Dependent Types for Assembly Code Safety

Matthew Thomas Harren

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-65
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-65.html

May 18, 2007

Dependent Types for Assembly Code Safety

by

Matthew Thomas Harren

B.S. (Cornell University) 2001
M.S. (University of California, Berkeley) 2004

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor George C. Necula, Chair
Professor Rastislav Bodík
Professor Leo Harrington

Spring 2007

The dissertation of Matthew Thomas Harren is approved:

Chair Date

Date

Date

University of California, Berkeley

Spring 2007

Dependent Types for Assembly Code Safety

Matthew Thomas Harren

Abstract

Dependent Types for Assembly Code Safety

by

Matthew Thomas Harren

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor George C. Necula, Chair

When proving that programs adhere to various safety and security properties, it is often useful to work with binary code instead of the high-level source code from which it was compiled. Proving safety for binary code means that we need not trust the compiler and that end users can check the safety property without access to the source code. But most of today's safety tools operate only on source code, where analysis and program transformation are easier. We need a way to take safety results that have been established for source code and propagate them to the binary level. This dissertation proposes such a system for certified assembly code, and shows how it can be used with three source-code safety tools that operate on existing C programs: CCured, Deputy, and Cqual.

A key feature of this system is a novel dependent type system suitable for assembly code. It is difficult to use dependent types safely in languages where heap values can be modified, because modifying a value can change the type of some other memory location.

We address this problem by limiting in-memory dependent types so that they only refer to other fields of the same object. We show that it is possible to safely update such an object by allowing memory to temporarily be in a "bad" state until enough fields have been written so that the object is again internally consistent. And we show examples from CCured and Deputy where such a type system is necessary.

The second key contribution of this dissertation is a type inference system for assembly code that has been generated by a "black box" compiler. This algorithm discovers the (possibly dependent) types of registers at each program point, and therefore reconstructs the invariants that were shown to hold during source-code verification. We use abstract interpretation of symbolic expressions for this inference, and discuss how to deal with pointer arithmetic.

Finally, we have an implementation of our verifier for CCured and Cqual. This verifier parses x86 assembly code generated by GCC and ensures that the program was correctly analyzed and instrumented by CCured or Cqual prior to being compiled, and that the object code has not been tampered with in a way that would affect type safety. We present experimental results for our verifier which show that verification can be done in an efficient manner, but certain C constructs such as complicated array index expressions are difficult to analyze.

Professor George C. Necula
Dissertation Committee Chair

Contents

List of Figures

Acknowledgments

Thank you first and foremost to my advisor George Necula for his continuous help and advice over the last six years. In addition to making considerable contributions to this dissertation, George has taught me an enormous amount about computer science, research, and careers. I greatly value our conversations and sailing trips.

Both the theory and implementation of this project rely on work done on the Open Verifier by Bor-Yuh Evan Chang, Adam Chlipala, Kun Gao, George Necula, and Robert Schneck. Their framework shielded me from much of the pain of assembly code, and their advice was invaluable. I thank Sumit Gulwani for his assistance with abstract interpretation and for other useful conversations. Jeremy Condit, Scott McPeak, and Wes Weimer were a pleasure to work with on CCured and many other projects, and Wes's dinners fueled many enjoyable evenings.

Dan Wilkerson provided a great deal of help for the Cqual verifier, by changing his implementation of Cqual to emit the annotations that I needed. His feature implementations always came faster and worked better than I could have hoped. Discussions with Dan, Rob Johnson, and the other Cqual developers helped me understand Cqual's algorithm and how it was applied.

I had a fantastic experience at Berkeley thanks to the people in this department. This includes work and play, and the members of the Open Source Quality group helped with both. Ras Bodik, Leo Harrington, and David Wagner asked insightful questions about this project during my Qualifying Exam.

Last but certainly not least, thank you to my family. My parents made me who I am today, and I'll always be grateful. And I thank Ann for her encouragement, for spending many hours sitting in airports, and for the future.

Chapter 1

Introduction

There are many ongoing efforts to design static analyses or instrumentation tools to ensure various safety and security properties of software. However, there is usually no independent way to ensure that the analysis or instrumentation tool was actually run on a given program. Since most of today's software security tools operate only on source code, a concerned user must obtain the source for the program in question, must run the tool himself, and is forced to trust that the tool and the compiler are working as advertised. In this dissertation, we describe our efforts to develop an independent verification strategy for static analyses and instrumentation tools.

A well-known example of the strategy that we advocate is the verification of type safety in Java and .NET bytecode. A compiler verifies that the original source code is type-safe, and uses this typing information to generate typed bytecode. The bytecode can then be checked for safety independently from the source code. We want to push this strategy to lower-level languages, such as assembly, and to allow more language-based enforcement

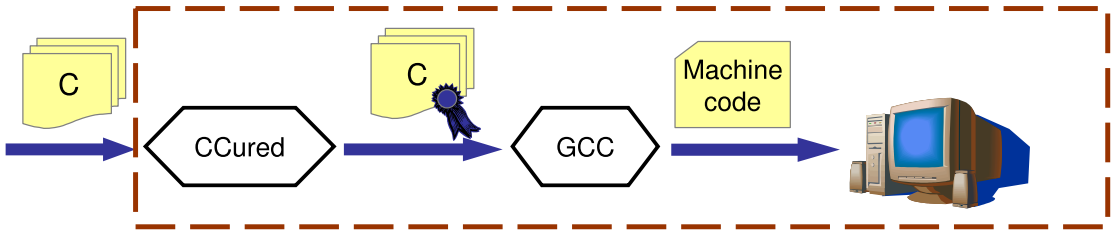

Figure 1.1: Existing CCured usage model. The dashed box shows the components that must be trusted.

tools to make use of it. Working at the assembly-language level makes our technique fit well with the common practice of distributing object code. Furthermore, it does not require the program source code, is applicable to more source languages, and eliminates the compiler from the trusted computing base.

Consider CCured, a source-to-source translator that guarantees type safety in legacy C code by inserting runtime checks before potentially unsafe operations [NCH+05]. Where necessary, it modifies data structures to accommodate metadata such as array bounds information. The resulting code is type safe and won't "go wrong" at run time.

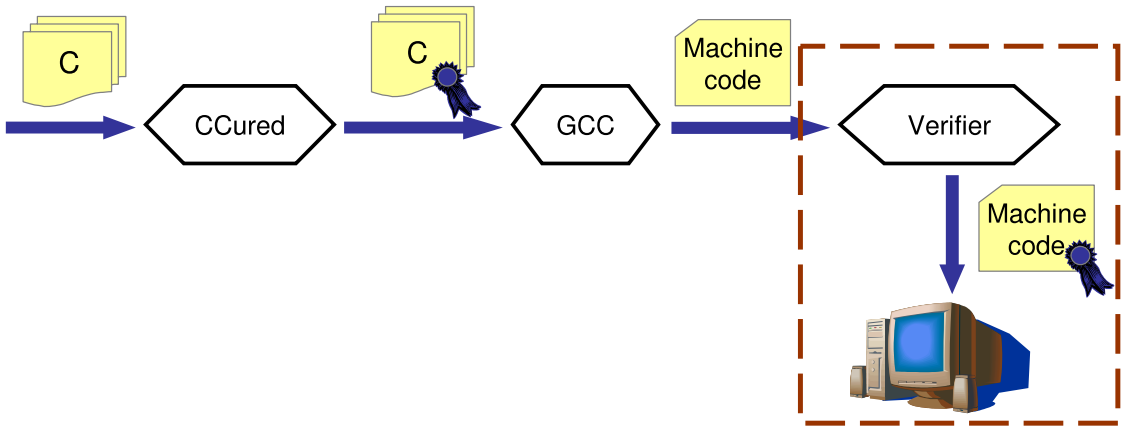

Figure 1.2: CCured usage model with machine-code verification. The dashed box shows the components that must be trusted.

But this code is then compiled with an off-the-shelf compiler that discards all of the typing information (Figure 1.1), so there is no way to tell if the resulting machine code is safe. We present in this dissertation a verifier (Figure 1.2) that can check that assembly code is safe without access to the source code. The two key contributions of this verifier are a *dependent type* system for assembly code that can encode the invariants of systems like CCured, and a *type inference* system for assembly code that can infer these invariants for code that has been instrumented by CCured.

Dependent types

A dependent type is a type that depends on a runtime value. A common example is an array whose length is stored separately (Figure 1.3): the type of the array pointer depends on the value of the length variable. CCured uses dependent types to relate the data in the original program (such as arrays) to the metadata it adds (such as array lengths). Traditional Typed Assembly Languages don't include dependent types [MWCG99], or don't allow dependently-typed data to be modified [XH01]. Chapter 2 discusses a type framework that includes dependent record types, allows dependent memory locations to be overwritten, and is suitable for use with assembly code, where we don't have the convenience of being able to add new operations to the language.

Type inference

In order to verify object code, we need some way to understand its behavior and relate its low-level instructions with the high-level safety policy we are trying to enforce. One way to do this is with a certifying, type-preserving compiler that emits annotations

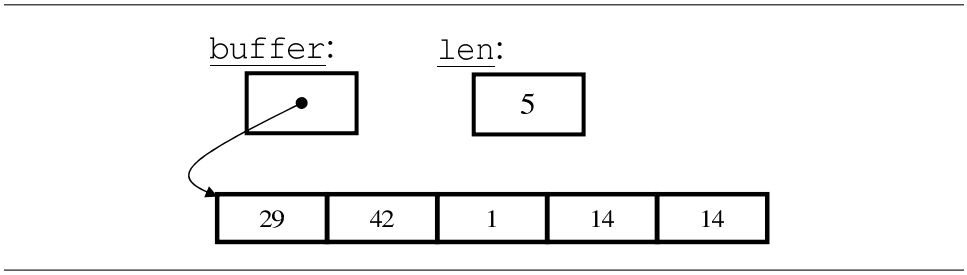

Figure 1.3: The variable `buffer` has the dependent type "array of `len` integers."

with the object code to partially explain the program's behavior. This approach is used by other typed assembly languages and proof-carrying code systems [MWCG99, NL98] as well as bytecode languages like Java and MSIL [LY97, GS01]. However, a special compiler may be too great a change to existing toolchains. Many safety tools for legacy code, such as CCured and Cqual [FFA99], operate on source code and use an off-the-shelf compiler to generate object code. It would take a significant amount of work to write a custom compiler that is aware of various source-level safety policies, would preserve this information during code generation, and contains all of the features and optimizations of existing compilers. In Chapter 3, we propose an assembly code analysis that can infer the types of Chapter 2 for code that was generated by a "black box" compiler.

Another alternative to our verification proposal would be to not check the source code at all. Waiting until the code is in object form before checking a safety property eliminates the redundant analysis step. But it is hard to analyze object code, and it is hard to report object code errors in ways that are helpful to the developer. The fact that the code has already been made safe at the source level means that verification should always succeed (unless there are bugs in the toolchain or the programs have been tampered with), so

error reporting is less important. Source code tools are also essential if you want to modify code, since binary-code instrumentation tools such as Purify and Valgrind [HJ91, NS07] have a significant performance cost. We let source code tools handle the hard parts — error reporting, instrumentation, whole-program analysis — so that object code analysis becomes a local verification problem.

There are a few issues that this thesis does not address. Our implementation works on assembly code, not binary. Disassembling binary code poses its own set of challenges that are orthogonal to the ones we tackle here. Our work on type safety also ignores allocation and deallocation. CCured uses the Boehm-Demers-Weiser conservative garbage collector [BW88] for deallocation, and we trust the garbage collector here as well. Finally, we do not address shared-memory multithreading. CCured is unsound for multithreaded code (unless all pointer values are guarded by locks, as in [Gro03]) and our type system is unsound for the same reasons.

We present our system of dependent types in Chapter 2, and explain how the type framework is parameterized on the source-code policy being checked. The main contributions of this chapter are a novel system for dependent record types in a low-level imperative language, and the outline of a soundness proof for the system. Chapter 3 presents a type inference system and discusses challenges related to pointer-indexing expressions and abstract interpretation of dependent types. We show how this type framework can be used with three source-code tools (CCured, Deputy, and Cqual) in Chapters 4, 5, and 6, and discuss the verifiers that we have implemented for CCured and Cqual.

Chapter 2

Dependent Types for Assembly Code

The first step in the assembly code verification is to design a type system that can encode invariants such as CCured's. This chapter describes a novel dependent type system suitable for assembly code. In addition to providing dependent types, our system addresses several difficulties that assembly code presents:

- Our type system is flow-sensitive, since registers and stack locations will be reused by the compiler.
- We support writes to dependent memory locations. Most dependent type systems forbid such writes, but they are needed by CCured. Working with assembly code makes this challenge harder since only one word can be written at a time.
- We support intraprocedural dependent type inference for optimized code.

Our type system is parameterized by a *type policy* that describes the invariants enforced by the safety tool you wish to use (CCured, for example). Factoring our type system in this way provides modularity and allows us to extend the system to different safety tools. It also allows us to focus our initial discussion on dependent types, without worrying about the specific safety policies we enforce. This chapter and the one following it describe the typechecking framework, while Chapters 4, 5, and 6 show type policies for three source-code tools: CCured, Deputy, and Cqual.

We begin the chapter by describing the operations that a type policy must support. We then present in Section 2.2 a flow-insensitive type system that supports dependencies among adjacent memory words. Our type system uses these flow-insensitive types for global variables and the heap. Sections 2.3–2.5 show how to use symbolic evaluation of each basic block to infer flow-sensitive types and dependencies for registers. We outline a proof of soundness for this framework in Section 2.6, and discuss related type systems in Section 2.7.

2.1 Type Policies

We assume that each type policy is described as a set of type constructors and associated operations such as subtyping. An n-ary type constructor applied to n expression arguments describes the type of a machine word, as described in the following sections.

A type policy consists of the following:

- A finite set $\mathbb{T}$ of type constructors C. These constructors are used to build policy-specific types for word-sized values. If a constructor has nonzero arity, it defines dependent types.

- A finite set $\mathbb{F}$ of fact constructors F. These are used to encode flow-sensitive information needed by the policy, such as "LessThan(x,y)" or "NotAliased(x,y)". Our typing context will include a set of facts that are known to be true at a given program point. This information could be encoded in the type constructors, but it's often more natural to store it separately.

- A subtyping relation `IsSubtype` for the types generated by the type constructors, and the associated upper bound function `TJoin` that returns a supertype of its arguments.

- Typing rules for constants and binary arithmetic operations.

- A `Constrain` operation that refines a typing context after a certain boolean expression has been tested to be true.

For example, a type policy could define the nullary type constructors "Int" for integers that will fit in a machine word and "MaybeNullPtr$_\sigma$" for possibly-NULL pointers to records with type σ. We'll see below that the framework defines the type "Ptr$_\sigma$" for pointers to σ. Then the policy will likely define both `IsSubtype`(p, Ptr$_\sigma$, MaybeNullPtr$_\sigma$()) and `IsSubtype`(p, MaybeNullPtr$_\sigma$(), Int()) to be `true` for all values p. The definition of `Constrain` for this policy may promote one or more values of type MaybeNullPtr$_\sigma$() to Ptr$_\sigma$ following an appropriate NULL check. We defer the more detailed discussion of the `IsSubtype`, `TJoin`, and `Constrain` operators until the presentation of our typechecking algorithm in Section 2.3.

We do not verify the soundness of the type policy, although the soundness of our typechecking relies on the type policy being sound. Section 2.6 states the properties that must hold for a type policy in order for our framework to be sound.

field types	t	::=	$C(d_1, \ldots d_n) \mid \mathrm{Ptr}_\sigma$
dependencies	d	::=	$c \mid s.i$
record types	σ	::=	$\Sigma_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1}\rangle$
constants	c		
type constructors	C	$\in$	$\mathbb{T}$

Figure 2.1: The types that are assigned to registers and memory locations.

2.2 Flow-insensitive types

Figure 2.1 shows the language of memory types in our framework. Field types t describe the contents of a word in memory whereas σ types describe a mutable record consisting of a sequence of related fields.

The type of a word-sized location is either the instantiation of a type constructor C given by the type policy or a pointer to a mutable record. We saw above a few examples of nullary constructors for non-dependent types; constructors for dependent types are parameterized on one or more values. When used for field types t, we instantiate dependent constructors with references to nearby memory locations; later we will introduce register types τ by instantiating the constructors with symbolic expressions instead.

Our type system also includes a built-in type "Ptr_σ" for each σ in the type policy so that we can refer to pointers in our judgments for memory access. These pointer types encode a fundamental property of a heap-based programming language: that the given value points to a memory location of a certain type.

The notation $\Sigma_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1}\rangle$ denotes a very-dependent [Hic96] record type with n mutable fields, each of whose types may depend on the runtime values of other fields. For simplicity, fields are labeled with their index in the record. The dependent

type constructor "Σ_s" binds a variable s that can be thought of as the "self pointer" for the record. We use s to encode dependencies among the fields of the record: the special expression $s.i$ refers to the value stored in the i^{th} word of the current record, where i is a constant. We say that a field type $C(d_1, \ldots d_n)$ *refers to* field i iff at least one expression d_j is "$s.i$". A record type $\sigma = \Sigma_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1}\rangle$ is *well-formed* if for all terms $s.j$ appearing in σ, we have $0 \leq j < n$. In other words, dependencies must refer to fields that actually exist. We require that all types used in this framework be well-formed.

For example, a type policy may define the singleton type constructor Single(e), and then can define a dependent record containing two identical integers as

$$\Sigma_s.\langle\ 0 : \text{Int}();\ 1 : \text{Single}(s.0)\ \rangle$$

If we define the type constructor "Array(len)" to be the type of a pointer to an array of Ints with length len, then a record containing an array pointer and the length of that array has the type

$$\Sigma_s.\langle\ 0 : \text{Array}(s.1);\ 1 : \text{Int}()\ \rangle$$

as seen in Figure 2.2. Circular dependencies are also allowed, so

$$\Sigma_s.\langle\ 0 : \text{Single}(s.1);\ 1 : \text{Single}(s.0)\ \rangle$$

is another valid definition for our record containing two identical integers.

We therefore have two kinds of memory locations in the language. *Dependent* fields have types that refer to the self pointer or other fields, or are referred to by the types of sibling fields. *Non-dependent* fields have types of the form $C()$ (or $C(c_1, \ldots c_n)$, where each c_i is a constant) that do not refer to, and are not referred to by, any other field. We must be

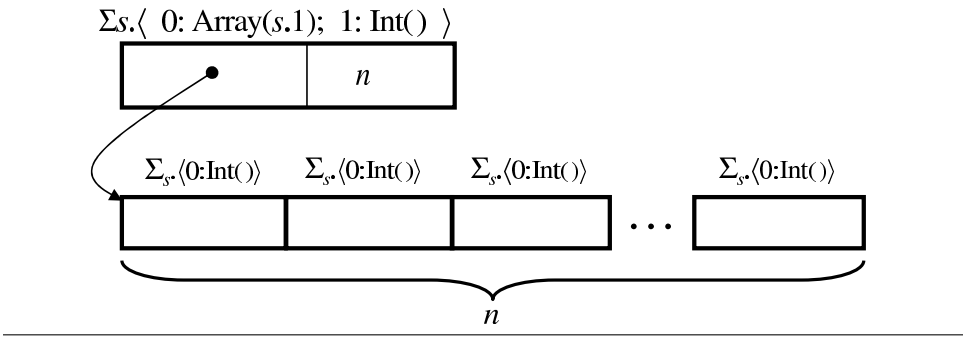

Figure 2.2: A record of type $\Sigma_s.\langle\ 0 : \text{Array}(s.1);\ 1 : \text{Int}()\ \rangle$, which contains a pointer to an integer array, and the length of the array.

careful when a dependent field is updated, to ensure that the dependencies are respected. However, we can modify non-dependent fields in place without additional checking.

2.3 Symbolic evaluation of assembly language

We describe here the process of typechecking assembly code when the start of each basic block has been annotated with an invariant, as is done in TAL [MCG+99]. Chapter 3 will describe how such invariants can be inferred.

Figure 2.3 shows the simple MIPS-like assembly language that we will be typechecking. A basic block is a sequence of instructions whose entry is denoted by some label, and whose exit is a branch or a jump. "`bnz` r, L_{true}, L_{false}" jumps to L_{true} when r $\neq 0$ and to L_{false} otherwise. We assume that all jumps are to labels in the same function, and that the machine has a finite set of registers $r_1, \ldots r_k$. Section 4.1.1 will discuss how we handle function calls and stack usage. Throughout this dissertation, we use $\oplus$, $\ominus$, and $\otimes$ to denote addition, subtraction, and multiplication performed modulo the word size of the target machine. Arithmetic overflow is one of the greatest challenges to soundness in many analyses;

instructions	I	::=	$\texttt{mov}\ \mathrm{r}_{dest}, c \mid \texttt{mov}\ \mathrm{r}_{dest}, \mathrm{r}_{s1}\ op\ \mathrm{r}_{s2}$
			$\mid \texttt{load}\ \mathrm{r}_{dest} \leftarrow [\mathrm{r}_a] \mid \texttt{store}\ \mathrm{r}_{src} \rightarrow [\mathrm{r}_a]$
binary operations	op	::=	$\oplus \mid \ominus \mid \otimes \mid \texttt{xor} \mid < \mid \ldots$
labels	L		
constants	c	::=	$L \mid \textit{numeric value}$
jumps	J	::=	$\texttt{bnz}\ \mathrm{r}, L_{true}, L_{false} \mid \texttt{jump}\ L$
basic blocks	B	::=	$I, B \mid J$
functions	$func$	::=	$\langle L_1 : B_1,\ \ldots,\ L_m : B_m \rangle$

Figure 2.3: The target assembly language.

in our discussions of overflow it will be necessary to distinguish the machine operations $\oplus$, $\ominus$, and $\otimes$ from the integer operations $+$, $-$, and $\cdot$. Finally, we will assume that memory locations are addressed as words rather than bytes.

We must track the memory state explicitly in order to reason about writes to dependent fields. "$\texttt{upd}(m, e_1, e_2)$" denotes the memory state that results from modifying memory state m by writing value e_2 at address e_1, while "$\texttt{sel}(m,\ e)$" is the result of reading address e in memory state m. We define "`ValidMem`" to be the type of a memory heap that is in a *consistent* state: one where all allocated locations contain a value that adheres to the type that the location was assigned when it was allocated. Consistency may be temporarily broken when we write a dependent field, since in general we will have to write to all of the fields in a dependent group before we can conclude that the group is consistent.

Our typechecker performs symbolic evaluation on one basic block at a time, using abstract values v for any unknown word-sized values and v_{mem} for unknown memory states. As seen in Figure 2.4, a state in our checker is $\langle \Delta, \Phi, m, \Gamma \rangle$, where Δ is a mapping from registers to symbolic expressions, Φ is a set of facts relevant to the type policy, m is the

states	S	::=	$\langle \Delta, \Phi, m, \Gamma \rangle$
register states	Δ	::=	$\mathrm{r}_1 = e_1,\ \ldots,\ \mathrm{r}_k = e_k$
fact states	Φ	::=	$F(e_1, \ldots, e_n), \Phi \mid \cdot$
memory states	m	::=	$\texttt{upd}(m, e_1, e_2) \mid v_{mem}$
type states	Γ	::=	Γ^r, Γ^m
register type states	Γ^r	::=	$v \mapsto \tau, \Gamma^r \mid \cdot$
memory type states	Γ^m	::=	$v_{mem} \mapsto \texttt{ValidMem}, \Gamma^m \mid \cdot$
abstract values	v		
abstract memory states	v_{mem}		
fact constructors	F	$\in$	$\mathbb{F}$
symbolic expressions	e	::=	$c \mid v \mid e_1\ op\ e_2 \mid \texttt{sel}(m,\ e)$
register types	τ	::=	$C(e_1, \ldots e_n) \mid \mathrm{Ptr}_\sigma$

Figure 2.4: The states of our symbolic execution algorithm for typechecking.

current memory state, and Γ is a mapping from abstract values to types. We use Γ^r for the portion of Γ containing word-sized abstract values, and Γ^m for the set of valid abstract memory states. While the previous section used field types t to represent the flow-insensitive types of memory locations, for abstract values in registers we use types τ that can depend on symbolic expressions, including other abstract values and the results of memory reads. If the type policy contains the Array constructor mentioned earlier, we could represent a checker state in which memory is valid, r_2 is known to equal $\mathrm{r}_1 \oplus 1$, and r_3 points to an array of length r_1 as $\langle \Delta_0, \Phi_0, v_{mem0}, \Gamma_0 \rangle$, where:

$$\begin{aligned}
\Delta_0 &= \{\mathrm{r}_1 = v_1;\ \mathrm{r}_2 = v_1 \oplus 1;\ \mathrm{r}_3 = v_2\} \\
\Phi_0 &= \emptyset \\
\Gamma_0 &= \{v_1 \mapsto \mathrm{Int}();\ v_2 \mapsto \mathrm{Array}(v_1);\ v_{mem0} \mapsto \texttt{ValidMem}\}
\end{aligned}$$

Informally, this state can be considered syntactic sugar for the following logical formula.

$$\langle \Delta, \Phi, m, \Gamma \rangle \vdash \texttt{mov}\ \mathrm{r}_{dest}, c \rightsquigarrow \langle \Delta[\mathrm{r}_{dest} \mapsto c],\ \Phi,\ m,\ \Gamma \rangle$$

$$\langle \Delta, \Phi, m, \Gamma \rangle \vdash \texttt{mov}\ \mathrm{r}_{dest}, \mathrm{r}_{s1}\ op\ \mathrm{r}_{s2} \rightsquigarrow \langle \Delta[\mathrm{r}_{dest} \mapsto \Delta(\mathrm{r}_{s1})\ op\ \Delta(\mathrm{r}_{s2})],\ \Phi,\ m,\ \Gamma \rangle$$

$$\langle \Delta, \Phi, m, \Gamma \rangle \vdash \texttt{load}\ \mathrm{r}_{dest} \leftarrow [\mathrm{r}_{addr}] \rightsquigarrow \langle \Delta[\mathrm{r}_{dest} \mapsto \texttt{sel}(m,\ \Delta(\mathrm{r}_{addr}))],\ \Phi,\ m,\ \Gamma \rangle$$

$$\langle \Delta, \Phi, m, \Gamma \rangle \vdash \texttt{store}\ \mathrm{r}_{src} \rightarrow [\mathrm{r}_{addr}] \rightsquigarrow \langle \Delta,\ \Phi,\ \texttt{upd}(m, \Delta(\mathrm{r}_{addr}), \Delta(\mathrm{r}_{src})),\ \Gamma \rangle$$

Figure 2.5: Symbolic evaluation rules for instructions.

(Section 2.6 describes the relationship between states and logic formulas more precisely.)

$$\exists v_1 \exists v_2 \exists v_{mem0}\ .\ v_1 \in \mathrm{Int}() \wedge v_1 \in \mathrm{Array}(v_1) \wedge v_{mem0} \in \texttt{ValidMem}$$
$$\wedge (\mathrm{r}_1 = v_1) \wedge (\mathrm{r}_2 = v_1 \oplus 1) \wedge (\mathrm{r}_3 = v_2)$$

All symbolic expressions are relative to a particular type state Γ that defines types for the abstract values that appear in the expression. Therefore, any abstract value in an expression should have a mapping in the associated type state:

Definition 2.1 (Well-formed expressions and states) *An expression e* is well-formed in Γ *if every abstract value appearing in e (including abstract memory states) is included in* $\mathrm{dom}(\Gamma)$. *A type state Γ is* well-formed *if for all $C(e_1, \ldots, e_n)$ in the range of Γ, each e_i is well-formed in Γ. A memory state m is* well-formed in Γ *if every abstract memory value in m is in* $\mathrm{dom}(\Gamma)$, *and every expression appearing in an* `upd` *term is well-formed in Γ. Finally, a state $\langle \Delta, \Phi, m, \Gamma \rangle$ is* well-formed *if Γ and m are well-formed and every expression appearing in Δ and Φ is well-formed in Γ.*

□

States in our type system must always be well-formed.

