\mathcal{M}(\rho(x_a)) = v; \text{def}(x_a, \tau_a); \text{def}(x_l, \tau_l)], s \rangle \rightarrow \mathcal{S}'\end{array}}{\langle \mathcal{S}, f(e) \rangle \rightarrow \mathcal{S}'}$$

Figure 3.2: Transition Relation

3.3 Problem Statement

As described in Section 3.1, the C standard imposes restrictions on the types of lvalue expressions that may access objects in memory. Using the notation established in Section 3.2, these restrictions can be characterized as follows:

Definition 5. Define the type compatibility relation $\equiv_t$ such that $\tau_i \equiv_t \tau_j$ iff types τ_i and τ_j belong to the same equivalence class: that is, if they have the same C type, ignoring qualifiers (`const, volatile, restrict`) and the signedness of integer types.

Definition 6. Define the can-access relation $\triangleright$ such that for two types τ_l and τ_o, $\tau_l \triangleright \tau_o$ iff:

- $\tau_l \equiv_t \tau_o$;
- τ_l is an structure or union type, and there exists a member $m = (x_m, \tau_m) \in \tau_l$ such that $\tau_o \equiv_t \tau_m$; or
- τ_l is a character type.

Definition 7. Define $\mathsf{safe}(l)$ to be true for an lvalue l if l is permitted to access the value of the object to which it refers. Given that $\Gamma \vdash l : \tau_l$, l is permitted to access the value of the object in memory at address $\rho(l)$, with effective type $\mathcal{E}(\rho(l)) = \tau_o$, iff $\tau_l \triangleright \tau_o$.

Less formally: an lvalue l is permitted to access an object in memory o if l and o have the same type, ignoring qualifiers and signedness. An exception is made if l is an expression with structure or union type. The intent of this exception is to acknowledge the existence of *structural aliasing*, in which aggregate and union variables are by nature aliased to their member variables. An assignment to a struct variable will necessarily modify the value of

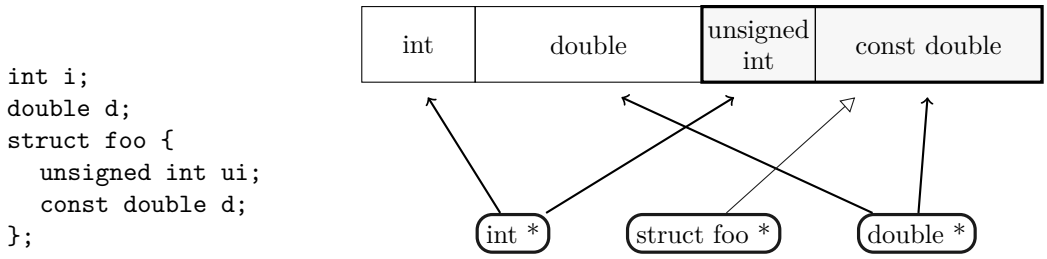

Figure 3.3: Typing Rules

its member variables. Finally, an exception is made for character pointers, to allow for their common use in bytewise manipulation of objects in memory.

A visual example of Definition 6 can be seen in Figure 3.3. A set of variables is defined on the left. Those variables are laid out as objects in memory on the right. Below that representation of memory, there is a set of pointer types. An arrow is drawn from a pointer type in the bottom right to an object in memory if a dereference of a pointer of that type is permitted to access that memory object. The arrow from `struct foo *` is drawn differently to indicate that it does not directly access either member of the struct; instead, it directly accesses the struct as a whole, which implicitly accesses each member. A `char *` pointer would have an arrow to each of the four objects in memory, but is omitted for clarity.

Figure 3.4 defines the conditions under which $\mathsf{safe}(l)$ is true for an lvalue l:

(VAR) : An lvalue consisting only of a variable use x is always permitted to access the memory to which it refers. Variables are stored on the stack, and effective types on the stack do not change. Γ is also constant within a function. Thus, because $\mathcal{E}(\rho(x)) = \tau_o$ and $\Gamma \vdash x : \tau_l$ are assigned the same type τ by (CALL) upon function entry, it will always be the case that $\tau_l \equiv_t \tau_o$, and thus $\tau_l \triangleright \tau_o$.

(MEMBER) : If l is a structure or union, and $\mathsf{safe}(l)$, then $\rho(l)$ is the address of a struct or union in memory, and its members can be accessed.

(DEREF) : A dereference $*l$ is permitted iff the type of l is a pointer to some type τ_1, the effective type of the object in memory to which l points is τ_2, and $\tau_1 \triangleright \tau_2$. The address of the object in memory to which l points is $\mathcal{M}(\rho(l))$. Thus, unlike (VAR) and (MEMBER), (DEREF) depends on $\mathcal{M}$.

(SUBSCRIPT) : According to the C standard, $a[b]$ is semantically identical to $*(a + b)$. As described in Chapter 1, SafeType assumes that array-bounds violations do not occur. Thus, $a + b$ will have the same type as a, and the same restrictions are applied in this case as in (DEREF).

An inspection of the rules in Figure 3.4 shows that the only non-trivial premises are the

$$\text{(VAR)}\quad \frac{x \text{ is a variable}}{\mathsf{safe}(x)}$$

$$\text{(MEMBER)}\quad \frac{\Gamma \vdash l : \text{struct or union type} \qquad \mathsf{safe}(l)}{\mathsf{safe}(l.x)}$$

$$\text{(DEREF)}\quad \frac{\mathsf{safe}(l) \qquad \mathcal{E}(\rho(l)) = *\tau_1 \qquad \mathcal{E}(M(\rho(l))) = \tau_2 \qquad \tau_1 \triangleright \tau_2}{\mathsf{safe}(*l)}$$

$$\text{(SUBSCRIPT)}\quad \frac{\mathsf{safe}(l) \qquad \mathcal{E}(\rho(l)) = *\tau_1 \qquad \mathcal{E}(M(\rho(l))) = \tau_2 \qquad \tau_1 \triangleright \tau_2}{\mathsf{safe}(l[e])}$$

Figure 3.4: Lvalue Safety

identical conditions found in (DEREF) and (SUBSCRIPT). Thus, we can precisely define the type-safety property for a program P:

Definition 8. A program P will violate the C standard's type-based restrictions on memory access iff P accesses memory through an lvalue l such that $\mathcal{E}(\rho(l)) \not\triangleright *\mathcal{E}(\mathcal{M}(\rho(l)))$.

Less formally: a program is safe unless it dereferences a pointer pointing to an object of an incompatible type. Thus, to determine whether or not a program violates the C standard's type-based restrictions on memory, it is sufficient to determine whether such a pointer is ever dereferenced. In turn, the presence or absence of violations in a program is sufficient to determine whether the application of type-based alias analysis is sound.

3.4 Static Analysis

By itself, Definition 8 is not particularly helpful. This section shows that this property can be verified at compile time by examining individual assignment statements, and verifying that they preserve the invariant that every non-null pointer in memory points to an object that it is permitted to access. We name the resulting analysis SafeType.

Definition 9. Given a state $\mathcal{S} = (\mathcal{M}, \Gamma, \mathcal{E})$, the safe-memory property $\mathsf{safe\text{-}mem}(\mathcal{S})$ is true iff for every address a in the domain of $\mathcal{E}$, if $\mathcal{E}(a) = *\tau$, then $\mathcal{M}(a) = 0 \vee \big(\tau \triangleright \mathcal{E}(\mathcal{M}(a))\big)$. That is, for every object in memory whose effective type is a pointer type $*\tau_{\text{ptr}}$, either it is a null pointer, or the value of that pointer is the address of an object in memory with effective type τ_{obj} such that $\tau_{\text{ptr}} \triangleright \tau_{\text{obj}}$. By definition, $\mathsf{safe\text{-}mem}(\mathcal{S}_0)$ is true for the initial state of a program $\mathcal{S}_0$.

