Dependent Types for Safe Systems Software

Jeremy Paul Condit

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-117
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-117.html

September 19, 2007

Acknowledgement

I am indebted to George Necula, my advisor, as well as Eric Brewer, Matt Harren, Zach Anderson, Feng Zhou, Rob Ennals, David Gay, Ilya Bagrak, Bill McCloskey, and Xavier Rival for their many contributions to the work described in this dissertation. This work was supported by the National Science Foundation under Grant Nos. CCR-0326577, CCF-0524784, and CNS-0509544, as well as gifts from Intel Corporation.

Dependent Types for Safe Systems Software

by

Jeremy Paul Condit

A.B. (Harvard University) 2000
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 Eric Brewer
Professor Jack Silver

Fall 2007

The dissertation of Jeremy Paul Condit is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2007

Dependent Types for Safe Systems Software

Jeremy Paul Condit

Abstract

Dependent Types for Safe Systems Software

by

Jeremy Paul Condit

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor George C. Necula, Chair

The C language is the dominant language for writing systems software, but it permits large classes of programming errors that could be prevented with a more modern language. Unfortunately, it is impractical to simply rewrite all of our existing software; rather, we must focus on a transition path to a more reliable systems programming language. As the first step in an incremental transition to a new language, we must allow programmers to annotate higher-level idioms typically used by C programmers, such as bounded pointers and tagged unions.

This thesis presents Deputy, a dependent type framework that allows programmers to annotate existing C code with more refined types. In particular, dependent types allow programmers to relate multiple state elements, such as a pointer and its bounds, or a union and its tag. In doing so, we address several technical hurdles involved with dependent types. First, we address the issue of mutation in the presence of dependencies by using insights

from axiomatic semantics. Second, we handle the problem of undecidability by using run-time checks where necessary. Third, we address usability challenges with a novel inference mechanism called automatic dependencies.

Once this framework is established, we show how it can be instantiated to support the dependent types required by modern systems code. In addition, we discuss several real-world applications of Deputy, including small and medium-size benchmarks as well as the Linux kernel itself. These benchmarks allow us to evaluate Deputy's annotation burden and performance in the context of real-world code.

Professor George C. Necula
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgments

I am deeply grateful to all of the people who have helped me complete this dissertation, either directly or indirectly. First and foremost is my advisor, George Necula, whose astute technical insights, gracious career advice, and boundless patience have been invaluable during my six years in graduate school. This dissertation would not have been possible without his guidance.

Thanks also to Eric Brewer, who has provided both guidance and inspiration in the realm of operating systems. I am grateful for the long-lasting collaboration between Eric, George, and the rest of our research group, which has provided the foundation for most of my research projects.

I am deeply indebted to my fellow students and researchers from the Ivy research group for their generous contributions to the work described in this dissertation and for their permission to use our shared results and coauthored work in this dissertation. First, many thanks to Matt Harren, who has made invaluable contributions to both the theory and implementation of Deputy. Thanks also to Zach Anderson, whose work on Deputy's optimizer was essential to Deputy's success, and to Feng Zhou, whose work on SafeDrive and Linux helped put Deputy through its paces. Thanks to Rob Ennals and David Gay, who worked tirelessly to apply Deputy to various software packages and who generously tolerated my use of their computers, and thanks to Ilya Bagrak, Bill McCloskey, and Xavier Rival for using and providing feedback on Deputy. Finally, many thanks to Intel Research Berkeley for their participation in and support of this project.

Portions of this dissertation, including Chapter 2 and Section 4.2, were previously

published in *Lecture Notes in Computer Science* as "Dependent Types for Low-Level Programming" [CHA+07], which is copyrighted by Springer. This material is reproduced (and updated) in this thesis with kind permission of Springer Science and Business Media.

I am grateful to have had a wonderful group of fellow grad students who have kept me sane during the last six years. I have shared many enjoyable conversations, meals, and games with Simon Goldsmith, Matt Harren, Wes Weimer, Tachio Terauchi, Evan Chang, Sumit Gulwani, Scott McPeak, George Porter, Holly Sorkin, and Steve Sorkin. Thanks also to the students and faculty of the Open Source Quality project for many enjoyable lunches and retreats.

Finally, thanks to my parents and family for believing in me and for always being there. Last but not least, thanks to Kristin for her love and support, and for making it all worthwhile.

Chapter 1

Introduction

The C programming language is the dominant programming language used for writing low-level systems software, as it has been for the past two to three decades. Most of today's widely-used operating systems, database systems, and internet servers are written in C. In the time since the C language was first developed, these systems have become both extremely complex and inextricably integrated into our daily lives.

Unfortunately, the C programming language makes a number of trade-offs between safety and expressiveness that are no longer suitable for the needs of modern systems. For example, C requires programmers to reason about the safety of array bounds and type casts without the help of the compiler. Although this approach is more permissive and more flexible than other languages available for these tasks, it also permits entire classes of bugs that would not be possible in modern, higher-level languages. These bugs, including the notorious buffer overflow, are responsible for a significant fraction of the failures exhibited by modern systems software; indeed, of the twenty most severe vulnerabilities listed by

CERT in July 2007, twelve are a result of buffer overflows [CER].[1]

In an ideal world, we could resolve this problem by rewriting our systems software in a more modern language; however, given the size of our existing systems and the degree to which we depend on them (bugs and all), it would be impractical to simply throw them out and replace them. Rather, we must focus on an *incremental* evolution of our systems software. If we can provide a smooth transition from our existing source code to an improved version in a higher-level language, we can achieve our safety and reliability goals while preserving our current investment in these systems.

Fortunately, it turns out that existing systems software is already *mostly* correct. Among other things, programmers have devised a number of common idioms that allow them to deal with C's inadequacies in structured and checkable ways. For example, C programmers will typically store a pointer to an array along with an integer indicating the length of that array, and these two values will be used and updated in tandem. Another common idiom is the use of a null terminator to indicate the end of an array of data. Unfortunately, since these idioms are not evident to standard C compilers, there is currently no way to verify that the programmer has checked the appropriate safety properties, even if they have done so in a rigorous and mechanical way. If we could *annotate* these idioms, and thereby make them visible to the compiler, we could provide the safety of a higher-level language along with the expressiveness of C—and without abandoning our investment in existing systems.

This thesis presents Deputy, a new type system and compiler for existing C code

[1] Vulnerabilities in this list, which range in date from July 1998 to March 2007, are scored by factors such as awareness of the vulnerability, ease of exploitation, and risk to internet infrastructure.

that serves as a first step in this evolution of our systems software. Deputy allows programmers to add simple annotations to existing code in order to describe common idioms used by C programmers. Deputy then uses a combination of compile-time and run-time checks to verify that these idioms are used correctly. This approach enables an incremental evolution of existing code toward a more reliable and mechanically-checked code base.

In the remainder of this chapter, we will discuss the properties that Deputy enforces (type and memory safety), the challenges in enforcing these properties in systems software (compatibility and incrementality), the main technique used to enforce these properties (dependent types), and an overview of our evaluation (small benchmarks plus the Linux kernel).

1.1 Enforcing Type and Memory Safety

The first step in a transition to a more reliable code base, and the primary goal of this thesis, is to enforce type and memory safety in existing C programs. Informally, *type safety* means that the run-time values of all memory locations correspond to their compile-time type, and *memory safety* means that all memory accesses are within the bounds of an allocated memory region. We concern ourselves only with *spatial memory safety*, or the lack of bounds violations, as opposed to *temporal memory safety*, or the lack of dangling pointers due to incorrect deallocation.

Enforcing these safety properties is an important step toward improving C programs for three reasons. First, enforcing these properties eliminates many common security vulnerabilities, such as buffer overflows. Second, these properties can be used to improve

isolation between components of a large system; for example, a type-safe and memory-safe device driver is less likely to corrupt data in the host kernel. Third, and perhaps most importantly, type safety and memory safety are basic assumptions made by almost any further program analysis; in other words, enforcing type and memory safety lays the foundation for all further improvements to these programs. For example, CCount [ABC+07] assumes type and memory safety (as enforced by Deputy) in order to verify memory deallocation.

1.2 Compatibility and Incrementality

Somewhat surprisingly, the central issue in designing a sound and scalable tool for enforcing type and memory safety in C code is the challenge of *compatibility* and *incrementality*. In short, an effective tool for enforcing these properties must maintain a compatible binary representation, and it must allow the user to apply the tool to their programs incrementally. In this section, we explore how these issues arise in the context of type and memory safety.

To make this discussion concrete, consider the following example derived from the Linux source code (`include/linux/socket.h`). This example shows portions of a structure used to describe network data packets:

```
struct msghdr {
    struct iovec * msg_iov;  /* Data blocks      */
    size_t msg_iovlen;       /* Number of blocks */
};
```

As indicated by the comments, `msg_iov` points to an array of data blocks of type `struct iovec`. In order to access these data blocks, we must index into this array; however,

the C language will allow us to index beyond the end of the array if we are not careful. So, given this structure and code that uses it, we would like to verify that accesses to the `msg_iov` pointer are within the bounds of some correctly-allocated memory region.

One approach to this challenge is to use static analysis, and indeed, many research projects have attempted to verify memory accesses in C programs using static checkers [CCF+05, DRS03, Che02]. Unfortunately, pure static analysis is typically insufficient for this task, so the resulting tools are either unsound or do not scale to large programs. Thus, tools that aim for practicality *and* soundness typically opt for some form of run-time checking.

The major challenge in performing run-time checking is storing the *metadata*—that is, extra information such as pointer bounds and run-time type information that is required to verify safety. One approach is to place all metadata in a *global table*, using a data structure such as a splay tree [JK97, RL04]. Accesses to memory locations are verified by looking up information about the corresponding memory region in this global table. Unfortunately, such lookups tend to be extremely slow, with overheads that are a factor of 5 to 10; as a result, these checks can only be used in debugging situations, which makes them far less effective. Recent work has reduced this overhead to a level that is more acceptable for production use, but only by relinquishing control over the allocator [DA06].

A final alternative is to use *fat pointers*. In this approach, the program's data structures are modified in order to store bounds and type information right next to the relevant data. For example, the above structure might become:

```
struct fat_msghdr {
    struct iovec * msg_iov;       /* Data blocks      */
    struct iovec * msg_iov_base;  /* Start of blocks  */
    struct iovec * msg_iov_end;   /* End of blocks    */
    size_t msg_iovlen;            /* Number of blocks */
};
```

Note that this new structure has two new pointers to hold the true start and end of this array. Since the compiler inserts and manages these fields, their contents can be trusted; furthermore, since they are added to the structure itself, access to this bounds information is fast.

Unfortunately, fat pointers change data structure layouts, which makes the compiled code incompatible with previously compiled libraries or modules built with another compiler. For example, if part of the Linux kernel was compiled using the original `msghdr` and another part was compiled with `fat_msghdr`, the linked kernel would not work properly. Furthermore, converting between these two representations is both tricky and expensive. As a result, these tools cannot be applied to existing code *incrementally*; instead, existing code must be ported all at once or not at all. Since most tools that use fat pointers do not provide a totally automatic transformation, a nontrivial amount of programmer intervention is required before the program can even be tested. This lack of incrementality limits the scalability of fat pointers to small, self-contained applications.