Figure 2.5 shows how basic blocks are symbolically evaluated. The judgment $\langle \Delta, \Phi, m, \Gamma \rangle \vdash I \rightsquigarrow \langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle$ means that in checker state $\langle \Delta, \Phi, m, \Gamma \rangle$, symbolically evaluating I yields the new checker state $\langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle$. Note that $\Gamma = \Gamma_1$ and $\Phi = \Phi_1$ in each of these rules, meaning that the types of abstract variables and the set of known facts are constant in a given basic block.

2.3.1 Typechecking symbolic expressions

We assign types to symbolic expressions with help from the typing rules provided by the type policy. The judgment $\Gamma, \Phi \vdash e : \tau$ means that expression e has type τ in type context Γ using assumptions Φ. Abstract values can be typechecked simply by looking them up in the context:

$$\frac{}{\Gamma, \Phi \vdash v : \Gamma(v)} \text{(ABSTRACT VALUE)}$$

The typing rules for constants and binary operations come from the type policy. Many of these rules will be straightforward: a label constant that refers to a Foo object in the data segment would be given type Ptr $_{\text{Foo}}$, and the sum of two Ints is an Int. But other operations may have preconditions that must be verified. For these rules, we rely on the set of facts Φ, which remembers information from branch instructions and any function preconditions.

For example, suppose a safety policy includes the Array type constructor and allows pointer arithmetic on arrays, but only if the result is within bounds. We can add to the type policy a fact constructor LessOrEq(a, b) meaning that $a \leq b$ (unsigned). Now the rule for pointer arithmetic is as follows:

(ARRAY ARITHMETIC)

$$\frac{\Gamma, \Phi \vdash e_1 : \text{Array}(e_{len}) \qquad \Gamma, \Phi \vdash e_2 : \text{Int} \qquad \text{LessOrEq}(e_2, e_{len}) \in \Phi}{\Gamma, \Phi \vdash e_1 \ \oplus \ e_2 : \text{Array}(e_{len} \ominus e_2)}$$

This rule requires that the program establish $e_2 \leq e_{len}$ before performing the pointer arithmetic. For example, if the code includes a branch on $e_2 \leq e_{len}$, the type policy's `Constrain` operation can add this fact to Φ. This requirement would make it difficult to verify arbitrary assembly code, but remember that that is not a goal of the project. Instead, we consider only statically verifiable programs and type policies. If a policy can't be verified statically, such as type safety in legacy C code, you must use CCured or another tool to insert runtime checks so that the code is verifiable.

For now, we won't be overly concerned with the implementation of Φ. Clearly, maintaining a list of every known fact is impractical: for example, if $\text{LessOrEq}(10, x) \in \Phi$, then we'd need $\text{LessOrEq}(9, x) \in \Phi$, $\text{LessOrEq}(8, x) \in \Phi$, and so on. Rather than maintain a closed set of facts, our implementation has achieved good results by using a few axioms when looking up information in Φ: $\text{LessOrEq}(a, b)$ holds if $\text{LessOrEq}(a, b) \in \Phi$, or a and b are constants where $a \leq b$, or there exists c such that $\text{LessOrEq}(a, c) \wedge \text{LessOrEq}(c, b)$. If needed, we could add more power to the static analysis by feeding Φ and the goal to an automated decision procedure such as Simplify [DNS03]. The only two requirements about Φ's implementation are that it is possible to look up information in it, and that it is possible to enumerate Φ as a finite set of facts, so that we can check that basic block preconditions hold.

We allow type policies to use the facts in Φ in the definition of `IsSubtype` where

necessary. When typechecking expressions we use this subsumption rule:

(SUBSUMPTION)

$$\frac{\Gamma, \Phi \vdash e : \tau' \qquad \Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \tau',\ \tau)}{\Gamma, \Phi \vdash e : \tau}$$

The judgment $\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \tau',\ \tau)$ is defined by the type policy and means that, if e has type τ', then it also has type τ. In the CCured type policy, we will see a case where the `IsSubtype` judgment uses the value e to look up information in Φ, but in many other cases the judgment will not depend on e, only on τ'.

For example, a type policy might allow long arrays to be used where shorter arrays are expected:

(ARRAY SUBTYPE)

$$\frac{\text{LessOrEq}(e'_{len}, e_{len}) \in \Phi}{\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{Array}(e_{len}),\ \text{Array}(e'_{len}))}$$

And since Array s are lists of integers, a nonempty array can be dereferenced as an Int() pointer:

(ARRAY DEREFERENCE)

$$\frac{\text{LessOrEq}(1, e_{len}) \in \Phi}{\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{Array}(e_{len}),\ \text{Ptr}_{\Sigma_s.\langle\ 0:\text{Int}()\ \rangle})}$$

2.3.2 Memory reads

The final form of expression is a read from memory. When reading a dependent field with type $C(s.j)$, we must replace dependency $s.j$ with a symbolic expression that explicitly encodes the current value of the j^{th} field. Consider a record that contains an array pointer and its length, and suppose we read the array field into r_1 and the length field

into r_2:

$$\begin{aligned}\Delta &= \{r_1 = \texttt{sel}(m_0,\ v);\ r_2 = \texttt{sel}(m_0,\ v \oplus 1)\} \\ \Gamma(v) &= \mathrm{Ptr}_{\Sigma_s.\langle\ 0{:}\mathrm{Array}(s.1);\ 1{:}\mathrm{Int}()\ \rangle}\end{aligned}$$

The value in r_1 will have type "Array($\texttt{sel}(m_0,\ v \oplus 1)$)," to reflect the fact that the length of the array is located at address $v \oplus 1$ in memory state m_0. We can now use r_2 as the length of array r_1. Even if memory is later changed, for example by updating this record with a new array and different length, we will still be able to use r_2 as the length of r_1 since we remember that they were read from the same memory state m_0.

We generalize the above intuition into the following rule:

$$\frac{\begin{array}{l}\text{(MEMORY READ)} \\ \Gamma, \Phi \vdash e : \mathrm{Ptr}_{\Sigma_s.\langle 0:t_0;\ \ldots\ ;\ n-1:t_{n-1}\rangle} \\ \tau = t_i\left[\texttt{sel}(m,\ e \oplus 0)\big/_{s.0}\right] \cdots \left[\texttt{sel}(m,\ e \oplus (n-1))\big/_{s.(n-1)}\right] \\ \Gamma, \Phi \vdash m : \texttt{ValidMem}\end{array}}{\Gamma, \Phi \vdash \texttt{sel}(m,\ e \oplus i) : \tau}$$

The binding step $\tau = t_i[\texttt{sel}(m,\ e \oplus 0)/_{s.0}] \cdots [\texttt{sel}(m,\ e \oplus n{-}1)/_{s.(n-1)}]$ will, for example, convert the field type Array($s.1$) in the previous example to the register type Array($\texttt{sel}(m,\ v \oplus 1)$). The requirement $\Gamma, \Phi \vdash m : \texttt{ValidMem}$ ensures that we are not in the middle of a dependent update. Note that because e points to a valid object with n fields, we can assume the arithmetic will not overflow: for all i between 0 and $n{-}1$, $e \oplus i = e + i$.

2.4 Memory consistency

After writing a value to memory, we must see whether $\Gamma, \Phi \vdash m : \texttt{ValidMem}$ for the resulting memory state m. If the `store` wrote to a dependent field, then other fields in the record may have to be updated as well in order for the record to be internally consistent

$$\frac{}{\vdash \texttt{sel}(\texttt{upd}(m, e_a, e_v),\ e_a) = e_v} \qquad \frac{e_a \neq e'_a}{\vdash \texttt{sel}(\texttt{upd}(m, e'_a, e_v),\ e_a) = \texttt{sel}(m,\ e_a)}$$

Figure 2.6: The standard axioms for interpreting `sel` and `upd`.

once again. For simplicity, our framework requires that all the relevant dependent fields of a record be mutated in the same basic block, with no other intervening writes to the heap. However, it would not be hard to extend the type system to allow invalid memory states that span basic block boundaries.

The rule for stores is below. Starting from a consistent state m, a basic block can perform a series of writes to some object that starts at address e_a. The notation $\texttt{upd}(m, e_a \oplus c_i, e_i)$ represents the result of storing e_i into this object's ${c_i}^{th}$ field; we check that each c_i is in bounds in the type rule below. Regardless of which fields have been overwritten, we can reestablish consistency for this object by checking whether *every* field $e_a \oplus i$ of the object has the type it should in the new memory state. First, we define a function that computes a canonical form for the result of a memory read using standard axioms for memory, which are shown in Figure 2.6:

$$\text{Read}(m, e_a \oplus i) = \begin{cases} e & \text{if } m = \texttt{upd}(m', e_a \oplus i, e) \\ \text{Read}(m', e_a \oplus i) & \text{if } m = \texttt{upd}(m', e_a \oplus j, e) \text{ and } i \neq j \\ \texttt{sel}(m,\ e_a \oplus i) & \text{otherwise} \end{cases}$$

With this function we can write the typing judgment that validates a sequence of

j writes to the same record:

$$\frac{\begin{array}{l}\text{(MEMORY UPDATE)}\\ m' = \texttt{upd}((\ldots\texttt{upd}(m, e_a \oplus c_1, e_1)\ldots), e_a \oplus c_j, e_j)\\ \Gamma, \Phi \vdash e_a : \mathrm{Ptr}_{\Sigma_s.\langle 0{:}t_0;\ \ldots\ ;\ n-1{:}t_{n-1}\rangle}\\ \forall 0 \le i < j\ .\ 0 \le c_i < n\\ \forall 0 \le i < n\ .\ \Gamma, \Phi \vdash \mathrm{Read}(m', e_a \oplus i) : \tau_i\\ \qquad \text{where } \tau_i = t_i\left[\mathrm{Read}(m', e_a \oplus 0)/_{s.0}\right]\cdots\left[\mathrm{Read}(m', e_a \oplus n{-}1)/_{s.(n-1)}\right]\\ \Gamma, \Phi \vdash m : \texttt{ValidMem}\end{array}}{\Gamma, \Phi \vdash m' : \texttt{ValidMem}}$$

For example, consider a record that contains an array reference, its length, and one other field of type Foo(). Suppose r_1 points to such a record, r_2 contains some array pointer, and r_3 contains the length of r_2:

$$\begin{aligned}\Delta &= \{\, \mathrm{r}_1 = v_{ptr};\ \mathrm{r}_2 = v_2;\ \mathrm{r}_3 = v_3\}\\ \Gamma &= \{\, v_{ptr} \mapsto \mathrm{Ptr}_{\Sigma_s.\langle\ 0{:}\mathrm{Array}(s.1);\ 1{:}\mathrm{Int}();\ 2{:}\mathrm{Foo}()\ \rangle};\\ &\qquad v_2 \mapsto \mathrm{Array}(v_3);\\ &\qquad v_3 \mapsto \mathrm{Int}();\\ &\qquad v_{mem0} \mapsto \texttt{ValidMem}\}\end{aligned}$$

Now we update the memory state v_{mem0} writing v_2 at address r_1 and v_3 at address $\mathrm{r}_1{+}1$, therefore mutating both the array and length fields of the record. These two store instructions produce the memory state

```
store r2 → [r1];
mov r_tmp, r1 ⊕ 1;
store r3 → [r_tmp];
```

$$m' = \texttt{upd}(\texttt{upd}(v_{mem0}, v_{ptr}, v_2),\ v_{ptr} \oplus 1,\ v_3)$$

The intermediate memory state $\texttt{upd}(v_{mem0}, v_{ptr}, v_2)$ is not consistent, and in general it must not be used for `load` instructions. But m' is consistent. Observe that we

get

$$\begin{aligned}
\mathrm{Read}(m', v_{ptr} \oplus 0) &= v_2 \\
\mathrm{Read}(m', v_{ptr} \oplus 1) &= v_3 \\
\mathrm{Read}(m', v_{ptr} \oplus 2) &= \mathtt{sel}(v_{mem0},\ v_{ptr} \oplus 2)
\end{aligned}$$

Each of these three fields has the correct type. v_2 has type

$$\begin{aligned}
\mathrm{Array}(s.1)\left[\mathrm{Read}(m', v_{ptr} \oplus 1)\big/_{s.1}\right] &= \mathrm{Array}(\mathrm{Read}(m', v_{ptr} \oplus 1)) \\
&= \mathrm{Array}(v_3)
\end{aligned}$$

while v_3 has type Int(). Location $v_{ptr} \oplus 2$ was not modified, so we rely on the fact that "$\Gamma, \Phi \vdash v_{mem0} : \mathtt{ValidMem}$" holds to ensure that $\mathtt{sel}(v_{mem0},\ v_{ptr} \oplus 2)$ has a value of type Foo().

For simplicity, we have presented a version of the (MEMORY UPDATE) rule in which all relevant writes to a dependent group are done consecutively. Our implementation uses a more general rule that takes advantage of simple type-based nonaliasing facts (a value with type Ptr_{σ_1} cannot alias a value with type Ptr_{σ_2} if $\sigma_1 \neq \sigma_2$). With alias information, we can allow writes to distinct objects to be interleaved. We can also allow reads from inconsistent memory, so long as the object being read from is consistent.

2.5 Typechecking basic blocks

As mentioned above, we assume for now that each basic block is annotated with a well-formed symbolic state that serves as a precondition. Let $\mathrm{Pre}(L)$ be this annotation for label L. Given the initial symbolic state for a label, we apply the instruction rules of

Jump instruction:	Requirement:
`jump` L	$\langle \Delta, \Phi, m, \Gamma \rangle \prec: \mathrm{Pre}(L)$
`bnz` r, L_{true}, L_{false}	$(\texttt{Constrain}(\langle \Delta, \Phi, m, \Gamma \rangle, \Delta(\mathrm{r}) \neq 0) \prec: \mathrm{Pre}(L_{true})) \wedge$
	$(\texttt{Constrain}(\langle \Delta, \Phi, m, \Gamma \rangle, \Delta(\mathrm{r}) = 0) \prec: \mathrm{Pre}(L_{false}))$

Figure 2.7: Symbolic evaluation rules for jumps, where $\langle \Delta, \Phi, m, \Gamma \rangle$ is the symbolic state before the jump is executed.

Figure 2.5 repeatedly to find the state at the end of the basic block. We then ensure that the resulting state implies the preconditions of successor blocks, as shown by the jump rules in Figure 2.7. These rules use the notation $S_1 \prec: S_2$ to mean that state S_1 is at least as strong as state S_2. In this section, we show how this ordering can be defined.

The branch instruction in Figure 2.7 also shows where the type policy's `Constrain` operation is used: $\texttt{Constrain}(S, b)$ is the state that results from refining S with the information that b is true. Usually, this involves adding a representation of b to the fact set Φ, or doing nothing at all if the branch is not relevant. `Constrain` could also modify the expressions or types in the state directly. For example, $\texttt{Constrain}(\langle \Delta, \Phi, m, \Gamma \rangle, v = 0)$ would be the state where every occurrence of v in Δ, Φ, or m is replaced by 0.

Example Figure 2.8 shows a loop that walks backwards through an array while maintaining the invariant that r_b points to the $\mathrm{r}_x{}^{th}$ element of array r_a. $\mathrm{Pre}(L_1)$, the precondition

```
//Preconditions:  0 ≤ r_x ≤ 100, and
//r_a points to an Array of length 100.
r_b = r_a + r_x;                          mov r_b, r_a ⊕ r_x
while(r_x != 0) {                  L1 :   bnz r_x, L2, L3
  r_x--;                           L2 :   mov r_x, r_x ⊖ 1
  r_b--;                                  mov r_b, r_b ⊖ 1
  ...                                     ...
}                                         jump L1
                                   L3 :
```

Figure 2.8: Example: A loop that walks backwards through an array, maintaining the invariant that r_b points to the $\mathrm{r}_x{}^{th}$ element of array r_a. The C code for this loop is on the left; the assembly version on the right.

at the beginning of the loop, is $\langle \Delta_1, \Phi_1, v_{mem}, \Gamma_1 \rangle$ where

$$\begin{aligned}
\Delta_1 &= \{\, \mathrm{r}_a = v_a; \\
&\quad\ \ \mathrm{r}_x = v_x; \\
&\quad\ \ \mathrm{r}_b = v_a \oplus v_x\} \\
\Phi_1 &= \{\, \mathrm{LessOrEq}(v_x, 100)\} \\
\Gamma_1 &= \{\, v_a \mapsto \mathrm{Array}(100); \quad v_b \mapsto \mathrm{Int}; \quad v_{mem} \mapsto \texttt{ValidMem}\}
\end{aligned}$$

Remember that we defined LessOrEq as an unsigned comparison, so $\mathrm{LessOrEq}(0, v_x)$ is implied. The branch command jumps to L_2 when $\mathrm{r}_x \neq 0$, so the state at L_2 is the same as L_1 except that the `Constrain` operator can add the fact $\mathrm{LessOrEq}(1, v_x)$ to Φ. As shown in Figure 2.5, evaluating the two decrement operations produces the symbolic state $\langle \Delta_{new}, \Phi_{new}, v_{mem}, \Gamma_{new} \rangle$, where

$$\begin{aligned}
\Delta_{new} &= \{\, \mathrm{r}_a = v_a; \\
&\quad\ \ \mathrm{r}_x = v_x \ominus 1; \\
&\quad\ \ \mathrm{r}_b = v_a \oplus v_x \ominus 1\} \\
\Phi_{new} &= \{\, \mathrm{LessOrEq}(1, v_x);\ \mathrm{LessOrEq}(v_x, 100)\} \\
\Gamma_{new} &= \{\, v_a \mapsto \mathrm{Array}(100); \quad v_b \mapsto \mathrm{Int}; \quad v_{mem} \mapsto \texttt{ValidMem}\}
\end{aligned}$$

The last line of the loop jumps back to L_1, so we have to show that this new state implies $\mathrm{Pre}(L_1)$. The relevant facts are that $\mathrm{r}_b = \mathrm{r}_a \oplus \mathrm{r}_b$ and $0 \leq \mathrm{r}_x \leq 100$. Recall that each

symbolic state is an existential statement: there exist values v_a, v_x, and v_{mem} such that the registers have the given values and the predicates in Φ hold. We can therefore show that a state implies $\mathrm{Pre}(L_1)$ by constructing values for the abstract variables in Γ_1. Using primes on the new abstract values for clarity, we get:

$$\begin{aligned} v'_a &= v_a \\ v'_x &= v_x \ominus 1 \\ v'_{mem} &= v_{mem} \end{aligned}$$

By substituting these values for the abstract variables of $\mathrm{Pre}(L_1)$, it's easy to see that $\langle \Delta_{new}, \Phi_{new}, v_{mem}, \Gamma_{new} \rangle$ satisfies all of the requirements of $\mathrm{Pre}(L_1)$, namely:

- v'_a, v'_x, and v'_{mem} have the correct types.
- $\mathrm{r}_a = v'_a = v_a$; $\mathrm{r}_x = v'_x = v_x \ominus 1$; and $\mathrm{r}_b = v'_a \oplus v'_x = v_a \oplus v_x \ominus 1$. [1]
- $\mathrm{LessOrEq}(v_x \ominus 1, 100)$, since the fact $\mathrm{LessOrEq}(1, v_x) \in \Phi_{new}$ means that $v_x \ominus 1$ does not underflow.

□

To formalize the procedure of checking that a state $S_1 = \langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle$ implies $S_2 = \langle \Delta_2, \Phi_2, m_2, \Gamma_2 \rangle$, let M be a mapping from abstract values in Γ_2 to expressions and memory states that are well-formed in Γ_1. Specifically, for each word-sized abstract value $v \in \mathrm{dom}(\Gamma_2^r)$, $M(v)$ is a well-formed expression in Γ_1; and for each abstract memory state $v_{mem} \in \mathrm{dom}(\Gamma_2^m)$, $M(v_{mem})$ is a well-formed memory state in Γ_1. Let $M[\![e]\!]$ be the result of replacing each abstract value v in e with $M(v)$. Observe that if e is well-formed in Γ_2, then

[1] For the most part, we treat arithmetic operators as uninterpreted functions when checking preconditions. Note that that's not quite enough here, since we need to know that $(v_a \oplus v_x) \ominus 1$ in the new state is equivalent to $v_a \oplus (v_x \ominus 1)$ in the precondition. Section 3.2 will describe how we handle such issues.

$M[\![e]\!]$ is well-formed in Γ_1. If such a mapping exists that satisfies all of the requirements of S_2, then S_1 implies S_2:

Definition 2.2 (State ordering) *State* $\langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle$ *is at least as strong as state* $\langle \Delta_2, \Phi_2, v_{mem2}, \Gamma_2 \rangle$*, written* $\langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle \prec: \langle \Delta_2, \Phi_2, v_{mem2}, \Gamma_2 \rangle$*, if each of the following hold:*

- $\mathrm{dom}(\Delta_1) = \mathrm{dom}(\Delta_2)$
- $\Gamma_2(v_{mem2}) = \texttt{ValidMem}$.
- There exists a mapping M from the values of $\mathrm{dom}(\Gamma_2)$ to expressions and memory states that are well-formed in Γ_1 such that
  1. $M(v_{mem2}) = m_1$.
  2. For all r $\in \mathrm{dom}(\Delta_2)$, $M[\![\Delta_2(\mathrm{r})]\!] = \Delta_1(\mathrm{r})$.
  3. For all $F(e_1, \ldots, e_n) \in \Phi_2$, $F(M[\![e_1]\!], \ldots, M[\![e_n]\!]) \in \Phi_1$.
  4. For each "$v \mapsto C(e_1, \ldots, e_n)$" $\in \Gamma_2^r$, let τ_1 be the type of $M(v)$ in state 1 (i.e. $\Gamma_1, \Phi_1 \vdash M(v) : \tau_1$) and let $\tau_2 = C(M[\![e_1]\!], \ldots, M[\![e_n]\!])$. It must be true that $\Gamma_1, \Phi_1 \vdash \texttt{IsSubtype}(M(v), \tau_1, \tau_2)$.

     Likewise, for each "$v \mapsto \mathrm{Ptr}_\sigma$" $\in \Gamma_2^r$, $\Gamma_1, \Phi_1 \vdash \texttt{IsSubtype}(M(v), \tau_1, \mathrm{Ptr}_\sigma)$ must be true.
  5. For each "$v_{mem} \mapsto \texttt{ValidMem}$" $\in \Gamma_2^m$, it must be that $\Gamma_1, \Phi_1 \vdash M(v_{mem}) :$ `ValidMem`.

□

Note that we define this relation only when the weaker state's memory state is an abstract value. Our preconditions at basic block boundaries will require only that memory is in a valid state. We could relax this requirement by generalizing Algorithm 2.1, but that has not been necessary in practice.

The following algorithm constructs a map M that satisfies requirements 1 and 2 in Definition 2.2, if possible. It works by recursively comparing symbolic expressions in the two states to determine which symbolic expression in state 1 corresponds to each abstract value in Γ_2. At the end of the algorithm, if $\mathrm{dom}(M) \neq \mathrm{dom}(\Gamma_2)$ then there is some abstract value in Γ_2 that does not appear in any of the expressions in Δ_2. We fail in this case, since there is no reason for a precondition to contain dead values. If $\mathrm{dom}(M) = \mathrm{dom}(\Gamma_2)$, then M is a complete map and we can easily check that the other requirements of Definition 2.2 hold.

Algorithm 2.1 (Construction of the abstract value map M)

When deciding whether $\langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle \prec: \langle \Delta_2, \Phi_2, v_{mem2}, \Gamma_2 \rangle$*:*

Let M *be a map from abstract values of* $dom(\Gamma_2)$ *to expressions and memory states that are well-formed in* Γ_1*.* M *is initially empty except that* $M(v_{mem2}) = m_1$*.*

Call (`compare` $\Delta_1(\mathrm{r}_i)$ $\Delta_2(\mathrm{r}_i)$) *for each* $\mathrm{r}_i \in dom(\Delta_1)$*. If each call to* `compare` *returns Success, then* M *will be a map satisfying* $M[\![\Delta_2(\mathrm{r}_i)]\!] = \Delta_1(\mathrm{r}_i)$ *for all* i*. If any call to* `compare` *returns Fail, or* $dom(M) \neq dom(\Gamma_2)$ *at the end of the algorithm, then conclude* $\langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle \not\prec: \langle \Delta_2, \Phi_2, v_{mem2}, \Gamma_2 \rangle$*.*

```
compare e1 e2 =
  match e1, e2 with
    | e1, v → if v ∈ dom(M) then
                if M(v) = e1 then return Success
                else return Fail
              else
                Add (v ↦ e1) to M; return Success.
    | e1' op e1'', e2' op e2'' →  if (compare e1' e2' = Success)
                                  and (compare e1'' e2'' = Success)
                                    then return Success
                                    else return Fail
    | sel(m, e1'), sel(vmem2, e2') → if m = m1
                                       and (compare e1' e2' = Success)
                                         then return Success
                                         else return Fail
    | c, c → return Success
    | otherwise → return Fail
```

□

2.6 Soundness

This section outlines soundness proofs for the typing judgments, memory consistency judgment, and the state order $\prec:$. For the portions of these rules that are handled by the type policy, such as arithmetic operators and `IsSubtype`, we state lemmas that will need to be proved for each type policy.

Since the soundness of the type system is with respect to a concrete execution environment, we begin by introducing some notation for concrete machines. Let Word be the set of machine words. An allocation state ρ_A is a partial function from Word values to record types σ that gives the type of each location in memory. These records may not

overlap: if $\rho_A(a) = \sigma$ where σ has n fields, then $a+1, \ \ldots \ a+n-1$ may not have any mapping in ρ_A. Furthermore the records must all fit in the address space, so that $a+(n-1)$ does not overflow, and NULL must not be used as a pointer: $0 \notin \mathrm{dom}(\rho_A)$. A memory store ρ_S : Word $\rightarrow$ Word represents the contents of memory at a particular time. For a given type state Γ, Vars is a map from the abstract values dom(Γ) to concrete values. Vars(v) $\in$ Word when $v \in \mathrm{dom}(\Gamma^r)$, and Vars($v$) is a memory store when $v \in \mathrm{dom}(\Gamma^m)$. The concretization functions $[\![e]\!]_{\mathrm{Vars}}$, $[\![m]\!]_{\mathrm{Vars}}$ and $[\![\tau]\!]_{\mathrm{Vars}}$ shown in Figure 2.9 convert our symbolic representations into concrete values using the map Vars.