Lemma 1. For a program state $\mathcal{S}$, if $\mathsf{safe\text{-}mem}(\mathcal{S})$, then the evaluation of an expression e in state S will never violate the C standard's type-based restrictions on memory.

By Definition 9, every pointer value in memory points to an object that it is permitted to access. As can be seen in Figure 3.1, no mechanism is available to an expression that could generate a pointer value that points to an object that it is not permitted to access.[1] Thus, evaluating an expression e in a state $\mathcal{S}$ cannot violate the C standard if $\mathsf{safe\text{-}mem}(\mathcal{S})$.

Lemma 2. For a program state $\mathcal{S}$, if $\mathsf{safe\text{-}mem}(\mathcal{S})$, then for every lval l, where $\Gamma \vdash l : \tau$, $\tau \triangleright \mathcal{E}(\rho(l))$.

Lemma 2 follows directly from the definition of safe-mem and the rules in Figure 3.4. The possibility of using SafeType to verify that a program is safe follows directly from Lemmas 1 and 2. The program starts in a state where memory is safe – that is, $\mathsf{safe\text{-}mem}(\mathcal{S}_0)$. Then, SafeType attempts to prove that for every step $\langle \mathcal{S}, a \rangle \to \mathcal{S}'$, $\mathsf{safe\text{-}mem}(\mathcal{S})$ implies $\mathsf{safe\text{-}mem}(\mathcal{S}')$. Therefore, if SafeType succeeds in its proof, then the entire program is safe by induction.

As can be seen in Figure 3.2, in each transition rule that updates memory, the update takes the form $\mathcal{S}[\mathcal{M}(\rho(l)) = v; \mathcal{E}(\rho(l)) \leftharpoonup \tau_v]$ for some lvalue l and value v of type τ_v. By Lemma 1, if τ_v is a pointer type, then v must be an address a such that $\tau_v \triangleright \mathcal{E}(\mathcal{M}(a))$. Thus, to determine whether $\mathsf{safe\text{-}mem}(\mathcal{S})$ implies $\mathsf{safe\text{-}mem}(\mathcal{S}')$ for the resulting state $\mathcal{S}'$, it is sufficient to determine whether the effective type τ_l of the object accessed by l is permitted to access an object of type τ_v: in short, whether $\tau_l \triangleright \tau_v$.

In summary: SafeType determines the presence or absence of violations in a program P by examining each assignment statement in P independently. If every such statement assigns an expression with a type τ_e to an lvalue with a type τ_l, and $\tau_l \triangleright \tau_e$, then the program does not violate the C standard's type-based restrictions on memory. If, for any statement, $\tau_l \triangleright \tau_e$, then a violation may occur. This approach is in general context- and flow-insensitive: there are no dependencies between individual assignments. The necessity of on-demand flow-sensitive queries to determine the type of some expressions is described in Section 4.3.

[1] It is for this reason that typecasts are isolated to assignment statements in our abstract syntax.

Chapter 4

A Prototype Implementation of SafeType

This section describes a prototype implementation of the approach to an analysis that identifies code that makes type-based alias analysis unsafe presented in Chapter 3. Section 4.1 describes the necessary inputs to the SafeType analysis and the representation of types. Section 4.2 describes the basic flow-insensitive analysis. Section 4.3 explains the need for a flow-sensitive analysis to correctly handle void pointers and outlines that analysis. Section 4.4 extends that analysis to be fully context-sensitive. Section 4.5 provides a small example illustrating the flow-sensitive and context-sensitive aspects of SafeType.

4.1 Preliminaries

In addition to the abstract syntax tree of a program P, SafeType requires data flow information for each function in P. For a function f, let $S(f)$ be the set of statements in f, augmented with a dummy statement s_0 representing the entry point of f. For a use of a variable v in a statement $s \in \mathcal{S}(f)$, the use-def function $\mathcal{D}(v, s)$ is the subset of statements in $S(f)$ that define a value for v such that this value may reach s without an intervening assignment to v. These definitions may occur as the result of an explicit assignment, or as the result of a side-effect of a function call. They may also represent the pre-existing value of v at the entry point of the function. The implementation of SafeType takes advantage of the factored single static assignment (SSA) representation generated by the XL compiler. In principle, the use of factored SSA is not necessary. The required information could be obtained from another form of SSA, from def-use and use-def chains, or by traversals of the abstract syntax tree.

SafeType also requires the construction of the static call graph of P. The call graph for a program P is a directed multi-graph $C_P = (V, E, \mathit{Call})$, where V is the set of functions, E is the set of call edges, and Call is the set of call sites within those functions. Each edge $(f, g, c) \in E$ represents a possible call from a function call statement $c \in \mathit{Call}$ in function

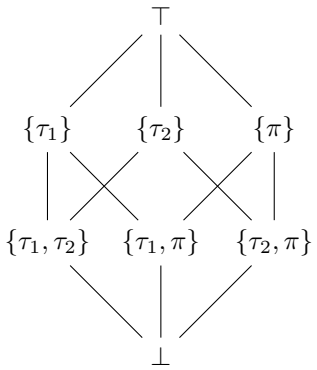

Figure 4.1: Type Lattice

f to function g, where $f, g \in V$. Cycles in the call graph represent function recursion; a self-recursive function will be represented by an edge (f, f, c). Multiple edges may exist from a function f to a function g if f calls g from multiple distinct call sites. Multiple edges may also exist from a call site c in a function f if the call site may call more than one function (due to the use of function pointers).

Let T_C be the set of C types present in a program. Let T_{P_f} be the set of placeholder types present in a function f, as defined in Section 4.4. For a function f, let $T_f = T_C \cup T_{P_f}$ be the set of types present in f. Let the power set of T_f be ordered such that for two sets $A, B \in 2^{T_f}$, $A \leq B$ iff $A \supset B$. The result is a lattice $\mathcal{L}(f)$ bounded by the null set ($\top$) and by T_f itself ($\bot$). Each element of this lattice represents a subset of T_f. The lattice for a program with two C types τ_1 and τ_2 and a placeholder type π can be seen in Figure 4.1.

4.2 Flow-Insensitive Analysis

Section 3.4 justifies an approach in which each assignment statement is inspected to determine whether it may create a pointer in memory that cannot be legally dereferenced. SafeType examines four types of assignment statements. Each statement generates one or more constraints.