In this thesis, we observe that most existing C programs are already safe and therefore already contain the information required to check them. For example, in the `msghdr` structure, the programmer has already stored the length of the data in a separate variable, because the programmer herself needs this information in order to verify safety

manually. The only catch is that we have no way of knowing whether this metadata has been set correctly. However, if we make the relationship between these fields known to the compiler, then the compiler can check that both of these fields are used and modified consistently without making significant changes to them.

In essence, the key observation driving this work is that *all of the extra data we need to check a program's safety is already in the program—just not in a form the compiler understands.* If we take advantage of this observation, we can provide efficient safety checks while preserving binary compatibility, which in turn gives us the incremental approach needed to scale to large programs. The goal of this thesis is to provide convenient and principled ways to make this information available to the compiler so that existing C code can be checked.

1.3 Dependent Types

The primary mechanism that we use to annotate existing C programs is a dependent type. A *dependent type* is a type whose meaning depends on the run-time value of some other program expression. Such types are extremely expressive, and they are a very effective way to add the sort of annotations we need in order to properly check C programs. Unfortunately, they are also correspondingly difficult to reason about.

As described in the previous section, the main challenge in verifying safety under the original C type system is that the information required to verify safety is often split into multiple state elements and is therefore invisible to the compiler. For example, a string may be represented by a pointer and its length, and a disjoint union may be represented by a

data variable along with a tag indicating its type. Dependent types allow us to assign types that relate these two state elements and permit the compiler to perform the appropriate checks.

For example, in the structure declaration for `msghdr`, the programmer clearly wishes to associate two state elements: `msg_iov` points to an array of data blocks, and `msg_iovlen` indicates the number of blocks. However, since the C compiler does not understand this intended relationship, it cannot enforce correct and consistent usage of this data. Indeed, the C compiler maintains no information about the length of the array to which `msg_iovlen` points, so the programmer is responsible for simply "getting it right".

With Deputy, however, the programmer can annotate this structure as follows:

```
struct msghdr {
    struct iovec * COUNT(msg_iovlen) msg_iov;  /* Data blocks      */
    size_t msg_iovlen;                         /* Number of blocks */
};
```

The `COUNT` annotation modifies the preceding pointer type, indicating that this pointer is actually a pointer to an array of `msg_iovlen` characters. As a result, the new type of `msg_iov`, which is `struct iovec * COUNT(msg_iovlen)`, depends on the run-time value of `msg_iovlen`, and it is therefore a dependent type. Because we can now express this dependency, the compiler can now check that we set and use these two fields in a consistent manner. Note that we do not trust this annotation—rather, the programmer declares their intent, and the compiler checks that the programmer's actions are consistent with this declared intent.

Of course, this approach presents its own set of challenges. Because types can now depend on the run-time values of other data in the program, type checking becomes

significantly more challenging—even undecidable. Previous work on dependent types in the context of practical languages [XP99, Xi00] has not been designed for existing code, and in particular has not allowed for the mutation of variables involved in dependencies. This thesis presents a dependent type framework for use with systems code that addresses many of these challenges. First, we use insights from axiomatic semantics to allow a more permissive approach to mutation. Second, we use run-time checks to avoid the problem of undecidability. And finally, we propose a technique called *automatic dependencies* that, when coupled with other inference mechanisms, reduces the programmer's annotation burden significantly. Using these techniques, we can allow programmers to annotate and check existing code without making significant modifications to the code and without sacrificing compatibility.

Of course, Deputy's annotations cannot anticipate all possible idioms, so ultimately there will be some code that cannot be annotated in its current form. Our goal is to design Deputy's annotations to handle the vast majority of common C idioms, with the hope that the remaining ones can be rewritten or redesigned as necessary.

1.4 Applications

To evaluate the effectiveness of Deputy, we need to see how well it applies to real code in practice. We examine several smaller benchmark suites that allow us to observe Deputy's annotation burden and performance in comparison to previous tools such as CCured [NCH+05]. In addition, we look at the Linux operating system as a case study for using Linux on a large, real-world system.

Our smaller benchmark suites include SPEC 95, Olden, Ptrdist, and MediaBench, as well as several components of TinyOS [HSW+00], an operating system intended for embedded systems. Many of these benchmarks allow us to draw comparisons between Deputy and CCured, and others allow us to show how Deputy is used (and how it can find bugs) in small to medium-sized software packages.

The acid test, however, is Linux, which offers the best example of complex, real-world code that can benefit significantly from additional program analysis. Our evaluation looks at two aspects of Linux.

First, we examine Linux device drivers in the context of the SafeDrive recovery system. SafeDrive uses Deputy to detect errors in device drivers, and when an error is detected, SafeDrive restarts the driver. Our experiments with SafeDrive demonstrate Deputy's effectiveness in detecting type and memory errors, and it shows how Deputy can be used to improve isolation and reliability in a commodity operating system.

Second, we examine the Linux kernel itself. Although recovery is more difficult in this context, using Deputy in the context of the kernel allows us to rule out potential security flaws and to lay the foundation for further program analysis. We examine the programmer effort required to annotate an entire working kernel as well as the performance penalty for introducing the necessary run-time checks.

1.5 Summary

The contributions of this thesis are as follows:

- A practical technique for enforcing safety in existing systems code (i.e., dependent

type annotations).

- A novel and extensible dependent type framework that addresses the challenges of mutation, decidability, and usability in a dependent type system for existing low-level code.

- A specific set of dependent types for use with existing C programs, including bounded pointers, null-terminated pointers, and tagged unions.

- A set of experiments using small and medium-size benchmarks as well as the Linux kernel that validate this approach in the context of real-world code by investigating its annotation burden and performance impact.

The remainder of this thesis is organized as follows. In Chapter 2, we lay the theoretical foundation for Deputy's type system, independent of the specific type annotations needed for C programs. In Chapter 3, we examine the specific annotations used by our implementation of Deputy in order to annotate real C programs, including bounded pointers, null-terminated pointers, and tagged unions. Then, in Chapter 4, we explore the various applications of Deputy in order to evaluate Deputy's effectiveness. Finally, we discuss related work in Chapter 5, and then we conclude.

Chapter 2

Deputy

In order to address the problem of type and memory safety in existing C programs, we have designed Deputy, a type system and compiler for annotated C code that enforces both type safety and spatial memory safety. As described in the previous chapter, we use dependent types to enforce these properties while preserving backward compatibility. In this chapter, we present the dependent type framework that provides the foundation for Deputy. This framework addresses a number of previous unsolved problems regarding the use of dependent types in low-level programs.

Types are perhaps the most successful mechanism currently used by programming languages to specify program invariants. Conventional types, while not particularly expressive, are both widely used and well-understood by programmers and language designers alike. Dependent types, which extend these conventional types with the ability to express invariants relating multiple state elements, are both more powerful and more difficult to use. However, such dependencies also play a fundamental role in low-level programming.

For example, the following widespread low-level programming practices all involve dependencies: an array represented as a count of elements along with a pointer to the start of the buffer; a pointer to an element inside an array along with the array bounds; and a disjoint union type along with a tag that identifies the active field of the union. Understanding these dependencies and verifying that the appropriate invariants are upheld is critical to proving the type and memory safety of even the most basic low-level programs.

At its core, Deputy is a framework for adding dependent types to imperative programs. These dependent types will allow programmers to describe a number of common C programming idioms, thus allowing the compiler to verify that these idioms are being used properly. In this chapter, we focus on the framework itself, describing the general principles that can be used to add dependent types to *any* low-level programming language. In the following chapter, we will show how this framework can be instantiated for specific C programming idioms.

2.1 Goals and Contributions

The primary challenges to consider when adding dependent types to imperative programs are as follows:

- *Soundness:* Mutation of variables or heap locations, used heavily in low-level programs, can invalidate the types of some state elements. For example, given the `msghdr` structure defined in Chapter 1, it would be unsafe to set the field `msg_iovlen` to a value that is larger than the length of the array to which `msg_iov` points. Previous dependent type systems have not addressed this problem, as they focus primarily on

pure functional languages [Aug98, XP99]. Dependent type systems that focus on imperative languages often simply disallow mutable variables in dependent types [Xi00]. In this thesis, we show that it is possible to combine mutation and dependencies in a more flexible manner by using a type rule inspired by Hoare's rule for assignment. This approach can be used for dependencies between variables and between fields of heap-allocated structures.

- *Completeness:* Dependent type checking involves reasoning about the run-time values of expressions, a task which is not typically required for type checking. In order to be useful, a dependent type system must be able to perform the appropriate reasoning for most common programs. For example, if the `msg_iovlen` field is set to the length of the array to which `msg_iov` points, this operation should be allowed. In most previous dependent type systems, dependencies are restricted to the point where all checking can be done statically, which rules out many common programs, or else type checking is not guaranteed to terminate at all. Instead, we propose the use of run-time checks where static checking is not sufficient. This hybrid type-checking strategy, which has also been used recently by Flanagan [Fla06], provides a reasonable compromise between completeness, compile-time speed, and run-time speed.

- *Usability:* Writing complete dependent type declarations can be a considerable burden. If the programmer were required to write explicit annotations on every pointer in the Linux source code, this technique would be both messy and impractical. We describe a technique for automatic dependency inference for local variables, starting from existing declarations for global variables, data structures, and functions. We

also describe inference techniques for specific dependent types.

We have applied these general principles for low-level dependent types to create the Deputy type system for the C programming language. As discussed in the previous chapter, adding these dependent types to C programs allows us to check existing idioms efficiently and scalably without changing the representation of data in the program. Essentially, our dependent type framework allows us to expose a lower-level program representation without losing the safety benefits of higher-level languages.

This remainder of this chapter proceeds as follows. First, we present an overview to describe the basic components of our type system. Then, we present a simple imperative language that is designed to be extended with dependent types, and we present a type system that allows us to address the challenges listed above for any dependent types added to our system. Then, we present a number of extensions to this framework that support common imperative language features such as structure types and function calls, and we discuss automatic dependencies, a feature of our system that improves usability significantly.