For a register type τ that depends only on concrete values, $[\![\tau]\!]_{\rho_A} \in 2^{\mathrm{Word}}$ is the set of values corresponding to τ under allocation state ρ_A. $[\![\mathrm{Ptr}_\sigma]\!]_{\rho_A}$ is defined as $\{a \in \mathrm{Word} \mid \rho_A(a) = \sigma\}$, while $[\![C(i_1, \ldots, i_n)]\!]_{\rho_A}$ is defined by the type policy for each C. For example, $[\![\mathrm{Array}(n)]\!]_{\rho_A} = \{a \in \mathrm{Word} \mid \forall\, i \ .\ 0 \le i < n \implies \rho_A(a+i) = \Sigma_s.\langle 0 : \mathrm{Int}() \rangle\}$, and $[\![\mathrm{Int}()]\!]_{\rho_A} = \mathrm{Word}$. For register types depending on symbolic values, we first use the $[\![\cdot]\!]_{\mathrm{Vars}}$ operator, as in $[\![[\![\tau]\!]_{\mathrm{Vars}}]\!]_{\rho_A}$. Finally, we define a notion of a well-typed memory store ρ_S:

$[\![\texttt{ValidMem}]\!]_{\rho_A} =$

$$\{\rho_S \in \mathrm{Word} \rightarrow \mathrm{Word} \mid \forall a \in \mathrm{dom}(\rho_A) :$$
$$\forall\, i < n, \quad \rho_S(a+i) \in \Big[\!\Big[t_i\,[\rho_S(a+0)/s.0] \cdots [\rho_S(a+n-1)/s.(n-1)] \Big]\!\Big]_{\rho_A}$$
$$\text{where } \rho_A(a) = \Sigma_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1} \rangle.\ \ \}$$

An allocation state ρ_A is well-formed (written $\models \rho_A$) if it is a map from (Word$-\{0\}$) to record types and it has the nonoverlapping and nonoverflowing properties specified earlier: for all a, σ, n, i such that $\rho_A(a) = \sigma$, σ has n fields, and $0 < i < n$, then $(a+i) \notin \mathrm{dom}(\rho_A)$ and $a + i < |\mathrm{Word}|$. A typing context Γ,Φ is well-formed under a particular concretization

Expressions: $[\![e]\!]_{\text{Vars}} \in \text{Word}$

$$\begin{aligned}
[\![v]\!]_{\text{Vars}} &= \text{Vars}(v) \\
[\![c]\!]_{\text{Vars}} &= c \\
[\![e_1\ op\ e_2]\!]_{\text{Vars}} &= [\![op]\!]\,([\![e_1]\!]_{\text{Vars}}, [\![e_2]\!]_{\text{Vars}}) \\
[\![\texttt{sel}(m,\ e)]\!]_{\text{Vars}} &= [\![m]\!]_{\text{Vars}}\,([\![e]\!]_{\text{Vars}})
\end{aligned}$$

Memory states: $[\![m]\!]_{\text{Vars}} \in \text{Word} \to \text{Word}$

$$\begin{aligned}
[\![v_{mem}]\!]_{\text{Vars}} &= \text{Vars}(v_{mem}) \\
[\![\texttt{upd}(m, e_1, e_2)]\!]_{\text{Vars}} &= [\![m]\!]_{\text{Vars}}\Big[[\![e_1]\!]_{\text{Vars}} \mapsto [\![e_2]\!]_{\text{Vars}}\Big]
\end{aligned}$$

Concretize the expressions in register types: $[\![\tau]\!]_{\text{Vars}}$

$$\begin{aligned}
[\![\text{Ptr}_\sigma]\!]_{\text{Vars}} &= \text{Ptr}_\sigma \\
[\![C(e_1, \ldots e_n)]\!]_{\text{Vars}} &= C([\![e_1]\!]_{\text{Vars}}, \ldots [\![e_n]\!]_{\text{Vars}})
\end{aligned}$$

Concretize machine arithmetic operators: $[\![op]\!] \in (\text{Word} \times \text{Word}) \to \text{Word}$

$$\begin{aligned}
[\![\oplus]\!](x, y) &= x + y \quad \textbf{mod } |\text{Word}| \\
[\![\otimes]\!](x, y) &= x \cdot y \quad \textbf{mod } |\text{Word}| \\
&\ldots
\end{aligned}$$

Figure 2.9: Definition of the concretization functions $[\![e]\!]_{\text{Vars}}$ for symbolic expressions e, $[\![m]\!]_{\text{Vars}}$ for symbolic memory states m, and $[\![\tau]\!]_{\text{Vars}}$ for register types τ.
We use $[\![op]\!] \in (\text{Word} \times \text{Word}) \to \text{Word}$ for the concrete arithmetic operation corresponding to the symbolic expression op.

(Vars, ρ_A) (written $\text{Vars}, \rho_A \models \Gamma, \Phi$) if the following hold: $\models \rho_A$; $\text{dom}(\Gamma) = \text{dom}(\text{Vars})$; $\text{Vars}(v) \in [\![[\![\Gamma(v)]\!]_{\text{Vars}}]\!]_{\rho_A}$ for all $v \in \text{dom}(\text{Vars})$; and $\rho_A \models F([\![e_1]\!]_{\text{Vars}}, \ldots, [\![e_n]\!]_{\text{Vars}})$ for all $F(e_1, \ldots, e_n) \in \Phi$. The meaning of $\rho_A \models F(i_1, \ldots, i_n)$ must be defined by the type policy for each fact constructor F. With these notions of well-formedness, we can state a soundness theorem for the judgment $\Gamma, \Phi \vdash e : \tau$.

Theorem 1 (Soundness of expression typing) *If* $\text{Vars}, \rho_A \models \Gamma, \Phi$ *and* $\Gamma, \Phi \vdash e : \tau$, *then* $[\![e]\!]_{\text{Vars}} \in [\![[\![\tau]\!]_{\text{Vars}}]\!]_{\rho_A}$.

Proof: The proof is by induction on e. There are several cases according to the different rules for deriving $\Gamma, \Phi \vdash e : \tau$. The case $e = v$ follows from $\text{Vars}, \rho_A \models \Gamma, \Phi$ and the (ABSTRACT VALUE) rule. The (SUBSUMPTION) rule relies on the correctness of the type policy's `IsSubtype` rule:

Required Lemma 1 (`IsSubtype`) *If* $\text{Vars}, \rho_A \models \Gamma, \Phi$, $[\![e]\!]_{\text{Vars}} \in [\![[\![\tau']\!]_{\text{Vars}}]\!]_{\rho_A}$, *and* $\Gamma, \Phi \vdash \texttt{IsSubtype}(e, \tau', \tau)$, *then* $[\![e]\!]_{\text{Vars}} \in [\![[\![\tau]\!]_{\text{Vars}}]\!]_{\rho_A}$.

Likewise the constant and arithmetic cases must be handled by the type policy, since it defines the typing rules for those cases:

Required Lemma 2 (Expression cases) *The constant* ($e = c$) *and arithmetic* ($e = e_1$ *op* e_2) *cases of Theorem 1 hold.*

The final case, memory reads, follows directly from the (MEMORY READ) rule, the definitions of $[\![\text{Ptr}_\sigma]\!]_{\rho_A}$ and $[\![\texttt{ValidMem}]\!]_{\rho_A}$, and the correctness of the judgment $\Gamma, \Phi \vdash m :$ `ValidMem`, shown below.

□

Theorem 2 (Soundness of memory updates) *If* $\text{Vars}, \rho_A \models \Gamma, \Phi$ *and* $\Gamma, \Phi \vdash m' : \texttt{ValidMem}$*, then* $[\![m']\!]_{\text{Vars}} \in [\![\texttt{ValidMem}]\!]_{\rho_A}$.

This theorem is proved by induction on the symbolic state m' using the (MEMORY UPDATE) rule. Therefore we assume that there exists a state m and expression e_a such that $[\![m]\!]_{\text{Vars}} \in [\![\texttt{ValidMem}]\!]_{\rho_A}$ and m' is the result of writing to m at one or more offsets of address e_a, as stated in (MEMORY UPDATE). Further, we can assume $\rho_A([\![e_a]\!]_{\text{Vars}}) = \Sigma_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1}\rangle$ and all of the write offsets are less than n.

First, note that for all $i \in [0, n)$, $[\![\text{Read}(m', e_a \oplus i)]\!]_{\text{Vars}} = [\![m']\!]_{\text{Vars}}([\![e_a \oplus i]\!]_{\text{Vars}})$. There are three cases to the proof of this fact, corresponding the three rules in the definition of Read on page 19. The first and third hold by definition of $[\![\texttt{upd}(m, e_a \oplus i, e)]\!]_{\text{Vars}}$ and $[\![\texttt{sel}(m,\ e_a \oplus i)]\!]_{\text{Vars}}$, respectively. The second case:

$$\text{Read}(\texttt{upd}(m'', e_a \oplus j, e), e_a \oplus i) = \text{Read}(m'', e_a \oplus i) \text{ when } i \neq j$$

holds because the offsets i and j are less than n, so $\oplus$ does not overflow. Therefore, $i \neq j \implies [\![e_a \oplus i]\!]_{\text{Vars}} \neq [\![e_a \oplus j]\!]_{\text{Vars}}$.

To show that $[\![m']\!]_{\text{Vars}} \in [\![\texttt{ValidMem}]\!]_{\rho_A}$, we must consider all $a \in \text{dom}(\rho_A)$ and prove

$$\forall\, i < n',\quad [\![m']\!]_{\text{Vars}}(a+i) \in \Big[\!\Big[t_i \left[[\![m']\!]_{\text{Vars}}(a+0)/s.0\right] \cdots \left[[\![m']\!]_{\text{Vars}}(a+n'-1)/s.(n'-1)\right] \Big]\!\Big]_{\rho_A}$$

where $\rho_A(a) = \Sigma_s.\langle 0 : t_0;\ \ldots\ ;\ n'-1 : t_{n'-1}\rangle$. When $a = [\![e_a]\!]_{\text{Vars}}$, this is true by the premise of the (MEMORY UPDATE) rule using the fact that $[\![\text{Read}(m', e_a \oplus i)]\!]_{\text{Vars}} = [\![m']\!]_{\text{Vars}}([\![e_a \oplus i]\!]_{\text{Vars}})$.

The important case of this proof is the one where $a \neq [\![e_a]\!]_{\text{Vars}}$. We must show that our memory rules are sufficient to ensure that writing to memory won't change the (dependent) type of any value outside of the current object. For this, we use the nonoverlapping property of well-formed allocation states ρ_A, and observe that the statement above is true for $[\![m']\!]_{\text{Vars}}$ because it is true for the original memory state $[\![m]\!]_{\text{Vars}}$, and no intervening writes to the object at $[\![e_a]\!]_{\text{Vars}}$ have changed anything relevant to a. Whenever $j' < n'$ and $c < n$, the nonoverlapping property implies that $(a + j') \neq ([\![e_a]\!]_{\text{Vars}} + c)$. Therefore, $a + i$ points to the same value, and depends on the same values, in m' as it does in m.

□

2.6.1 Soundness of state ordering

Finally, we discuss the correctness of the state ordering test $\langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle \prec: \langle \Delta_2, \Phi_2, m_2, \Gamma_2 \rangle$. A concrete execution state is a triple (ρ_A, ρ_S, ρ_R) where $\rho_A : \text{Word} \to \sigma$ is an allocation state, $\rho_S : \text{Word} \to \text{Word}$ represents the contents of memory, and ρ_R represents the contents of the machine registers. Let $\gamma(\langle \Delta, \Phi, m, \Gamma \rangle)$ be the set of concrete machine states corresponding to abstraction $\langle \Delta, \Phi, m, \Gamma \rangle$:

$$\gamma(\langle \Delta, \Phi, m, \Gamma \rangle) = \{(\rho_A, \rho_S, \rho_R) \mid \text{There exists a concretization map Vars such that } \text{Vars}, \rho_A \models \Gamma, \Phi;\ \ \rho_S = [\![m]\!]_{\text{Vars}};\ \ \text{and for each machine register } r,\ \rho_R(r) = [\![\Delta(r)]\!]_{\text{Vars}}\}$$

Theorem 3 (Soundness of state ordering) *If* $\langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle \prec: \langle \Delta_2, \Phi_2, m_2, \Gamma_2 \rangle$, *then* $\gamma(\langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle) \subseteq \gamma(\langle \Delta_2, \Phi_2, m_2, \Gamma_2 \rangle)$.

Proof: Assume $\langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle \prec: \langle \Delta_2, \Phi_2, m_2, \Gamma_2 \rangle$, and $(\rho_A, \rho_S, \rho_R) \in \gamma(\langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle)$. We will show $(\rho_A, \rho_S, \rho_R) \in \gamma(\langle \Delta_2, \Phi_2, m_2, \Gamma_2 \rangle)$ by finding a map Vars_2 that satisfies the requirements above.

Let M be the map of abstract values in Γ_2 satisfying $\langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle \prec: \langle \Delta_2, \Phi_2, m_2, \Gamma_2 \rangle$, and let Vars_1 be the concretization satisfying $(\rho_A, \rho_S, \rho_R) \in \gamma(\langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle)$. Define $\text{Vars}_2(v) = [\![M(v)]\!]_{\text{Vars}_1}$ for all $v \in \Gamma_2$. It is then easy to prove the following lemma about the substitution $M[\![\cdot]\!]$:

Lemma 3 (Substitution lemma) *If e is well-formed in Γ_2, then $[\![e]\!]_{\text{Vars}_2} = [\![M[\![e]\!]]\!]_{\text{Vars}_1}$. (Proof omitted.)*

Now, we prove $\text{Vars}_2, \rho_A \models \Gamma_2, \Phi_2$:

- $\models \rho_A$, because $\text{Vars}_1, \rho_A \models \Gamma_1, \Phi_1$.
- $\text{dom}(\Gamma_2) = \text{dom}(\text{Vars}_2)$ by definition of Vars_2.
- For each $v \in \text{dom}(\text{Vars}_2)$, we want $\text{Vars}_2(v) \in [\![[\![\Gamma_2(v)]\!]_{\text{Vars}}]\!]_{\rho_A}$:

  Assume $\Gamma_2(v) = C(e_1, \ldots, e_n)$; the case where $\Gamma_2(v) = \text{Ptr}_\sigma$ is similar. Let τ_1 be such that $\Gamma_1, \Phi_1 \vdash M(v) : \tau_1$, as in Requirement 4 of Definition 2.2. By Theorem 1, $\text{Vars}_2(v) = [\![M(v)]\!]_{\text{Vars}_1} \in [\![[\![\tau_1]\!]_{\text{Vars}_1}]\!]_{\rho_A}$. From Requirement 4 and Required Lemma 1, we know $[\![[\![\tau_1]\!]_{\text{Vars}_1}]\!]_{\rho_A} \subseteq [\![[\![C(M[\![e_1]\!], \ldots, M[\![e_n]\!])]\!]_{\text{Vars}_1}]\!]_{\rho_A}$. And by the Substitution Lemma, $[\![[\![C(M[\![e_1]\!], \ldots, M[\![e_n]\!])]\!]_{\text{Vars}_1}]\!]_{\rho_A} = [\![C([\![e_1]\!]_{\text{Vars}_2}, \ldots, [\![e_n]\!]_{\text{Vars}_2})]\!]_{\rho_A}$. So $\text{Vars}_2(v) \in [\![[\![C(e_1, \ldots, e_n)]\!]_{\text{Vars}_2}]\!]_{\rho_A}$.
- For all $F(e_1, \ldots, e_n) \in \Phi_2$, $F(M[\![e_1]\!], \ldots, M[\![e_n]\!]) \in \Phi_1$ by Requirement 3. $\rho_A \models F([\![M[\![e_1]\!]]\!]_{\text{Vars}_1}, \ldots, [\![M[\![e_n]\!]]\!]_{\text{Vars}_1})$ because $\text{Vars}_1, \rho_A \models \Gamma_1, \Phi_1$. Using the Substitution

Lemma, we conclude $\rho_A \models F(\llbracket e_1 \rrbracket_{\text{Vars}_2}, \ldots, \llbracket e_n \rrbracket_{\text{Vars}_2})$.

$\rho_S = \llbracket m_1 \rrbracket_{\text{Vars}_1} = \llbracket m_2 \rrbracket_{\text{Vars}_2}$ by the Substitution Lemma, since m_2 is the abstract value v_{mem2}. Finally, for any register r, $\Delta_1(\text{r}) = M\llbracket\Delta_2(\text{r})\rrbracket$ by Requirement 2 of Definition 2.2. With one last application of the Substitution Lemma, we get $\rho_S(\text{r}) = \llbracket\Delta_1(\text{r})\rrbracket_{\text{Vars}_1} = \llbracket M\llbracket\Delta_2(\text{r})\rrbracket\rrbracket_{\text{Vars}_1} = \llbracket\Delta_2(\text{r})\rrbracket_{\text{Vars}_2}$.

□

2.7 Related type systems

There are many dependent type systems that are more expressive than the one presented here. The Xanadu language, for example, provides dependent types for an imperative, source-level language [Xi00]. Xanadu supports dependencies between different objects, which lets the language express more interesting properties about heap structures than ours can. The cost of such expressiveness is that dependently-typed locations cannot be modified, because the type system cannot keep track of which other types might refer to the location being modified.

Xanadu can be compiled to DTAL, a dependently-typed assembly language [XH01]. DTAL focuses largely on array types and array-bound check elimination. Basic blocks are annotated with invariants to reduce the need for type inference, and a type-preserving compiler is used. Like Xanadu, and for the same reasons, DTAL does not support the modification of dependently-typed locations in the heap.

Our dependent record types are related to Hickey's very dependent function types [Hic96]. Hickey encodes immutable records as functions from labels to values. By using very

dependent types for these functions, one can impose dependencies among the object's fields. Hickey uses these types to formalize a theory of objects, including methods and inheritance. Our type system has a similar focus on dependencies among fields and function arguments, but in the context of a low-level imperative language with mutable structures.

Grossman [Gro02] discusses the difficulty in supporting existential types in imperative languages: you can allow mutation of existentially-typed objects, or you can allow objects to be aliased after unpacking the existential, but not both. This is similar to the difficulty that our system addresses for dependent types. The dependent-type analog to the unpack operation is the type rule for memory reads, so we fix the problem by remembering in the `sel` expression from which memory state the value was read, and we don't assume that values read from two different memory states have any correlation.

Deputy [CHA+07] is a recently-developed, source-level dependent type system for C code that imposes similar restrictions as ours: types can depend only on other values in the same structure, variable scope, or formal parameter list. We discuss Deputy further in Chapter 5.

Chapter 3

Type Inference

So far, we have described how to check that assembly code adheres to a safety policy when each label in the assembly code is annotated with an invariant describing the abstract state that should hold at that label. There are two ways to generate such annotations: 1) Use a certifying compiler that emits the invariants during code generation, or 2) Rediscover these invariants after code generation by analyzing the assembly code. As you will see in this chapter, our system rediscovers the basic block invariants given a small amount of information by the code generator, such as the types of global variables and function signatures.

It would be very difficult to write a sound, whole-program type inference system for assembly code without any high-level information about the code. All of the difficulties of source code static analysis are magnified by the low-level instructions and lack of typing information in assembly code. CodeSurfer/x86 [BR04] is one tool that will analyze binary code with the goal of helping a human understand it, and will do so without annotations

or debugging information. (This is especially useful when studying viruses and other code for which the author has no interest in helping you decompile it.) But soundness is not a goal of decompilation tools like CodeSurfer/x86; instead they aim to give the best possible decompilation while warning about certain suspect operations such as potential buffer overflows. We assume that the authors of the program are more helpful, and will provide us with certain annotations just as they have already agreed to run a specific safety tool on their source code. In exchange for these annotations, we get soundness, greater precision, and faster verification than we could otherwise achieve.

To make our assembly code analysis tractable we analyze one function at a time, using the source code tool to annotate function boundaries for us. The following annotations are required:

- The type of each global variable.
- For each function: the type of each formal parameter, the type of the return value, and the stack address and type of any local variable that has its address taken.
- For each heap allocation, the type that will be applied to the resulting pointer (e.g. "`T*`" if the corresponding source instruction is "`T* x = (T*)malloc(n)`").
- Any extra information needed by the type policy. For example, CCured supports a form of dynamic typing in which Run-Time Type Information (RTTI) is attached to pointers and used in checked downcasts. To support assembly analysis, CCured emits an annotation describing which type tag is used with each type.

These annotations are untrusted, so any erroneous annotation results in a verifi-

cation failure rather than unsoundness. We implement the annotations by inserting inline assembly instructions into the source code, which are then preserved by the compiler. Other methods of preserving this data would also work. With the exception of allocation sites, none of these annotations are inserted into function bodies, so they do not hinder the optimizer. It would also have been possible to also use inline assembly or a similar mechanism to insert annotations for loop invariants, the types of local variables, and so forth. We decided against adding such annotations because they would pin variables to certain locations and constrain the opportunities for optimization. Of course, no inference algorithm can be complete for all valid optimizations. As seen in Section 4.3, ours is robust enough to handle most code, but some complex pieces of code such as multidimensional array indices will need a stronger analysis or additional annotations.

Abstract interpretation

Given this global information, we use abstract interpretation [CC79] on each function to determine a symbolic state describing each basic block. The abstract state $\langle \Delta, \Phi, m, \Gamma \rangle$ for this analysis is the same as the typechecking state described in the previous chapter. At the start of each function, we use the information about formal parameters that is given by the annotations to create an initial state. We then perform symbolic execution over the body of the function. The hard part is dealing with *join points*, which are labels with two or more predecessors. Here, we must use a *widening* operator to determine a suitable upper bound for the symbolic states of each predecessor. This operation must preserve any information that will be needed in the future to check the type policy, while assuring that the abstract interpretation does not enter an infinite loop. In general

a widening operator must be defined for each type policy, but the problems and solutions discussed here should apply to most policies for C and related languages.

We begin by adapting Algorithm 2.1 to be widening operator that generates an upper bound of its input states. As is common in abstract interpretation, the widening operator also determines whether the upper bound is at least as strong as the previous state at this label; if so we are at fixed point and do not need to check state ordering separately. We also eagerly add information to the state Φ that would otherwise be lost during widening.

The next challenge we discuss is arithmetic. The `compare` algorithm in the previous chapter treats arithmetic operators as uninterpreted functions, and makes no assumptions about their behavior. This would not be sufficient during widening, as we will demonstrate. Support for pointer arithmetic requires more careful treatment of arithmetic operators.

Finally, we discuss the performance of the system. By careful design of the widening operator, we ensure the abstract interpretation terminates. By using a liveness analysis for registers and optimizing the worklist algorithm, we ensure that it does so quickly.

3.1 Widening

We perform abstract interpretation in the standard manner. At each label (the start of a basic block) we remember the current symbolic state. When we reach a label that has already been explored, we compare the current state $S_1 = \langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle$ to the saved state $S_2 = \langle \Delta_2, \Phi_2, m_2, \Gamma_2 \rangle$. If $S_1 \prec: S_2$, then we are done exploring this path since we have already typechecked this basic block using a precondition S_2 that is weaker than

the current one. Otherwise, we must find a new state S_3 that is weaker than both S_1 or S_2, and use this state to symbolically execute the basic block. This section describes how to find such an S_3.

We start by combining the symbolic expressions for each register in Δ_1 and Δ_2 in such a way as to preserve as much of the symbolic structure as possible. For example, suppose S_1 and S_2 each contain a pointer to the second field of a two-Int record.

$$\Delta_1(\mathrm{r}_1) = v_1 \oplus 1 \qquad \Delta_2(\mathrm{r}_1) = \texttt{sel}(m_2,\ e_2) \oplus 1$$

$$\Gamma_1, \Phi_1 \vdash v_1 : \mathrm{Ptr}_{\Sigma_s.\langle\ 0{:}\mathrm{Int}();\ 1{:}\mathrm{Int}()\ \rangle} \qquad \Gamma_2, \Phi_1 \vdash \texttt{sel}(m_2,\ e_2) : \mathrm{Ptr}_{\Sigma_s.\langle\ 0{:}\mathrm{Int}();\ 1{:}\mathrm{Int}()\ \rangle}$$

Since $\Delta_1(\mathrm{r}_1)$ and $\Delta_2(\mathrm{r}_1)$ each point to an Int, it would be sound to assign to $\Delta_3(\mathrm{r}_1)$ a fresh abstract value of type pointer to Int. This would satisfy the requirement that $S_1 \prec: S_3$ and $S_2 \prec: S_3$. However, such a value for $\Delta_3(\mathrm{r}_1)$ loses useful information: that $\mathrm{r}_1 \ominus 1$ also points to an Int. If S_3 is too weak, we may not be able to typecheck the code that follows. Instead, we can create a fresh abstract value of type $\mathrm{Ptr}_{\Sigma_s.\langle\ 0{:}\mathrm{Int}();\ 1{:}\mathrm{Int}()\ \rangle}$ and keep the symbolic structure that's common to S_1 and S_2.

$$\Delta_3(\mathrm{r}_1) = v_3 \oplus 1$$

$$\Gamma_3(v_3) = \mathrm{Ptr}_{\Sigma_s.\langle\ 0{:}\mathrm{Int}();\ 1{:}\mathrm{Int}()\ \rangle}$$

Section 3.2.3 will give an example where this preservation of information is important.

To preserve expression structure, we will change the `compare` function from Algorithm 2.1 into a new function `join` that returns an expression weaker than its two argument expressions, while creating abstract values only when necessary. This function is based on existing join algorithms for the theory of uninterpreted functions [GTN04, CL05]; in Section 3.2 we discuss how to handle arithmetic in the common case where treating arithmetic operators as uninterpreted is too imprecise.

In the new algorithm, M is a map from abstract values in the new state to pairs (e_1, e_2), where e_1 is well-formed in Γ_1 and e_2 is well-formed in Γ_2. In the vocabulary of Single Static Assignment (SSA) form [CFR+91], M represents Phi nodes. Each $M(v) = (e_1, e_2)$ can be thought of as the assignment $v \leftarrow \phi(e_1, e_2)$, meaning that v takes the value e_1 when the S_1 path is used, or e_2 when the S_2 path is used. We define M_L and M_R to be maps containing the left and right projections, respectively, of each of these pairs, so $M_L(v) = e_1$ and $M_R(v) = e_2$ when $M(v) = (e_1, e_2)$. We will construct M such that M_L is a map demonstrating $S_1 \prec: S_3$ and M_R is a map demonstrating $S_2 \prec: S_3$ according to Definition 2.2.

Figure 3.1 shows the procedure for joining expressions. The `join` function abstracts incompatible parts of the symbolic expressions and remembers the abstractions using M. We memoize the construction of M so that we preserve as many equivalences as possible. For example, suppose we are combining two states in which r_a equals r_b:

$$\Delta_1(\mathrm{r}_a) = v_1 \oplus 1 \qquad \Delta_2(\mathrm{r}_a) = \texttt{sel}(m_2,\ v_2)$$

$$\Delta_1(\mathrm{r}_b) = v_1 \oplus 1 \qquad \Delta_2(\mathrm{r}_b) = \texttt{sel}(m_2,\ v_2)$$

Here, we will create a new v such that $M(v) = (v_1 \oplus 1,\ \texttt{sel}(m_2,\ v_2))$ and set $\Delta_3(\mathrm{r}_a) = \Delta_3(\mathrm{r}_b) = v$ to preserve the only fact that these states have in common: $\mathrm{r}_a = \mathrm{r}_b$.

Finally, we need a way to construct types for the new abstract values in $S_3 = \langle \Delta_3, \Phi_3, m_3, \Gamma_3 \rangle$. For this, the type system must provide an operation `TJoin` such that $\texttt{TJoin}(\tau_1, \tau_2)$ is a supertype of both τ_1 in S_1 and τ_2 in S_2. Specifically, when the following hold

- $\texttt{TJoin}(\tau_1, \tau_2) = \tau$,

- Vars_1, Vars_2, Vars_3, and ρ_A are such that $\text{Vars}_1, \rho_A \models \Gamma_1, \Phi_1$ and $\text{Vars}_2, \rho_A \models \Gamma_2, \Phi_2$, and $\text{Vars}_3, \rho_A \models \Gamma_3, \Phi_3$, and

then it must be true that $(\llbracket\llbracket\tau_1\rrbracket_{\text{Vars}_1}\rrbracket_{\rho_A} \cup \llbracket\llbracket\tau_2\rrbracket_{\text{Vars}_2}\rrbracket_{\rho_A}) \subseteq \llbracket\llbracket\tau\rrbracket_{\text{Vars}_3}\rrbracket_{\rho_A}$. The `TJoin` operation depends on S_1 and S_2, and it can call `join` as needed. When $M(v) = (e_1, e_2)$ we will set $\Gamma_3(v)$ to a supertype of e_1 and e_2, so v can be used as a conservative approximation of these expressions.