- **Explicit assignments:** for an lvalue l and an expression e, where $\Gamma \vdash l : \tau_l$ and $\Gamma \vdash e : \tau_e$, an assignment of the form $l = e$ generates the constraint $\tau_l \triangleright \tau_e$.
- **Function calls:** for a function f with type $\langle \tau_R, \{\tau_{f_1}, \ldots, \tau_{f_n}\}\rangle$ and a set of expressions $e_1, \ldots, e_n$, where $\Gamma \vdash e_i : \tau_{e_i}$, a function call of the form $f(e_1, \ldots, e_n)$ generates the set of constraints $\{\tau_{f_i} \triangleright \tau_{e_i} \mid 1 \leq i \leq n\}$.
- **Return statements:** for an expression e evaluated in a function f with return type τ_R, where $\Gamma \vdash e : \tau_e$, a return statement of the form `return` e generates the constraint $\tau_R \triangleright \tau_e$.

```
1  int foo(int *arg) {
2   return *arg + 1;
3  }
4
5  void main() {
6   int i = 1;
7   double *dp;
8   i = foo(&i);
9   dp = (double *) &i;
10  *dp = 1.0;
11 }
```

```
return *arg + 1   ⟶   int ▷ int
          i = 1   ⟶   int ▷ int
       foo(&i)    ⟶   int* ▷ int*
    i = foo(&i)   ⟶   int ▷ int
  (double *) &i   ⟶   double* ▷ int*
dp = (double *) &i  ⟶  double* ▷ double*
      *dp = 1.0   ⟶   double ▷ double
```

Figure 4.2: Constraint Generation

Listing 4.1: Non-compliant

```
1  void *vp;
2  int i;
3  double *dp;
4
5  vp = &i;
6  dp = (double *) vp;
7  *dp = 1.0;
```

Listing 4.2: Compliant

```
1  void *vp;
2  double d;
3  double *dp;
4
5  vp = &d;
6  dp = (double *) vp;
7  *dp = 1.0;
```

Figure 4.3: Void Pointers

- **Type casts:** for a type τ_c and an expression e, where $\Gamma \vdash e : \tau_e$, a typecast of the form $(\tau_c)\ e$ generates the constraint $\tau_c \triangleright \tau_e$.

Each constraint is evaluated according to the definition of $\triangleright$ in Section 3.3. For any constraint that is not satisfied, a warning is produced to indicate that the assignment may make type-based alias analysis unsound. This warning includes the location of the statement in the source code, and the two types being compared.

An example of the constraint generation process can be seen in Figure 4.2. The source code on the left generates the set of constraints on the right. In line 10, `i` is accessed through an lvalue of type `double`. SafeType detects this violation and attributes it to the cast on line 9, due to the unsatisfiability of the constraint `double* ▷ int*`.

4.3 Flow-Sensitive Analysis

The preceding discussion implicitly assumes that each object in memory representing a pointer has an effective type. This is not, however, the case for objects of type `void *`. A void pointer can never be dereferenced without casting it to a non-void type, but is permitted to point to any object in memory. This flexibility creates a problem. Consider the example in Listing 4.1. The code includes three assignments: the assignment of `&i` to `vp`, the cast of `vp` to `double *`, and the assignment of the cast expression to `dp`. This code

violates the C standard. After line 6, `dp` points to `i`. When `dp` is dereferenced on line 7, an object of type `int` is accessed through an lvalue of type `double`, which violates the rules laid out in Definition 6. Conversely, in Listing 4.2, `dp` points to d, with type `double`, and the dereference in line 7 complies with the C standard.

As described above, the safe-mem invariant is maintained by ensuring that each pointer always points to an object that it is permitted to access. This approach is insufficient for void pointers, because they can point to any address and can never be dereferenced. As can be seen in Listings 4.1 and 4.2, the type of a void pointer, at the point at which it is cast to a non-void pointer, is a flow-sensitive property depending on previous assignments. Thus, it is necessary, in the specific case of void pointers, to extend SafeType to make it flow-sensitive.

Flow-sensitivity is attained in SafeType using on-demand flow-sensitive queries. Consider an expression e that uses a variable v with declared type τ. If $\tau =$ `void *`, then a flow-sensitive query is necessary to determine the actual type of the object to which v points.

Definition 10. Given a variable v and a definition d of that variable (such that $d \in \mathcal{D}(v, s)$ for some statement s), the types function $\mathcal{T}(v, d)$ is a lattice element representing the possible types of the value assigned to v by definition d. This section describes the implementation of $\mathcal{T}$ in the case where d is an assignment statement. Section 4.4 describes the implementation of $\mathcal{T}$ in the case where d is a function call statement or a function entry point.

Definition 11. Given a variable v and a statement s, the flow-sensitive query function $\mathcal{F}(v, s)$ is defined as follows:

$$\mathcal{F}(v, s) = \bigvee_{d \in \mathcal{D}(v,s)} \mathcal{T}(v, d)$$

The definition of $\mathcal{T}(v, d)$ is straightforward for assignment statements. Given a statement $v = e$, where $\Gamma \vdash e : \tau_e$, $\mathcal{T}(v, v = e) = \{\tau_e\}$. If e also uses a void pointer, additional flow-sensitive queries may be necessary to determine τ_e. SafeType uses a worklist algorithm and caches results to enable flow-sensitive queries to be performed efficiently.

4.4 Context-Sensitive Analysis

The definition of $\mathcal{T}$ in Section 4.3 is sufficient for void pointers that are defined and used entirely within a single function. A flow-sensitive query $\mathcal{F}(v, s)$ for a variable v of type `void *` used in a statement s may require interprocedural dataflow in two situations:

- A call site defines v, and that definition reaches s.
- v is live upon entry to the function, and the entry definition reaches s.

To handle these cases, SafeType uses two ideas: placeholder types, and transfer functions. Placeholder types are used to represent the unknown types of void pointer variables which are live upon function entry. Each placeholder type is defined in the context of a single function. For a function f, placeholder types are used in two cases:

1. A formal parameter v_p has type `void *`. For any call to f, the type of the object to which v_p points depends on the expression assigned to that parameter at the call site.
2. A global variable v_g used in a function has type `void *`. For any call to f, the type of the object to which v_g points depends on the value of that global variable at the point immediately preceding the call site.

Each function f has a set of placeholder types T_{P_f}. Let V_p be the set of formal parameters of f with type `void *`. Let V_g be the set of global variables with type `void *` that are used or defined in f, or in a function called by f. For each variable $v \in V_g \cup V_p$, the placeholder type $\pi_v \in T_{P_f}$ represents the type of v for an arbitrary call to f.

Transfer functions summarize the effects of a function call at a call site. Unlike points-to analysis, which depends on the value of each pointer and is not amenable to succinct summarization [31], SafeType concerns itself with types. As a result, it can represent each function in a compact form, and use that summary at each individual call site to efficiently achieve full context-sensitivity. For a function f, the transfer function $T(f)$ is a function $A \to (\tau_R, C, E)$, where:

- A is an assignment of types to the placeholder types in f, such that for each input variable $v \in V_g \cup V_p$, $A(v)$ is a type τ_{in_v}.
- τ_R is the return type of F.
- C is a set of delayed constraints of the form $\tau_1 \triangleright \tau_2$, where τ_2 includes at least one placeholder type $\pi \in T_{P_f}$.
- E is a map representing the exit types of each global variable used or defined in f, such that for each input variable $v \in V_g$, $E(v)$ is a type τ_{out_v}.

The transfer function for a function f is constructed while performing the analysis of f:

- **Return type:** If $S_R = \{\texttt{return}\ e_1, \ldots, \texttt{return}\ e_n\}$ is the set of return statements in f, and for each e_i, $\Gamma \vdash e_i : \tau_i$, then
$$\tau_R = \bigvee_{s \in S_R} \tau_i$$
- **Delayed Constraints:** If a constraint $\tau_1 \triangleright \tau_2$ is generated such that τ_2 is a placeholder type, it cannot be evaluated outside a particular calling context. Instead, the constraint is added to C, and evaluated at each call site of f independently.

- **Exit types:** For each variable v in V_g, $E(v) = \mathcal{F}(v, s_{\text{exit}})$, where s_{exit} is a dummy node representing the exit of f.