2.2 Overview

In this section, we will provide a brief overview of Deputy's architecture. As shown in Figure 2.1, we begin with source files and header files that contain programmer annotations, which are merged using the standard C preprocessor. After merging, we *infer* additional annotations that the programmer did not supply directly. Then, we *add run-time checks* that enforce the safety properties we require, and finally, we *optimize* the code, which eliminates any checks that can be verified at compile time. These last two steps are

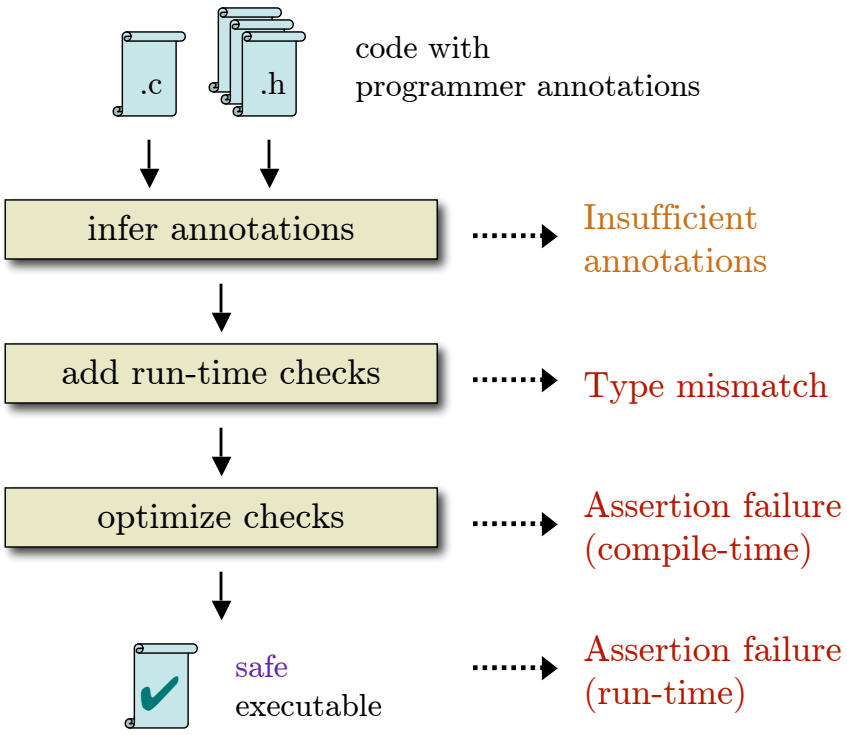

Figure 2.1: Deputy's architecture.

comparable to generating and discharging a verification condition; the main difference here is that our "verification condition" is designed to be emitted as executable code if it cannot be proved or disproved by the compiler.

The arrows on the right side of Figure 2.1 show the warnings and errors that this process can produce. The inference phase can produce warnings of insufficient annotations; often, such warnings are generated by function parameters where the callees or callers are unavailable. The next phase, adding run-time checks, can fail with a type mismatch, much like a standard C compiler; these failures occur because many types must match at compile time in order for appropriate checks to be generated. Finally, we are left with assertion failures, which can be found either by the optimizer (at compile-time) or during execution (at run-time).

In order to provide more intuition for this process, we will consider an example C program with Deputy annotations, and we will show how this program is modified in each phase of compilation. The original program, which sums the elements in an array, is shown on the left side of Figure 2.2. The programmer has supplied one Deputy annotation, written `COUNT(end - buf)`, which modifies the type of the parameter `buf`. This annotated type indicates that this variable represents a pointer to an array of at least `end - buf` integers. Adding this annotation is the only work the programmer needs to do in order to run Deputy on this example. Note that the temporary variable used in this program is shown in order to better demonstrate Deputy's features; however, the programmer may omit this temporary variable if they so choose.

Once annotated, Deputy processes this program as follows:

```
1 int sum (int * COUNT(end - buf) buf,
2          int * end) {
3   int sum = 0;
4   while (buf < end) {
5
6     sum += * buf;
7
8
9     int * tmp = buf + 1;
10
11    buf = tmp;
12  }
13  return sum;
14 }
```

```
1 int sum (int * COUNT(end - buf) buf,
2          int * end) {
3   int sum = 0;
4   while (buf < end) {
5     assert(0 < end - buf);
6     sum += * buf;
7     int tmplen = (end - buf) - 1;
8     assert(0 <= 1 <= end - buf);
9     int * COUNT(tmplen) tmp = buf + 1;
10    assert(0 <= end - tmp <= tmplen);
11    buf = tmp;
12  }
13  return sum;
14 }
```

Figure 2.2: A sample Deputy program with inferred code (underlined) and run-time checks (italicized).

- **Pass 1: Inference of missing annotations.** For each pointer type without bounds annotations (e.g., `tmp`), Deputy introduces a fresh local variable to hold the bounds, along with appropriate assignments for this variable whenever the pointer is assigned. In this example, the inferred code is shown underlined on the right-hand side of Figure 2.2. Here, the inference algorithm has introduced the variable `tmplen`, which is updated to store the length of the array pointed to by `tmp`, and we have used this variable to annotate `tmp`. This technique, which we call *automatic dependencies*, will be described in detail in Section 2.5. We also provide inference mechanisms that are specific to the dependent types on which our framework is instantiated, which will be discussed along with those types in the following chapter.

- **Pass 2: Flow-insensitive type checking and instrumentation.** Next, Deputy type checks the program using a flow-insensitive type system. Any checks that involve reasoning about run-time values of expressions are emitted as run-time assertions. In this example, the assertions added in this phase are shown in italics on the right-hand side of Figure 2.2. For example, the assertion in line 5 ensures that the `buf` array is nonempty and can therefore be safely dereferenced.

  The check on line 10 is particularly interesting because it shows the power of Deputy's handling of mutation in presence of dependent types. Previous dependent type systems would disallow any assignments to `buf` because there exist types in the program that depend on it. Instead, Deputy inserts checks that ensure that `buf`'s type invariant will still hold after the assignment. Specifically, we ensure that `tmp` has at least `end - tmp` elements and thus will satisfy `buf`'s type invariant when assigned to `buf`. In this

example, we have actually taken advantage of a self-dependency; in our experience, such dependencies are particularly useful when designing flexible types for low-level code, since they allow general annotations to be specialized for common cases. The rules for type checking and for inserting run-time checks are described in Section 2.3.

- **Pass 3: Flow-sensitive optimization of checks.** Because our flow-insensitive type checker has limited ability to recognize redundant checks, we follow type checking with a flow-sensitive optimization phase. In Figure 2.2, all checks could reasonably be eliminated by using standard data-flow techniques. For example, the guard condition on the loop will ensure that the first assertion always holds. We are also capable of discovering that certain checks will always fail, in which case an error is emitted at compile time.

  By separating the flow-insensitive type checker from the flow-sensitive optimizer, we simplify both the implementation and the programmer's view of the type system. From the implementer's point of view, we can perform all reasoning about dependent types in a flow-insensitive context, while using more standard dataflow techniques in the flow-sensitive optimizer. The flow-insensitive pass is responsible for enforcing correctness, and the flow-sensitive pass is responsible for improving performance while maintaining correctness. Indeed, the amount of static memory safety enforcement depends directly on the quality of the optimizer; essentially, the optimizer encapsulates all of the static, flow-sensitive reasoning about the safety properties we wish to enforce. Since optimization depends heavily on the details of the checks themselves, we defer a discussion of the optimizer until Chapter 3, where we discuss dependent types for

C programs.

We can use this example to contrast our approach with safe C type systems that use fat pointers [JMG+02, NCH+05]. With these systems, the pointer `buf` might be stored as a two-word pointer containing both the original pointer value and a pointer to the buffer's end. As a result, the function now takes three parameters, so all callers of this function must be instrumented as well. However, in this case, the programmer has already supplied the required metadata. Deputy allows the programmer to identify that metadata, which allows the program to be verified without any change in the callers to this function. This approach is crucial for large systems that must be annotated and ported incrementally.

It is also important to note that Deputy can use existing run-time checks in addition to using existing metadata. For example, our optimizer can take advantage of the conditional in line 4 in order to eliminate the unnecessary assertions that have been inserted. As a result, in this example, *the Deputy-compiled program is identical in performance to the original program.* In tools based on fat pointers, all the checks would refer to components of the fat pointers, which would have no apparent relationship to the existing metadata; therefore, we would not be able to optimize away these checks based on the run-time checks that are already present in the program.

2.3 Dependent Type Framework

This section presents the key components of our dependent type framework. We begin with a simple imperative language supporting dependencies between local variables, and we present our type rules along with a sketch of our soundness proof.

$$
\begin{array}{lrcl}
\text{Constructors} & C & ::= & \text{int} \mid \text{ref} \mid \ldots \\
\text{Types} & \tau & ::= & C \mid \tau_1\ \tau_2 \mid \tau\ e \\
\text{Kinds} & \kappa & ::= & \text{type} \mid \text{type} \to \kappa \mid \tau \to \kappa \\
\text{L-expressions} & \ell & ::= & x \mid *e \\
\text{Expressions} & e & ::= & n \mid \ell \mid e_1\ \text{op}\ e_2 \\
\text{Commands} & c & ::= & \text{skip} \mid c_1; c_2 \mid \ell := e \mid \text{assert}(\gamma) \mid \\
 & & & \text{let } x : \tau = e \text{ in } c \mid \text{let } x = \text{new } \tau(e) \text{ in } c \\
\text{Predicates} & \gamma & ::= & e_1\ \text{comp}\ e_2 \mid \text{true} \mid \gamma_1 \wedge \gamma_2
\end{array}
$$

$$
\begin{array}{rcl}
n & \in & \text{Integer constants} \\
x, y & \in & \text{Variables} \\
\text{op} & \in & \text{Binary operators} \\
\text{comp} & \in & \text{Comparison operators}
\end{array}
$$

Figure 2.3: The grammar for a simple dependently-typed imperative language.

2.3.1 Language

Although our implementation uses the concrete syntax of C, as shown in the previous section, for the purposes of our formalism we use the simpler language shown in Figure 2.3. In this language, types are specified using one of several type constructors, which represent type families indexed by types or by expressions. The built-in constructors are the nullary type constructor "int" (a prototypical base type) and the unary type constructor "ref". The "ref" constructor allows the creation of types such as "ref int", which is an ML-style reference to an integer; this reference type is introduced here so that we can show how our type system works in the presence of memory reads and writes. In later sections, we will introduce additional type constructors, such as more expressive pointer types. The built-in constructors do not yield dependent types, but the additional constructors will.

Types are classified into kinds. The kind "type" characterizes complete types, whereas the functional kinds characterize type families that have to be applied to other complete types, or to expressions of a certain type, to eventually form complete types. Constructors that are applied to expressions (as opposed to types) yield dependent types. For the two constructors we have seen so far, the kind of "int" is "type", and the kind of "ref" is "type → type"; thus, "ref" can be applied to "int" to get the complete type "ref int".

To show how this system can be extended with additional type constructors, consider the `COUNT` annotation used in Figure 2.2. To represent this annotated pointer type, we can introduce the constructor "array" with kind "type → int → type", such that "array τ e_{len}" is the type of arrays of elements of type τ and length at least e_{len}. In the concrete syntax this type is written as "`t * COUNT(e_len)`".

The remainder of this language is standard. Note that $*$ represents pointer dereference, as in C. Also note that assertions are present only for compilation purposes and do not appear in the input program. Finally, note that we omit loops and conditionals, which are irrelevant to our flow-insensitive type system, and we omit function calls, which will be added later as an extension.

In the remainder of this thesis, we will use this language to present formal rules for checking Deputy's dependent types; however, we will also present brief examples from real code using the concrete syntax in order to show how these concepts are applied in the real world.