Algorithm 3.1 (Widening) *To find a state* $S_3 = \langle \Delta_3, \Phi_3, m_3, \Gamma_3 \rangle$ *that is weaker than both* $S_1 = \langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle$ *and* $S_2 = \langle \Delta_2, \Phi_2, m_2, \Gamma_2 \rangle$*:*

- Let M, Γ_3, and Φ_3 be initially empty.
- **Construct m_3:** Require $\Gamma_1, \Phi_1 \vdash m_1 : \texttt{ValidMem}$ and $\Gamma_2, \Phi_2 \vdash m_2 : \texttt{ValidMem}$. Let $m_3 = v_{mem3}$, $\Gamma_3(v_{mem3}) = \texttt{ValidMem}$, and $M(v_{mem3}) = (m_1, m_2)$.
- **Construct Δ_3 and M:** Require $\text{dom}(\Delta_1) = \text{dom}(\Delta_2)$. For each $\text{r}_i \in \text{dom}(\Delta_1)$ let $\Delta_3(\text{r}_i) = (\texttt{join}\ \ \Delta_1(\text{r}_i)\ \ \Delta_2(\text{r}_i))$. `join` will build M in a memoized fashion.
- **Construct Γ_3:** For each $(v \mapsto (e_1, e_2)) \in M$ (aside from v_{mem3}), let $\Gamma_3(v) = \texttt{TJoin}(\tau_1, \tau_2)$ where $\Gamma_1, \Phi_1 \vdash e_1 : \tau_1$ and $\Gamma_2, \Phi_2 \vdash e_2 : \tau_2$.
- **Construct Φ_3:** Add true facts to Φ_1 and Φ_2 as necessary (see Section 3.1.2). For all pairs $\big(F_1(e_1, \ldots, e_n),\ F_2(e'_1, \ldots, e'_n)\big) \in (\Phi_1 \times \Phi_2)$ where $F_1 = F_2$, try to compute $e''_i = (\texttt{optionalJoin}\ e_i\ e'_i)$ for each $i \leq n$, as explained below. If none of these calls to `optionalJoin` raises the Failure exception, add $F_1(e''_1, \ldots, e''_n)$ to Φ_3.

□

```
join e1 e2 =
  match e1, e2 with
    | e1' op e1'', e2' op e2'' →
                return (join e1' e2') op (join e1'' e2'').
    | sel(m1', e1'), sel(m2', e2') when m1' = m1 and m2' = m2 →
                return sel(v_mem3, (join e1' e2')).
    | c, c → return c.
    | otherwise → if there exists v such that M(v) = (e1, e2) then
                    return v.
                  else
                    Create a fresh v.
                    Add (v ↦ (e1, e2)) to M
                    return v.
```

Figure 3.1: The `join` function used by Algorithm 3.1.

The final step of this algorithm tries to find as many valid facts as possible by doing a comparison of each fact in Φ_1 with each fact in Φ_2. If there is some way to combine the expressions in each fact using M, we add an appropriate fact to Φ_3. One way to compare e_i and e'_i would be to use `join`, but `join` would create new abstract values for any pair of incompatible expressions that isn't in M. Facts about abstract values are almost always useless unless that abstract value also appears in the register or memory state. For example, LessOrEq($v_x, 100$) is useful when some register's value involves v_x, but it means nothing if v_x is fresh.[1] To keep Φ_3 from being cluttered with useless facts, we use a function `optionalJoin`. This function is identical to `join` except that it does not create fresh abstract values. Given two incompatible expressions that are not already in M, `optionalJoin` raises the exception `Failure`. While a quadratic increase in the size of Φ is theoretically possible at each join point, in our experiments with CCured `optionalJoin` prevents this problem. The number of facts in Φ_3 is generally comparable to the number of facts in Φ_1 and Φ_2, whichever is smaller.

If any part of Algorithm 3.1 other than `optionalJoin` fails, we consider it a typechecking error and refuse to certify the program. Such failures include m_1 or m_2 not being in a valid state, or a `sel` expression not being typeable because the address expression is not a valid pointer. If Algorithm 3.1 succeeds, then we are guaranteed that $S_1 \prec: S_3$ and $S_2 \prec: S_3$.

[1] While it would be possible for a type policy to use $F(v)$ when v is fresh to remember a piece of useful information, it is easier to just use a 0-ary constructor.

3.1.1 Testing for Fixpoint

To determine if we are at a fixed point, we must test whether $S_1 \prec: S_2$, where S_1 is the current state of the abstract interpreter and S_2 is the saved state at this label. (Testing $S_3 \prec: S_2$ would also be sufficient, since $S_1 \prec: S_3$.) We can speed up this test, however, by using the information gathered by the widening algorithm. Recall that for a map M, $M_L(v)$ is the expression in S_1 corresponding to v, and $M_R(v)$ is the expression in S_2. If any expression in the range of M_R is an arithmetic operator, `sel` expression, or constant, then we are not at fixed point. In this case, there is some $M(v) = (e_1, e_2)$ such that e_2 is a complex expression that's incompatible with e_1. So S_1 is weaker than (or incomparable to) S_2.

On the other hand, suppose M_R is 1–1, the range of M_R contains only abstract values, and $\text{range}(M_R) = \text{dom}(\Gamma_2)$. Then M is a collection of pairs (e_1, v_2) where each $v_2 \in \Gamma_2$ corresponds to a single expression in S_1. We can build a map $M' = M_L \circ {M_R}^{-1}$ from $\text{dom}(\Gamma_2)$ to expressions that are well-formed in Γ_1. Now M' can be used in Definition 2.2 to test whether $S_1 \prec: S_2$. Moreover, the construction of M insures that $M'[\![\Delta_2(\mathrm{r})]\!] = \Delta_1(\mathrm{r})$ for each register, so we only need to check that the facts and subtyping requirements hold. These steps too can be integrated into Algorithm 3.1.

3.1.2 Adding extra facts

The disadvantage of the pairwise comparison of facts from Algorithm 3.1 is that a simple enumeration of "facts" may not include all true information. Tautologies such as $\text{LessOrEq}(1, 2)$ will usually not appear explicitly in Φ, nor will facts that are implied by

the transitivity or reflexivity of various relations. Ideally, Φ would be closed under logical consequence, but to support enumeration we must choose a sparser representation.

As an example of this problem, consider this code:

```
// Precondition:  r_x is an Int(), and
// r_a points to an Array of length 100.
if (r_x ≥ 100) {
  r_x = 50;
} else {
  // Do nothing
}
r_y = r_a[r_x]
```

We need to ensure the array access on the last line is within bounds. At the end of the `if` statement we perform a join of these two states:

Then branch:	Else branch:
$\Delta_{then}(\mathrm{r}_x) = 50$	$\Delta_{else}(\mathrm{r}_x) = v_x$
$\Delta_{then}(\mathrm{r}_a) = v_a$	$\Delta_{else}(\mathrm{r}_a) = v_a$
$\Phi_{then} = \{\, \mathrm{LessOrEq}(100, v_x) \,\}$	$\Phi_{else} = \{\, \mathrm{LessOrEq}(v_x, 99) \,\}$

where v_x is the initial value of r_x and v_a points to the array. The `if` condition generates the fact $\mathrm{LessOrEq}(v_x, 99)$ in the `else` branch, but Φ_{then} contains no useful facts since v_x is dead. The join of these two states yields

$$\Delta_3(\mathrm{r}_x) = {v_x}'$$

$$\Delta_3(\mathrm{r}_a) = {v_a}'$$

$$\Phi_3 = \emptyset$$

where $M({v_x}') = (50,\ v_x)$ and $M({v_a}') = (v_a,\ v_a)$. Unfortunately, this state is not strong enough to typecheck the array access. We've lost the information $\mathrm{r}_x \leq 99$, even though it's true in both branches.

To solve this problem, we allow type policies to add true facts to the representations of Φ_1 and Φ_2 before they are joined. In this example, the type policy must add LessOrEq$(50, 99)$ to Φ_{then}. This fact, when paired with LessOrEq$(v_x, 99)$ in Φ_{else}, yields LessOrEq$(v_x', 99)$ in the joined state since $M(v_x') = (50,\ v_x)$. Depending on what information is needed across basic blocks, type policies might need add information about dependencies among values, as in this case, or close each set of facts under transitivity or other algorithms. We give examples of one useful heuristic for facts about constants. This heuristic is the only one needed by the type policies we have implemented.

Constants. One way to determine true facts that should be added to Φ_1 (the Φ_2 case is symmetric) is to consider all of the pairings $(c, e),\ (c', e'),\ \ldots$ in the range of M where the left expression is a constant. Here, constants in S_1 have been paired with incompatible expressions from S_2 (which might themselves be constants). Since we will soon be abstracting the constants into abstract values in the new state, we should make note of any useful information about them.

For each $\phi \in \Phi_2$, let $\phi' = \phi\left[{}^{c}\!/\!_{e}\right]\left[{}^{c'}\!/\!_{e'}\right]\ldots$ be the result of inserting these constants into ϕ in place of the expressions from S_2 that they are paired with. If ϕ' is a constant fact (it does not refer to any abstract values) and it is true in S_1, then add ϕ' to Φ_1. When performing the join, $\phi' \in \Phi_1$ will match $\phi \in \Phi_2$, and a corresponding fact will be added to the new state. In the example above, $(50, v_x) \in \text{range}(M)$ and LessOrEq$(v_x, 99) \in \Phi_{else}$, so we consider $\phi' = \text{LessOrEq}(50, 99)$. This is a true statement, so we add it to Φ_1. Note that if ϕ' contains an abstract value then it makes no sense to ask whether it is true in S_1, since the abstract values from Γ_2 have no meaning in S_1.

```
struct Foo {
   char* name;
   int info;
};


struct Foo data[100];
```

```
int* foo(unsigned int x) {
   int* p;
   if (x < 100) {
      pp = &data[x].info;
   } else {
      pp = &data[0].info;
   }
   return pp;
}
```

Figure 3.2: Pointer arithmetic example: A function that creates a pointer to the second field of one of the elements of the `data` array.

3.2 Arithmetic

One of the most important issues in abstract interpretation of assembly code is how to deal with arithmetic in memory addresses. Because the compiler generates low-level, optimized code for memory access operations, it is important that the abstract interpreter understand the same arithmetic identities that the compiler does. The `join` and `compare` functions presented so far treat arithmetic operators as uninterpreted, but by itself this is not precise enough for many memory operations. As with the addition of extra facts in Section 3.1.2, our solution will take the form of a preprocessing step that allows the join of interpreted facts to preserve the necessary information.

Consider the program in Figure 3.2. Function `foo` returns a pointer to the second field of either the $\texttt{x}^{th}$ or 0^{th} element of the `data` array, depending on whether x is within bounds. If $\_data$ is the assembly label corresponding to the start of the global array, the assembly code in the `then` branch will assign to `pp` the symbolic value $\_data \oplus (2 \otimes v_x) \oplus 1$. The coefficient of 2 for `x` reflects the fact each element of `data` is two words in size, and the "+1" reflects the fact that `pp` points to the second word in one of those elements. (Recall

that for simplicity our examples assume that memory is accessed by words, not by bytes. Assembly labels such as *_data* can appear in symbolic expressions like any other constant.) The `else` branch, meanwhile, yields the symbolic expression $\_data \oplus 1$. At the join point before the `return` statement, how do we merge the two expressions?

Invoking `join` on $\_data \oplus (2 \otimes v_x) \oplus 1$ and $\_data \oplus 1$ will result in either $v' \oplus 1$ or $data \oplus v'$, depending on whether the first expression is represented as $(\_data \oplus (2 \otimes v_x)) \oplus 1$ or $\_data \oplus ((2 \otimes v_x) \oplus 1)$. Neither of these represents the information that they should: `pp` points to an `info` field in the `data` array. The best way to fix this while still using the efficient join procedure of Algorithm 3.1 is to rewrite the second expression as $\_data \oplus (2 \otimes 0) \oplus 1$ before widening. The join of uninterpreted functions on these two expressions is now $\_data \oplus (2 \otimes v') \oplus 1$ where $M(v') = (v_x,\ 0)$, which is exactly what we'd like. The machine arithmetic operators $\oplus$ and $\otimes$ form a commutative ring, so we can rewrite symbolic expressions using the familiar associative, commutative, and distributive identities.

3.2.1 Complicated indexing expressions

We use this approach for all expressions involving pointer arithmetic. If a value points to an array, we canonicalize the symbolic expression to make each array index explicit, as in $\_data \oplus (2 \otimes 0) \oplus 1$ above. Given a term that is the sum of one or more expressions, we ask the type policy to determine which addend is the *pointer term*. This term (*_data* in the above example) is an expression not involving arithmetic that points to an object in memory. All of the other addends are assumed to represent offsets within this object. By looking at the type of a pointer term p, the type policy can identify any arrays that occur within it and know what kind of indexing to expect. In code compiled from C, we might

$$
\begin{array}{rrcl}
\text{object layout} & \kappa & ::= & \sigma :: \kappa \;\mid\; (\kappa \times n) :: \kappa \;\mid\; \mathbf{nil} \\
\text{constant} & n, \text{ where } & & n > 1 \\
\text{object size:} & |\sigma :: \kappa_2| & = & |\sigma| \;+\; |\kappa_2| \\
& |(\kappa_1 \times n) :: \kappa_2| & = & n \cdot |\kappa_1| \;+\; |\kappa_2| \\
& |\mathbf{nil}| & = & 0 \\
& |\sigma| & = & \text{the number of fields in } \sigma
\end{array}
$$

Figure 3.3: The object layouts κ supported by our pointer arithmetic canonicalization, and the size $|\kappa|$ in words of an object κ.

find single-dimensional arrays such as the one in Figure 3.2, multidimensional arrays, or arrays of `structs` that themselves contain arrays.

When dealing with the common case of single-dimensional arrays, it is relatively easy to rewrite symbolic expressions to make the array indices explicit, as in the above example. We consider here how this canonicalization step works in the general case, when we have nesting of arrays and C structures.

Figure 3.3 shows the memory objects we consider. An object is a **nil**-terminated sequence of possibly-dependent records σ from Chapter 2 and/or arrays. An array $(\kappa \times n)$ is a sequence of n objects that each have layout κ. While the fields of a heterogeneous object $\sigma_1 :: \sigma_2 :: \ldots :: \mathbf{nil}$ can only be accessed via a constant offset, elements of an array subobject can be indexed by a runtime value.

For example, consider the C declaration

```
struct element{
   int * f0;
   int f1[10];
};
struct element bar[7];
struct element bar2[5];
```

The arrays `bar` and `bar2` consist of consecutive $\kappa_{\texttt{element}}$ objects, where $\kappa_{\texttt{element}} = \text{Ptr}_{\text{Int}()} :: \big((\Sigma_s.\langle\ 0 : \text{Int}()\ \rangle :: \textbf{nil}) \times 10\big) :: \textbf{nil}$.[2] We use f_i to refer to the i^{th} field of an object, whose type is specified by the i^{th} element in the layout list. The C address `bar[`x`].`f_1`[`y`]` would be compiled as

$$\begin{aligned}\texttt{bar[}x\texttt{].}f_1\texttt{[}y\texttt{]} &= \_bar \oplus (|\kappa_{\texttt{element}}| \otimes x) \oplus \text{offset}(f_1, \kappa_{\texttt{element}}) \oplus (|\Sigma_s.\langle\ 0 : \text{Int}()\ \rangle| \otimes y) \\ &= \_bar \oplus (11 \otimes x) \oplus 1 \oplus (1 \otimes y)\end{aligned}$$

where $|\kappa|$ is the size of an object as defined in Figure 3.3 and $\text{offset}(f_i, \kappa)$ is the distance between the start of a κ object and the start of the i^{th} field in the object. In this case, $|\kappa_{\texttt{element}}| = 11$ and $\text{offset}(f_1, \kappa_{\texttt{element}}) = 1$.

The arrays embedded within structures or other arrays, such as f_1 in `struct element`, must generally have a fixed size or else the compiler would not know how to generate indexing expression $|\kappa_{\texttt{element}}| \otimes x$. However, "top-level" arrays can be dynamically allocated and have a size that's determined at runtime. The fact that `bar` contains 7 elements is not used anywhere when generating index expressions for `bar[`x`].`f_1`[`y`]`. Therefore we identify `bar` simply as "a pointer to consecutive $\kappa_{\texttt{element}}$ objects" rather than as a pointer to a $(\kappa_{\texttt{element}} \times 7) :: \textbf{nil}$ object. The length of `bar` is useful only to the type policy, for checking that the array indices of a memory access are within bounds. The Array

[2] For the sake of readability, we abbreviate the types of pointers to one-word records, using as "Ptr_t" instead of $\text{Ptr}_{\Sigma_s.\langle\ 0:t\ \rangle}$.

constructor from Chapter 2 is an example of a variable-length, top-level array that cannot, and need not, be expressed as $(\kappa \times n) :: \mathbf{nil}$ for constant n.

Suppose during a widening operation we needed to join the expression $\_bar \oplus (11 \otimes x) \oplus 1 \oplus (1 \otimes y)$ with a pointer to $\texttt{bar2[5].}f_1\texttt{[2]}$, a constant offset into a different array of the same type that compiles to $\_bar2 \oplus 58$. To preserve the index structure, we first rewrite $\_bar2 \oplus 58$ as $\_bar2 \oplus (11 \otimes 5) \oplus 1 \oplus (1 \otimes 2)$ and then join the corresponding parts of the expressions, yielding

$$\texttt{join}(\_bar, \_bar2) \ \oplus \ (11 \otimes \texttt{join}(x, 5)) \ \oplus \ 1 \ \oplus \ (1 \otimes \texttt{join}(y, 2)).$$

Notice that after the rewriting step, this is still just a join of uninterpreted functions. We use the ternary (in this case) function "$[\cdot] \oplus (11 \otimes [\cdot]) \oplus 1 \oplus (1 \otimes [\cdot])$" instead of the binary functions $\oplus$, $\otimes$, etc. When joining two expressions, if the coefficients in the canonical expressions are identical then we join the subexpressions; otherwise we abstract everything to a fresh value. If the pointer terms in the two expressions being joined have the same type, then the canonical expressions will have the same form. When the pointer terms have different types it may be possible to do better than complete abstraction, but this occurs rarely in practice and we leave it for future work.

Figure 3.4 shows how to canonicalize expressions. Given a symbolic expression e, the type policy first identifies the pointer term p and offset e' such that $e = p \oplus e'$, using type policy-specific knowledge about which abstract values represent pointers into the heap. The type policy also provides the object layout κ corresponding to the type of p (*i.e.* p points to one or more consecutive κ objects). We rewrite e as $p \oplus \texttt{canon}(e', \kappa)$, where $\texttt{canon}(e', \kappa)$ is an expression equivalent to e' that has been canonicalized according to layout κ.

```
//"canon (e, κ)" takes an expression that indexes into an array of
// κ objects, and returns an expression equivalent to e in canonical form.
canon(e, κ ) =
   let e_q, e_r = divide(e, |κ|) in
   //e is an offset into element e_q of this array.
   return (|κ| ⊗ e_q) ⊕ canonField(e_r, κ).


//"canonField (e, κ)" takes an expression representing an offset
// into a single κ object, and returns an expression equivalent to e
// in canonical form.
canonField(e, κ ) =
   let (c1⊗e1) ⊕ (c2⊗e2) ⊕ ... ⊕ c be the flattened form of e in
   match κ with
      |  σ :: κ2 → if c ≥ |σ| then
                      return |σ| ⊕ canonField(e − |σ|, κ2).
                   else
                      //e is an offset into σ, so it must be constant.
                      if e = c then return c.
                      else Error.
      |  (κ1 × n) :: κ2 → if c ≥ n · |κ1| then
                             return (n⊗|κ1|) ⊕ canonField(e − n⊗|κ1|, κ2).
                          else
                             return canon(e, κ1).
      |  nil → Error.


//"divide(e, n)" (where n is a constant) returns symbolic expressions
// (e_q, e_r) such that e is equivalent to e_q ⊗ n ⊕ e_r and every coefficient
// in the flattened form of e_r is within the range [0, n).
divide(e, n) =
   let (c1⊗e1) ⊕ (c2⊗e2) ⊕ ... ⊕ c be the flattened form of e in
   let e_q = ⌊c1/n⌋ ⊗ e1 ⊕ ⌊c2/n⌋ ⊗ e2 ⊕ ... ⊕ ⌊c/n⌋ in
   let e_r = (c1 − n·⌊c1/n⌋) ⊗ e1 ⊕ (c2 − n·⌊c2/n⌋) ⊗ e2 ⊕ ... ⊕ (c − n·⌊c/n⌋) in
   return (e_q, e_r).
```

Figure 3.4: The expression canonicalization function.

`canon` rewrites expressions as suggested by the examples above. The key step is the division, where we break an array offset e into an expression $n \otimes e_q \oplus e_r$ where e_q is an index into the array and e_r is an offset within the ${e_q}^{th}$ element of the array. The helper function $\texttt{canonField}(e_r, \kappa)$ handles the case where e_r is an offset into a single object with layout κ, rather than an array of such objects. `canon` always terminates because each successive call to `canonField` uses a smaller type layout κ. For example, the expression $\_bar2 \oplus 58$ would be canonicalized as follows:

$$
\begin{aligned}
\_bar2 \oplus 58 &= \_bar2 \oplus \texttt{canon}(58, \kappa_{element}) \\
&= \_bar2 \oplus (11 \otimes 5) \oplus \texttt{canonField}(3, \kappa_{element}) \\
&= \_bar2 \oplus (11 \otimes 5) \oplus 1 \oplus \texttt{canonField}(2, \left((\Sigma_s.\langle\ 0 : \text{Int}()\ \rangle :: \textbf{nil}) \times 10\right) :: \textbf{nil}) \\
&= \_bar2 \oplus (11 \otimes 5) \oplus 1 \oplus \texttt{canon}(2, (\Sigma_s.\langle\ 0 : \text{Int}()\ \rangle :: \textbf{nil})) \\
&= \_bar2 \oplus (11 \otimes 5) \oplus 1 \oplus (1 \otimes 2) \oplus \texttt{canonField}(0, (\Sigma_s.\langle\ 0 : \text{Int}()\ \rangle :: \textbf{nil})) \\
&= \_bar2 \oplus (11 \otimes 5) \oplus 1 \oplus (1 \otimes 2) \oplus 0
\end{aligned}
$$

The $\oplus 0$ term in the last line can be omitted from the final result.

Figure 3.4 also shows how we do division of symbolic expressions. The first step is to convert the expression to *flattened form*. A symbolic expression in flattened form is written as $(c_1 \otimes e_1) \oplus (c_2 \otimes e_2) \oplus \ldots \oplus (c_k \otimes e_k) \oplus c$ where each c_i is a constant, each e_i is not an addition, a subtraction, or a constant, and for all $i < j$, $e_i \prec e_j$ according to some strict order $\prec$.[3] The e_i expressions represent (parts of) array indices, so we leave them untouched during canonicalization; we care only about constant terms and coefficients. A few arithmetic operators are dealt with in a preprocessing step: a left bitshift by a constant is changed to multiplication by a power of two, and the xor of a value and itself equals zero (this last idiom is common in x86 code). After preprocessing all arithmetic operations

[3]The choice of an order $\prec$ is arbitrary. We use it to guarantee that each e_i is unique, and that there is a unique flattened form for each expression.

other than $\oplus$, $\ominus$, and $\otimes$ are treated as uninterpreted. Complicated operators may appear in array index expressions in the original program, but they are unlikely to be generated by the compiler during optimization.

3.2.2 Completeness

The soundness of the arithmetic canonicalization is easy to verify by inspecting Figure 3.4. Since $\texttt{canon}(e, \kappa)$ is equivalent to e, we can replace e with $\texttt{canon}(e, \kappa)$ at any time without changing the meaning of a symbolic state. A more interesting question is whether this approach is complete: can `canon` correctly decompose all arithmetic expressions? Unfortunately, the answer is no. Consider the C struct

```
struct twoArrays {
    int  f0[10];
    int* f1[20];
};
```

The expression $p \oplus x \oplus 10$, where p points to a `twoArrays` object, could mean either $p.f_1[x]$ or $p.f_0[10 \oplus x]$ depending on whether x is negative. `canon` implicitly assumes that abstract values are nonnegative, so it will canonicalize this expression as an access to f_1. But $p.f_0[10 \oplus x]$ is perfectly legal C code when $-10 \leq x < 0$. In this section, we give a set of sufficient conditions for `canon` to correctly identify the fields being accessed.

Since our object layouts model the nested arrays and structs of C, we introduce C-like notation to discuss how these objects are accessed in source code. As seen in Figure 3.5, a source code offset α is an alternating sequence of array accesses $[e]$ and field accesses $.f_i$. The field access at the end of the list denotes a field in an atomic record σ. Given an offset α into an array of κ objects, we define the *standard compilation* of α as the expression e

$$\text{offset} \quad \alpha \quad ::= \quad [e]\beta$$

$$\text{offset within an array element} \quad \beta \quad ::= \quad .f_i\alpha \mid .f_i$$

$$\frac{\kappa \vdash_F \beta : e'}{\kappa \vdash [e]\beta : |\kappa| \otimes e \ \oplus \ e'}$$

$$\frac{\kappa \vdash \alpha : e}{(\kappa \times n) :: \kappa' \vdash_F .f_0\alpha : e} \qquad \frac{0 \leq i < |\sigma|}{\sigma :: \kappa' \vdash_F .f_i : i}$$

$$\frac{\kappa' \vdash_F .f_i\alpha : e}{\sigma :: \kappa' \vdash_F .f_{i+1}\alpha : |\sigma| \oplus e} \qquad \frac{\kappa' \vdash_F .f_i\alpha : e}{(\kappa \times n) :: \kappa' \vdash_F .f_{i+1}\alpha : n \otimes |\kappa| \oplus e}$$

Figure 3.5: The source-code offsets α and how they are compiled. When α is an offset into an array of κ objects, its standard compilation is given by $\kappa \vdash \alpha : e$. When β is an offset into a single κ object, its standard compilation is given by $\kappa \vdash_F \beta : e$.

given by the judgment $\kappa \vdash \alpha : e$. The mutual definitions of α and β parallel the mutual definitions of `canon` and `canonField`, and for good reason: we will show that the output of canonicalization matches the standard compilation for that offset.

We assume that compilers will generate code for any offset α in the source code by taking the standard compilation of α and simplifying or optimizing the result using arithmetic identities. If the original offset is *simple*, as defined below, we will be able to correctly decompose the optimized version.

Definition 3.1 (Simple offsets) *An offset α is* simple *if each array index expression in α is either a constant that is within the array bounds, or an uninterpreted expression such as an abstract value or* `sel` *expression.*

We assume compilers do not do any optimizations within uninterpreted expressions. By forbidding arithmetic in indices, we rule out the $p.f_0[10 \oplus x]$ example from earlier. Simplicity of offsets is not necessary for correct decomposition, but it is a sufficient condition for the theorem stated here. We now present the completeness result and the lemmas that are needed for its proof. We formalized Lemmas 4 and 5 and proved them correct using the Coq prove assistant [Coq06].

Lemma 4 (Uniqueness of the flattened form) *Let e_1 and e_2 be symbolic expressions involving only addition, subtraction, and multiplication of constants and uninterpreted expressions. Then e_1 and e_2 are equivalent (they have the same value for any possible assignment to the uninterpreted terms) iff they have the same flattened form.*

□

Lemma 5 (Limitation on the constants in a simple offset) *Let β be a simple offset within an array element, and suppose κ and e are such that $\kappa \vdash_F \beta : e$.*

Let $(c_1 \otimes e_1) \oplus \ldots \oplus (c_n \otimes e_n) \oplus c_0$ be the flattened form of e. Then for all $0 \leq i \leq n$, $0 \leq c_i < |\kappa|$.

This lemma states that if e is an offset within a single object with layout κ, then any constants appearing in e are less than $|\kappa|$. Intuitively, this should hold because e is an offset into an κ object, so the value of e itself is less than $|\kappa|$ whenever all array accesses are within bounds.

This lemma is proved by induction on the length of offset β. The corresponding invariant for array offsets α is that when α is a simple offset into an array of n objects of type κ, $n > 1$, and $\kappa \vdash \alpha : e$, then $0 \leq c < n \otimes |\kappa|$ for all constants c in the flattened form of e. When the array index is a symbolic value, there exists e' such that $e = (|\kappa| \otimes v) \oplus e'$. $|\kappa| < n \otimes |\kappa|$ because $n > 1$ by definition of object layouts κ, and κ cannot be empty. Furthermore, the constants in e' are strictly less than $|\kappa|$ by induction. When the array index is a constant, we take advantage of the requirement that constant array indices in simple offsets are within bounds.