Each of the three outputs of a transfer function may include one or more placeholder types. Where this is the case, each placeholder type is replaced with the actual type of the corresponding input argument at the call site. The type of a formal parameter is available directly from the call statement. When the type of a global variable is required, a flow-sensitive query is performed for that variable, starting at the call site.

The use of transfer functions makes it possible to define the types function $\mathcal{T}$ for function call statements and function entry points. Given a statement $f(e_1, \ldots, e_n)$ which defines a variable v, if $\Gamma \vdash e_i : \tau_i$ and $T(f) = (\tau_R, C, E)$, then $\mathcal{T}(v, f(e_1, \ldots, e_n)) = E(v)$. For a function entry point d that defines v, $\mathcal{T}(v, d) = \pi_v$.

To ensure that the transfer function for a called function f is available at its call site in each calling function g, SafeType uses a post-order traversal of the call graph to generate the order in which each function is analyzed. However, self-recursive or mutually recursive functions create cycles in the call graph and make a post-order traversal impossible. To solve this problem, SafeType collapses strongly connected components of the call graph into a single node, and uses a post-order traversal on the modified call graph (which is cycle-free).

Within strongly connected components, transfer functions may have cyclic dependencies. SafeType therefore uses a worklist algorithm to find a fixed point for the transfer functions. This algorithm works as follows. Initially, each transfer function is initialized to be blank: $\tau_R = \top, C = \emptyset$, and for all $v \in V_g$, $E(v) = \top$. Each function in the strongly connected component is analyzed. Whenever the analysis of a function causes its transfer function to change, the callers of that function are inserted into the worklist for reanalysis. This continues until the worklist is empty, meaning that a fixed point has been reached.

Termination is addressed in three ways. First: define the size of a transfer function $T(f)$ as the sum of the number of possible return types in τ_R, the number of delayed comparisons in C, and the total number of possible exit types in E. The size of a transfer function is monotically increasing over the course of the analysis: return types and exit types may increase in size, and new delayed comparisons may be added, but existing types and comparisons are never removed. If T_f is the (finite) set of types present in f and T_{P_f} is the (finite) set of placeholder types present in f, then the maximum size of a transfer function $T(f)$ is reached when the return type $\tau_R = \bot$, $E(v) = \bot$ for all $v \in V_g$, and a delayed constraint $\tau_1 \triangleright \tau_2$ exists for each $\tau_1 \in T_f$, $\tau_2 \in T_{P_f}$. Because the size of each transfer function monotonically increases towards a finite maximum, termination is guaranteed. Second: to ensure termination in practice, an iteration count is maintained. If the analysis of a strongly connected component exceeds a set number of iterations (which scales with the size of the strongly connected component), a warning is produced. Each transfer function in

the strongly connected component is then conservatively approximated such that $\tau_R = \bot$, $E(v) = \bot$ for all $v \in V_g$, and $C = \emptyset$. Third: as described in Section 5.2, it is empirically rare that this algorithm causes a function to be analyzed more than twice. Although the conservative approximation escribed above could cause spurious warnings to appear at each call site of a function, in practice the iteration limit is simply precautionary and has no impact on the results of the analysis.

4.5 Example

Consider the following (highly contrived) code:

Listing 4.3: Context-sensitive Example

```
1  void *g_in, *g_out;
2
3  void *foo(void *p_in, void *cond) {
4   g_out = p_in;
5   int i = *(int *) cond;
6   if (i)
7    return p_in;
8   else
9    return g_in;
10 }
```

- `foo` may return either `p_in` or `g_in`. The return type of `foo` is therefore the lattice element $\pi_{\texttt{p\_in}} \vee \pi_{\texttt{g\_in}}$. The values of these placeholder types will vary between call sites of `foo`.
- The only global variable defined in `foo` is `g_out`. The exit type of `g_out` is $E(\texttt{g\_out}) = \pi_{\texttt{p\_in}}$. The global variable `g_in` is used, but not defined, in `foo`. Its exit type is $E(\texttt{g\_in}) = \pi_{\texttt{g\_in}}$.
- There is one delayed constraint in `foo`, arising in line 5: `int *` $\triangleright \pi_{\texttt{cond}}$. At each call site of `foo`, that constraint must be evaluated to verify the safety of casting `cond` to an `int *`.

The combination of the return type, the exit types, and the delayed constraints of `foo` are sufficient to create a transfer function that precisely characterizes the behaviour of `foo`. Using that transfer function, the effects of a call to `foo` can be determined for any number of call sites without requiring any further analysis of `foo`.

Chapter 5

Experimental Evaluation

This experimental evaluation uses a 64-bit, 32-processor, 2.3 GHz POWER 5 machine with 640 GB of memory running AIX 6.1.7.15. The experiments evaluate SafeType on C benchmarks from SPEC CPU2006. Additionally, it evaluates SafeType on Python 2.7.5 and GNU Emacs 24.3 to increase the number of large programs in the test set. Other large programs were considered, but were incompatible either with the XL compiler (for example, the Linux kernel) or with the Power architecture (for example, WINE). Lines of code are counted using the open-source tool CLOC. For emacs and python, lines of code written in Emacs Lisp and Python, respectively, are subtracted from the overall total, because SafeType is only applicable to the portion of a program written in C.

During compilation, compiler flags are chosen to disable other optimizations and minimize the amount of work done outside SafeType. This is done to reflect the expected use case of SafeType. To speed development, programs are typically compiled and tested with a minimal set of optimizations. After a program has been written and tested, higher levels of optimization are used to create a production version of the program. SafeType is best used during the transition from development to optimization, to verify that it is safe to use type-based alias analysis to enable the transformations that are turned on at higher levels of optimization. The actual compilation of the optimized version of a program should not use SafeType. Instead, the optimized compilation will benefit from being able to safely use a type-based alias analysis to enable other optimizations.

5.1 Output

Table 5.1 contains the experimental results for the prototype implementation of SafeType. For each benchmark, four counts are listed. The **Total** column lists the number of possible violations flagged for each benchmark. These violations are classified into three disjoint sets: true positives, necessary false positives, and implementation-based false positives. The **True** column represents true positives: statements reported as potential violations which do

Table 5.1: Violations Detected by SafeType

Name	Total	True	Nec	Impl
400.perlbench	52938	3463	166	49309
401.bzip2	14	1	0	13
403.gcc	16463	0	2	16461
429.mcf	13	0	0	13
433.milc	179	0	0	179
445.gobmk	10255	0	0	10255
456.hmmer	53	0	0	53
458.sjeng	321	0	0	321
462.libquantum	0	0	0	0
464.h264ref	2721	0	0	2721
470.lbm	57	30	0	27
482.sphinx3	1149	0	0	1149
emacs	74402	0	0	74402
python	16554	9089	0	7465

Listing 5.1: Necessary False Positive

```
1  int i;
2  int *ip = &i;
3  double *dp = (double *) ip;
4  ip = (int *) dp;
5  *ip = 1;
```

violate the C standard's type-based restrictions on memory access. These are the statements that SafeType is designed to detect. If a program does not have any true positives, it is safe to use a type-based alias analysis while compiling that program.