$\Gamma \vdash_L \tau :: \kappa$	In type environment Γ, τ is a local, well-formed type with kind κ.

$$\frac{}{\Gamma \vdash_L C :: \mathrm{kind}(C)}\ (\textsc{type ctor}) \qquad \frac{\Gamma \vdash_L \tau :: (\tau' \to \kappa) \quad \Gamma \vdash_L e : \tau'}{\Gamma \vdash_L \tau\ e :: \kappa}\ (\textsc{type exp}) \qquad \frac{\Gamma \vdash_L \tau_1 :: (\mathrm{type} \to \kappa) \quad \emptyset \vdash_L \tau_2 :: \mathrm{type}}{\Gamma \vdash_L \tau_1\ \tau_2 :: \kappa}\ (\textsc{type type})$$

$\Gamma \vdash_L e : \tau$	In type environment Γ, e is a local, well-typed expression with type τ.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau}\ (\textsc{local name}) \qquad \frac{}{\Gamma \vdash_L n : \mathrm{int}}\ (\textsc{local num}) \qquad \frac{\Gamma \vdash_L e_1 : \mathrm{int} \quad \Gamma \vdash_L e_2 : \mathrm{int}}{\Gamma \vdash_L e_1\ \mathrm{op}\ e_2 : \mathrm{int}}\ (\textsc{local int arith})$$

$\Gamma \vdash e : \tau \Rightarrow \gamma$	In type environment Γ, e is a well-typed expression with type τ, if γ is satisfied.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \Rightarrow \mathrm{true}}\ (\textsc{var}) \qquad \frac{}{\Gamma \vdash n : \mathrm{int} \Rightarrow \mathrm{true}}\ (\textsc{num}) \qquad \frac{\Gamma \vdash e_1 : \mathrm{int} \Rightarrow \gamma_1 \quad \Gamma \vdash e_2 : \mathrm{int} \Rightarrow \gamma_2}{\Gamma \vdash e_1\ \mathrm{op}\ e_2 : \mathrm{int} \Rightarrow \gamma_1 \wedge \gamma_2}\ (\textsc{int arith}) \qquad \frac{\Gamma \vdash e : \mathrm{ref}\ \tau \Rightarrow \gamma}{\Gamma \vdash *e : \tau \Rightarrow \gamma}\ (\textsc{deref})$$

Figure 2.4: Judgments for checking types, expressions, and local expressions.

2.3.2 Type Rules

In this section, we present the type rules for the core language. Figure 2.4 and Figure 2.6 show these rules and summarize the judgment forms involved.

To understand these rules, we must first discuss our strategy for handling mutation in the presence of dependent types, which has two important components. First, we use a typing rule inspired by the Hoare axiom for assignment to ensure that each mutation operation preserves well-typedness of the state. Second, dependencies in types are restricted such that we can always tell statically which types can be affected by each mutation operation. For this purpose, we restrict types to contain only expressions formed using constants, local variables, and arbitrary arithmetic operators. In other words, we do not allow memory dereferences in types, because if we did, we would require perfect alias information in order to determine which locations are affected by mutation operations. We refer to these restricted notions of expressions and types as *local expressions* and *local types*. Our type rules will require that all types written by the programmer be local types. Note that when we add structures to the language in the next section, we will extend this notion to allow field types to refer to other fields of the same structure. However, in the full version of Deputy for C, local expressions exclude function calls, references to fields of other structures, and variables whose address is taken.

We now consider the well-formedness rules for types, shown at the top of Figure 2.4. If Γ is a mapping from variables to their types, we say that a type τ is well-formed in Γ if τ depends only on the variables in Γ. The judgment $\Gamma \vdash_L \tau :: \kappa$ states that τ is well-formed in Γ and has kind κ. Note that expression arguments must be well-typed in Γ, as shown

in rule (TYPE EXP), but type arguments must be well-formed in the empty environment, as shown in rule (TYPE TYPE). This latter restriction is essential for the "ref" constructor. If we allowed variables in Γ to appear in the base type of a reference, then we would need perfect aliasing information to ensure that we can find all references to a certain location when its type is invalidated through mutation.

We have two judgments for checking expressions: one for local expressions and one for non-local expressions. The rules for local expressions are standard, but the rules for non-local expressions produce a condition γ that must hold in order for the judgment to be valid. This condition is generated during type checking and will be emitted as a run-time check unless it is discharged statically by the optimizer.

The rules presented in Figure 2.4 do not generate any interesting guard conditions themselves. Our intent is that an instantiation of this type system will provide additional type constructors whose typing rules include non-trivial guards. For example, to access arrays using the array constructor introduced earlier, we might add new typing rules for pointer arithmetic and dereference, which are shown in Figure 2.5.

The first two rules in Figure 2.5 handle array dereference and arithmetic, both of which involve non-trivial assertions. The first rule adds an assertion that ensures that at least one element is present in the array being dereferenced, and the second rule allows arithmetic within the bounds of the array. Note that we allow zero-length arrays to be constructed, but we check for this case at dereference; this approach is useful in programs that construct pointers to the end of an array, as allowed by ANSI C. To see how these rules work, consider the example in Figure 2.2. The assertion on line 5 is generated when

(ARRAY DEREF)

$$\frac{\Gamma \vdash e : \text{array } \tau\ e_{len} \Rightarrow \gamma_e}{\Gamma \vdash *e : \tau \Rightarrow \gamma_e \wedge (0 < e_{len})}$$

(ARRAY ARITH)

$$\frac{\Gamma \vdash e : \text{array } \tau\ e_{len} \Rightarrow \gamma_e \qquad \Gamma \vdash e' : \text{int} \Rightarrow \gamma_{e'}}{\Gamma \vdash e + e' : \text{array } \tau\ (e_{len} - e') \Rightarrow \gamma_e \wedge \gamma_{e'} \wedge (0 \leq e' \leq e_{len})}$$

(ARRAY COERCE)

$$\frac{\Gamma \vdash e : \text{array } \tau\ e_{len} \Rightarrow \gamma_e \qquad \Gamma \vdash e'_{len} : \text{int} \Rightarrow \gamma_{e'_{len}}}{\Gamma \vdash e : \text{array } \tau\ e'_{len} \Rightarrow \gamma_e \wedge \gamma_{e'_{len}} \wedge (0 \leq e'_{len} \leq e_{len})}$$

(ARRAY WRITE)

$$\frac{\Gamma \vdash e_1 : \text{array } \tau\ e_{len} \Rightarrow \gamma_1 \qquad \Gamma \vdash e_2 : \tau \Rightarrow \gamma_2}{\Gamma \vdash *e_1 := e_2 \quad \Rightarrow \quad \text{assert}(\gamma_1 \wedge \gamma_2 \wedge (0 < e_{len}));\ \ *e_1 := e_2}$$

Figure 2.5: Rules for array dereference, arithmetic, coercion, and update.

checking the pointer dereference in the following line, and the assertion on line 8 is generated when checking the arithmetic on the following line.

The third rule in Figure 2.5 is a coercion rule that allows long arrays to be used where short arrays are expected. For example, if we wanted to use an expression of type "array τ 5" where the type "array τ n" was expected, we would generate the assertion $0 \leq n \leq 5$, which ensures that this coercion is valid. Note that the base type τ must be identical in both cases; a mismatch in this base type will result in a compile-time type error. Also note that in our implementation, we ensure that type checking is syntax-directed by invoking coercion rules only from the rules for commands. The fourth rule in Figure 2.5 is a command for array writes that will be discussed later.

Finally, we have the judgment for checking commands, described in Figure 2.6. This judgment, written $\Gamma \vdash c \Rightarrow c'$, says that in environment Γ, command c is compiled to command c' by adding assertions with the necessary guard conditions. These two commands must have identical semantics if no assertion in c' fails. The first two rules, (SKIP) and (SEQ), are trivial, but the rest require some discussion.

First, the (VAR WRITE) rule is responsible for updates to variables in the presence of dependent types and is a key contribution of our type system. This rule says that when updating a variable x with the value of expression e, we check all variables y in the current environment to see that their types still hold after substituting e for x. This rule essentially verifies that the assignment does not break any dependencies in the current scope.

The intuition for this rule is based on the Hoare axiom for assignment, which says that an assignment $x := e$ preserves an invariant ϕ if one can prove that $\phi \implies \phi[^{e}/_{x}]$,

$\Gamma \vdash c \Rightarrow c'$	In type environment Γ, command c compiles to c', where c' is identical to c except for added assertions.

(SKIP)

$$\frac{}{\Gamma \vdash \text{skip} \Rightarrow \text{skip}}$$

(SEQ)

$$\frac{\Gamma \vdash c_1 \Rightarrow c_1' \qquad \Gamma \vdash c_2 \Rightarrow c_2'}{\Gamma \vdash c_1; c_2 \Rightarrow c_1'; c_2'}$$

(VAR WRITE)

$$\frac{x \in \text{Dom}(\Gamma) \qquad \text{for all } (y : \tau_y) \in \Gamma, \quad \Gamma \vdash y[e/x] : \tau_y[e/x] \Rightarrow \gamma_y}{\Gamma \vdash x := e \quad \Rightarrow \quad \text{assert}(\bigwedge_{y \in \text{Dom}(\Gamma)} \gamma_y);\ x := e}$$

(MEM WRITE)

$$\frac{\Gamma \vdash e_1 : \text{ref } \tau \Rightarrow \gamma_1 \qquad \Gamma \vdash e_2 : \tau \Rightarrow \gamma_2}{\Gamma \vdash *e_1 := e_2 \quad \Rightarrow \quad \text{assert}(\gamma_1 \wedge \gamma_2);\ *e_1 := e_2}$$

(LET)

$$\frac{x \notin \text{Dom}(\Gamma) \qquad \Gamma, x : \tau \vdash_L \tau :: \text{type} \qquad \Gamma \vdash e : \tau[e/x] \Rightarrow \gamma \qquad \Gamma, x : \tau \vdash c \Rightarrow c'}{\Gamma \vdash \text{let } x : \tau = e \text{ in } c \quad \Rightarrow \quad \text{assert}(\gamma);\ \text{let } x : \tau = e \text{ in } c'}$$

(ALLOC)

$$\frac{x \notin \text{Dom}(\Gamma) \qquad \emptyset \vdash_L \tau :: \text{type} \qquad \Gamma \vdash e : \tau \Rightarrow \gamma \qquad \Gamma, x : \text{ref } \tau \vdash c \Rightarrow c'}{\Gamma \vdash \text{let } x = \text{new } \tau(e) \text{ in } c \quad \Rightarrow \quad \text{assert}(\gamma);\ \text{let } x = \text{new } \tau(e) \text{ in } c'}$$

Figure 2.6: Judgments for checking commands.

where the latter predicate is the weakest precondition of $x := e$ with respect to ϕ. If we view the type environment Γ as an invariant predicate on the state of the program, the (VAR WRITE) rule states that assignments maintain the invariant. In particular, for every variable-type pair $(y, \tau_y) \in \Gamma$, this rule ensures that the new state, with e substituted for x everywhere, can be derived using the old environment Γ.

One way to look at this rule is to split it into two parts. When $y = x$, we have the premise $\Gamma \vdash e : \tau_x[^e/_x] \Rightarrow \gamma_x$, which essentially says that e has the type of x. For all other $y \neq x$, we have the premise $\Gamma \vdash y : \tau_y[^e/_x] \Rightarrow \gamma_y$, which makes sure that this mutation does not break dependencies elsewhere in the environment. Note that if there are no dependencies in the environment, the substitutions have no effect, leaving us with $\Gamma \vdash e : \tau_x \Rightarrow \gamma_x$ and $\Gamma \vdash y : \tau_y \Rightarrow \gamma_y$. Since the latter is trivial, we see that we are left with the standard rule for type checking assignment in an imperative language.