□

Theorem 4 (Canonicalization of optimized expressions) *Let α be a simple offset, and suppose κ and e are such that $\kappa \vdash \alpha : e$. Let e' be equivalent to e. Then* `canon`(e', κ) *is syntactically equal to e.*

Similarly, if β is a simple offset into a single object, $\kappa \vdash_F \beta : e$, and e' is equivalent to e, then `canonField`(e', κ) *is syntactically equal to e.*

This theorem states that even in the presence of arithmetic optimizations, `canon` recovers all of the structure in e, which itself directly reflects the structure in α.

Theorem 4 is proved by induction on the lengths of object layouts. First, note that $\texttt{canon}(e', \kappa) = \texttt{canon}(e, \kappa)$ and $\texttt{canonField}(e', \kappa) = \texttt{canonField}(e, \kappa)$. The first step in both procedures is to convert the symbolic expressions to flattened form, and e' and e have the same flattened form by Lemma 4. Therefore, we can ignore the difference between e and e'.

The cases of β follow directly from the definition of `canonField` and Lemma 4. Consider then the two cases of α: $\alpha = [v]\beta$ and $\alpha = [i]\beta$ for constant i. For the variable-index case, $e = (|\kappa| \otimes v) \oplus e_{rest}$, where $\kappa \vdash_F \beta : e_{rest}$. Let e'_{rest} be the flattened form of e_{rest}. If v does not occur in e_{rest}, then the flattened form of e is $(|\kappa| \otimes v) \oplus e'_{rest}$, or some permutation thereof. Therefore $\texttt{divide}(e, |\kappa|) = (v, e'_{rest})$ thanks to Lemma 5.[4] By the inductive hypothesis, $\texttt{canonField}(e'_{rest}, \kappa) = e_{rest}$, and we have verified that $\texttt{canon}(e, \kappa) = (|\kappa| \otimes v) \oplus e_{rest}$. If v does occur in e'_{rest} (because some nested array also uses v as an index), then the flattened form of e is $(|\kappa| + c) \otimes v \oplus \ldots$ for some constant c. By Lemma 5, $0 \leq c < |\kappa|$ because e_{rest} is an offset into a β object, so the result of the division is the same: $\texttt{divide}(e, |\kappa|) = (v, e'_{rest})$ and $\texttt{canon}(e, \kappa) = (|\kappa| \otimes v) \oplus e_{rest}$.

In the constant-index case $\alpha = [i]\beta$, $e = (|\kappa| \otimes i) \oplus e_{rest}$. Let e'_{rest} be the flattened form of e_{rest}. The flattened form of e will have a constant term $(|\kappa| \cdot i + c)$ where c is the constant term of e'_{rest}. By Lemma 5, $0 \leq c < |\kappa|$ because e_{rest} is an offset into a β object, so as before we get $\texttt{divide}(e, |\kappa|) = (i, e'_{rest})$ and $\texttt{canon}(e, \kappa) = |\kappa| \otimes i \oplus e_{rest}$. □

[4] `divide` as presented in Figure 3.4 would return $(1 \otimes v, e'_{rest})$. To get exact syntactic equality between `canon` and the standard compilation, we simplify the result of `divide` to drop the unnecessary "$1\otimes$".

Now that we have a theorem that `canon` can account for compiler transformations, we can discuss the completeness of widening. Recall that our widening operation involves first canonicalizing each expression and then using the join of uninterpreted functions. Let α_1 and α_2 be simple offsets for the same object layout κ that differ only in the values of array indices, and define $\texttt{join}(\alpha_1, \alpha_2)$ as the offset with the same structure as α_1 and α_2 where each array index is the join of the corresponding array indices in α_1 and α_2. Since `canon` recovers all of the structure in an offset, we know that the structure is present in the joined expression as well:

Corollary 1 (Join of canonicalized expressions) *Let α_1 and α_2 be simple offsets for the same object layout κ that differ only in the values of array indices. For $i \in \{1, 2\}$, let $\kappa \vdash \alpha_i : e_i$ and let ${e_i}'$ be equivalent to e_i.*

Then $\kappa \vdash \texttt{join}(\alpha_1, \alpha_2) : \texttt{join}(\texttt{canon}({e_1}', \kappa), \texttt{canon}({e_2}', \kappa))$.

Proof is by induction on the structure of α_1 and α_2. By Theorem 4, $\texttt{canon}({e_i}', \kappa) = e_i$. It's easy to see in Figure 3.5 the one-to-one correspondence between an offset α and the standard compilation e. Since α_1 and α_2 differ only in array indices, the same is true for e_1 and e_2.

□

In addition to the sufficient condition given by the definition of simple offsets, Theorem 4 and Corollary 1 should also hold if the array indices are a sum or product of nonnegative terms, although we have not proved this formally. The common case of objects containing only one array (and no nested arrays) can also be handled in a complete fashion.

```
struct Foo {
   char* name;
   int info;
};
struct Foo grid[7][5];

//Precondition:  col < 5
int verticalSum(int col) {
   int i = 0;
   int sum = 0;
   while(i < 7) {
      sum += grid[i][col].info;
      i++;
   }
   return sum;
}
```

```
int verticalSum(int col) {
   int i = 0;
   int sum = 0;
   // let p = &grid[0][col].info:
   int * p = (int*)grid + 2*col + 1;
   while(i < 7) {
      sum += *p;
      p += 10;
      i++;
   }
   return sum;
}
```

Figure 3.6: Multidimensional array example: `verticalSum` is a function that computes the sum of the `info` fields for a given column. On the right, a version of the code as it might appear after optimization by the compiler.

Here, there is no ambiguity about which array is accessed by which variable term, so the problem described at the beginning of this chapter does not arise.

Also note, however, that the completeness results in this section apply only if the pointer terms in the two expressions have the same type (*i.e.* the same object layout). Given pointer terms of different types — say, a pointer to the f_1 field of a `struct twoArrays` and a pointer to an array of `int` pointers — the join function will fail because the canonical expressions have different lengths. To handle this case, we would need to replace our naive join of uninterpreted functions with a join function that looks for a common suffix shared by the two terms.

3.2.3 Multidimensional array example

We give an example of a program using a two-dimensional array to show how the canonicalization helps during joins. This program will also demonstrate how our abstract interpreter helps recover information that is obscured by compiler optimizations. Figure 3.6 shows two versions of a function that computes the sum of the `info` fields of a given column in the global array `grid`. The version on the left is how a programmer might implement this function. The instruction "`sum += grid[i][col].info`" reads from memory at address $\_grid \oplus (10 \otimes \texttt{i}) \oplus (2 \otimes \texttt{col}) \oplus 1$, but computing this address during each iteration of the loop would be unnecessarily expensive. To fix this, a compiler might perform a loop induction variable optimization [CK77] as shown on the right in Figure 3.6. Here, `p` is a pointer to the next `info` field to be read, and we advance `p` by an entire row in each iteration of the loop. (The next step in the optimization might be to eliminate the variable `i` completely, but this isn't always possible if `i` is used elsewhere in the loop.)

When the abstract interpreter first enters the loop, the `p` register will have symbolic value $\_grid \oplus (2 \otimes v_{\texttt{col}}) \oplus 1$. After symbolically executing the body of the loop, we return to the top with $\mathrm{r}_{\texttt{p}} = \_grid \oplus (2 \otimes v_{\texttt{col}}) \oplus 1 \oplus 10$, and must join this with the initial state for the loop. Since the pointer term $\_grid$ in each of these expressions has element sizes {10, 2}, we canonicalize the expressions as $\_grid \oplus (10 \otimes 1) \oplus (2 \otimes v_{\texttt{col}}) \oplus 1$ and $\_grid \oplus (10 \otimes 0) \oplus (2 \otimes v_{\texttt{col}}) \oplus 1$. The join of these expressions is

$$\mathrm{r}_{\texttt{p}} \quad = \quad \_grid \ \oplus \ (10 \otimes v_i) \ \oplus \ (2 \otimes v_{\texttt{col}}{}') \ \oplus \ 1$$

where $M(v_i) = (1, 0)$ and $M(v_{\texttt{col}}{}') = (v_{\texttt{col}}, v_{\texttt{col}})$. This result preserves all of the commonality between the two expressions. A second iteration through the loop using this value for

r_p will determine that this state is a fixed point for the abstract interpretation.

This example also illustrates why the memoization of M in `join` is necessary. The register holding the value of `i` is 0 initially, and 1 after the first iteration through the loop. When joining these expressions, the result is the same v_i that appears in r_p, since M already has the binding $v_i \mapsto (1, 0)$. On the second iteration the loop guard generates the fact $LessEq(v_i, 6)$; together with the function precondition $LessEq(v_{\texttt{col}}, 4)$ it is easy for the type policy to verify that the array indices in r_p are within bounds. If we had compared 10 with 0 instead of comparing $10 \otimes 1$ with $10 \otimes 0$ when joining r_p, the abstract interpreter would not have discovered the relationship between r_p and r_i.

3.3 Performance of abstract interpretation

To show that our abstract interpretation terminates, we must show that you can only widen an abstract state a finite number of times without reaching a fixpoint. We impose an additional constraint on the type policy's `TJoin` function: that the new types generated by `TJoin` be monotonically increasing in the τ_2 parameter, and that repeated calls to `TJoin` must reach a fixed point.

Required Lemma 6 (Widening of `TJoin`) *There exists a function* $\text{rank}(\tau)$ *from register types in the type policy to* $\mathbb{N}$ *such that for all* $S_1, S_2, \tau_1,$ *and* τ_2*, if* $\texttt{TJoin}(\tau_1, \tau_2) = \tau_3$ *then* $\tau_3 = \tau_2$ *or* $\text{rank}(\tau_3) < \text{rank}(\tau_2)$

Intuitively, the rank is the distance in the TJoin lattice between a type and Int(), or whatever the greatest possible supertype is in the policy. Using this requirement, we can state that repeated calls to Algorithm 3.1 must eventually reach a fixed point when

results of a previous iteration are used as state S_2 in the next widening. Define $|\Delta|$ to be the total number of symbolic nodes other than abstract values among the expressions in Δ. For any widening $S_3 = \langle \Delta_3, \Phi_3, m_3, \Gamma_3 \rangle$ of input states $S_1 = \langle \Delta_1, \Phi_1, m_1, \Gamma_1 \rangle$ and $S_2 = \langle \Delta_2, \Phi_2, m_2, \Gamma_2 \rangle$, either we are at a fixed point ($S_3 \prec: S_2$) or one of the following holds:

- The map M produced by Algorithm 3.1 does not satisfy the requirement "for all r $\in$ dom(Δ_2), $M[\![\Delta_2(\mathrm{r})]\!] = \Delta_3(\mathrm{r})$" from Definition 2.2. Therefore, some expression node in Δ_2 has been abstracted away, and $|\Delta_3| < |\Delta_2|$.

- M does satisfy the requirement above, but there is some "$v \mapsto \tau_3$" $\in \Gamma_3^r$ where τ_3 is different from the type of $M_R(v)$ in Γ_2^r. By the `TJoin` requirement above, this can only happen a finite number of times for each type in the range of Γ_3^r.

- There is some fact $F(e_1, \ldots, e_n) \in \Phi_2$ that is not present in Φ_3.

Since each successive call to widen must weaken the state in at least one of these ways, we can construct a (very) conservative upper bound for the number of iterations needed to reach fixed point from an initial state $\langle \Delta_0, \Phi_0, m_0, \Gamma_0 \rangle$:

$$|\Delta_0| \cdot n \cdot (|\Delta_0| + |\mathrm{dom}(\Delta_0)|) \; + \; |\Phi_0|$$

where n is the smallest value such that for any $\Gamma_1, \Phi_1, \tau_1, \Gamma_2, \Phi_2, x_0$, `TJoin` terminates in at most n steps; $(|\Delta_0| + |\mathrm{dom}(\Delta_0)|)$ is a conservative approximation of the size of Γ at each step, and $|\Phi_0|$ is the number of facts in the enumeration of Φ_0.

While this guarantees eventual termination, we also want a widening operation that terminates quickly in practice. One way to do this is to limit the height of the `TJoin` lattice. In the CCured policy without physical subtyping, this lattice has height 3: arrays

(rank 2), pointers to single objects (rank 1), and integers (rank 0). We have also greatly improved the performance of our abstract interpretation through the use of liveness analysis and worklist optimizations.

3.3.1 Worklist optimizations

When doing abstract interpretation, it is important to implement the worklist carefully. Exploring basic blocks in the wrong order can mean that much unnecessary work will be done before fixed point is reached. Consider the control-flow graph in Figure 3.7. The function is acyclic, so each basic block only needs to be evaluated once, e.g. by processing them in the order they are labeled in the figure. However, a naïve depth-first traversal of the graph (*i.e.* a stack-based worklist) would likely take exponential time. The abstract interpretation would first evaluate the entire DAG rooted at D using the state that results from evaluating A,B. After evaluating C, the engine would redo the DAG rooted at D using the join of B's poststate and C's poststate, making the first evaluation of D a waste of time. A breadth-first traversal (*i.e.* a queue-based worklist) is better, but still worst-case exponential even for acyclic code, because some paths in the DAG are longer than others, as in DEH *vs.* DFGH.

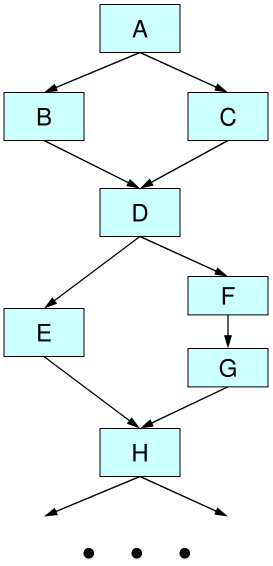

Figure 3.7: The start of a sample function's control-flow graph.

A good way to organize the worklist would be to first perform an interval analy-

Program	Depth-first		Depth-first with Liveness		Sorted by PC with Liveness		Total Speedup
	joins per BB	time (sec)	joins per BB	time (sec)	joins per BB	time (sec)	
unoptimized `go`	4.81	646	4.30	379	2.09	113	82%
optimized `go`	4.38	168	4.16	131	3.46	96	43%

Table 3.1: Performance improvement in the CCured verifier due to abstract interpretation optimizations on the Spec95 `go` program. "Joins per BB" is the number of times each basic block was explored until reaching a fixed point, averaged over all of the basic blocks in the program.

sis [Muc97], which will break the control-flow graph into various loops and acyclic regions. However, we have had good luck in our implementation with a much simpler heuristic that requires no control-flow analysis: sort the worklist such that blocks that appear first in the object file are explored first. Here, we rely on the fact that our target compiler emits basic blocks in a sensible order corresponding the source-level control-flow structures such that the only backwards jumps are the backedges of proper loops. This ordering is optimal for acyclic code, and works well for loops.

Our original implementation used a depth-first ordering of the worklist, because it is often easier to debug the abstract interpretation by considering an entire path through the function. As seen in Table 3.1, however, the performance of the system improved significantly when we began sorting the worklist in the order described above. The "Depth-first" column in this table shows the performance of our original algorithm, while the "Depth-first with Liveness" column shows the performance when using a liveness analysis. (By determining which registers are live at the start of each basic block, we can avoid joining symbolic expressions that will not be used. Besides the computational cost of the join, joining mismatched, dead expressions could cause us to believe that the abstract interpre-

tation is not at a fixed point even though the register in question is irrelevant.) The final results show the performance with both a liveness analysis and worklist sorting. Note that unoptimized code takes longer to verify than optimized code, due to increased code size.

3.4 Related work

Type Inference

Our type inference system uses many ideas from Chang et al.'s Coolaid verifier [CCNS05]. Coolaid was designed to help students in undergraduate compiler courses check whether their compilers are generating type-safe assembly code. Therefore Coolaid, like our system, has no control over the compiler and must infer register types at each program point. Checking that certain safety properties hold, as we and Coolaid do, is a simpler version of translation validation [PSS98, Nec00], which seeks to prove that the compiled code has the same semantics as the source code.

As is usually the case with abstract interpretation, the hardest part of the algorithm is the join of two states. Much of the information we need to maintain across join points can be represented as expressions of uninterpreted functions [GTN04, RKS99]. However, Section 3.2 shows that our join algorithm also needs to have some understanding of linear arithmetic to support pointer indexing expressions. For this we chose a domain-specific operation that canonicalizes pointer expressions rather than more general logics such as [GN04] that would be harder to incorporate into our domain.

Other methods of certifying object code

There has been much work done to certify that binary code adheres to various safety properties. Colby et al. [CCH+03] survey several approaches, such as TAL and PCC, and describe the general problem of certifying safety in mobile code, including issues of how such certifications can be communicated to the end user.

Typed Assembly Language [MCG+99, MWCG99] is used as a compilation target for Popcorn, a subset of C. TAL includes many useful features, including flow-sensitive types for registers so that register types can change from one instruction to the next; typechecking that is done one basic block at a time; existential types; and support for stack-based compilation schemes [MCGW98]. But TAL does not support the dependent types that we need for CCured, and it assumes that assembly code is generated by a specially-written, type-preserving compiler that can emit the invariant for each basic block.

Proof-Carrying Code [Nec97, CLN+00] packages object code with a checkable proof of safety. The original implementations of PCC targeted specific type policies, such as Java's type system [CLN+00]. Recent projects such as LTT [CV02] and work by Shao et al. [SSTP02] seek a general type system for certified code that is not tied to any one source language. A low-level type system permits use of a wide variety of proofs and proof techniques, and it allows code from multiple source languages to be combined safely. But these two systems do not yet target imperative languages, making them impractical for the applications we are considering. An advantage of proof-carrying code is that machine-checked proofs provide a greater assurance of soundness than you get with our system; our soundness proofs exist only on paper and so are disconnected from the implementation.

Walker [Wal00] proposes an extensible certification system for safety properties that can be expressed as security automata [ES00]. A single, global automaton maintains the state of a program, and this state is explicitly represented during execution. Static analysis ensures that the run-time representation of the state is correctly maintained, and that no actions are taken that would be illegal in the current state. For example, they encode the requirement that an applet not leak information contained on a user's computer by maintaining a state variable that records whether the applet has read from the hard drive. If so, no further network communication is allowed.

Our goal of certifying customizable security properties is the same as Walker's, but the notion of a global automaton that transitions on operations such as system calls is too coarse for our applications. We need a more expressive framework to handle issues such as type safety, which require knowing about each variable in the program. Instead of assembly code, Walker targets an intermediate language where control flow is expressed with continuation-passing style.

Decompilation

The beginning of this chapter compared our work to existing decompilation tools, and explained that we are willing to impose more requirements on the object code than other decompilers. When working with unannotated binary code, Balakrishnan and Reps's algorithm for analyzing memory accesses [BR04] is probably the most advanced in the literature. It uses a *value set analysis* to determine the possible values at each abstract location, and thus reconstruct a model of the heap.

Mycroft [Myc99] gives a unification-based algorithm for reconstructing recursive

C datatypes from object code, but without annotation it is difficult to recover array types faithfully. Other decompilers, such as work by Cifuentes *et al.* [CSF98] and Propan [KW02], focus primarily on control-flow and local variables. Because we can get type information for function pointers – and because we don't deal with deliberately-obfuscated object code – control-flow analysis is the easiest part of our verification.

Chapter 4

CCured

Most of our experiments with this analysis framework were done using CCured's type invariants as the safety policy being enforced. CCured [NCH+05] is a source-level tool that enforces type safety in legacy C code by adding additional metadata and runtime checks. It includes a whole-program static analysis to determine which operations can be proved safe statically and which need runtime checks. In this chapter, we present a type policy that allows the framework of Chapters 2 and 3 to verify that CCured's invariants are correctly maintained in the assembly code, and hence that the result is typesafe. We also discuss our implementation of the framework and CCured policy.

We begin by discussing the implementation, which analyzes x86 assembly code. This section includes discussion of various issues that were omitted from earlier chapters for simplicity, such as stack handling and function calls. We then discuss in detail CCured's type system, with its support for dependent array types and dynamic typing, and show how to encode it as a type policy. We conclude with experiments to evaluate the performance

and completeness of our analysis.

4.1 Implementing the analysis

4.1.1 Parsing assembly code

Our implementation is based on the assembly code parser from the Open Verifier project [CCNS05]. This tool converts both x86 and MIPS assembly into SAL, a simplified assembly language. SAL is very similar to the language presented in Figure 2.3 except that basic blocks can fall through to the following block; `jump` instructions can jump to computed addresses and not just constant labels; and SAL permits expressions of arbitrary complexity, unlike the `mov` operation of Figure 2.3 that allows only one arithmetic operation per instruction. SAL's limited set of opcodes makes it easy to do symbolic evaluation, since all side effects are made explicit. For example, the x86 `call` instruction pushes the next PC value onto the stack, to be used as a return pointer, and then jumps to its target. SAL compiles "`call` f" into three instructions and a label:

$$\begin{aligned} &\texttt{mov } \mathrm{r}_{esp},\ \mathrm{r}_{esp} \ominus 4 \\ &\texttt{store } L_{retaddr} \rightarrow [\mathrm{r}_{esp}] \\ &\texttt{jump } f \\ L_{retaddr}: \end{aligned}$$

In SAL, each bit of the x86 EFLAGS status register is treated as a separate one-bit register. So the x86 "compare" instruction also becomes multiple SAL instructions: "`mov` $\mathrm{r}_{ZF},\ (\mathrm{r}_1 = \mathrm{r}_2);$ `mov` $\mathrm{r}_{BF},\ (\mathrm{r}_1 < \mathrm{r}_2);\ \ldots$". All arithmetic operations and comparisons in SAL are done on 32-bit integers. 8- and 16- bit operations are encoded in SAL using the

appropriate bitmasking.

Our parser does not yet accept any floating point operations, since floating-point numbers are not interesting for the safety policies we have worked with.

Decompiling stack usage

The Open Verifier project also provides support for function calls and stack usage, as described in [Sch04]. The Stack and Function modules decompile their respective assembly idioms into code that is at a slightly higher level and is therefore easier to analyze.

Many of the values that are stored in stack memory are placed there due to register spilling: these are local variables and temporary values that would have been stored in physical registers had there been enough registers. This is especially common in x86 code, which has only 6 or 7 general purpose registers available, depending on whether %`ebp` is used as a frame pointer. Other stack locations are used to remember register values during a function call. We'd like to treat reads and writes to the stack differently from reads and writes to the heap: the stack is only accessed locally, so we can do a precise analysis of it without worrying about aliasing. (Section 4.1.3 describes how we handle stack-allocated objects whose addresses are taken.)

By tracking changes to the stack pointer, as well as the frame pointer and any other register that is used to access the stack, the Stack module can determine which `load` and `store` instructions are accessing the stack, and which offsets within the stack (a.k.a. *stack slots*) they are accessing. For each stack slot, the Stack module creates a *pseudoregister* to represent it, and replaces any `load`/`store` instructions referencing that stack slot with a `mov` instruction referencing the pseudoregister. Now our abstract interpreter can treat these

pseudoregisters the same as physical registers and achieve a precise handling of the stack with no additional analysis. Of course, this is only sound if every `load` and `store` referencing the stack is correctly decompiled — mixing `load`/`store` access with pseudoregister access would change the semantics of the program. Therefore, we rely on the type policy to ensure that any memory operation that is not handled by the Stack module refers to an address in the heap. Since the stack and heap are disjoint, this is sufficient for soundness.

[CHN06] elaborates on this strategy of decompiling assembly code into higher-level languages one step at a time.

Decompiling function calls

As with stack spilling, it is also useful to verify function calling conventions separately from the main type system. Since we do an intraprocedural analysis, we want to treat function calls as opaque instructions, subject only to any pre- and post-conditions annotated by the safety tool. The Function module decompiles SAL into a slightly higher-level language that includes a new instruction and a new jump:

$$I \ ::= \ \ldots \ | \ \mathrm{r}_{ret} \leftarrow \texttt{call}_{cc}\ e_f(\mathrm{r}_{arg1}, \ldots, \mathrm{r}_{argN});\ \texttt{clobber}(\mathrm{r}_{c1}, \ldots, \mathrm{r}_{cK})$$

$$J \ ::= \ \ldots \ | \ \texttt{return}\ \mathrm{r}_{ret}$$

The instruction "$\mathrm{r}_{ret} \leftarrow \texttt{call}_{cc}\ e_f(\mathrm{r}_{arg1}, \ldots, \mathrm{r}_{argN});\ \texttt{clobber}(\mathrm{r}_{c1}, \ldots, \mathrm{r}_{cK})$" means that the code is using calling convention CC to call the function pointed to by e_f with arguments $\mathrm{r}_{arg1}, \ldots, \mathrm{r}_{argN}$, and on return the values of caller-save registers $\mathrm{r}_{c1}, \ldots, \mathrm{r}_{cK}$ are undefined. During symbolic evaluation, the $\texttt{call}_{cc}$ instruction will require the type policy to show that e_f is a pointer to a valid function with N parameters that uses calling convention CC, and each argument can be coerced to the appropriate type for that parameter. Also,

$\Delta(\mathrm{r}_{ret})$ is assigned a fresh abstract value whose type is determined by the annotated return type of the function, and for all $i \leq K$, $\Delta(\mathrm{r}_{ci})$ is assigned a fresh abstract value of type Int(). The type policy may make other changes to the symbolic state if the target function has any special postconditions.

For example, consider the following SAL code:

$$
\begin{aligned}
& \texttt{mov } \mathrm{r}_{esp},\ \mathrm{r}_{esp} \ominus 12 \\
& \texttt{store } e_2 \rightarrow [\mathrm{r}_{esp} \oplus 8] \\
& \texttt{store } e_2 \rightarrow [\mathrm{r}_{esp} \oplus 4] \\
& \texttt{store } L_{retaddr} \rightarrow [\mathrm{r}_{esp}] \\
& \texttt{jump } L_f \\
L_{retaddr} :\ &
\end{aligned}
$$

The Stack module replaces the stack writes with writes to pseudoregisters, and the Function module replaces the `jump` with a `call`, resulting in

$$
\begin{aligned}
& \texttt{mov } \mathrm{r}_{esp},\ \mathrm{r}_{esp} \ominus 12 \\
& \texttt{mov } \mathrm{r}_{pseudo2},\ e_2 \\
& \texttt{mov } \mathrm{r}_{pseudo1},\ e_1 \\
& \mathrm{r}_{eax} \leftarrow \texttt{call}_{gcc\_std}\ L_f(\mathrm{r}_{pseudo1}, \mathrm{r}_{pseudo2});\ \texttt{clobber}(\mathrm{r}_{pseudo1}, \mathrm{r}_{pseudo2}, \mathrm{r}_{ecx}, \mathrm{r}_{edx}, \ldots) \\
& \texttt{mov } \mathrm{r}_{esp},\ \mathrm{r}_{esp} \oplus 4
\end{aligned}
$$

The new code is written at a higher level than the original, and can be understood more easily. This translation deals with several details of the x86 calling convention so that later analyses won't have to: arguments are pushed onto the stack in reverse order; r_{ecx} and r_{edx} are caller-save; the function being called can overwrite the parts of the stack where its parameters are stored; the return value is usually stored in r_{eax}; and after returning the stack pointer is 4 bytes higher than it was on function entry. The Function module determines whether a jump is a function call (yes, in this case), the number of arguments of the function (two), and the calling convention being used (*gcc_std*), by querying the type policy about the expression L_f.

When returning from a function, the Function module again ensures that the x86 calling convention is correctly used: callee-save registers contain the same value they did at the start of the function, the stack pointer has been incremented by 4, etc. The type policy, meanwhile, will check when it sees a `return` r_{ret} instruction that $\Delta(\mathrm{r}_{ret})$ can be coerced to the declared return type for the function.

We have extended the Functions module from the Open Verifier project with new features that are needed for C programs. We support functions with multi-word return types, where the return values are stored on the stack. And we support tail calls, where a function jumps to another function (or to itself) and reuses the same stack frame. This is decompiled into a `call` followed by a `return`, so the type policy needs no special handling of tail calls. The remainder of this section discusses other new features that we added to the Open Verifier project to support our CCured tests: dependently-typed function calls, stack-allocated objects, and polymorphic subroutines.