The prototype implementation of SafeType also produces a number of false positives: assignments that are flagged as potentially unsafe, but that are actually safe. These false positives are subdivided into two categories: necessary false positives, and implementation-based false positives. A false positive is necessary if it is caused by an inherent limitation of the SafeType analysis. For example, consider the code in Listing 5.1. The `int` pointer `ip` is cast to a `double` pointer, cast back to an `int` pointer, and then dereferenced. Because the value of `i` is never accessed through the `double` pointer, this code does not violate the C standard's type-based restrictions. However, SafeType works by proving that non-null pointers in memory are always safe to dereference. Thus, SafeType must conservatively emit a warning for lines 3 and 4 of Listing 5.1. These cases are sufficiently rare that the substantial increase in tractability justifies the small decrease in precision. Necessary false positives are listed in column **Nec** of Table 5.1.

In addition to necessary false positives, the current implementation of SafeType also produces a number of implementation-based false positives, listed in column **Impl** of Table 5.1. SafeType requires data-flow and call-graph information to accurately perform flow-

and context-sensitive analysis. This information is not available until the optimization phase of the compilation, after the source code has already been converted into an internal representation. This internal representation is non-language-specific, and does not perfectly preserve type information. As a result, some type information is lost. To remain sound, it is therefore necessary for SafeType to conservatively emit a warning in cases where the missing type information is necessary to prove correctness. This requirement is a limitation of the prototype implementation of SafeType, rather than the underlying analysis.

SafeType automatically flags many false positives as implementation-based during the course of its analysis. The remaining cases are categorized using a combination of manual inspection and automated post-processing scripts.

5.1.1 Case Studies

This section highlights a selection of true positives and necessary false positives.

Python and Perl: As mentioned in Chapter 1, the implementation of Python objects in versions of Python prior to 3.0 violated the C standard's restrictions on memory access, and the internal data structures representing Perl's variables are accessed in such a way as to violate aliasing rules in `400.perlbench`. As can be seen in Table 5.1, SafeType correctly identifies these violations. These violations typically take the form of a struct of one type being cast to a struct of another type. In addition to the large number of true positives, `400.perlbench` also includes an example of another important type of necessary false positive. As seen in Listing 5.2, `400.perlbench` uses tagged void pointers to represent scalar values. The `SV` struct includes a void pointer (`sv_any`) that represents a value, and a tag (`sv_flags`) to indicate the type of the variable being pointed to. The value of `sv_flags` is used in control-flow statements to determine the type of `sv_any`, as in line 9 of Listing 5.2. This code construct causes difficulties for SafeType. As a result of the test in line 9, the block beginning at line 10 may be able to safely cast `sv_any` to a particular type. However, to determine if such a cast is safe, SafeType would have to examine the condition in line 9, and then prove the existence of an invariant relationship between that condition and the type of the object pointed to by `sv_any`. In the general case, computing such invariants is infeasible. As a result, SafeType must conservatively emit a warning for a cast that occurs within the block beginning on line 10.

gcc: gcc contains a real-life instance of the example provided in Listing 5.1. Listing 5.3 demonstrates the situation. In `cfg.c`, gcc manages a pool of structs representing basic blocks. In the functions `expunge_block_nocompact` and `alloc_block`, unused basic blocks are added to and removed from a linked list. To save memory, gcc uses a pointer variable that already exists as a member of the `basic_block` struct to store the links in the linked list. However, the member in question is defined as a pointer to a different type of struct,

Listing 5.2: Necessary False Positive in 400.perl

```
1  struct SV {
2      void* sv_any;        /* pointer to something */
3      U32   sv_refcnt;     /* how many references to us */
4      U32   sv_flags;      /* what we are */
5  };
6
7  struct SV sv;
8  ...
9  if (sv->sv_flags == ...) {
10   ...
11 }
```

Listing 5.3: Necessary False Positive in 403.gcc

```
1  struct basic_block_def {
2    ...
3    struct edge_def *succ;
4    ...
5  };
6
7  struct basic_block_def *block, *head;
8  block->succ = (struct edge_def *) head;
```

the `edge_def` struct, which represents an edge between basic blocks. SafeType identifies the assignment in line 8 of Listing 5.3 as a violation. However, because the value of block->succ is never dereferenced, no actual violation occurs.

lbm: SafeType identifies a true type violation in `470.lbm` that, to our knowledge, has not been previously reported. The header file `lbm_1d_array.h` defines a macro `SET_FLAG`. A use of that macro expands to the code in Listing 5.4. The variable `grid` is an array of `double`. The address of a member of `grid` is taken in line 4. That pointer is cast, first to `void *`, then to `int *`. Finally, the new `int *` value is used to set a flag bit. That use constitutes a modification of a `double` object in memory by an lvalue of type `int`; as described in Chapter 3, such a modification is a violation of the C standard. Although we are not aware of a compiler that takes advantage of the incorrect invariant generated by a type-based alias analysis in this case, a sufficiently aggressive optimizing compiler could theoretically eliminate or reorder uses of `SET_FLAG` in a way that does not preserve the semantics of the program.

5.2 Performance

Table 5.2 shows the time necessary to run SafeType on each benchmark. Column **LoC** is the number of lines of code in each benchmark. Column **Total** is the time taken to compile the benchmark. The following two columns show the time spent inside SafeType: first as an

Listing 5.4: Violation in 470.lbm

```
1 double grid[N];
2 int flag = 1 << M;
3 {
4   int* const _aux_ = ((int*) ((void*) (&(grid[...]))));
5   (*_aux_) |= flag;
6 }
```

Table 5.2: Performance of SafeType

Name	**LoC**	**Total (s)**	**SafeType (s)**	**%**
400.perlbench	138480	3939.0	3819.9	96.9
401.bzip2	5882	10.6	1.7	15.9
403.gcc	387777	650.9	304.8	46.8
429.mcf	1669	2.7	0.1	3.5
433.milc	9746	15.5	1.1	7.2
445.gobmk	157792	109.6	25.8	23.5
456.hmmer	20793	29.1	1.8	6.1
458.sjeng	10630	13.8	1.0	7.2
462.libquantum	2732	4.2	0.2	4.3
464.h264ref	36162	74.8	20.7	27.7
470.lbm	1006	2.9	0.15	5.1
482.sphinx3	13240	22.2	1.6	13.9
emacs	326779	338.2	90.1	26.7
python	514345	390.8	263.9	67.5

Table 5.3: Timing Breakdown

	Flow-Sensitive			**Context-Sensitive**	
Name	**Data (%)**	**FSQuery (%)**	**Other (%)**	**Rec Time (%)**	**Rec Freq (%)**
400.perlbench	0.4	95.3	4.3	99.6	81.3
401.bzip2	59.8	21.0	19.2	0.0	0.0
403.gcc	22.3	38.6	39.1	74.3	75.5
429.mcf	38.6	10.9	50.5	2.0	2.8
433.milc	33.6	28.2	38.2	0.0	0.0
445.gobmk	18.7	32.0	49.3	76.3	80.9
456.hmmer	38.1	20.4	41.5	3.8	2.9
458.sjeng	27.0	18.6	54.4	10.0	4.4
462.libquantum	43.7	10.9	45.4	9.8	11.3
464.h264ref	20.9	50.1	29.0	2.5	6.7
470.lbm	24.1	23.1	52.8	0.0	0.0
482.sphinx3	24.3	11.9	63.8	1.6	2.9
emacs	16.3	13.8	69.9	75.9	70.3
python	7.7	55.8	36.5	91.8	82.4

absolute value, and then as a percentage of the total compilation time. As mentioned above, compilation time is measured for a compilation with optimizations turned off. With less compilation time spent on other optimizations, the percentage of compilation time spent inside SafeType is larger.