To understand this rule in more detail, consider the following code:

$$\begin{aligned}&\text{let } n : \text{int} = \ldots \text{ in}\\&\text{let } a : \text{array int } n = \ldots \text{ in}\\&n := n - 1\end{aligned}$$

In this example, decrementing n should be safe as long as $n \geq 1$, because if a is an array of length n, it is also an array of length $n - 1$. When we apply the (VAR WRITE) rule to this assignment, the premises are as follows:

$$\begin{aligned}&\Gamma \vdash n[^{n-1}/_n] : \text{int}[^{n-1}/_n] \Rightarrow \gamma_n\\&\Gamma \vdash a[^{n-1}/_n] : (\text{array int } n)[^{n-1}/_n] \Rightarrow \gamma_a\end{aligned}$$

The first premise is trivial, with $\gamma_n = \text{true}$. The second premise is more interesting. After substitution, it becomes $\Gamma \vdash a : \text{array int } (n-1) \Rightarrow \gamma_a$. If we apply the (ARRAY COERCE) rule shown above, we can derive this judgment with $\gamma_a = 0 \leq n - 1 \leq n$. After

static optimization, this check can be reduced to $0 \leq n - 1$, which is precisely the check we expected.[1]

As a second example, consider the following code, which is derived from line 11 of the example in Figure 2.2:

$$\begin{aligned}&\text{let } buf : \text{array int } (end - buf) = \ldots \text{ in}\\&\text{let } tmp : \text{array int } tmplen = \ldots \text{ in}\\&buf := tmp\end{aligned}$$

According to the (VAR WRITE) rule, the premises when type checking the assignment are as follows:

$$\begin{aligned}&\Gamma \vdash buf[{}^{tmp}/_{buf}] : (\text{array int } (end - buf))[{}^{tmp}/_{buf}] \Rightarrow \gamma_{buf}\\&\Gamma \vdash tmp[{}^{tmp}/_{buf}] : (\text{array int } tmplen)[{}^{buf}/_{len}] \Rightarrow \Gamma_{tmp}\end{aligned}$$

The first premise simplifies to $\Gamma \vdash tmp : \text{array int } (end - tmp) \Rightarrow \gamma_{buf}$, which is derivable using the coercion rule if $\gamma_{buf} = 0 \leq end - tmp \leq tmplen$. The second premise simplifies to $\Gamma \vdash tmp : \text{array int } tmplen \Rightarrow \gamma_{tmp}$, which is satisfied by the (VAR) rule with $\gamma_{tmp} = \text{true}$. Note that the overall assertion, $0 \leq end - tmp \leq tmplen$, is the very same assertion that appears in line 10 of Figure 2.2.

This example demonstrates a particularly important instance of mutation: self-dependencies. In this particular case, buf actually depends on itself, and because of this dependency, we are able to make the array type more expressive: we have converted a type representing an array with n entries into a type representing an array with a pointer to its upper bound. In fact, our experience with real code suggests that these self-dependencies, along with the mutation rule that enables variables with such types to be updated, provide a

[1] We take care to account for possible overflow of machine arithmetic, which is simple when reasoning about array indices that must be bounded by the length of an array.

very powerful mechanism for creating general and useful dependent type annotations. These self-dependencies are in many ways the most important application of the (VAR WRITE) rule.

In summary, the (VAR WRITE) rule allows us to verify that dependencies in the local environment have not been broken, and the "local expression" restriction on base types of pointers ensures that there are no dependencies from the heap. Essentially, a combination of the Hoare-inspired assignment rule and the local type restriction have allowed us to verify mutation in the presence of dependent types.

The remainder of the rules for commands are more conventional. The (MEM WRITE) rule checks to make sure that the type being written matches the base type of the pointer; since the well-formedness rule for reference types requires that the contents of a reference be independent of its environment, there is no need to reason about dependencies here. Note that this rule only covers writes to references; in order to write to arrays, we need the (ARRAY WRITE) rule shown in Figure 2.5, which checks to make sure the array contains at least one element. The (LET) rule requires a substitution when checking e in order to account for self-dependencies; however, we need not check the rest of the environment in this case because we are introducing a fresh variable. Finally, the (ALLOC) rule requires no substitution, since the type τ is not allowed to depend on any other data in its environment.

2.3.3 Soundness

We sketch here the key soundness results, with the goal of exposing the various formal requirements on the framework for ensuring sound handling of mutation in presence of dependent types.

Let Val be the set of machine values, with Addr a synonym used when dealing with memory addresses. Let Var be the set of variable names. The state of the execution ρ contains at least three elements: ρ_E : Var $\rightarrow$ Val giving the value of each variable, ρ_S : Addr $\rightarrow$ Val representing the contents of memory (i.e., the store), and ρ_A : Addr $\rightarrow \tau$ indicating the type of each memory location (i.e., the allocation state). Additional elements may be added in order to support additional type constructors. We define a distinguished ρ_{fail} value to be the unique state resulting from a failed assertion.

The evaluation of expressions and commands is standard: $[\![e]\!]\rho$ denotes the value $v \in$ Val of expression e in state ρ, and $[\![c]\!]\rho$ denotes the state ρ' that results when command c executes in state ρ. We define $[\![c]\!]\rho_{fail} = \rho_{fail}$ for all commands c. We assume termination of all commands, in order to simplify the presentation.

An essential element of the formalization is that for each type τ we can define the set of values of that type in state ρ as $[\![\tau]\!]\rho$, as follows:

$$\begin{aligned}
[\![\text{int}]\!]\rho &= \text{Val} \\
[\![\text{ref}]\!]\rho &= \lambda\tau.\{a \in \text{Dom}(\rho_A) | \tau = \rho_A(a)\} \\
[\![\tau_1\ \tau_2]\!]\rho &= ([\![\tau_1]\!]\rho)\ \tau_2 \\
[\![\tau\ e]\!]\rho &= ([\![\tau]\!]\rho)\ ([\![e]\!]\rho)
\end{aligned}$$

In particular, note that each constructor C must have some meaning given by $[\![C]\!]\rho$. If additional constructors are added, their meanings must be given as well; in some cases, these definitions may require an augmented notion of state (e.g., with a history of the locking operations when we have a type constructor characterizing the state of locks). The fact that types have state-based meanings is essential for the adequacy of using the Hoare's assignment axiom for type checking.

Note also the difference between the way in which the type application and ex-

pression application rules are evaluated. The former rule is important because the rules for type constructors often need to distinguish types by their syntactic form, not their value; for example, the "ref" constructor needs to compare types directly. The latter rule is important because we must be sure to evaluate expressions in a suitable environment. In some sense, this distinction corresponds to the distinction in the type rules where type arguments are checked in the empty environment and expression arguments are checked in Γ.

We say that a type environment Γ is well-formed if for all $x \in \mathrm{Dom}(\Gamma)$, $\Gamma \vdash_L \tau :: \mathrm{type}$. We say that an execution state ρ is well-formed if $\mathrm{Dom}(\rho_S) = \mathrm{Dom}(\rho_A)$ and for all $a \in \mathrm{Dom}(\rho_A)$, we have $\rho_S(a) \in [\![\rho_A(a)]\!]\rho$ and $\emptyset \vdash_L \rho_A(a) :: \mathrm{type}$.

We say that $\rho \models \Gamma$ if ρ and Γ are well-formed and if for all $x \in \mathrm{Dom}(\Gamma)$, $[\![x]\!]\rho \in [\![\Gamma(x)]\!]\rho$. Finally, we say that $\rho \models \gamma$ holds if predicate γ is satisfied in state ρ.

Our soundness theorems (provable by induction on the structure of the typing derivations) are as follows:

Theorem 1 (Soundness for expressions) *If* $\rho \models \Gamma$, $\Gamma \vdash e : \tau \Rightarrow \gamma$, *and* $\rho \models \gamma$, *then* $[\![e]\!]\rho \in [\![\tau]\!]\rho$.

Theorem 2 (Soundness for commands) *If* $\rho \models \Gamma$, $\Gamma \vdash c \Rightarrow c'$, *then* $[\![c']\!]\rho = \rho'$ *and either* $\rho' = \rho_{fail}$ *or* $\rho' \models \Gamma$.

The interesting cases in the proof are for variable update and memory write. In the former case, the proof works as for the soundness of Hoare's assignment with Γ playing the role of an invariant predicate on the state (by means of the $\rho \models \Gamma$ judgment). In the case of memory write the proof relies on the fact that there are no dependencies on the

contents of store locations because the base type for the "ref" constructor must have no external dependencies. More details are presented in the appendix.

Finally, we know that the only change to the behavior of the program is the possibility of failed assertions:

Theorem 3 (Correctness for commands) *If* $\Gamma \vdash c \Rightarrow c'$, $[\![c']\!]\rho = \rho'$, *and* $\rho' \neq \rho_{fail}$, *then* $[\![c]\!]\rho = \rho'$.

This theorem is easily proved by noting that all of the derivation rules transform a command c into a command of the form "assert(γ); c" for some predicate γ. Thus, the only possible change in the program's behavior is to have a new assertion failure, which means that the final state is either ρ' or ρ_{fail}.

2.4 Extensions

Now that we have presented our core type framework, we will present a formal description of three additional features that are very important in using Deputy on real programs: parallel assignment, structures, and function calls.

2.4.1 Parallel Assignment

One potentially troublesome issue with mutation and dependent types is the order in which several dependent variables are mutated. Consider again the declaration "a : array int n" and the sequence of assignments "$a := a'; n := n'$", where a' has type "array int n'". This sequence of assignments is sound, but the type invariant for a (i.e., "a has length n") may be temporarily violated between these two assignments. Because our

type system is flow-insensitive, it may report an error after the first assignment. In this example, the assignment $a := a'$ will fail if $n' < n$. Changing the order of these assignment statements does not help; in that case, the assignment $n := n'$ would fail if $n' > n$.

To support this operation, we add *parallel assignment* to the language. Modifying multiple local variables simultaneously allows programmers to avoid temporarily violating a dependent type relationship that holds between the variables. Specifically, we introduce a new command to our language:

$$\text{Commands} \quad c \quad ::= \quad \ldots \mid x_1, \ldots, x_n := e_1, \ldots, e_n$$

The runtime semantics of this operation are standard: e_1 through e_n are evaluated before any variable is modified, and then each x_i gets the value of the corresponding e_i.

We type check this operation by generalizing the (VAR WRITE) rule: for each variable y in the environment, we check that its type still holds after substituting the new values of x_1 through x_n. Formally, the type rule is as follows:

(PARALLEL VAR WRITE)

$$\frac{x_1, \ldots, x_n \text{ distinct} \qquad \{x_1, \ldots, x_n\} \subseteq \mathrm{Dom}(\Gamma) \qquad \text{for all } (y : \tau_y) \in \Gamma, \quad \Gamma \vdash y \left[{}^{e_i}\!/_{x_i}\right]_{1 \le i \le n} : \tau_y \left[{}^{e_i}\!/_{x_i}\right]_{1 \le i \le n} \Rightarrow \gamma_y}{\Gamma \vdash x_1, \ldots, x_n := e_1, \ldots, e_n \quad \Rightarrow \quad \mathrm{assert}(\textstyle\bigwedge_{y \in \mathrm{Dom}(\Gamma)} \gamma_y);\ x_1, \ldots, x_n := e_1, \ldots, e_n}$$

We use the notation $\left[{}^{e_i}\!/_{x_i}\right]_{1 \le i \le n}$ to mean the parallel substitution of each x_i with e_i. That is, each occurrence of x_i in the original expression is replaced by e_i, but occurrences of the x_i within the e_i are not recursively expanded.