4.1.2 Dependently-typed function calls

Now that our target language includes function call instructions, we must extend the typing rules of Chapter 2 to deal with them. We handle dependent function arguments the same way we do dependent records: parameter types can refer to the values of other parameters, but not to any other values. We present here the typechecking rules for functions that return a single word-sized value with a nondependent type (although it is also possible to have multiple-word return values or return types that depend on the parameters). Our functions therefore have types of the form $\Sigma_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1}\rangle \rightarrow_{cc} \Sigma_s.\langle\ 0 : t_{ret}\ \rangle$, using the same σ notation that was used earlier for dependent memory types.

Initial function state. First we define the initial state of the abstract interpretation for a dependently-typed function. Suppose the function has type $\Sigma_s.\langle 0 : t_0; \ldots; n-1 : t_{n-1}\rangle \rightarrow_{cc} \Sigma_s.\langle 0 : t_{ret}\rangle$ and that the parameters are passed in registers r_{arg_0} through r_{arg_n-1} in calling convention CC. Then the initial state of the symbolic evaluation will be $\langle \Delta_0, \Phi_0, v_{mem0}, \Gamma_0\rangle$, where:

$$\begin{aligned}
\Delta_0 &= \{\mathrm{r}_{arg_0} = v_0;\ \ldots;\ \mathrm{r}_{arg_n-1} = v_{n-1}\} \\
\Phi_0 &= \emptyset \\
\Gamma_0 &= \{v_0 \mapsto \tau_0;\ \ldots;\ v_{n-1} \mapsto \tau_{n-1};\ v_{mem0} \mapsto \texttt{ValidMem}\}
\end{aligned}$$

and for all i, $\tau_i = t_i\left[{}^{v_0}\!/_{s.0}\right] \cdots \left[{}^{v_{n-1}}\!/_{s.(n-1)}\right]$.

Function call and return. Figure 4.1 shows the symbolic evaluation rules for function calls and returns. For simplicity we evaluate the `call` and `clobber` instructions separately, although they will always appear together. We extend field and register types with type $\sigma_{args} \rightarrow_{cc} \sigma_{ret}$ denoting a pointer to a function with calling convention CC, and give the appropriate type to the assembly labels of each function. As in Chapter 2, the judgment $\langle \Delta, \Phi, m, \Gamma\rangle \vdash I \rightsquigarrow \langle \Delta_1, \Phi_1, m_1, \Gamma_1\rangle$ means that in checker state $\langle \Delta, \Phi, m, \Gamma\rangle$, symbolically evaluating I yields the new checker state $\langle \Delta_1, \Phi_1, m_1, \Gamma_1\rangle$. Unlike earlier evaluation rules, however, these rules have preconditions that must be satisfied. Note how the (FUNCTION CALL) rule resembles the rule for memory writes, just as the initial function state above resembles the rule for reading dependent memory.

$$\text{field types} \quad t \quad ::= \quad \ldots \mid \sigma_{args} \rightarrow_{cc} \sigma_{ret}$$

$$\text{register types} \quad \tau \quad ::= \quad \ldots \mid \sigma_{args} \rightarrow_{cc} \sigma_{ret}$$

(FUNCTION CALL)

$\Gamma, \Phi \vdash e_f : \Sigma_s.\langle 0 : t_0;\ \ldots\ ;\ n-1 : t_{n-1} \rangle \rightarrow_{cc} \Sigma_s.\langle\ 0 : t_{ret}\ \rangle$

t_{ret} has no dependencies

$\forall 0 \leq i < n\ .\ \Gamma, \Phi \vdash \Delta(\mathrm{r}_{arg\_i}) : \tau_i$

where $\tau_i = t_i \left[{\Delta(\mathrm{r}_{arg\_0})}/{s.0} \right] \cdots \left[{\Delta(\mathrm{r}_{arg\_n-1})}/{s.(n-1)} \right]$

$\Gamma, \Phi \vdash m : \texttt{ValidMem}$

$v_{mem}{}', v_{ret}$ fresh in Γ

$$\langle \Delta, \Phi, m, \Gamma \rangle \vdash \quad \mathrm{r}_{ret} \leftarrow \texttt{call}_{cc}\ e_f(\mathrm{r}_{arg\_0}, \ldots, \mathrm{r}_{arg\_n-1})$$
$$\rightsquigarrow \langle\ \Delta[\mathrm{r}_{ret} \mapsto v_{ret}],\ \Phi,\ v_{mem}{}',\ \Gamma[v_{ret} \mapsto t_{ret}][v_{mem}{}' \mapsto \texttt{ValidMem}]\ \rangle$$

(CLOBBER REGISTERS)

$$v_1, \ldots, v_K \text{ fresh in } \Gamma$$

$$\langle \Delta, \Phi, m, \Gamma \rangle \vdash \quad \texttt{clobber}(\mathrm{r}_{c1}, \ldots, \mathrm{r}_{cK})$$
$$\rightsquigarrow \langle\ \Delta[\mathrm{r}_{c1} \mapsto v_1] \ldots [\mathrm{r}_{cK} \mapsto v_K],\ \Phi,\ m,\ \Gamma[v_1 \mapsto \mathrm{Int}()] \ldots [v_K \mapsto \mathrm{Int}()]\ \rangle$$

(RETURN)

$$\text{The current function has type } \ \sigma_{args} \rightarrow_{cc} \Sigma_s.\langle\ 0 : t_{ret}\ \rangle \qquad \Gamma, \Phi \vdash \Delta(\mathrm{r}_{ret}) : t_{ret}$$

$$\langle \Delta, \Phi, m, \Gamma \rangle \vdash\ \texttt{return}\ \mathrm{r}_{ret} \rightsquigarrow \textbf{Halt}$$

Figure 4.1: Typing/evaluation rules for function calls.

4.1.3 Stack-allocated objects

In C, it is possible to allocate an array or other object on the stack, then take the address of the object using C's & operator, and then treat the address the same way that heap pointers are treated. This is commonly used to implement call-by-reference semantics in C: for example, if a function `bar` takes a pointer to a Foo object as an argument, one could invoke it by declaring a (stack-allocated) local variable with type Foo, and passing the address of this variable to `bar`. Allocating an object on the stack is semantically equivalent to allocating the object on the heap and freeing it at the end of the current scope, but stack allocation is much faster than calling `malloc` and `free`. Unfortunately, stack allocation poses a challenge for program analysis because any read or write to memory could actually be accessing a stack location whose address was taken, so such locations cannot be treated as pseudoregisters by our Stack decompilation. The good news is that compilers must also treat such local variables conservatively (and perform fewer optimizations), so type policies can usually analyze these variables even without the convenience of coverting them to pseudoregisters.

To support stack allocation, we modify the Stack decompilation described earlier to treat stack-allocated objects as if they are part of the heap. CCured emits an annotation for each function describing which parts of the stack frame are used for objects whose address is taken, and what the types of those arguments are. The Stack module and the type policy both read this annotation so that they agree on which parts of the stack are managed by the type policy and which are only accessed by the current function and can be decompiled by the Stack module. For any particular stack state, we therefore have two

logical memory regions: the "heap" region consists of the heap, stack-allocated objects, and global variables; the "stack" region consists of all stack slots that were not annotated as stack-allocated objects by CCured. The Stack module does not rewrite any accesses to stack-allocated objects, so the type policy sees these as normal memory accesses.

In order for stack allocation of objects to be safe, we must ensure that pointers to the stack are not dereferenced after that stack frame is deallocated. Objects created by `malloc` use garbage collection for safe deallocation, but that's not an option on the stack. CCured addresses this issue by requiring that pointers to stack-allocated objects never be stored in memory (aside from pseudoregisters) or returned from a function.[1] Instead, these pointers can be used locally or passed as arguments to a function, as is commonly done in C to implement call-by-reference. CCured performs a whole-program analysis to determine which formal parameters might point to stack objects, and labels these pointers as part of its annotation of parameter types. The assembly analyzer can then check that pointers with this "stack-allocated" qualifier are not stored into memory.

4.1.4 Subroutines

CCured allows a function to be declared as "context-sensitive" if there is no way to infer a single type that works for all locations where the function is invoked. Examples include functions that wrap allocator calls, or the identity function from `void*` to `void*`. Context sensitive functions are analyzed separately for each call site, but CCured generally only emits one copy of the code for each function.

[1] Actually, CCured offers dynamic checks that can safely permit other uses of stack pointers, such as storing a pointer to a stack-allocated object inside another stack-allocated object which is in a lower frame. However, these checks do not work well in the presence of function inlining and are not needed very often in practice.

In assembly code, we analyze such functions separately at each call site, just as CCured does. When reaching a call to such a function, we continue the abstract interpretation within the body of the function as if the code had been inlined. However, if a context sensitive function is called twice within a function, we must take care to keep the two analyses separate rather than conflate the analysis state at join points, as we normally would for performance.

In terms of control-flow, context sensitive functions resemble the subroutines that are used to implement try/finally in Java bytecode [SA99]. In both cases, polymorphic function types are used as an alternative to code duplication. However, they are analyzed differently. Java subroutine types are polymorphic only over the types of local variables that they do not access; any accessed variable must have the same type in any call to the subroutine. Therefore, the subroutine only needs to be typechecked once. CCured's context-sensitive functions can manipulate polymorphic values in certain ways. Since these functions are usually short, it is easier to analyze them separately at each call site than to attempt to infer a polymorphic type signature for them.

4.2 The CCured type policy

CCured enforces type safety for legacy C code by classifying pointers according to their usage. Depending on a pointer's classification, or *kind*, CCured may change the pointer to a "fat" pointer structure that stores *metadata* such as array bounds and run-time type information. Figure 4.2 shows two fat pointer kinds that we support in our implementation: Sequence pointers ("SEQ"), which are used for arrays; and "RTTI" pointers,

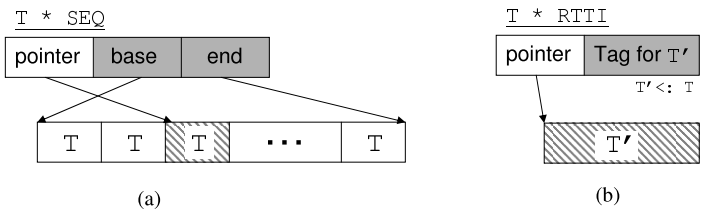

Figure 4.2: Two "fat" pointer kinds used by CCured: (a) a sequence pointer (array), and (b) a pointer with run-time type information. The current targets of the pointers are shown with stripes, and the metadata added by the CCured code transformation is in grey.

which hold Run-Time Type Information specifying the dynamic type of the object being pointed to. The metadata is used to support run-time checks that CCured inserts when the pointer is dereferenced (for SEQ) or cast (for RTTI pointers). These fat pointers are dependent types: the pointer field's type depends on the value of the metadata fields. When a program modifies a fat pointer, it may have to update all of the fields in the fat pointer using the (MEMORY UPDATE) rule of the type policy that allows memory to be temporarily inconsistent.

CCured has several other kinds beyond SEQ and RTTI. If a pointer needs neither form of metadata (because no future operation will perform arithmetic on it or downcast it), we give it kind "SAFE". SAFE pointers have the same representation they did in the original program (one word wide, with no metadata) and need only a NULL check before being dereferenced. In the published CCured experiments, 65–100% of the pointer declarations are classified as SAFE. The "SEQR" kind is a four-word fat pointer containing both bounds metadata and RTTI information. And the "FSEQ" kind is an optimization of Sequence pointers for the common case where pointers are incremented but not decremented: here, we need to carry an upper bound but not a lower bound. In this dissertation, we focus on

$$\text{SEQ pointer to } \sigma \quad = \quad \Sigma_s.\langle\ 0 : \text{Seq}_\sigma(s.1, s.2);\ 1 : \text{Int}();\ 2 : \text{Int}()\ \rangle$$

where

$$\begin{aligned}[\![\text{Seq}_\sigma(b,e)]\!]_{\rho_A} \triangleq \quad \{p \in \text{Word} \quad | \quad & (b \le e) \\ & \wedge\ (e-b) \bmod |\sigma| = 0 \\ & \wedge\ (p-b) \bmod |\sigma| = 0 \\ & \wedge\ \forall i.(b \le (p + i{\cdot}|\sigma|) < e) \implies \rho_A(p + i{\cdot}|\sigma|) = \sigma\}\end{aligned}$$

Figure 4.3: The meanings of the Seq type constructors used by CCured. The $<$ and $\le$ operators used here are unsigned comparisons.

the SEQ and RTTI kinds.

4.2.1 Sequence pointers

CCured uses Sequence pointers to support arrays and pointer arithmetic in C. A Sequence pointer is a three-word fat pointer, as shown in Figure 4.2(a), consisting of the actual pointer and pointers to the two ends of the array.

To encode Sequence pointers in a type policy for our framework, we define a type constructor $\text{Seq}_\sigma(base, end)$ for each base type σ used by the program, just as we defined the family of constructors Ptr_σ. The assembly-level encoding of the fat pointers pointers is shown in Figure 4.3. As in Section 2.6 we use the set comprehension notation $\{p|\ldots\}$ to show the meaning of the type constructor, and $\rho_A(e) = \sigma$ means that e points to a σ object in the current allocation state. The definition of Seq_σ directly follows the invariants that CCured maintains for its SEQ pointers: both the end pointer and the actual pointer are aligned on multiples of the element size with respect to the base pointer, although the

(SEQ DEREFERENCE)

$$\frac{\text{LessOrEq}(e_b,\ e_p) \in \Phi \qquad \text{LessThan}(e_p,\ e_e) \in \Phi}{\Gamma, \Phi \vdash \texttt{IsSubtype}(e_p,\ \text{Seq}_\sigma(e_b, e_e),\ \text{Ptr}_\sigma)}$$

(SEQ ARITHMETIC)

$$\frac{\Gamma, \Phi \vdash e_1 : \text{Seq}_\sigma(e_b, e_e) \qquad \Gamma, \Phi \vdash e_2 : \text{Int} \qquad \text{DividesDiff}(|\sigma|,\ (e_1 \oplus e_2),\ e_b) \in \Phi}{\Gamma, \Phi \vdash e_1 \oplus e_2 : \text{Seq}_\sigma(e_b, e_e)}$$

(SEQ ARITHMETIC SPECIAL CASE)

$$\frac{\Gamma, \Phi \vdash e_1 : \text{Seq}_\sigma(e_b, e_e) \qquad \Gamma, \Phi \vdash e_2' : \text{Int} \qquad |\sigma| \text{ is a power of two}}{\Gamma, \Phi \vdash e_1 \oplus (|\sigma| \otimes e_2') : \text{Seq}_\sigma(e_b, e_e)}$$

Figure 4.4: Dereference and arithmetic rules for Sequence pointers.

pointer p itself may sometimes be out of bounds. We can dereference a $\text{Seq}_\sigma(b, e)$ pointer p if it is within its bounds b and e. Moreover, we can apply pointer arithmetic to this value, so long as the quantity being added is a multiple of the element size. NULL pointers are represented as the triple $\langle 0, 0, 0 \rangle$.

Before dereferencing a sequence pointer, CCured will insert a check that the pointer is within bounds. When the abstract interpretation sees this conditional jump, the `Constrain` operation will add the facts $\text{LessOrEq}(e_b,\ e_p)$ and $\text{LessThan}(e_p,\ e_e)$ to Φ. Figure 4.4 shows the subsumption judgment (SEQ DEREFERENCE) that allows a Sequence pointer to be coerced to a Ptr, and hence be used in the (MEMORY READ) and (MEMORY UPDATE) rules.

Figure 4.4 also shows the rules for pointer arithmetic. The definition of $[\![\text{Seq}_\sigma(b, e)]\!]_{\rho_A}$ in Figure 4.3 implies that if p has type $\text{Seq}_\sigma(b, e)$, then we can add any multiple of $|\sigma|$ to p and the result will also have type $\text{Seq}_\sigma(b, e)$. In C pointer arithmetic is done with respect to

the element size, so "`p + x`" in the source code is compiled "$p \oplus (|\sigma| \otimes x)$" in the assembly version. While designing this type policy, however, we realized that $|\sigma| \otimes x$ will not always be a multiple of $|\sigma|$. (Recall that $\oplus$, $\otimes$, ... refer to machine arithmetic instructions that are subject to overflow.) Consider for example a pointer p with type $\text{Seq}_\sigma(b, e)$, where $|\sigma| = 12$. The expression "$p \oplus (|\sigma| \otimes x)$" might *not* have type $\text{Seq}_\sigma(b, e)$ because on a 32-bit machine with $x = 357,913,943$, $12 \otimes x$ equals 20 and $p \oplus 20$ would not be properly aligned.

The rules in Figure 4.4 support two methods of dealing with overflow. The more general rule, (SEQ ARITHMETIC), requires an explicit alignment check be performed, and uses a new fact $\text{DividesDiff} \in \mathbb{F}$ to encode this check. We define $\rho_A \models \text{DividesDiff}(k_1, k_2, k_3)$ to be true when $k_1, k_2, k_3 \in \text{Word}$, and k_1 divides $(k_2 - k_3)$. The requirement $\text{DividesDiff}(|\sigma|,\ e_1 \oplus e_2,\ b)$ ensures the alignment invariants of $[\![\text{Seq}_\sigma(b, e)]\!]_{\rho_A}$ are maintained even if there is overflow in the addition $e_1 \oplus e_2$ or in the multiplication within e_2. (In practice, e_2 will always have the from $|\sigma| \otimes e_2'$ because of C's pointer arithmetic.) The runtime check that detects overflow and generate the DividesDiff fact is relatively expensive, since it requires a division. However, the (SEQ ARITHMETIC SPECIAL CASE) rule removes the need for a runtime check in the common case where $|\sigma|$ is a power of two. In that case, if e_1 has type $\text{Seq}_\sigma(b, e)$ then $\text{DividesDiff}(|\sigma|,\ e_1 \oplus (|\sigma| \otimes e_2'),\ b)$ will always hold, as explained in the soundness discussion below.

Allocation of heap-based objects, including arrays, is handled by a trusted allocator such as `malloc`. Each call to `malloc` in a program is annotated by CCured with a representation of the object type σ being allocated and a flag saying whether this is an array allocation or a single object. For array allocations, we ensure that the size of the requested

allocation area is a multiple of σ. Shortly after each call to `malloc`, CCured will insert a conditional testing whether the result is NULL. If so, we give it type $\text{Seq}_\sigma(0,0)$. If not, the result has type $\text{Seq}_\sigma(v_p, v_p \oplus len)$ where v_p is the fresh abstract value created to hold the result, and len is the argument to `malloc`.

Soundness of Sequence pointers

To show that the rules in Figure 4.4 are sound, we must prove that Required Lemma 1 and Required Lemma 2 from Section 2.6 hold for all cases relating to the Seq_σ constructors. The `IsSubtype` judgment in (SEQ DEREFERENCE) is easy: this follows directly from the definition of $[\![\text{Seq}_\sigma(e_b, e_e)]\!]_{\rho_A}$ in Figure 4.3, using $i = 0$.

For (SEQ ARITHMETIC), we must show that if $\text{Vars}, \rho_A \models \Gamma, \Phi$, $\Gamma, \Phi \vdash e_1 \oplus e_2 : \text{Seq}_\sigma(e_b, e_e)$, and the premises of (SEQ ARITHMETIC) hold, then $[\![e_1 \oplus e_2]\!]_{\text{Vars}} \in [\![[\![\text{Seq}_\sigma(e_b, e_e)]\!]_{\text{Vars}}]\!]_{\rho_A}$. By assumption, $\Gamma, \Phi \vdash e_1 : \text{Seq}_\sigma(e_b, e_e)$ holds, so we know that $[\![e_b]\!]_{\text{Vars}}$ is less than or equal to $[\![e_e]\!]_{\text{Vars}}$ and $(([\![e_e]\!]_{\text{Vars}} - [\![e_b]\!]_{\text{Vars}}) \bmod |\sigma|)$ equals 0. $([\![e_1 \oplus e_2]\!]_{\text{Vars}} - [\![e_b]\!]_{\text{Vars}}) \bmod |\sigma|$ is also 0 because $\rho_A \models \text{DividesDiff}(|\sigma|, [\![e_1 \oplus e_2]\!]_{\text{Vars}}, [\![e_b]\!]_{\text{Vars}})$. Finally, because $|\sigma|$ divides both $([\![e_1]\!]_{\text{Vars}} - [\![e_b]\!]_{\text{Vars}})$ and $[\![e_1 \oplus e_2]\!]_{\text{Vars}} - [\![e_b]\!]_{\text{Vars}}$, it can be shown that there exists k such that $[\![e_1 \oplus e_2]\!]_{\text{Vars}} = [\![e_1]\!]_{\text{Vars}} + |\sigma| \cdot k$. For all i, if $e_b \leq ([\![e_1 \oplus e_2]\!]_{\text{Vars}} + i \cdot |\sigma| < e_e$, then $e_b \leq [\![e_1]\!]_{\text{Vars}} + (k+i) \cdot |\sigma| < e_e$ and $\rho_A([\![e_1 \oplus e_2]\!]_{\text{Vars}} + i|\sigma|) = \sigma$.

Finally, we consider the (SEQ ARITHMETIC SPECIAL CASE) rule. From the premises, we know that $|\sigma|$ is a power of two and $|\sigma|$ divides $([\![e_1]\!]_{\text{Vars}} - [\![e_b]\!]_{\text{Vars}})$. Let k be such that $[\![e_1]\!]_{\text{Vars}} - [\![e_b]\!]_{\text{Vars}} = |\sigma| \cdot k$. On a 32-bit machine,

$$[\![e_1 \oplus (|\sigma| \otimes e_2')]\!]_{\text{Vars}} - [\![e_b]\!]_{\text{Vars}} \equiv ([\![e_1]\!]_{\text{Vars}} - [\![e_b]\!]_{\text{Vars}} + |\sigma| \cdot [\![e_2']\!]_{\text{Vars}}) \quad \textbf{mod } 2^{32}$$

so there exists k' such that

$$\begin{aligned}\llbracket e_1 \oplus (|\sigma| \otimes e_2') \rrbracket_{\text{Vars}} - \llbracket e_b \rrbracket_{\text{Vars}} &= \llbracket e_1 \rrbracket_{\text{Vars}} - \llbracket e_b \rrbracket_{\text{Vars}} + |\sigma| \cdot \llbracket e_2' \rrbracket_{\text{Vars}} + 2^{32} \cdot k' \\ &= |\sigma| \cdot k + |\sigma| \cdot \llbracket e_2' \rrbracket_{\text{Vars}} + 2^{32} \cdot k'\end{aligned}$$

Therefore, $|\sigma|$ divides $\llbracket e_1 \oplus (|\sigma| \otimes e_2') \rrbracket_{\text{Vars}} - \llbracket e_b \rrbracket_{\text{Vars}}$ if $|\sigma|$ is a power of two less than 2^{32} (or, more generally, if $|\sigma|$ is a factor of the size of the ring in which arithmetic is performed). We ensure that $|\sigma|$ never exceeds the size of the address space, so we conclude that $|\sigma|$ divides $\llbracket e_1 \oplus (|\sigma| \otimes e_2') \rrbracket_{\text{Vars}} - \llbracket e_b \rrbracket_{\text{Vars}}$. The rest of this case is identical to the (SEQ ARITHMETIC) case.

4.2.2 Run-time type information

CCured also supports casts between pointers to different base types. If the base types do not contain pointers or other special types, the cast is always safe. For example, it is legal to coerce a pointer to an array of characters into a pointer to an array of integers provided that an alignment check is performed on the upper bound. For casts where the base type does involve pointers, additional steps are needed to ensure type safety.

Upcasts

Because C has no native support for subtyping, programmers are forced to give each object type its own `struct` definition, and cast between them using C's unsafe cast operator. Consider a program that uses the two objects shown in Figure 4.5. A Manager is a special type of Employee that includes a list of the employees that he or she supervises. The program can use Manager pointers in places where Employee pointers are expected,

```
struct Employee {
  int employeeNumber;
  struct Date * SAFE hireDate;
};
```

```
struct Manager {
  int employeeNumber;
  struct Date * SAFE hireDate;
  struct Employee * SAFE * SEQ reports;
};
```

Figure 4.5: An example of two C object types where `Manager` is a logical subtype of `Employee`. The pointer kinds SAFE and SEQ were inferred by CCured. A reference to an Employee is denoted "`struct Employee * SAFE`", so "`struct Employee * SAFE * SEQ`" is an array of pointers to employees. This toy example omits other useful fields, such as the number of reports that a Manager has.

such as in the `reports` field of a second-level manager. Therefore, it should be possible to cast a Manager pointer to an Employee pointer.

In our type policy, we can represent an Employee as

$$Employee \triangleq \Sigma_s.\langle\ 0 : \mathrm{Int}();\ 1 : \mathrm{Safe}_{Date}()\ \rangle$$

where $Date$ is the layout of `struct Date` and $\mathrm{Safe}_\sigma()$ is similar to Ptr_σ except that it allows NULL pointers.[2] Similarly, we could represent a Manager as

$$Manager \triangleq \Sigma_s.\langle\ 0 : \mathrm{Int}();\ 1 : \mathrm{Safe}_{Date}();\ 2 : \mathrm{Seq}_{EmployeeP}(s.3, s.4);\ 3 : \mathrm{Int}();\ 4 : \mathrm{Int}()\ \rangle$$

where $EmployeeP$ is a record containing only a $\mathrm{Safe}_{Employee}()$ pointer. (Note that CCured has added two fields to the struct to hold the metadata for the Sequence pointer.) However, this definition would not allow a cast from a $\mathrm{Ptr}_{Manager}$ to a $\mathrm{Ptr}_{Employee}$, since each part of the address space must have a unique type in the allocation state ρ_A. Instead, we can define

$$ManagerSuffix \triangleq \Sigma_s.\langle\ 0 : \mathrm{Seq}_{EmployeeP}(s.1, s.2);\ 1 : \mathrm{Int}();\ 2 : \mathrm{Int}()\ \rangle$$

[2] Therefore, $[\![\mathrm{Safe}_\sigma()]\!]_{\rho_A} = \{0\} \cup [\![\mathrm{Ptr}_\sigma]\!]_{\rho_A}$ for all σ and ρ_A.

and treat a Manager object as an *Employee* record followed by a $ManagerSuffix$ record.

To support this, we define new families of constructors $\mathrm{Ptr}_{\vec{\sigma}}$, $\mathrm{Safe}_{\vec{\sigma}}()$, and $\mathrm{Seq}_{\vec{\sigma}}(b, e)$ where the base type is not a single record but a list of records. The record lists $\vec{\sigma}$ are flattened versions of the object layouts κ from Figure 3.3. $\mathrm{Ptr}_{\vec{\sigma}}$ is recursively defined as:

$$[\![\mathrm{Ptr}_{\sigma_1::\vec{\sigma}}]\!]_{\rho_A} = \{\ a \in \mathrm{Word} \mid \rho_A(a) = \sigma_1 \ \wedge\ (a+|\sigma_1|) \in [\![\mathrm{Ptr}_{\vec{\sigma}}]\!]_{\rho_A}\ \}$$

$$[\![\mathrm{Ptr}_{\mathbf{nil}}]\!]_{\rho_A} = \mathrm{Word}$$

$\Gamma, \Phi \vdash \texttt{IsSubtype}(e, \mathrm{Ptr}_{\vec{\sigma}_1}, \mathrm{Ptr}_{\vec{\sigma}_2})$ and $\Gamma, \Phi \vdash \texttt{IsSubtype}(e, \mathrm{Safe}_{\vec{\sigma}_1}(), \mathrm{Safe}_{\vec{\sigma}_2}())$ hold for all Γ, Φ, and e iff $\vec{\sigma}_2$ is a prefix of $\vec{\sigma}_1$, so a value with type $\mathrm{Safe}_{Employee::ManagerSuffix::\mathbf{nil}}()$ can be cast to $\mathrm{Safe}_{Employee::\mathbf{nil}}()$. In this example, it is sufficient to break the Manager type into two records, but in general we split an object into as many records as possible to allow the greatest flexibility in subtyping. Note, however, that dependent fields such as those in $ManagerSuffix$ cannot be separated or the resulting types would be ill-formed: the type system cannot allow the metadata fields to be mutated independently of the pointer that depends on them. We use σ records for the smallest, indivisible units of the allocation state.