Table 5.3 categorizes the time spent inside SafeType in two ways. The first three columns divide the time spent inside SafeType into three categories that demonstrate the behaviour of the flow-sensitive analysis. **Data** is the percentage of the execution time of SafeType spent building the data structures necessary to analyze data flow. In a normal compilation, the time spent building these data structures would normally be amortized over each analysis that uses them. However, as described above, no other analysis using these data structures was enabled during the evaluation. **FSQuery** is the percentage of the execution time of SafeType spent performing flow-sensitive queries as described in Section 4.3. **Other** is the percentage of the execution time of SafeType that does not fall into one of the previous two categories.

The remaining columns of Table 5.3 illuminate the behaviour of the context-sensitive analysis. **Rec Time** is the percentage of the execution time of SafeType spent analyzing recursive functions: that is, functions that are part of a strongly connected component in the call graph. This category includes both self-recursive and mutually recursive functions. As a point of comparison, **Rec Freq** is the percentage of functions in the program that are recursive, measured as a static count.

The values in Tables 5.2 and 5.3 can be contextualized by an examination of the theoretical complexity of the SafeType analysis. The complexity of the non-flow-sensitive, non-context-sensitive component of SafeType is $O(n)$ in the size of the program: it examines each statement once and performs an amount of work proportional to the size of the abstract syntax tree for that statement.

The complexity of the context-sensitive component is more complicated to characterize. As described in Section 4.4, recursive functions must be analyzed until their transfer functions reach a fixed point. Each function must be analyzed at least once. If the analysis of a function F updates the transfer function for F, then the callers of F must be reanalyzed using the more precise transfer function. Thus, after the first analysis of F, F will be reanalyzed only if the previous analysis of F updated the transfer function for F and (directly or indirectly) caused an update to a callee of F. For a strongly connected component of the call graph containing $O(n)$ functions, each of which has $O(m)$ arguments, the pathological worst-case behaviour of the fixed point computation is $O(nmt)$ reanalyses of each function, where t is the number of types in the program. In practice, however, the worst-case behaviour does not occur. Figures 5.1 and 5.2 demonstrate the empirical behaviour of SafeType on the test programs. Each bar in Figures 5.1 and 5.2 represents the

total number of recursive functions that were analyzed a certain number of times, across the entire set of benchmarks. For example, the first bar of Figure 5.1 shows the number of recursive functions that were only analyzed once. Within each bar, the total number of functions is partitioned by colour to represent the number of functions from each individual benchmark program. The difference between the highest and lowest values makes it impossible to display all of the results in a single graph. Thus, Figure 5.1 shows the precise number of functions with iteration counts 1 to 5, and collects the remaining functions in the 6+ bar. Figure 5.2 shows the precise iteration counts for the functions in the 6+ bar of Figure 5.1. Functions that are neither self-recursive nor mutually recursive are always analyzed exactly once, but are omitted from Figure 5.1 to avoid compressing the scale.

More than half of all recursive functions are analyzed a single time. In the majority of test programs, no function must be analyzed more than twice. The benchmark `445.gobmk` includes three functions that must be analyzed three times each. `emacs` includes a number of functions that must be analyzed three times, and two that must be analyzed four times each. `gcc` includes functions that must be analyzed up to five times. In each of these cases, the functions that must be analyzed more than twice are part of large mutually recursive cycles. Amortized across each of the functions in a strongly connected component of the call graph, the average number of iterations per function necessary to find a fixed point in these cases is always two or less.

This is not true for `400.perlbench` or `python`. Figure 5.1 shows that the benchmarks `400.perlbench` and `python` both have a significant number of functions that must be analyzed six or more times. The distribution of these functions is shown in Figure 5.2. As in Figure 5.1, higher numbers of iterations are substantially less commmon than lower numbers. However, `400.perlbench` includes a number of functions that require many iterations to find a fixed point, including one function that requires 33 iterations to find a fixed point. Empirically, functions that require a large number of iterations occur in strongly connected components of the call graph that are large and contain a large number of calls through function pointers. It is possible that the number of iterations could be reduced in these cases by carefully ordering the reanalysis of functions, or by improving the precision of the points-to sets of function pointers. Regardless, amortized over an entire strongly connected component, the average number of iterations per function necessary to find a fixed point in `400.perlbench` or `python` never surpasses 3.5. Thus, the process of finding a fixed point for recursive transfer functions contributes only a constant factor to the overall complexity of SafeType.

The complexity of the flow-sensitive analysis is highly implementation-dependent. As can be seen in Table 5.2, the benchmarks for which SafeType requires the greatest percentage of compilation time are also the benchmarks for which FSQuery represents the greatest

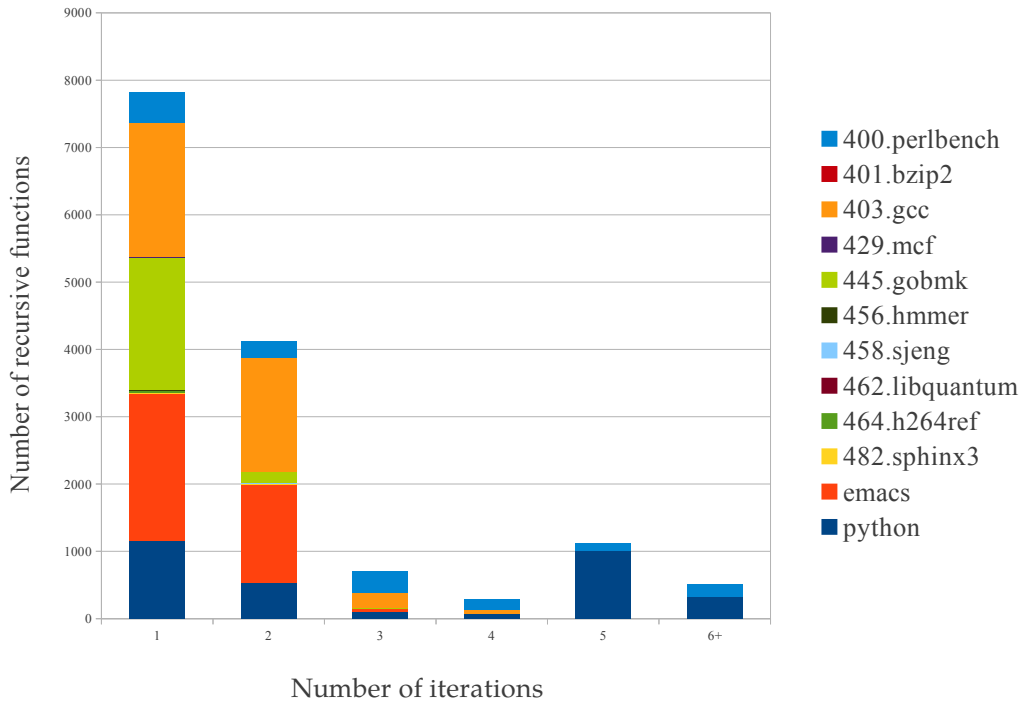

Figure 5.1: Iterations necessary to find a fixed point

percentage of the execution of SafeType. Further work may be necessary to improve the performance of the flow-sensitive analysis. However, even the current prototype scales to programs of hundreds of thousands of lines of code.