To see how this rule works, consider the example from above. When applying the

parallel assignment rule to the statement "$a := a'; n := n'$", we get the following premises:

$$\Gamma \vdash a[{}^{a'}\!/_{a}, {}^{n'}\!/_{n}] : (\text{array int } n)[{}^{a'}\!/_{a}, {}^{n'}\!/_{n}] \Rightarrow \gamma_a$$
$$\Gamma \vdash n[{}^{a'}\!/_{a}, {}^{n'}\!/_{n}] : \text{int}[{}^{a'}\!/_{a}, {}^{n'}\!/_{n}] \Rightarrow \gamma_n$$
$$\Gamma \vdash a'[{}^{a'}\!/_{a}, {}^{n'}\!/_{n}] : (\text{array int } n')[{}^{a'}\!/_{a}, {}^{n'}\!/_{n}] \Rightarrow \gamma_{a'}$$
$$\Gamma \vdash n'[{}^{a'}\!/_{a}, {}^{n'}\!/_{n}] : \text{int}[{}^{a'}\!/_{a}, {}^{n'}\!/_{n}] \Rightarrow \gamma_{n'}$$

After the substitutions, these premises become:

$$\Gamma \vdash a' : \text{array int } n' \Rightarrow \gamma_a$$
$$\Gamma \vdash n' : \text{int} \Rightarrow \gamma_n$$
$$\Gamma \vdash a' : \text{array int } n' \Rightarrow \gamma_{a'}$$
$$\Gamma \vdash n' : \text{int} \Rightarrow \gamma_{n'}$$

All of these premises are trivially satisfied by the (VAR) rule, which simply looks the corresponding variables up in Γ.

For a real-world example where parallel assignment can be useful, consider the following example derived from code found in the Linux source files `include/linux/blkdev.h` and `drivers/ide/ide-cd.c`:

```
struct request {
    unsigned int data_len;
    void * COUNT(data_len) data;
} * rq;
*rq->data++ = 0;
--rq->data_len;
```

Here, `data_len` stores the length of `data`, and we wish to both increment `data` and decrement `data_len`. The only change required from the original Linux source code is the addition of the `COUNT` annotation, as long as there is some form of parallel assignment to handle the related updates in these two statements. Without parallel assignment, it

becomes much trickier to handle this example, because the invariant of the `COUNT` annotation is temporarily violated between these two statements.

The major challenge of using the parallel assignment rule is that it is not often apparent where this rule should be used. Since no such statement exists in C itself, we must be conservative about which statements are considered parallel. For example, we can group adjacent assignment expressions in a basic block as long as their right-hand sides do not refer to their left-hand sides. However, this conservative approach is often insufficient; for example, in some cases, the related assignments are split across basic block boundaries.

Ultimately, there are two ways to get reliable parallel assignment. First, we can have the programmer explicitly identify assignments that should be treated as parallel assignments. Such cases are rare enough in practice that the additional annotation burden is relatively small. In addition, parallel assignments are frequently introduced by the automatic dependency transformation, which is discussed later in this chapter.

2.4.2 Structures

We now extend our presentation to allow mutable C-like structures as a natural extension of our dependent types for local variables. We allow field types to depend on other fields of the same structure, which enables us to express common idioms such as a structure containing a pointer to an array along with its length.

To add structures to our language, we extend our language with several new syn-

(TYPE STRUCT)

$$\frac{\text{for all } 1 \le i \le n, \quad (f_1 : \tau_1,\ \ldots\ f_n : \tau_n) \vdash_L \tau_i :: \text{type}}{\Gamma \vdash_L \text{struct}\ \{f_1 : \tau_1;\ \ldots\ f_n : \tau_n\} :: \text{type}}$$

(STRUCT LITERAL)

$$\frac{\text{for all } 1 \le i \le n, \quad \Gamma \vdash e_i : \tau_i \left[{}^{e_j}\!/_{f_j} \right]_{1 \le j \le n} \Rightarrow \gamma_i \qquad \gamma = \bigwedge_{1 \le j \le n} \gamma_i}{\Gamma \vdash \{f_1 = e_1; \ldots; f_n = e_n\} : \text{struct}\ \{f_1 : \tau_1;\ \ldots\ f_n : \tau_n\} \Rightarrow \gamma}$$

(STRUCT READ)

$$\frac{\Gamma \vdash \ell : \text{struct}\ \{f_1 : \tau_1;\ \ldots\ f_n : \tau_n\} \Rightarrow \gamma_\ell}{\Gamma \vdash \ell.f_i : \tau_i \left[{}^{\ell.f_j}\!/_{f_j} \right]_{1 \le j \le n} \Rightarrow \gamma_\ell}$$

(STRUCT WRITE)

$$\frac{\begin{array}{c}\Gamma \vdash \ell : \text{struct}\ \{f_1 : \tau_1;\ \ldots\ f_n : \tau_n\} \Rightarrow \gamma_\ell \\ \text{for all } 1 \le j \le n, \quad \Gamma \vdash \rho(f_j) : \rho(\tau_j) \Rightarrow \gamma_j \qquad \text{where } \rho(e') = e' \left[{}^{e}\!/_{f_i}, {}^{\ell.f_j}\!/_{f_j} \right]_{1 \le j \le n, j \ne i}\end{array}}{\Gamma \vdash \ell.f_i := e \ \Rightarrow\ \text{assert}(\gamma_\ell \wedge \textstyle\bigwedge_{1 \le j \le n} \gamma_j);\ \ell.f_i := e}$$

Figure 2.7: Structure type checking rules.

tactic constructs:

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \ldots \mid \text{struct}\ \{f_1 : \tau_1;\ \ldots\ f_n : \tau_n\} \\ \text{L-expressions} & \ell & ::= & \ldots \mid \ell.f \\ \text{Expressions} & e & ::= & \ldots \mid \{f_1 = e_1; \ldots; f_n = e_n\} \end{array}$$

The first addition is a new type, "struct $\{f_1 : \tau_1;\ \ldots\ f_n : \tau_n\}$", which defines a mutable record type in which the i^{th} field has label f_i and type τ_i. We also add an l-expression that allows the programmer to select a single field of this structure, and we add a structure literal expression that initializes field f_i to expression e_i.

Structures are intended to introduce a new context for naming dependencies. Pre-

viously, all identifiers found within types referred to other local variables, but here, identifiers refer to other fields of the same structure. For example, we could declare a structure with two fields such that field buf is an array whose length is one greater than the value in field len:

$$y \ : \ \text{struct} \ \{buf : \text{array int} \ (len + 1); \ len : \text{int}\}$$

In our base type system, we ensured that the "ref" constructor could never be applied to a type that had dependencies on external data. It is important to note that structure types can *never* depend on external data; any dependencies are restricted to fields within the structure itself. In fact, structures have the effect of introducing a new scope for naming dependencies between multiple state elements. Thus, it is legal to apply the "ref" constructor to a structure type whose fields depend on one another. This feature is actually quite important, because such structures are quite common in C programs. The `msghdr` example from Chapter 1, as well as the `request` example from Section 2.4.1, provide examples of pointers to structures that contain internal dependencies.

Figure 2.7 shows the rules for type checking structures. The (TYPE STRUCT) rule ensures that field types depend only on other fields in the *same* structure; note that in the premises for this rule, the usual Γ is replaced by an environment consisting of all the field names. The (STRUCT LITERAL) rule checks a literal to ensure that it has the appropriate type. Note that in the premises, we must substitute identifiers that refer to fields with the appropriate initializer expressions.

The (STRUCT READ) rule substitutes these field names with the appropriate expressions in a similar manner; by prepending the structure's l-value to all field names, we

generate the full expression for referring to each field. For example, using the declaration above, a read from $y.buf$ would have type "array int $(y.len + 1)$".

Finally, the (STRUCT WRITE) rule is analogous to the (VAR WRITE) rule; when a field is changed, we check all of the other fields in the current environment to make sure that any dependencies are satisfied. For example, if we wish to perform the field write "$y.len := n$", we would have the following premises:

$$\Gamma \vdash buf[{}^{n}\!/_{len}, {}^{y.buf}\!/_{buf}] : (\text{array int } (len + 1))[{}^{n}\!/_{len}, {}^{y.buf}\!/_{buf}] \Rightarrow \gamma_{buf}$$
$$\Gamma \vdash len[{}^{n}\!/_{len}, {}^{y.buf}\!/_{buf}] : \text{int}[{}^{n}\!/_{len}, {}^{y.buf}\!/_{buf}] \Rightarrow \gamma_{len}$$

After substitution, these premises become:

$$\Gamma \vdash y.buf : \text{array int } (n + 1) \Rightarrow \gamma_{buf}$$
$$\Gamma \vdash n : \text{int} \Rightarrow \gamma_{len}$$

The first premise can be derived using (ARRAY COERCE) and (STRUCT READ), with $\gamma_{buf} = 0 \leq n + 1 \leq y.len + 1$, which simplifies to $\gamma_{buf} = -1 \leq n \leq y.len$, which just says that the new length value must be shorter than the old one, but not so short as to make the total length negative. The second premise is trivial using (VAR).

Finally, note that we could also design a parallel version of this rule for parallel assignment to multiple fields of the same structure.

2.4.3 Function Calls

Finally, we add support for function calls to our language. To do so, we add four new syntactic constructs:

$$\begin{array}{llll} \text{Types} & \tau & ::= & \ldots \mid (x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau \\ \text{Commands} & c & ::= & \ldots \mid \text{let } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = c_{body} \text{ in } c \mid \\ & & & \text{let } x = e_f(e_1, \ldots, e_n) \text{ in } c \mid \text{return } e \end{array}$$

(FUN DEF)

$$\frac{\Gamma' = (x_1 : \tau_1, \ldots, x_n : \tau_n) \qquad \Gamma' \vdash_L \tau :: \text{type} \qquad \text{for all } 1 \leq i \leq n, \quad \Gamma' \vdash_L \tau_i :: \text{type}}{\begin{array}{c}\Gamma'; \tau \vdash c_{body} \Rightarrow c'_{body} \qquad x_i \text{ are not modified in } c_{body} \\ f \notin \text{Dom}(\Gamma) \qquad \Gamma, f : (x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau; \tau_{ret} \vdash c \Rightarrow c' \\ \hline \Gamma; \tau_{ret} \vdash \text{let } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = c_{body} \text{ in } c \quad \Rightarrow \\ \text{let } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = c'_{body} \text{ in } c'\end{array}}$$

(FUN CALL)

$$\frac{\begin{array}{c}\Gamma \vdash e_f : (x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau \Rightarrow \gamma \\ \text{for all } 1 \leq i \leq n, \quad \Gamma \vdash e_i : \tau_i[{}^{e_j}\!/_{x_j}]_{1 \leq j \leq n} \Rightarrow \gamma_i \\ \tau' = \tau[{}^{e_i}\!/_{x_i}]_{1 \leq i \leq n} \qquad x \notin \text{Dom}(\Gamma) \qquad \Gamma, x : \tau'; \tau_{ret} \vdash c \Rightarrow c'\end{array}}{\begin{array}{l}\Gamma; \tau_{ret} \vdash \text{let } x = e_f(e_1, \ldots, e_n) \text{ in } c \quad \Rightarrow \\ \quad \text{assert}(\gamma \wedge \bigwedge_{1 \leq i \leq n} \gamma_i);\ \text{let } x = e_f(e_1, \ldots, e_n) \text{ in } c'\end{array}}$$

(FUN RETURN)

$$\frac{\Gamma \vdash e : \tau_{ret} \Rightarrow \gamma}{\Gamma; \tau_{ret} \vdash \text{return } e \Rightarrow \text{assert}(\gamma);\ \text{return } e}$$

Figure 2.8: Function type checking rules.