Downcasts

Occasionally, the example program might need to cast an Employee pointer to a Manager pointer rather than the other way around. This *downcast* from a supertype to a subtype is safe only if the *Employee* record is followed by a $ManagerSuffix$ record. To support downcasts in a safe manner CCured uses RTTI fat pointers, which combine a SAFE pointer with a one-word tag (Run-Time Type Information) representing the true base

type of the pointer. In this example, an Employee pointer would be packaged with a flag indicating whether the Employee object being pointed to is actually a Manager object. Before each downcast, we compare the tag to the desired type and abort the program if there is a mismatch. A common use of these checked downcasts is casts from `void*` — a.k.a. a pointer whose base type is **nil** — to a specific type.

During compilation, CCured will associate a unique tag with each type that is involved in a downcast, and use these tags consistently throughout the program. The tags are actually pointers into a static tree-shaped data structure where each node points to its immediate parent in the subtype hierarchy. To do a tag check, CCured calls the trusted function `CHECK_RTTICAST` in its runtime library that traverses the RTTI data structure. We write $\texttt{typeof}(t)$ for the record list corresponding to tag t, and `typeof`'s inverse $\texttt{tagof}(\vec{\sigma})$ for the tag corresponding to record list $\vec{\sigma}$. For example, suppose a program attempts to casts a "`void * RTTI`" to a "`struct Employee *`". If the actual base type is Manager, then the original pointer's tag is $\texttt{tagof}(\mathit{Employee} :: \mathit{ManagerSuffix} :: \textbf{nil})$. `CHECK_RTTICAST` will notice that while this tag is different from the desired tag, the tag's parent in the hierarchy ($\texttt{tagof}(\mathit{Employee} :: \textbf{nil})$) does match the desired tag, so the check succeeds.

Figure 4.6 shows the definition of the $\text{Rtti}_{\vec{\sigma}}$ type constructors. As seen in Figure 4.2, an RTTI fat pointer is a two-word record containing a pointer and a tag. When a pointer is stored in a register, we can add or drop the RTTI information at any time:

(RTTI CREATE)

$$\frac{}{\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{Safe}_{\vec{\sigma}}(),\ \text{Rtti}_{\vec{\sigma}}(\texttt{tagof}(\vec{\sigma})))}$$

$$\text{RTTI pointer to } \vec{\sigma} \quad = \quad \Sigma_s.\langle\ 0 : \mathit{Rtti}_{\vec{\sigma}}(s.1);\ 1 : \text{RttiTag}()\ \rangle$$

where

$$[\![\text{Rtti}_{\vec{\sigma}}]\!]_{\rho_A}(t) \triangleq \{p \mid \ p = 0 \ \vee \ ((p \in [\![\text{Ptr}_{\vec{\sigma}}]\!]_{\rho_A}) \wedge (p \in [\![\text{Ptr}_{\texttt{typeof}(t)}]\!]_{\rho_A}))\}$$

Figure 4.6: The Rtti type constructors used by CCured.

(RTTI DESTROY)

$$\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{Rtti}_{\vec{\sigma}}(t),\ \text{Safe}_{\vec{\sigma}}())$$

Since $\text{Rtti}_{\vec{\sigma}}(t)$ is at least as informative a type as $\text{Safe}_{\vec{\sigma}}())$, we eagerly convert all $\text{Safe}_{\vec{\sigma}}())$ pointers to $\text{Rtti}_{\vec{\sigma}}(\texttt{tagof}(\vec{\sigma}))$ before widening as part of the "adding extra facts" step of Section 3.1.2.

When we encounter a call to $\texttt{CHECK\_RTTICAST}(\mathit{tagof}(\vec{\sigma}_1),\ t)$ it means that CCured is casting a pointer with runtime tag t to a new layout $\vec{\sigma}$. The first argument to CHECK_RTTICAST is always a constant, since CCured always knows what type it is casting to. We trust that $\texttt{CHECK\_RTTICAST}(\mathit{tagof}(\vec{\sigma}_1),\ \mathit{tagof}(\vec{\sigma}_2))$ will abort the program unless $\vec{\sigma}_1$ is a prefix of $\vec{\sigma}_2$. Therefore if $\texttt{CHECK\_RTTICAST}(\mathit{tagof}(\vec{\sigma}_1),\ t)$ returns, any pointer with run-time tag t points to an object whose layout has prefix $\vec{\sigma}_1$. So for each subexpression occurring in Δ which has type $\text{Rtti}_{\vec{\sigma}'}(t)$, we replace it with a fresh variable with type $\text{Rtti}_{\vec{\sigma}_1}(t)$, unless $\vec{\sigma}'$ is already stronger than $\vec{\sigma}_1$.

4.2.3 Bugs found

One of the main advantages of accompanying a source-level safety tool with a separate assembly-level checker is that duplicate verification gives protection against mistakes in the first tool that break soundness. We discovered three bugs in CCured while attempting to verify the safety of assembly code. Each of them could be exploited to subvert type safety in the program.

- When initializing the buffer returned by `malloc` call, CCured forgot to check that the pointer was nonnull.
- CCured's optimizer incorrectly removed bounds checks on the faulty assumption that two pointers could not alias.
- CCured originally omitted the alignment checks for Sequence pointer arithmetic described in Section 4.2.1. Cyclone [JMG+02], the typesafe C variant that CCured is closest too, has this same vulnerability. Accessing a misaligned pointer breaks type safety, since the code might access the wrong field of a structure. Unlike the previous two bugs, which were discovered while running experiments, this bug was caught while designing the type policy. Reasoning about the low-level semantics of assembly code forces one to pay attention to issues such as overflow that may be easier to overlook at the source level. Balakrishnan *et al.* similarly observe that many bugs can appear in low-level code that are not easily spotted in the source code or detected by source-code tools [BRMT05].

Program	LOC	number of functions	functions verified	time (sec)	joins per BB
unoptimized `go`	29,321	372	327	113	2.09
optimized `go`	29,321	372	300	96	3.46

Table 4.1: Experience using our verifier with `go`.

4.3 Experiments

As an initial test, we used our prototype on the `go` program in the Spec95 benchmark suite [SPE95]. Of the Spec95 programs, we chose `go` because it makes extensive use of arrays while avoiding floating-point instructions, which our x86 parser does not yet handle. The tests were run using the latest version of CCured and the assembly code was generated using GCC 3.4.4, using optimization level -O3.

Of the 378 functions in the 29,321 LOC program, we can successfully verify 300 of them(79%), while the others report false positives. The most common reason for failure was array indexing expressions that don't meet the completeness criteria described in Section 3.2. For such functions, additional annotations may need to be added to the function despite our goal to do without. Verifying the program takes 96 seconds on a 2 GHz Xeon with 2 GB of RAM, and each basic block was explored an average of 3.45 times before a fixed point was reached, with basic blocks in nested loops accounting for most of the cases where more than 2 iterations were needed. When not using compiler optimizations, our verifier is slightly better (88% of the functions can be verified), but some array indices are still too complicated.

4.3.1 CCured features supported

Our implementation handles the CCured features discussed in this chapter, as well as tagged unions and special handling for the initialization of newly-allocated objects. Tagged unions are similar to RTTI pointers in that the runtime value of a "tag" field describes the type of the data in the union. Unlike RTTI pointers, however, union tags are controlled by the programmer, who provides annotations describing the relation between tag values and union fields. As with all of our dependent types, we rely on the fact that the tag and the union must be located in the same C struct in order to ensure that one is not modified without the other.

However, there are several other CCured features that we do not support. Variable-argument functions are forbidden, and our implementation does not allow casting between arrays of different base types when those base types contain pointers. To simplify our implementation, we told GCC not to inline the runtime functions that perform most array bounds and alignment checking functions.

We do not support CCured's *Compatible Metadata Representation*, which allows metadata to be stored separately from the pointers it describes so that heap-based data structures will have the same representations they would without CCured. By maintaining a one-to-one correspondence between the pointer and metadata, CCured can ensure that the metadata is kept in sync with any changes the pointer, provided that the pointer is not modified by any external code. To support this, we would have to modify our type system with a more general notion of which memory locations can depend on each other. Since this feature of CCured was used only rarely, supporting it was not a priority. However,

in recent work with dependently-typed source code [CHA+07], we have encountered other cases where it would be useful to relax the requirement that dependent values be located in the same object, and we are considering solutions to the problem.

CCured's WILD pointers, which have no (useful) static typing information, are also unsupported because they too are little-used in later CCured experiments. As features such as RTTI pointers were added to CCured, it became possible to deal with "bad casts" in ways that didn't cause the performance and compatibility problems that WILD pointers did. If desired, supporting WILD pointers in our implementation would be relatively straightforward if the supporting runtime functions were trusted and not inlined.

Chapter 5

Deputy

In this chapter we show how our system can be used with Deputy [CHA+07], a successor to CCured. Like CCured, Deputy guarantees type safety for C programs. Unlike CCured, however, Deputy does not use fat pointers or make any other changes to a program's data structures. Instead, it allows the programmer to annotate code at module boundaries to express dependencies. Figure 5.1 shows a simple dependent type annotation in the `usb_interface` structure of the Linux kernel. Deputy also incorporates a novel mechanism for supporting C's null-terminated strings. Because it does not change data structures, Deputy can be applied incrementally to a program, while CCured requires a whole-program transformation.

Deputy uses the same approach to mutation of dependent data that our framework did: metadata and the values that depend on it are stored in the same context, *i.e.* heap object, parameter list, or local variable scope. As in our framework, Deputy can permit mutations of values stored in the heap because it knows that the only values that could

```
struct usb_interface {
   /* array of alternate settings for this interface */
   struct usb_host_interface * COUNT(num_altsetting) altsetting;
   struct usb_host_interface * cur_altsetting;
   unsigned num_altsetting;          /* number of alternate settings */
   ...
}
```

Figure 5.1: Part of the definition of `struct usb_interface` from include/linux/usb.h in the Linux kernel. The Deputy "`COUNT(...)`" annotation is similar to the Array constructor from Chapter 2, and indicates here that `altsetting` is a pointer to an array of `num_altsetting` objects.

depend on the modified location are other fields of the same heap object.

Deputy also encounters the same difficulty that our framework does with nonatomic updates of dependent values, but uses a different solution. In CCured, fat pointer updates are considered atomic at the source level, but in Deputy programmers update one memory word at a time using ordinary C instructions. For example, consider lines 5 and 6 in Figure 5.2. After the assignment on line 5, the programmer might break the invariant that `ptr->num_altsetting` is the length of array `ptr->altsetting`. Deputy's solution to this problem is that any metadata is considered valid for a NULL pointer. Therefore, the assignment of NULL to `ptr->altsetting` on line 4 ensures that line 5 is legal, and the assignment on line 6 is legal because `ptr->num_altsetting` == `n`. At the assembly level, however, this solution may not be enough. A compiler that is optimizing single-threaded code (or that knows `*ptr` is local to the thread) could reorder the memory writes or drop the write on line 4. Therefore, memory may be temporarily inconsistent even using Deputy's relaxed notion of consistency, and we must continue to use the approach to temporary inconsistencies described in Section 2.4.

We have not implemented an assembly-code checker for Deputy, but we show here

```
1 void changeSettings(struct usb_interface * ptr, int n) {
2   struct usb_host_interface * COUNT(n) newSetting = malloc(...);
3   // ...  initialize the members of newSetting ...
4   ptr->altsetting = NULL;
5   ptr->num_altsetting = n;
6   ptr->altsetting = newSetting;
7 }
```

Figure 5.2: Sample Deputy code that mutates the `altsetting` and `num_altsetting` fields from Figure 5.1.

how it can be done by presenting a type policy. We give the constructor definitions and type rules for Deputy's pointers, and we outline the soundness proofs. The first half of this chapter discusses regular pointers; the second half discusses null-terminated arrays.

5.1 Bounded pointers

Because Deputy does not use fat pointers, it does not make CCured's distinction between SAFE and SEQ pointers. Instead, it has a single form of bounded pointer, where the declaration "`T * BND(`b`,`e`) x;`" means that either `x` is NULL, or $b \leq$ `x` $\leq e$ and the memory range between b and e contains consecutive `T` objects. The pointer must be aligned on a object boundary, but unlike in CCured b and e need not be (Figure 5.3). As seen in Figure 5.1, Deputy uses "`T * COUNT(n) x;`" as syntactic sugar for "`T * BND(x, x+n) x;`", where the '`+`' is C-style pointer arithmetic. Pointers with no annotation are assumed to point to a single object (like CCured's SAFE pointers), and they have the implicit annotation "`COUNT(1)`".

In our framework, we can define a family of type constructors $\text{Bnd}_\sigma(b, e)$ for a bounded pointer to σ objects. As seen in Figure 5.4, Bnd_σ is similar to Seq_σ except that a

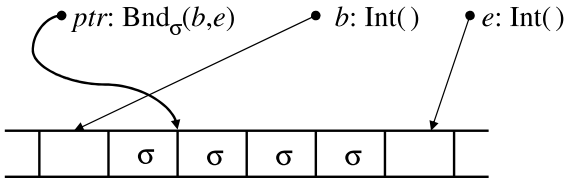

Figure 5.3: A bounded pointer in Deputy. The pointer itself must be aligned on an object boundary, but the bounds b and e need not be.

$$
\begin{aligned}
&[\![\mathrm{Bnd}_\sigma(b,e)]\!]_{\rho_A} \triangleq \\
&\quad \{0\} \cup \{p \in \mathrm{Word} \mid (b \le p \le e) \\
&\qquad\qquad \wedge (e-b) < (|\mathrm{Word}|/2 - 1) \\
&\qquad\qquad \wedge \forall i.(b \le (p + i \cdot |\sigma|) < e)) \Longrightarrow \rho_A(p + i \cdot |\sigma|) = \sigma\}
\end{aligned}
$$

Figure 5.4: The meanings of the Bnd_σ type constructors used by Deputy.

pointer must be either in-bounds or equal to its upper bound (CCured's pointers need only be in bounds when they are dereferenced), and if a pointer is NULL then its bounds need not denote a valid memory range. This second difference is needed to support updates in Deputy, as discussed earlier. The first difference was a design decision that makes it easier to guard against overflow, since Deputy will be able to incorporate the alignment check into the bounds check. As a result, Deputy won't need the bounds b, e to be aligned. The condition $(e - b) < (|\mathrm{Word}|/2 - 1)$ requires that no single object or array takes up half of the address space; we explain later why this is necessary.

Figure 5.5 shows the `IsSubtype` and arithmetic type-checking rules for Bnd types. The (BND DEREFERENCE) rule, which lets us use a Bnd pointer in memory operations, requires that a pointer be strictly less than its upper bound by using a new fact constructor

(BND DEREFERENCE)

$$\frac{\text{NotEq}(e,\ e_e) \in \Phi \qquad \text{NotEq}(e, 0) \in \Phi}{\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{Bnd}_\sigma(e_b, e_e),\ \text{Ptr}_\sigma)}$$

(BND NULL COERCE)

$$\frac{}{\Gamma, \Phi \vdash \texttt{IsSubtype}(0,\ \tau,\ \text{Bnd}_\sigma(e_{b2}, e_{e2}))}$$

(BND COERCE)

$$\frac{\begin{array}{cc}\text{LessOrEq}(e_{b1},\ e_{b2}) \in \Phi & \text{LessOrEq}(e_{b2},\ e) \in \Phi \\ \text{LessOrEq}(e,\ e_{e2}) \in \Phi & \text{LessOrEq}(e_{e2},\ e_{e1}) \in \Phi\end{array}}{\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{Bnd}_\sigma(e_{b1}, e_{e1}),\ \text{Bnd}_\sigma(e_{b2}, e_{e2}))}$$

(BND ARITHMETIC)

$$\frac{\begin{array}{c}\Gamma, \Phi \vdash e_1 : \text{Bnd}_\sigma(e_b, e_e) \qquad \Gamma, \Phi \vdash e_2 : \text{Int}() \qquad \text{NotEq}(e_1, 0) \\ \text{SignedLessOrEq}(-1 \otimes e_2,\ (e_1 \ominus e_b) \oslash |\sigma|) \in \Phi \\ \text{SignedLessOrEq}(e_2,\ (e_e \ominus e_1) \oslash |\sigma|) \in \Phi\end{array}}{\Gamma, \Phi \vdash e_1 \oplus (|\sigma| \otimes e_2) : \text{Bnd}_\sigma(e_b, e_e)}$$

(POINTERS ARE NON-NULL AFTER ARITHMETIC)

$$\frac{\Gamma, \Phi \vdash e_1 \oplus e_2 : \text{Bnd}_\sigma(e_b, e_e)}{\rho_A \models \text{NotEq}([\![e_1 \oplus e_2]\!]_{\text{Vars}}, 0)}$$

Figure 5.5: Dereference and arithmetic rules for Deputy's bounded pointers.

NotEq. (BND NULL COERCE) says that NULL can be used as a pointer of any type. The (BND COERCE) rule allows casting from a type $\text{Bnd}_\sigma(e_{b1}, e_{e1})$ to a type $\text{Bnd}_\sigma(e_{b2}, e_{e2})$ only if

1. The base type σ is the same in the two types.

2. The new type has a smaller range than the original type: $e_{b1} \leq e_{b2} \wedge e_{e2} \leq e_{e1}$.

3. The pointer e is within the new bounds: $e_{b2} \leq e \leq e_{e2}$.

To reduce the computational cost of the (BND COERCE) rule, Deputy does not check that the new bounds be aligned with respect to the pointer or the old bounds. Such an alignment requirement would correspond to adding the constraints $(e - b) \bmod |\sigma| = 0$ and $(p - b) \bmod |\sigma| = 0$ to the definition of $\rho_A(\text{Bnd}_\sigma(b, e))$. This would imply, among other things, that if $b \neq e$ then b is a valid pointer to a σ object. However Deputy does not actually need the bounds to be valid pointers, as they exist only for arithmetic comparisons. Therefore we don't need any alignment checks in (BND COERCE).

The final rules support pointer arithmetic in Deputy. Because the new pointer must be within bounds, we must ensure $e_b \leq (e_1 \oplus (|\sigma| \otimes e_2)) \leq e_e$. To guard against overflow in the multiplication step, we rewrite this as $((e_b \ominus e_1) \oslash |\sigma|) \leq e_2 \leq ((e_e \ominus e_1) \oslash |\sigma|)$. The version shown in the (BND ARITHMETIC) rule, which is the check used by the current Deputy implementation, takes into account the fact that the bounds are not necessarily aligned, so $(e_b \ominus e_1)/|\sigma|$ might not be a whole number. The "$\oslash$" operator is machine integer division, where values are rounded towards zero. For the actual inequality comparisons, we introduce a new fact constructor $\text{SignedLessOrEq}(e_1, e_2)$ corresponding to a signed integer comparison, assuming two's complement representation. Note that this

rule would be unsound if $((e_1 - e_b)/|\sigma|)$ or $((e_e - e_1)/|\sigma|)$ were too big to fit in a signed integer. This is why we require when buffers are allocated that they contain fewer than $2^{31} - 1$ bytes.

5.1.1 Implementing Deputy's checks

Deputy relies on an optimization phase to determine which required facts can be verified statically and which require runtime checks. Assembly code verification will need to have static analysis that is at least as powerful as Deputy uses in order to verify that the code is safe. Some of these optimizations are easy – the (BND COERCE) rule entails many requirements of the form LessOrEq(e, e), which are clearly true – while others are more sophisticated. Recently, Deputy has started using Miné's implementation of the octagon domain [Min01] to eliminate array checks, so any assembly code verification would also need to incorporate an analysis at least as strong as the octagon domain, or use additional annotations from the source tool.

Deputy also uses optimizations that are specific to its type rules. The last rule in Figure 5.5 shows that if $e_1 \oplus e_2$ typechecks, then the sum is nonnull and we can safely add NotEq$(e_1 \oplus e_2, 0)$ to Φ. This is true because Deputy prevents $e_1 \oplus e_2$ from overflowing to 0.

Finally, we note that if we implemented a Deputy policy in our framework we would relax the syntactic restrictions for dependent types on the heap to allow arithmetic. Chapter 2 says that the arguments to type constructors may only be field names and constants:

$$d \quad ::= \quad c \mid s.i$$

But we can extend that to

$$d \quad ::= \quad c \mid s.i \mid d_1 \; op \; d_2$$

so that, for example,

```
struct {
   T * COUNT(2*x) ptr;
   int x;
};
```

can be expressed as $\Sigma_s.\langle\ 0 : \text{Bnd}_\text{T}(s.0,\ s.0 \ \oplus \ (2 \otimes s.1));\ 1 : \text{Int}()\ \rangle$. This is simply a convenience (we could achieve a similar effect by defining new constructors for each `COUNT` annotation) that does not affect the soundness of the rules in Chapter 2.

5.1.2 Sketch of soundness proof

Lemma 7 (Soundness of the Bnd **constructors)** *Required Lemma 1 and Required Lemma 2 from Section 2.6 hold for all cases relating to the* Bnd_σ *constructors.*

The correctness of the (BND DEREFERENCE) rule follows directly from the definition of $[\![\text{Bnd}_\sigma(b,e)]\!]_{\rho_A}$ with $i = 0$. Similarly, the correctness of (BND NULL COERCE) and (BND COERCE) follows directly from the definition of $[\![\text{Bnd}_\sigma(b,e)]\!]_{\rho_A}$. The interesting case is (BND ARITHMETIC), for which we must show that if $\text{Vars}, \rho_A \models \Gamma, \Phi;\ \Gamma, \Phi \vdash e_1 \oplus (|\sigma| \otimes e_2) : \text{Bnd}_\sigma(e_b, e_e)$; and the premises of (BND ARITHMETIC) hold, then $[\![e_1 \oplus (|\sigma| \otimes e_2)]\!]_{\text{Vars}} \in [\![[\![\text{Bnd}_\sigma(e_b, e_e)]\!]_{\text{Vars}}]\!]_{\rho_A}$. Specifically, we will need to show that the premises of this rule are enough to prevent overflow.

Let MAX_INT be the largest value in Word when interpreting integers as signed numbers using two's complement notation (*i.e.* $|\text{Word}|/2 - 1$). In our allocation rule (not shown), we ensure that the size of every object and array is less than MAX_INT words. By

assumption, $\Gamma, \Phi \vdash e_1 : \text{Bnd}_\sigma(e_b, e_e)$ holds, so we know that $[\![e_b]\!]_{\text{Vars}} \leq [\![e_1]\!]_{\text{Vars}} \leq [\![e_e]\!]_{\text{Vars}}$ and $([\![e_e]\!]_{\text{Vars}} - [\![e_b]\!]_{\text{Vars}}) < \text{MAX\_INT}$. Therefore $[\![e_1]\!]_{\text{Vars}} - [\![e_b]\!]_{\text{Vars}}$ and $[\![e_e]\!]_{\text{Vars}} - [\![e_1]\!]_{\text{Vars}}$ are also nonnegative and less than MAX_INT, so the SignedLessOrEq comparisons imply $\left\lceil \frac{e_b \ominus e_1}{|\sigma|} \right\rceil \leq e_2 \leq \left\lfloor \frac{e_e \ominus e_1}{|\sigma|} \right\rfloor$ and we are assured that $|\sigma| \otimes e_2$ does not overflow. Therefore, $[\![e_b]\!]_{\text{Vars}} \leq [\![e_1 \oplus (|\sigma| \otimes e_2)]\!]_{\text{Vars}} \leq [\![e_e]\!]_{\text{Vars}}$. Finally, we prove that the last requirement of Bnd_σ holds where $p = [\![e_1 \oplus (|\sigma| \otimes e_2)]\!]_{\text{Vars}}$:

$$\forall i.([\![e_b]\!]_{\text{Vars}} \leq (p + i \cdot |\sigma|) < [\![e_e]\!]_{\text{Vars}})) \Longrightarrow \rho_A(p + i \cdot |\sigma|) = \sigma$$

Because there is no overflow in $e_1 \oplus (|\sigma| \otimes e_2)$, we get

$$\begin{aligned} p + i \cdot |\sigma| &= [\![e_1 + |\sigma| \otimes e_2]\!]_{\text{Vars}} + i \cdot |\sigma| \\ &= [\![e_1]\!]_{\text{Vars}} + |\sigma| \cdot [\![e_2]\!]_{\text{Vars}} + i \cdot |\sigma| \\ &= [\![e_1]\!]_{\text{Vars}} + |\sigma| \cdot ([\![e_2]\!]_{\text{Vars}} + i) \end{aligned}$$

If $[\![e_b]\!]_{\text{Vars}} \leq (p + i \cdot |\sigma|) < [\![e_e]\!]_{\text{Vars}}$, then $[\![e_b]\!]_{\text{Vars}} \leq ([\![e_1]\!]_{\text{Vars}} + ([\![e_2]\!]_{\text{Vars}} + i) \cdot |\sigma|) < [\![e_e]\!]_{\text{Vars}}$, and $\rho_A(p + i \cdot |\sigma|) = \sigma$ because $[\![e_1]\!]_{\text{Vars}} \in [\![[\![\text{Bnd}_\sigma(e_b, e_e)]\!]_{\text{Vars}}]\!]_{\rho_A}$.

□

5.2 Null-terminated arrays

One novel feature of Deputy is its typesafe method of supporting null-terminated arrays. Because C does not provide a way to determine at runtime the length of an array, programmers commonly denote the extent of an array by filling the last element of an array with zero (null). The most common example is the C convention that strings are represented

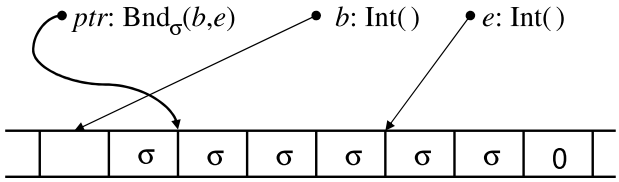

Figure 5.6: A null-terminated pointer in Deputy. The pointer and upper bound are aligned on object boundaries, and there is a terminating null somewhere after the upper bound.

as null-terminated arrays of characters, but this programming pattern is also used on arrays of pointers, structures, etc. Deputy includes a qualifier "`NT`" that indicates that a pointer is pointing to a null-terminated array. In this section, we explain the semantics of null-terminated pointers and how they are checked in assembly code.

When working with a null-terminated array, we must ensure that the program does not access any element beyond the terminating null, and that it does not overwrite the null with a non-zero value. However, there are situations where programs can safely overwrite a null: for example, reusing a buffer of known size with strings of varying lengths. Therefore, Deputy lets programs combine the NT and BND annotations to indicate that an array has both a known bound and a terminating null value. A variable declaration "`char * NT BND(b,e) ptr;`" means that `ptr` points to an array that includes the range [`b`,`e`] and that there is a terminating null value at address `e` or higher, as in Figure 5.6. Values in the range [`b`,`e`) can be safely overridden even if they are null, since the terminating null is somewhere after `e`. Most C strings have the annotated type "`char * NT COUNT(0)`" indicating that any null should be treated as a terminating null, but a few pointers will have known bounds that allow more flexibility in how the array is used.

For example, the entrypoint to a C program has the signature "`int main(int`

$$
\begin{aligned}
&[\![\mathrm{NTBnd}_\sigma(b,e)]\!]_{\rho_A} \triangleq \\
&\quad \{0\} \cup \{p \in \mathrm{Word} \mid (b \le p \le e) \\
&\qquad\qquad \wedge (e-p) \bmod |\sigma| = 0 \\
&\qquad\qquad \wedge \exists j \ge 0.(\ \rho_A(e + j\cdot|\sigma|) = \vec{0}_{|\sigma|} \\
&\qquad\qquad\qquad\qquad \wedge\ \forall i.(b \le (p + i\cdot|\sigma|) \le e + j\cdot|\sigma|) \Longrightarrow \rho_A(p + i\cdot|\sigma|) = \sigma\ )\}
\end{aligned}
$$

Figure 5.7: The invariant for the NTBnd_σ type constructors.