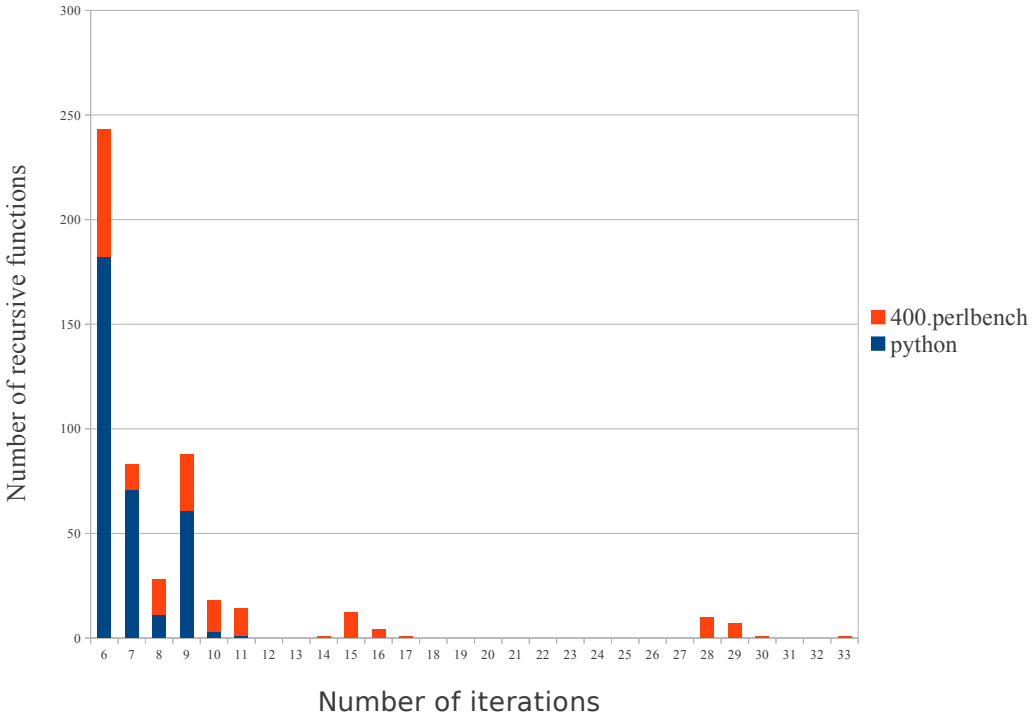

Figure 5.2: Iterations necessary to find a fixed point

Chapter 6

Related Work

This chapter summarizes related work on type-based alias analysis and points-to analysis.

6.1 Points-to Analysis

A closely related idea to alias analysis is points-to analysis. Where an alias analysis determines if two memory references point to the same location, a points-to analysis approximates, for each pointer in the program, the set of locations to which it could point at runtime [8]. Thus, a points-to analysis can be used to determine aliasing information, although the converse is not in general true.

A precise points-to analysis with multiple-level pointers is NP-hard even in restricted cases [19]. Such an analysis is undecidable for programs with dynamically allocated memory [18]. Many frameworks for approximating points-to information have therefore been proposed [16]. These frameworks may be classified based on the approximations they use and the degrees of sensitivity they emphasize.

The most scalable points-to analyses are flow- and context-insensitive. These analyses build up a points-to graph based on the set of assignments present in the analyzed code. Within this category, a division exists between unification-based and inclusion-based analyses, which differ based on how they represent assignments. In a unification-based analysis, such as Steensgaard's [29], an assignment such as `x = y` will unify the nodes representing `x` and `y` in the points-to graph. In an inclusion-based analysis, such as Andersen's [1], an assignment such as `x = y` will establish a subset constraint $\texttt{x} \subseteq \texttt{y}$, represented as an edge in a constraint graph. The dynamic transitive closure of the graph is then computed.

Steensgaard's unification-based analysis runs in nearly linear time ($O(N\alpha(N,N))$, where α is the inverse Ackermann's function) and scales to millions of lines of code. However, it is relatively imprecise. The scalability of inclusion-based alias analysis depends primarily on its ability to quickly compute the dynamic transitive closure of the constraint graph. The greatest gains come from detecting and collapsing cycles, which will have identical solutions.

Hardekopf and Lin present a state-of-the-art analysis [13] that scales to programs containing over a million lines of code.

Other approximations can also be made. Das presents a "one-level flow" analysis that is unification-based, except in the case of single-level pointers, under the assumption that multi-level pointers are less important in C programs [5]. Das' evaluation indicates that this analysis is nearly as scalable as Steensgaard's, and nearly as precise as Andersen's.

A more precise result can be obtained using a flow-sensitive analysis. The scalability of traditional flow-sensitive alias analyses has been limited by the need to propagate points-to information through every node in the control flow graph, which is prohibitively expensive both with respect to execution time and with respect to memory usage. One solution is to perform the analysis on a sparse graph instead of the full control-flow graph. Hardekopf and Lin present a semi-sparse analysis in which top-level (non-address taken) variables are represented in SSA form, and other variables are not [14]. They also present an improved sparse analysis in which a preliminary flow-insensitive points-to analysis is used to convert address-taken variables into SSA form, and a flow-sensitive analysis is then performed on the sparse representation [15]. The latter approach scales to programs with over a million lines of code.

An alternative approach is due to Lhotak [21]. Lhotak observes that the precision benefit of a flow-sensitive points-to analysis derives from its ability to represent points-to relationships which are overwritten by strong updates and are not true at every point in a program. Because strong updates are usually only applicable when a points-to set is a singleton, Lhotak's analysis is flow-sensitive with respect to singleton points-to sets, and flow-insensitive elsewhere. This approach, implemented in the LLVM compiler, scales as well as the existing non-flow-sensitive approach and improves points-to precision for 98% of the program points that are improved by a full flow-sensitive analysis.

Each of the preceding flow-sensitive analyses is context-insensitive (although Hardekopf and Lin discuss the possibility of extending their analyses to be both flow and context-sensitive). Research has also been done on context-sensitive points-to analysis. Foster examines the consequences of adding context-sensitivity to Steensgaard and Andersen's analyses, and concludes that context-sensitivity improves the precision of unification-based analyses more significantly than it improves inclusion-based analyses [9]. Recent research in flow-insensitive context-sensitive points-to analysis has focused on the use of binary decision diagrams (BDDs) to compactly represent the large number of contexts that can be generated [3]. Zhu and Calman present a BDD-based context-sensitive approach that scales to tens of thousands of lines of code [32]. However, to do so they merge strongly-connected components in the call graph, effectively analyzing function calls within strongly connected components in a context-insensitive manner. Independently, Whaley and Lam present an

identical approach[30]. Lhotak shows that this approximation has a significant negative effect on precision when applied to pointer analysis in Java [22]; however, no such evaluation has been done for C.

Finally, work has also been done on analyses that are both flow and context sensitive. Landi and Ryder present an algorithm that differentiates contexts for single-level pointers, but may propagate information to extraneous call sites in the presence of multi-level pointers [20]. Emami achieves context sensitivity by re-analyzing a procedure for each of its calling contexts [8]. This approach ensures precision, but is only practical for small programs. Wilson and Lam present an analysis that creates *partial transfer functions* and must only re-analyze a procedure if the aliasing relations among its inputs do not correspond to any pre-existing partial transfer function for that procedure [31]. This approach scales up to programs with thousands of lines of code.

6.2 Safety Analysis of C

Much work has been done on the more general question of analyzing the safety of C programs. Of particular relevance are analyses that aim to detect illegal memory operations: dereferences of undefined pointers, dereferenced of freed memory, and array-bounds violations.