The first addition is a new type, $(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau$, describing a function that takes n arguments x_i of type τ_i, returning a value of type τ. In this type, we allow the τ_i and τ to depend upon the x_i. We also add three new commands, one for defining functions, one for calling them, and one for returning from them.

In Figure 2.8, we give the typing rules for checking function definitions, calls, and returns. We update our command judgment to take the form $\Gamma; \tau_{ret} \vdash c \Rightarrow c'$, where τ_{ret} indicates the type of the current return value. All existing rules can be updated to simply

pass this type through to subcommands; it is only used and updated in the rules introduced here.

In (FUN DEF), we give the rule for defining a function. We check well-formedness of types as usual, we check the body of the function in the new environment, and we add the new function f to the environment of the subcommand c. The only unusual part of this rule is that we restrict the x_i from being modified in c_{body}, which makes sure that the return type τ will reflect the values at function entry when the function return is checked.

Next, (FUN CALL) checks a function call. Much like structure literals, we must check the type of each argument by substituting the expressions for the corresponding variable in each type. And of course, (FUN RETURN) checks the returned expression against the return type. Note that because the variables on which τ_{ret} depends cannot be modified, this check corresponds directly to the type of the return value seen by the caller.

Consider the following example:

let $copy(a : \text{array int } n, n : \text{int}) : \text{array int } n =$
    let $b = \text{new int}[n](0)$ in
    ... // copy a to b
    return b
in
let $x = \text{new int}[10](0)$ in
let $y = copy(x, 5)$ in
...

In this example, we create a function *copy* that creates a copy of an array a of length n. (This function creates an array using an extension of the "new" syntax that we have not formally defined.) Note that in the function definition, the first argument and the return type depend upon n, which is allowed by the premises in (FUN DEF) that check

the well-formedness of these types. Within the body of this function, "array int n" will be the return type, so in the "return" statement, the expression b will be checked against this type. Since n cannot be modified in the body of this function, the "return" statement is guaranteed to be checking the return value against the value of n that was originally passed to this function.

After definition this function, we create an array x of length 10 and then copy the first 5 elements. When calling the function, the first argument will generate the premise $\Gamma \vdash x : (\text{array int } n)[5/n] \Rightarrow \gamma_x$. Using the (ARRAY COERCE) rule, we see that this premise is derivable with $\gamma_x = 0 \leq 5 \leq 10$, which will be eliminated later by the optimizer. Finally, note that the return type of this function call is "array int 5", with the argument 5 substituted for n. Note once again that we must prevent n from being modified in the body of the function in order to make sure this assumption matches the check performed at the "return" statement.

Generally speaking, such dependent function calls are extremely useful in real code. For example, it is quite common to see functions that take a pointer to an array and the length of that array as parameters, as in the following declaration from Linux's socket filter code (`include/linux/filter.h`):

```
int sk_run_filter(struct sk_buff *skb,
                  struct sock_filter * COUNT(flen) filter,
                  int flen);
```

This syntax is particularly useful because there is no ambiguity about the order in which arguments are assigned to formal parameters. The issues that required us to introduce the (PARALLEL VAR WRITE) rule simply do not arise here.

2.5 Automatic Dependencies

Until now, we have presented our type checker under the assumption that all dependent types were fully specified. To reduce the programmer burden, our type system includes a feature called *automatic dependencies*, which automatically adds missing dependencies of local variables using fresh variables. As described in Section 2.2, this feature operates as a preprocessing step before type checking, and the output is guaranteed to type check properly; that is, automatic dependencies will not introduce any new type errors.

To implement this feature, we allow local variables to omit expressions in their dependent types. That is, local variables are allowed to have incomplete types as long as those incomplete types take only expression arguments, not type arguments. For example, a variable x might be declared to have type "array int", where the length of the array is unspecified. For every missing expression in a dependent type of a local variable, we introduce a new local variable that is updated along with the original variable. For example, we might introduce a new variable e_{len} that tracks the length of x, and we would change the type of x to "array int e_{len}". As a second example, recall that in Figure 2.2, we used this mechanism to automatically add `tmplen` to track the length of `tmp`, updating it as appropriate.

Formally, we maintain a mapping Δ from variables to the list of new variables that were added to track their dependencies. If a variable x had a complete type in the original program, $\Delta(x)$ is the empty list. We describe the automatic dependency inference as a judgment $\Gamma; \Delta \vdash c \rightsquigarrow c'$, which says that in the context $\Gamma; \Delta$, the command c can be transformed into command c' such that all types in c' are complete and such that c'

(AUTO LET)

$$\frac{\Gamma \vdash_L \tau :: \tau_1 \to \ldots \to \tau_n \to \text{type} \qquad \Gamma \vdash e : \tau\ e_1 \ldots e_n \Rightarrow \gamma \qquad \tau' = \tau\ x_1 \ldots x_n \qquad x_1, \ldots, x_n \text{ fresh} \qquad (\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n, x : \tau'); (\Delta, x \mapsto (x_1, \ldots, x_n)) \vdash c \rightsquigarrow c'}{\Gamma; \Delta \vdash \text{let } x : \tau = e \text{ in } c \rightsquigarrow \text{let } x_1 : \tau_1 = e_1 \text{ in } \ldots \text{let } x_n : \tau_n = e_n \text{ in let } x : \tau' = e \text{ in } c'}$$

(AUTO VAR WRITE)

$$\frac{\Gamma(x) = \tau\ x_1 \ldots x_n \qquad \Delta(x) = (x_1, \ldots, x_n) \qquad \Gamma \vdash e : \tau\ e_1 \ldots e_n \Rightarrow \gamma}{\Gamma; \Delta \vdash x := e \ \rightsquigarrow\ x,\ x_1, \ldots, x_n := e,\ e_1, \ldots, e_n}$$

Figure 2.9: Rules for automatic dependencies.

computes the same result as c.

The interesting rules for deriving this judgment are given in Figure 2.9. In the (AUTO LET) rule, we add new variables x_i to track any missing dependencies for x. In order to determine the correct values to assign to these variables, we use the type checking judgment to determine the type of e. (Note that we ignore the condition γ generated by this rule, since it is irrelevant to our current task; however, it will be checked appropriately during the type checking phase.) We maintain the invariant that all types in Γ are complete; thus, if the variable x is used to initialize another variable y with incomplete type, the type checking judgment will return the new variables x_i as the metadata for x.

In the (AUTO VAR WRITE) rule, we update all of the automatic variables associated with x using a similar approach. Once again, we use the type checking judgment to determine the expressions that should be used to update our fresh variables, and then we

use the parallel assignment operation to update the variable and its metadata in tandem. Note that because this transformation is performed by Deputy, we are free to make use of this parallel assignment construct!

To see how this technique works in practice, consider the following example:

$$
\begin{aligned}
&\text{let } a_1 : \text{array int } n_1 = \ldots \text{ in} \\
&\text{let } a_2 : \text{array int } n_2 = \ldots \text{ in} \\
&\text{let } x : \text{array int} = a_1 \text{ in} \\
&\text{if } (\ldots) \text{ then } x := a_2; \\
&*(x + 3) := 0;
\end{aligned}
$$

Note that we have not specified a bound for the array x. Depending on which way the conditional goes, x could be equal to a_1 or a_2 when the final statement is executed. Thus, it might be difficult (or even impossible) for the compiler to choose n_1 or n_2 as the length of the array. With automatic dependencies, the output is as follows:

$$
\begin{aligned}
&\text{let } a_1 : \text{array int } n_1 = \ldots \text{ in} \\
&\text{let } a_2 : \text{array int } n_2 = \ldots \text{ in} \\
&\underline{\text{let } n_x : \text{int} = n_1 \text{ in}} \\
&\text{let } x : \text{array int } \underline{n_x} = a_1 \text{ in} \\
&\text{if } (\ldots) \text{ then } x,\ \underline{n_x} := a_2,\ \underline{n_2}; \\
&*(x + 3) := 0;
\end{aligned}
$$

The newly-introduced code has been underlined in this example. We have introduced a new variable, n_x, which holds the length of x regardless of which way the conditional goes, and thus x will always have the appropriate bounds information.

By making this change, we have introduced some additional overhead, both in terms of memory and execution time. However, the optimizer can eliminate this overhead in cases where the outcome is statically determined. For example, if the condition will

always be true, then x will always be equal to a_2, and n_x will always be equal to n_2. By using standard optimization techniques such as copy propagation and dead code elimination, the optimizer can replace all uses of n_x with n_2 and get rid of n_x entirely. In other words, there is no performance penalty in cases where the bounds of x are statically computable.

In a sense, this technique takes the same approach as the rest of Deputy. First, we have deferred hard problems to run time where necessary in order to keep the compiler simple. Second, we use a flow-sensitive optimizer to encapsulate all of the available static reasoning. If the optimizer is capable of improving performance with static analysis, it will; however, when static analysis fails, we can fall back on dynamic analysis.

Also, this transformation recovers some of the flow-sensitivity that is absent in the core type system. In many cases, it is difficult to annotate a variable with a single dependent type that is valid throughout a function. By adding fresh variables that are automatically updated with the appropriate values, we provide the programmer with a form of flow-sensitive dependent type. We have found that this feature is critical when updating large amounts of preexisting C code. And as with the optimizer, we have found that separating this feature from the core type system simplifies both the implementation and the user's view of the type system.

For example, consider the following code:

$$\begin{aligned} &\text{let } a : \text{array int } 10 = \ldots \text{ in} \\ &a := a + 1 \end{aligned}$$

In this example, we would like to increment an array of 10 elements. Unfortunately, $a + 1$ has type "array int 9", which means that the assignment back to a will fail the check in (VAR WRITE), either at compile time or run time. This limitation frequently appears in

real code where function parameters are annotated using `COUNT` and then incremented in the body of the function.

We can use automatic dependencies to address this problem by copying this variable into a variable with incomplete type:

$$\begin{aligned}&\text{let } a : \text{array int } 10 = \ldots \text{ in}\\&\text{let } a' : \text{array int} = a \text{ in}\\&a' := a' + 1\end{aligned}$$

After processing this code with the automatic bounds transformation, we will get a fresh bounds variable that will track the bounds of this array automatically:

$$\begin{aligned}&\text{let } a : \text{array int } 10 = \ldots \text{ in}\\&\underline{\text{let } n' : \text{int} = 10 \text{ in}}\\&\text{let } a' : \text{array int } \underline{n'} = a \text{ in}\\&a', \underline{n'} := a' + 1, \underline{n' - 1}\end{aligned}$$

Now, when we update a', we get a new variable n' that tracks the length of the array. Because n' is updated in parallel with a', the bounds of a' are always correctly maintained. Thus, even though the original annotation for a was hampered by the flow-insensitivity of the base type system, the automatic dependency transformation can restore some of the flow-sensitivity required to handle real C code.