`argc, char* * argv)`" where `argv` points to an array of arguments to the program. The parameter `argc` holds the number of arguments in the array, and the last element of the array (*i.e.* `argv[argc]`) is always a NULL pointer. Therefore, programmers have two ways to safely traverse this array:

```
int main(int argc, char* * argv) {
  int i = 1;
  while (i < argc) {
    //process argument argv[i]
    i++;
  }
  ...
```

```
int main(int argc, char* * argv) {
  char** p = argv+1;
  while(*p != NULL) {
    //process argument *p
    p++;
  }
  ...
```

Since this array both has a known length and is null-terminated, we can give it the Deputy annotation "`NT COUNT(argc)`", or equivalently "`NT BND(argv, argv+argc)`". With these two pieces of information, Deputy is able to optimize away its checks no matter which method the program uses. Since the elements of the array are themselves null-terminated strings, `main` functions have this signature in Deputy:

```
int main(int argc, char* NT COUNT(0) * NT COUNT(argc) argv)
```

(NTBND DEREFERENCE)

$$\frac{\text{LessThan}(e,\ e_e) \in \Phi \qquad \text{NotEq}(e, 0)}{\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{NTBnd}_\sigma(e_b, e_e),\ \text{Ptr}_\sigma)}$$

(NTBND ENDPTR DEREFERENCE)

$$\frac{\text{NotEq}(e, 0)}{\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{NTBnd}_\sigma(e_b, e),\ \text{ReadPtr}_\sigma)}$$

(DROP NT)

$$\frac{}{\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{NTBnd}_\sigma(e_b, e_e),\ \text{Bnd}_\sigma(e_b, e_e))}$$

(NTBND SHRINK)

$$\frac{\begin{array}{c}\text{LessOrEq}(e_{b1},\ e_{b2}) \in \Phi \qquad \text{LessOrEq}(e_{b2},\ e) \in \Phi \\ \text{LessOrEq}(e,\ e_{e2}) \in \Phi \qquad \text{LessOrEq}(e_{e2},\ e_{e1}) \in \Phi \qquad (e_{e1} \ominus e_{e2}) \bmod |\sigma| = 0\end{array}}{\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{NTBnd}_\sigma(e_{b1}, e_{e1}),\ \text{NTBnd}_\sigma(e_{b2}, e_{e2}))}$$

(NTBND EXPAND)

$$\frac{\begin{array}{c}\text{LessOrEq}(e_{b1},\ e_{b2}) \in \Phi \qquad \text{LessOrEq}(e_{b2},\ e) \in \Phi \\ \text{LessOrEq}(e,\ e_{e2}) \in \Phi \qquad \text{NoNull}_{|\sigma|}(e_{e1},\ e_{e2}) \in \Phi\end{array}}{\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{NTBnd}_\sigma(e_{b1}, e_{e1}),\ \text{NTBnd}_\sigma(e_{b2}, e_{e2}))}$$

(NTBND NULL COERCE)

$$\frac{}{\Gamma, \Phi \vdash \texttt{IsSubtype}(0,\ \tau,\ \text{NTBnd}_\sigma(e_{b2}, e_{e2}))}$$

(DEFINITION OF NoNull_k)

$$\frac{a \le b \qquad (b - a) \bmod k = 0 \qquad \forall i \ge 0.((a + i{\cdot}k) < b) \implies \rho_A(a + i{\cdot}k) \neq \vec{0}_k}{\rho_A \models \text{NoNull}_k(a, b)}$$

Figure 5.8: The type rules for the NTBnd_σ type constructors. The arithmetic rule is the same as for the Bnd constructors in Figure 5.5.

5.2.1 Type rules for null-terminated arrays

To encode a pointer to a null-terminated array of σ objects, we use the NTBnd_σ constructor defined in Figure 5.7. NTBnd_σ is a subtype of Bnd_σ; the difference is that NTBnd_σ imposes more constraints on the upper bound and what lies beyond it. Let $\vec{0}_k$ be the record type denoting a memory area k words long containing only zeros. A null-terminated array is therefore a sequence of σ objects followed by a $\vec{0}_{|\sigma|}$ object, as seen in the definition of $[\![\text{NTBnd}_\sigma(b, e)]\!]_{\rho_A}$.

The key type rule here is (NTBND EXPAND), which lets us increase the upper bound of a null-terminated array. If at runtime we determine that the value pointed to by the upper bound is not null, we can safely increase the upper bound by one element, or $|\sigma|$ words. This knowledge that there is no null termination in a certain range is encoded in our framework with the fact constructor NoNull_k, where k is the size of array elements. As seen in Figure 5.8, $\text{NoNull}_k(e_1, e_2)$ is true if $e1 \leq e_2$, the addresses e_1 and e_2 are aligned with respect to k, and none of the objects in the range $[e_1,\ e_2)$ are null. We can increase the upper bound of a pointer from e_{e1} to e_{e2} if $\text{NoNull}_{|\sigma|}(e_{e1},\ e_{e2})$ is true. Typically, programs will read the upper bound e_e of an NT pointer and branch on whether its first field is null. If so, it is assumed to be a terminating null and the loop halts. If not, then the current object can't have allocation type $\vec{0}$, so we add the fact $\text{NoNull}_{|\sigma|}(e_e,\ e_e \oplus |\sigma|)$ to Φ. However, Deputy also uses a special runtime function that can scan a larger region for zero and test whether $\text{NoNull}_{|\sigma|}(e_{e1},\ e_{e2})$ holds even if e_{e1} and e_{e2} differ by more than $|\sigma|$.

Unlike the bounds of a Bnd pointer, the the upper bound of an NTBnd pointer must be aligned on an object boundary. If the upper bound were misaligned, then the

Deputy runtime function that tests NoNull facts would read misaligned values, since it starts scanning memory at the upper bound. Therefore, (NTBND SHRINK) has an extra alignment check $(e_{e1} \ominus e_{e2}) \bmod |\sigma| = 0$ that is not present in the corresponding (BND COERCE) rule for Bnd pointers. The initial release of the Deputy compiler omits the alignment checks; just as with the missing alignment requirement in CCured, this bug was discovered while attempting to write soundness proofs for the type policies in this dissertation.

Also unlike with Bnd pointers, it is legal to read the memory referred to by the upper bound of an NTBnd pointer, as seen in the (NTBND ENDPTR DEREFERENCE) rule. If the object read contains only zero, it is assumed to be the terminating null. If it contains some nonzero field, however, the program knows that this location is not the terminating null, and will be able to increase the upper bound by one using the (NTBND EXPAND) rule. Therefore the (NTBND ENDPTR DEREFERENCE) rule allows read but not write access to the upper bound, since it would not be safe to overwrite a terminating null with a nonnull value. (We implement this rule by defining a read-only pointer type ReadPtr_σ in the next section.) In order to write to the element at the end of the array, the program must first verify that it is not the terminating null, then expand the bounds and use the (NTBND EXPAND) rule to get a writable pointer. Finally, Deputy requires for NTBnd_σ pointers that an object containing only zeros be a valid σ object, so that the terminating null is also a σ object and the (NTBND ENDPTR DEREFERENCE) rule is safe.

For example, consider the loop shown in Figure 5.9. This is the canonical method of accessing a null-terminated array: a loop that is executed as long as the terminating null has not been reached. The inferred loop invariant is that p has type $\text{NTBnd}_{\text{Int}()}(p, p)$ and

```
1 // Precondition:  p is a nonnull pointer
2 // with type int* NT COUNT(0)
3 while(*p != 0) {
4   ...
5   p++;
6 }
7
```

L_1 : load $r_{tmp} \leftarrow [r_p]$
bnz r_{tmp}, L_2, L_3
L_2 : ...
mov r_p, $r_p + 1$
jump L_1
L_3 :

Figure 5.9: A loop that walks over a null-terminated array an array. The Deputy code for this loop is on the left; the assembly version on the right.

$\text{NotEq}(p, 0) \in \Phi$ where p is the value in r_p. Line 2 reads from p, using the rule that p's upper bound (which is p) can be read through a read-only pointer. The branch on line 3 lets us add the fact $\text{NoNull}_1(p, p+1)$ to the state at L_2[1]. The arithmetic on line 5 is legal because we can coerce the type of p to $\text{NTBnd}_{\text{Int}()}(p, p+1)$ with the (NTBND EXPAND) rule. Finally, during the join at L_1 r_p will be given a fresh abstract value p' with type $\text{NTBnd}_{\text{Int}()}(p, p')$, which is a fixed point.

5.2.2 Type rules for read-only pointers

In order to define a read-only pointer — needed for (NTBND ENDPTR DEREFERENCE) — we introduce a notion of subtyping of object types. When we read at address e using this rule, we may be reading either a σ object or a $\vec{0}_{|\sigma|}$ object. For simplicity, Deputy requires that for any base type σ used in a null-terminated array, an object containing only zeros is a valid σ object. So we'll define a new type ReadPtr_σ to mean an address whose

[1]The branch also adds the fact $\text{NotEq}(\texttt{sel}(m,\ p),\ 0)$, but this fact is not used

allocation state is either σ or some subtype of σ:

$$[\![\text{ReadPtr}_\sigma]\!]_{\rho_A} = \{\ p \mid \rho_A(p) = \sigma' \text{ where for all tuples } (i_1, \ldots, i_{|\sigma|}),$$
$$\text{if } (i_1, \ldots, i_{|\sigma|}) \text{ is a valid } \sigma' \text{ object, then it is a valid } \sigma \text{ object. } \}$$

We extend our type framework to allow ReadPtr_σ pointers to be used in the (MEMORY READ) rule in place of Ptr_σ. It is therefore safe to give the upper bound of an NTBnd_σ array the type ReadPtr_σ because we will either read a σ or a $\vec{0}_{|\sigma|}$.

5.2.3 Sketch of soundness proof for null-terminated pointers

Lemma 8 (Soundness of the NTBnd **constructors)** *Required Lemma 1 and Required Lemma 2 from Section 2.6 hold for all cases relating to the* NTBnd_σ *constructors.*

First, note that NTBnd_σ is a subtype of Bnd_σ as stated in rule (DROP NT) because for any $j \geq 0$, $\forall i.(b \leq (p + i \cdot |\sigma|) < e + j \cdot |\sigma|)$ implies $\forall i.(b \leq (p + i \cdot |\sigma|) < e)$. Therefore the (NTBND DEREFERENCE) rule is correct since `IsSubtype` is transitive. The NTBnd arithmetic rule(not shown) and the (NTBND SHRINK) and (NTBND NULL COERCE) rules are similar to the arithmetic and coercion rules for Bnd pointers, and are correct for analogous reasons.

The interesting case here is (NTBND EXPAND). We assume $\text{Vars}, \rho_A \models \Gamma, \Phi$, $e \in [\![\text{NTBnd}_\sigma([\![e_{b1}]\!]_{\text{Vars}}, [\![e_{e1}]\!]_{\text{Vars}})]\!]_{\rho_A}$, and the premises of (NTBND EXPAND) hold, and show $e \in [\![\text{NTBnd}_\sigma([\![e_{b2}]\!]_{\text{Vars}}, [\![e_{e2}]\!]_{\text{Vars}})]\!]_{\rho_A}$. If $e = 0$, this is trivial. Otherwise, note that because $\text{NoNull}_{|\sigma|}(e_{e1},\ e_{e2}) \in \Phi$, we know that there exists $j' \geq 0$ such that $e_{e2} = e_{e1} + j' \cdot |\sigma|$ and $\forall 0 \leq i < j'.\ \rho_A(e_{e1} + i \cdot |\sigma|) \neq \vec{0}_{|\sigma|}$. By the assumption $e \in [\![\text{NTBnd}_\sigma([\![e_{b1}]\!]_{\text{Vars}}, [\![e_{e1}]\!]_{\text{Vars}})]\!]_{\rho_A}$,

we know there exists $j \geq 0$ such that

$$\rho_A(e+j) = \vec{0}_{|\sigma|} \ \wedge \ \forall i.(e_{b1} \leq (e + i{\cdot}|\sigma|) < e_{e1} + j{\cdot}|\sigma|) \Longrightarrow \rho_A(e + i{\cdot}|\sigma|) = \sigma$$

Therefore, $j \geq j'$. If we substitute $e_{e2} - j'{\cdot}|\sigma|$ for e_{e1} in the formula above, we get

$$\rho_A(e+j) = \vec{0}_{|\sigma|} \ \wedge \ \forall i.(e_{b1} \leq (e + i{\cdot}|\sigma|) < e_{e2} + (j - j'){\cdot}|\sigma|) \Longrightarrow \rho_A(e + i{\cdot}|\sigma|) = \sigma$$

$(j - j')$ satisfies the existential as it is nonnegative. Finally, because $[\![e_{b1}]\!]_{\text{Vars}} \leq [\![e_{b2}]\!]_{\text{Vars}} \leq [\![e]\!]_{\text{Vars}} \leq [\![e_{e2}]\!]_{\text{Vars}}$, we have satisfied all of the requirements of
$e \in [\![\text{NTBnd}_\sigma([\![e_{b2}]\!]_{\text{Vars}}, [\![e_{e2}]\!]_{\text{Vars}})]\!]_{\rho_A}$.

□

Chapter 6

Cqual

To test our framework on a policy other than type safety, we used Cqual, a whole-program static analysis tool for C [FFA99]. Cqual infers *type qualifiers* that can represent various properties. Given an initial set of annotations, Cqual will propagate the qualifiers throughout the program and report erroneous usage of qualified types. For example, Cqual has been used with the following qualifiers:

- **Format-string vulnerabilities.** For security reasons, strings that are under the control of an adversary should not be used as format strings for `printf` and related functions [New00]. Therefore, strings that are read from files, program arguments, network connections and the like are annotated as `$tainted`. Cqual will infer to which other program locations tainted data could flow; it is an error for tainted values to be used as format strings [STFW01].

- **User-space/kernel-space pointers.** In protected-mode operating systems, pointers under the control of user-level programs must not be dereferenced inside the kernel,

except by special routines that perform the correct security checks. Cqual has been used to find errors in the Linux kernel by annotating inputs from user-level programs (such is the argument to `ioctl` and any pointer read from user-space memory) as `$user`; it is an error to dereference a pointer that is inferred as `$user` [JW04].

We implemented a type policy that encodes Cqual's `$tainted`/`$untainted` qualifiers in assembly code. These qualifiers can be used to prevent format-string vulnerabilities and similar security problems like SQL injection attacks, and they are representative of other qualifier lattices in Cqual. As with CCured, we use Cqual on the source code to do the interprocedural analysis, and then have Cqual emit annotated function signatures so that we need only an intraprocedural analysis on assembly code. It would be possible to do a whole-program analysis of assembly code, but doing the interprocedural analysis on source code is more efficient and is consistent with our objective merely to verify that assembly code has been compiled from a program that was checked by a source code tool. Cqual does not use dependent types, but we can still make use of our inference system.

Like nearly all static analyses for C code, Cqual relies on the type safety of the program and is unsound if the program violates type safety in any way, including out-of-bounds array accesses and unusual casting.[1] Our type policy makes the same assumptions that Cqual does: pointer arithmetic is assumed to always be within bounds, and casts, unions, etc. are not used to break type safety. This lets us replicate Cqual's analysis at the assembly level, but the type policy is not sound.

[1] Cqual's algorithm would be sound if you also use CCured or a similar tool to guarantee type safety.

6.1 The `$tainted` type qualifier

Cqual's type qualifiers are different from the types used by our framework and by most other type systems. Instead of representing a set of values, qualifiers denote a higher-level notion of where the value came from or how it will be used. For example, a variable with type "`T $tainted`" may hold the same set of possible values as a variable with type "`T $untainted`", but the former variable holds a value that was obtained from an untrustworthy source, so that value should not be used in any critical piece of program logic. Similarly, a pointer with Cqual's `$const` qualifier represents the same set of possible heap addresses as a pointer with the `$nonconst` qualifier, but the program promises not to use the former pointer to perform memory writes.

Cqual does not guarantee that `$tainted` values will have no effect on `$untainted` values. Although all *data dependencies* are handled, *control dependencies* – such as branching on a tainted condition and using the result to modify an `$untainted` value – are not. Ignoring control flow allows Cqual to be used with programs for which full-fledged secure information flow [SM03] systems would be too restrictive. However, this makes it harder to formally describe the guarantees that Cqual provides.

It's safe to use `$untainted` values where `$tainted` are expected, so `$untainted` types are subtypes of the corresponding `$tainted` types. Therefore, we use the existing type constructors Int and Ptr_σ to represent the tainted versions of these types, and introduce new constructors UInt and UPtr_σ to represent the untainted versions. As seen in Figure 6.1 we define the untainted types to be subtypes of tainted types but not vice versa, even though $[\![\text{Int}()]\!]_{\rho_A} = [\![\text{UInt}()]\!]_{\rho_A}$ and $[\![\text{Ptr}_\sigma]\!]_{\rho_A} = [\![\text{UPtr}_\sigma()]\!]_{\rho_A}$. We also allow untainted pointers to

(INT SUBTYPING)

$$\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{UInt}(),\ \text{Int}())$$

(PTR SUBTYPING)

$$\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{UPtr}_\sigma(),\ \text{Ptr}_\sigma)$$

(CAST TO INT)

$$\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{UPtr}_\sigma(),\ \text{UInt}())$$

(UNSOUND ARITHMETIC)

$$\frac{\Gamma, \Phi \vdash e_1 : \text{UPtr}_\sigma() \qquad \Gamma, \Phi \vdash e_2 : \text{Int}()}{\Gamma, \Phi \vdash e_1 \oplus |\sigma| \otimes e_2 : \text{UPtr}_\sigma()}$$

(UNSOUND TAINTED ARITHMETIC)

$$\frac{\Gamma, \Phi \vdash e_1 : \text{Ptr}_\sigma \qquad \Gamma, \Phi \vdash e_2 : \text{Int}()}{\Gamma, \Phi \vdash e_1 \oplus |\sigma| \otimes e_2 : \text{Ptr}_\sigma}$$

(UNSOUND CAST)

$$\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{UInt}(),\ \text{UPtr}_\sigma())$$

(UNSOUND TAINTED CAST)

$$\Gamma, \Phi \vdash \texttt{IsSubtype}(e,\ \text{Int}(),\ \text{Ptr}_\sigma)$$

Figure 6.1: Subtyping and arithmetic rules for `$tainted` pointers.

be cast to untainted integers, mirroring our rule from Section 2.1 that Ptr_σ is a subtype of Int. In their work on semantic type qualifiers, Chin, Markstrum and Millstein [CMM05] use the same approach for the semantics of tainted types: tainted types correspond to the same set of values as untainted types, and only the subtyping rule distinguishes them.

Figure 6.1 also shows four rules that are not type safe but are necessary to replicate Cqual. The (UNSOUND ARITHMETIC) and (UNSOUND TAINTED ARITHMETIC) rules say that pointer arithmetic is assumed to be legal, and the result of the arithmetic as the same taint

status as the original pointer. The (UNSOUND CAST) and (UNSOUND TAINTED CAST) rules say that we can cast any value to a pointer of our choosing. (By composing these rules with the rules that state $\text{UPtr}_{\sigma'}$ and $\text{Ptr}_{\sigma'}$ are subtypes of UInt() and Int(), programs can cast pointers from one base type to another.) These cast rules also preserve the taint status of the original value. Programs that use these rules are not statically type safe, and a malicious program could easily disguise a `$tainted` value as an `$untainted` value using them. However, these rules are usually sufficient for bug detection. If used in combination with a type-safe policy, we could drop the four unsound rules and instead introduce untainted and tainted versions of the constructors in the type-safe policy.

In Cqual, programmers can explicitly cast `$tainted` values to `$untainted` to indicate that the value has been validated and is safe to use. We require Cqual to annotate in the assembly code where these casts occur so that we can mark the values as untainted in our analysis. Source code casts that are not related to type qualifiers can present problems as well, because the assembly code may suddenly start using a value in a different way than its type suggests. (In CCured, such casts are represented in the assembly code as checks of RTTI metadata.) Here, our implementation does its best to continue using the old typing information, and preserves the taint qualifier as much as possible.

6.2 Leaf polymorphism

Cqual provides a mechanism for parametric polymorphism in the signatures of external (library) functions, and our implementation supports this polymorphism. Function prototypes can be annotated with qualifier variables that are implicitly quantified over the

current prototype. These annotations are useful for library functions that can be used on either `$tainted` or `$untainted` data. For example, the identity function would be annotated as:

q_1 `int identity(`q_1 `int arg)`

indicating that the return type has the same taint status as the argument. The user can also state the subtype relationship between quantifier variables. In this declaration,

q_2 `char * strcpy(`q_2 `char* dest,` q_1 `char * src) where` $q_1 <: q_2$

q_1 is declared as a subtype of q_2 to indicate that information flows from `src` to `dest` but not vice versa. There are three valid ways to instantiate this declaration:

`dest` and `src` both point to tainted data.
`dest` and `src` both point to untainted data.
`dest` points to tainted data, and `src` to untainted.

When typechecking the call to a polymorphic function, our implementation does two things:

- Ensures that there exists an instantiation of qualifier variables to {`$tainted`, `$untainted`} such that the "where" constraints hold and each actual argument to the function is a subtype of the corresponding formal argument. It is an error if no such instantiation is possible.

- Constructs a type for the return value of the function, by substituting `$tainted`/ `$untainted` for quantifier variables according to the instantiation. If any quantifier variable in the return type is unconstrained, it is treated as `$untainted`.

6.3 Experiments

We implemented the type policy for tainted/untainted data described in this chapter. Oink [WCM05], which contains an implementation of Cqual, generated the qualifiers that we used to annotate function boundaries for the sake of intraprocedural analysis. We ran the tool on bftpd, a 6160-line FTP server. An earlier version of bftpd had a format string bug detectable by Cqual [STFW01].

Our verifier correctly checks the tainted-ness of strings, but it has difficulty with casts, as mentioned earlier. Several instructions in bftpd cast from `void*` to a specific type, but there is no way for the assembly code analyzer to know which type is being cast to, and therefore no way of knowing if the base type is tainted. Of the 119 functions in bftpd, 102 were verified correctly and 17 could not be verified. Four unverifiable functions take a variable-length argument list, which we do not support, and the others had tricky casts. As with CCured, adding additional annotations (for casts, this time) would fix these false positives. However, such annotations would be unsound because Cqual cannot handle certain bad casts safely.

Bibliography

[BR04] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction (CC)*, April 2004.

[BRMT05] Gogul Balakrishnan, Thomas Reps, Dave Melski, and Tim Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *Proceedings of the IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, October 2005.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 269–282, 1979.

[CCH+03] Christopher Colby, Karl Crary, Robert Harper, Peter Lee, and Frank Pfenning.

Automated techniques for provably safe mobile code. *Theor. Comput. Sci.*, 290(2):1175–1199, 2003.

[CCNS05] Bor-Yuh Evan Chang, Adam Chlipala, George Necula, and Robert Schneck. Type-based verification of assembly language for compiler debugging. In *The 2nd ACM SIGPLAN Workshop on Types in Language Design and Implementation*, January 2005.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[CHA+07] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. Dependent types for low-level programming. In *European Symposium on Programming*, March 2007.

[CHN06] Bor-Yuh Evan Chang, Matthew Harren, and George C. Necula. Analysis of low-level code using cooperating decompilers. In *Thirteenth International Static Analysis Symposium (SAS'06)*, LNCS, August 2006.

[CK77] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, 1977.

[CL05] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *Sixth International Conference on*

Verification, Model Checking, and Abstract Interpretation (VMCAI'05), LNCS, January 2005.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107. ACM Press, 2000.

[CMM05] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 05)*, pages 85–95, New York, NY, USA, 2005. ACM Press.

[Coq06] Coq Development Team. The Coq proof assistant reference manual, version 8.0. http://coq.inria.fr/doc/main.html, 2006.

[CSF98] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *Proceedings of the International Conference on Software Maintenance (ICSM 98)*, page 228, Washington, DC, USA, 1998. IEEE Computer Society.

[CV02] Karl Crary and Joseph C. Vanderwaart. An expressive, scalable type theory for certified code. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 191–205. ACM Press, 2002.

[DNS03] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover

for program checking. Technical Report HPL-2003-148, HP Laboratories, July 2003.

[ES00] Úlfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: a retrospective. In *Proceedings of the 1999 workshop on New security paradigms*, pages 87–95. ACM Press, 2000.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation*, pages 192–203, New York, NY, USA, 1999. ACM Press.

[GN04] Sumit Gulwani and George C. Necula. Path-sensitive analysis for linear arithmetic and uninterpreted functions. In *11th Static Analysis Symposium*, volume 3148 of *LNCS*, pages 328–343. Springer-Verlag, August 2004.

[Gro02] Dan Grossman. Existential types for imperative languages. In *Proceedings of the 11th European Symposium on Programming Languages and Systems*, pages 21–35. Springer-Verlag, 2002.

[Gro03] Dan Grossman. Type-safe multithreading in cyclone. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 13–25. ACM Press, 2003.

[GS01] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles*

of Programming Languages (POPL), pages 248–260, London, United Kingdom, January 2001.

[GTN04] Sumit Gulwani, Ashish Tiwari, and George C. Necula. Join algorithms for the theory of uninterpreted functions. In *24th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *LNCS*. Springer-Verlag, December 2004.

[Hic96] Jason Hickey. Formal objects in type theory using very dependent types. In *Proceedings of the 3rd International Workshop on Foundations of Object-Oriented Languages*, 1996.

[HJ91] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 125–138, Berkeley, CA, USA, January 1991. Usenix Association.

[JMG+02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.

[JW04] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[KW02] Daniel Kästner and Stephan Wilhelm. Generic control flow reconstruction from assembly code. In *Proceedings of the joint conference on Languages, compilers*

and tools for embedded systems (LCTES/SCOPES 02), pages 46–55, New York, NY, USA, 2002. ACM Press.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Proceedings of the Second International Workshop on Types in Compilation*, pages 28–52. Springer-Verlag, 1998.

[Min01] Antoine Miné. The Octagon abstract domain. In *Analysis, Slicing and Transformation (part of the Working Conference on Reverse Engineering)*, pages 310–319. IEEE CS Press, October 2001. http://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[Myc99] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems (ESOP 99)*, pages 208–223, London, UK, 1999. Springer-Verlag LNCS.

[NCH+05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems. To Appear*, 2005.

[Nec97] George C. Necula. Proof-carrying code. In *The 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM, January 1997.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 83–94. ACM Press, 2000.

[New00] Tim Newsham. Format string attacks, September 2000. http://www.thenewsh.com/~newsham/format-string-attacks.pdf.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 333–344, June 1998.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 conference on Programming Language Design and Implementation (PLDI)*, June 2007.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166. Springer-Verlag, 1998.

[RKS99] Oliver Rüthing, Jens Knoop, and Bernhard Steffen. Detecting equalities of variables: Combining efficiency with precision. In *Proceedings of the 6th International Symposium on Static Analysis*, volume 1694 of *LNCS*, pages 232–247. Springer-Verlag, 1999.

[SA99] Raymie Stata and Martin Abadi. A type system for java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.

[Sch04] Robert R. Schneck. *Extensible Untrusted Code Verification.* PhD thesis, University of California, Berkeley, May 2004.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[SPE95] SPEC95. Standard Performance Evaluation Corporation Benchmarks, July 1995.

[SSTP02] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 217–232. ACM Press, 2002.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

[Wal00] David Walker. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 254–267. ACM Press, 2000.

[WCM05] Daniel S. Wilkerson, Karl Chen, and Scott McPeak. Oink: a collaboration of C++ static analysis tools. http://www.cubewano.org/oink, 2005.

[XH01] Hongwei Xi and Robert Harper. Dependently Typed Assembly Language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 169–180, Florence, Italy, September 2001.

[Xi00] Hongwei Xi. Imperative programming with dependent types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santa Barbara, June 2000.