Valgrind is an open-source instrumentation framework for building dynamic analysis tools [24]. Memcheck, the default Valgrind tool, instruments each memory access to detect a variety of common errors, including (among others) invalid pointer dereference and the use of uninitialized values. However, Memcheck is not designed to detect memory accesses through pointers of incompatible type. The addition of this instrumentation leads to a 10-50 times slowdown in program execution. Thus, Valgrind is useful as a developer's tool to detect bugs, rather than being used to run production code.

Cyclone is a type-safe programming language derived from C [12]. Cyclone aims to preserve the low-level control of data representation and resource management available in C, while eliminating safety violations such as incorrect type-casts, buffer overruns, dangling pointer dereferences, and memory leaks. This safety is accomplished by dividing memory into regions and requiring that pointer variables be annotated with information about the region into which they point. Empirically, porting C code into Cyclone requires alterations in approximately 8% of the lines of code in a program. Compared to the original C program, a network or I/O bound Cyclone application runs with little to no overhead, but compute-intensive applications may be three times slower than the C version. This overhead comes from bounds-checks inserted into the code, as well as garbage collection and fat pointers.

CCured is a program transformation system that adds type-safety guarantees to existing C programs. CCured extends the type system of C to separate pointer types based on

their usage; it then attempts to use static analysis to verify that programs adhere to that type system. Run-time checks are inserted in cases where static analysis is insufficient: these checks include null checks, bounds checks, and type checks. CCured replaces manual memory management with a garbage collector and requires the programmer to annotate custom allocators. CCured also requires wrappers around calls to external library functions to preserve the metadata associated with runtime checks. Like Cyclone, CCured imposes little overhead on network- and I/O-bound applications, but causes a slowdown of 5 to 87% on most compute-intensive benchmarks, increasing to a ten times slowdown in pathological cases.

SafeCode is a compilation strategy for C programs that uses static analysis and run-time checking to ensure the accuracy of the points-to graph, call graph, and the available type information [6]. SafeCode uses *automatic pool allocation* to partition heap memory into fine-grained pools, while retaining explicit memory management (rather than introducing a garbage collector). SafeCode requires no code modifications and imposes less than 30% overhead. Unlike Cyclone and CCured, SafeCode does not guarantee the absence of dangling pointer references.

In comparison to each of the above, our SafeType algorithm limits its scope to consider exclusively type violations. In doing so, SafeType does not introduce any runtime checks, does not impose any overhead on execution, and does not require any modifications to existing code. The previous approaches are useful in cases where users wish to exchange performance for memory safety. However, the primary use of type information is in alias analysis, which in turn is used by various transformations to improve performance. Thus, it is useful to have a specialized analysis that a programmer can use to verify the safety of type-based alias analysis without incurring a performance penalty. Certain aspects of the static analysis of CCured and SafeCode resemble the flow-insensitive component of SafeType. However, they do not have any static analogue to SafeType's flow-sensitive, context-sensitive approach to pointers of type `void *`. Instead, CCured and SafeCode verify the safety of such pointers using runtime checks.

Chapter 7

Conclusion

Type-based alias analysis is a useful tool for enabling and improving the precision of program transformations in optimizing compilers. However, type-based alias analysis is unsafe when applied to programs that violate the C standard's type-based restrictions on memory access. As a result, programs are frequently compiled without the benefits of type-based alias analysis.

SafeType is a sound and scalable static analysis that identifies violations of the C standard's type-based restrictions on memory access. To do so, SafeType verifies that each non-null pointer in the program can be safely dereferenced. This verification uses a flow-insensitive analysis in the general case, augmented with flow-sensitive queries to safely analyze pointer variables of type `void *`. These flow-sensitive queries are made context-sensitive through the use of precise, compact transfer functions.

A prototype implementation of SafeType in the IBM XL C compiler was created and evaluated. Although the architecture of the implementation leads to a number of false positives, this prototype implementation demonstrates the utility of the SafeType approach by identifying previously unreported violations in the `470.lbm` benchmark of SPEC CPU2006. The prototype implementation scales to programs with hundreds of thousands of lines of code. Work is underway to improve the performance further for programs with call graphs containing strongly connected components with a large number of function pointers.

The description of the SafeType approach in this thesis, along with the discoveries made with the performance evaluation of the prototype implementation, are a solid basis for a robust and precise analysis that will enable the safe use of type-based alias analysis in commercial compilers.

Bibliography

[1] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 243–262, Orlando, Florida, USA, 2009.

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

[4] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Programming Language Design and Implementation (PLDI)*, pages 296–310, White Plains, New York, USA, 1990.

[5] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Programming Language Design and Implementation (PLDI)*, pages 35–46, Vancouver, British Columbia, Canada, 2000.

[6] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: enforcing alias analysis for weakly typed languages. In *Programming Language Design and Implementation (PLDI)*, pages 144–157, Ottawa, Ontario, Canada, 2006.

[7] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Programming Language Design and Implementation (PLDI)*, pages 106–117, Montreal, Quebec, Canada, 1998.

[8] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Programming Language Design and Implementation (PLDI)*, pages 242–256, Orlando, Florida, USA, 1994.

[9] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium (SAS)*, pages 175–198, Santa Barbara, California, USA, 2000.

[10] Alias Analysis. `http://gcc.gnu.org/news/alias.html`, 1998. [Online; accessed 1-March-2013].

[11] Rakesh Ghiya. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Programming Language Design and Implementation (PLDI)*, pages 47–58, Snowbird, Utah, USA, 2001.

[12] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, pages 282–293, Berlin, Germany, 2002.

[13] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Programming Language Design and Implementation (PLDI)*, pages 290–299, San Diego, California, USA, 2007.

[14] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Principles of Programming Languages (POPL)*, pages 226–238, Savannah, GA, USA, 2009.

[15] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Code Generation and Optimization (CGO)*, pages 289–298, Chamonix, France, 2011.

[16] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, Snowbird, Utah, United States, 2001.

[17] ISO 9899. Programming languages – C, 2000.

[18] William Landi. Undecidability of static analysis. *Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, December 1992.

[19] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem classification. In *Principles of Programming Languages (POPL)*, pages 93–103, Orlando, Florida, USA, 1991.

[20] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Programming Language Design and Implementation (PLDI)*, pages 235–248, San Francisco, California, USA, 1992.

[21] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Principles of Programming Languages (POPL)*, pages 3–16, Austin, Texas, USA, 2011.

[22] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *Transactions on Software Engineering and Methodology (TOSEM)*, 18(1):1–53, September 2008.

[23] LLVM 2.9 Release Notes. `http://llvm.org/releases/2.9/docs/ReleaseNotes.html`, 2011. [Online; accessed 1-March-2013].

[24] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation (PLDI)*, pages 89–100, San Diego, California, USA, 2007.

[25] David J. Pearce. Efficient field-sensitive pointer analysis for c. In *Program Analysis for Software Tools and Engineering (PASTE)*, pages 37–42, Washington, DC, USA, 2004.

[26] Gordon D Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, September 1981.

[27] Python enhancement proposal 3123, April 2007.

[28] August G. Reinig. Alias analysis in the DEC C and DIGITAL C++ compilers. *Digital Technical Journal*, 10(1):48–57, 1998.

[29] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages (POPL)*, POPL '96, pages 32–41, St. Petersburg Beach, Florida, USA, 1996.

[30] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Programming Language Design and Implementation (PLDI)*, pages 131–144, Washington DC, USA, 2004.

[31] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Programming Language Design and Implementation (PLDI)*, pages 1–12, La Jolla, California, USA, 1990.

[32] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Programming Language Design and Implementation (PLDI)*, pages 145–157, Washington, DC, USA, 2004.