Finally, it is important to note that this technique is completely independent of the actual dependent types in use. Although it is frequently used to propagate bounds information in cases where the programmer does not want to manually provide bounds information for every variable, it can be used for any dependent type where metadata information is not already available.

2.6 Limitations

The dependent type framework presented in this chapter currently has several major limitations, including flow-insensitivity, local expressions, and "out" parameters.

Flow-insensitivity limits the power of the type system because two statements that temporarily violate an invariant and then immediately repair it will trigger an error, often at run time. As discussed in Section 2.4.1, this issue can be mitigated by using parallel assignment; however, even with parallel assignment, we often require the programmer's help to eliminate such errors. In addition, Section 2.5 shows how we can recover some flow-sensitivity for local variables using automatic dependencies; however, this approach does not extend to structures of a field without changing the layout of that data structure. Ultimately, we have found that many of the spurious errors (i.e., incompletenesses) that we have discovered with the current system require a manual change to the code being ported.

A second limitation is the use of local expressions. Although many dependencies can be annotated correctly using local expressions, there are a number of dependencies that cannot be directly expressed in this way. For example, the sample Linux driver `scull` contains structures similar to the following:

```
struct scull_qset {
    char * COUNT(?)  * data;
    struct scull_qset * next;
};
struct scull_dev {
    struct scull_qset * data;
    int quantum, qset;
};
```

In this case, the `quantum` and `qset` fields together indicate the size of the buffers pointed to by `data`, but they are stored in a parent structure. In order to use Deputy here, the programmer must rewrite the code or mark it as trusted. We believe that such rewrites are good practice even in the absence of a verifier such as Deputy; although the rewritten code may require more memory and may contain some redundancy, it will be significantly less likely to contain bugs.

One final example of code that is difficult to annotate in existing programs is functions that use "out" parameters. Such functions typically return data by having the caller pass a pointer to a location where the return data can be placed. If there are several pieces of return data, then it is difficult to annotate relationships between this return data because of the extra level of indirection. For example, Linux's security code (`include/linux/security.h`) has a declaration similar to the following:

```
int security_inode_init_security(struct inode *inode, struct inode *dir,
                                 char **name, void **value, size_t *len);
```

In this example, it is difficult to talk about the relationship between `value` and `len` because they are both returned as a parameter; in our current framework, the types of `*value` and `*len`, which are related, must type check in the empty environment.

Chapter 3

Dependent Types for C

Now that we have established a general framework for adding dependent types to imperative programs and for reasoning about them soundly, we must consider the specific dependent type constructors that are needed in order to annotate and check real C programs. This chapter presents type constructors that handle pointer bounds, null-terminated pointers, and tagged unions.

In addition, we present important details about the inference and optimization components for each of these type constructors. Since inference and optimization depend heavily on the meaning and use of each type constructor, it is important to tailor these components to the specific type constructors that we plan to use.

3.1 Pointer Bounds

The first and most important type that we will introduce is a bounded pointer type. Currently, our language only contains the "ref" constructor, which represents a non-null

pointer to a single object of the base type. In order to accommodate the true range of pointer usage in C programs, we introduce a new, more powerful dependent type constructor. This constructor is intended to represent pointers in their full generality. Note that we do not (and will not) provide a non-dependent pointer type constructor; all of these cases will be special cases of the pointer type we present here.

3.1.1 Definition

Our type constructor for bounded pointers is a generalization of the array constructor presented earlier. This new type, written "ptr τ *lo hi*", represents a possibly-null pointer to an array of elements of type τ, where *lo* and *hi* are expressions that indicate the bounds of this array. Specifically, *lo* is the address of the first accessible element of the array, and *hi* is the address of the first inaccessible element after the end of the area. Figure 3.1 illustrates this invariant; note that the pointer p may point anywhere between *lo* and *hi*, but all pointers must be aligned with respect to the size of τ. The kind of "ptr" is "type $\rightarrow$ int $\rightarrow$ int $\rightarrow$ type".

In addition to the new "ptr" constructor, we add to the language an operator $\oplus$ for C-style pointer arithmetic, which moves a pointer forwards or backwards by a certain number of elements rather than bytes. The $\oplus$ operator may be used in local expressions.

Finally, we add typing rules for all relevant operations on this type, including reads and writes, pointer arithmetic, dereference, and coercions. These rules will be presented in detail below.

Examples of the "ptr" type are as follows:

x : ptr int b $(b \oplus 8)$ // 8 integer area starting at b
x : ptr int x $(x \oplus n)$ // n integer area starting at x
x : ptr int x e // from x to e

Note that this pointer type is very general and can accommodate a wide variety of schemes for indicating the bounds of an array. The programmer is free to use additional pointers to designate bounds, or they can specify bounds relative to the pointer itself. Note that in the second example, both bounds for x are relative to x itself; in fact, this pointer type is very similar to the "array int n" type discussed in the previous chapter. (Our implementation uses the more general version, along with some syntactic sugar.)

In fact, self-dependencies are one of the most important features for designing a flexible pointer type. Without the ability for bounds to refer to the pointer itself, it would be very difficult to design a single bounded pointer type that encompasses so many different idioms. This feature is made possible by our handling of mutation (i.e., the (VAR WRITE) rule), and it represents perhaps the most common use of that rule.

Formally, we define the meaning of the bounded pointer type constructor as follows:

Definition 1 (Pointer types) *A value v has type "ptr τ lo hi" for some values lo and hi if either $v = 0$ or the following conditions hold:*

- $lo \leq v \leq hi$, *and*
- *there exists an allocated area of memory containing elements of type τ whose first byte is lo' and whose last byte is $hi' - 1$ such that $lo' \leq lo$ and $hi \leq hi'$, and*
- *v, lo, and hi, are correctly aligned relative to the elements in the allocation area*

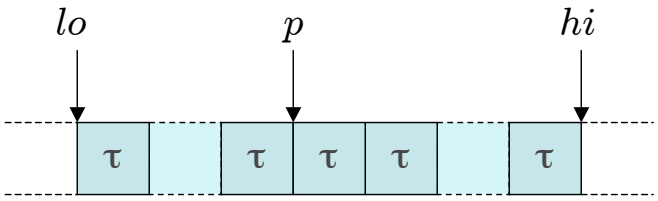

Figure 3.1: Memory layout for a pointer p of type "ptr τ *lo hi*".

described above (i.e., $(lo - lo') \bmod |\tau| = 0$*,* $(v - lo') \bmod |\tau| = 0$*, and* $(hi - lo') \bmod |\tau| = 0$*, where* $|\tau|$ *is the size of a* τ *object in bytes).*

This definition allows us to conclude that if a value v has type "ptr τ *lo hi*", $v \neq 0$, and $v \neq hi$, then v points to the start of a valid τ object. As in ANSI C, we allow a pointer to be equal to the upper bound of its buffer even though such pointers must not be dereferenced; support for such pointers is required to handle several common programming idioms.

3.1.2 Type Rules

Figure 3.2 shows the typing rules for the "ptr" constructor and its associated operations.

The first two rules, (LOCAL PTR ARITH) and (LOCAL INT COERCION), show that pointer arithmetic and integer coercion are allowed in local expressions. The former is important for many common idioms, as seen in the example types presented earlier. The latter is useful because the kind of the "ptr" constructor requires (for simplicity) that the bounds be specified as integers.

(LOCAL PTR ARITH)

$$\frac{\Gamma \vdash_L e_1 : \text{ptr } \tau \; lo \; hi \qquad \Gamma \vdash_L e_2 : \text{int}}{\Gamma \vdash_L e_1 \oplus e_2 : \text{ptr } \tau \; lo \; hi}$$

(LOCAL INT COERCION)

$$\frac{\Gamma \vdash_L e : \text{ptr } \tau \; lo \; hi}{\Gamma \vdash_L e : \text{int}}$$

(PTR ARITH)

$$\frac{\Gamma \vdash e_1 : \text{ptr } \tau \; lo \; hi \Rightarrow \gamma_1 \qquad \Gamma \vdash e_2 : \text{int} \Rightarrow \gamma_2}{\Gamma \vdash e_1 \oplus e_2 : \text{ptr } \tau \; lo \; hi \Rightarrow \gamma_1 \wedge \gamma_2 \wedge (e_1 \neq 0) \wedge (lo \leq (e_1 \oplus e_2) \leq hi)}$$

(PTR DEREF)

$$\frac{\Gamma \vdash e : \text{ptr } \tau \; lo \; hi \Rightarrow \gamma}{\Gamma \vdash *e : \tau \Rightarrow \gamma \wedge (e \neq hi) \wedge (e \neq 0)}$$

(PTR WRITE)

$$\frac{\Gamma \vdash e_1 : \text{ptr } \tau \; lo \; hi \Rightarrow \gamma_1 \qquad \Gamma \vdash e_2 : \tau \Rightarrow \gamma_2}{\Gamma \vdash *e_1 := e_2 \Rightarrow \text{assert}(\gamma_1 \wedge \gamma_2 \wedge (e_1 \neq hi) \wedge (e_1 \neq 0)); *e_1 := e_2}$$

(PTR ALLOC)

$$\frac{\begin{array}{c} x \notin \text{Dom}(\Gamma) \qquad \emptyset \vdash_L \tau :: \text{type} \qquad \Gamma \vdash e_{init} : \tau \Rightarrow \gamma_{init} \\ \Gamma \vdash_L e_{len} : \text{int} \qquad \Gamma, x : \text{ptr } \tau \; x \; (x \oplus e_{len}) \vdash c \Rightarrow c' \end{array}}{\begin{array}{l} \Gamma \vdash \text{let } x = \text{new } \tau[e_{len}](e_{init}) \text{ in } c \;\; \Rightarrow \\ \qquad \text{assert}(\gamma_{init} \wedge (e_{len} > 0)); \text{let } x = \text{new } \tau[e_{len}](e_{init}) \text{ in } c' \end{array}}$$

Figure 3.2: Type rules for bounded pointer operations.

$$\frac{\Gamma \vdash e : \text{ptr } \tau \; lo \; hi \Rightarrow \gamma}{\Gamma \vdash e : \text{int} \Rightarrow \gamma} \quad \text{(INT COERCION)}$$

$$\frac{\Gamma \vdash e : \text{ptr } \tau \; lo \; hi \Rightarrow \gamma}{\Gamma \vdash e : \text{ptr } \tau \; lo' \; hi' \Rightarrow \gamma \wedge (e = 0)} \quad \text{(NULL COERCION)}$$

$$\frac{\Gamma \vdash e : \text{ptr } \tau \; lo \; hi \Rightarrow \gamma}{\begin{array}{l}\Gamma \vdash e : \text{ptr } \tau \; lo' \; hi' \quad \Rightarrow \\ \quad \gamma \wedge (lo \leq lo' \leq e \leq hi' \leq hi) \\ \quad\quad \wedge ((lo' - lo) \bmod |\tau| = 0) \wedge ((hi - hi') \bmod |\tau| = 0)\end{array}} \quad \text{(PTR COERCION)}$$

Figure 3.3: Type rules for bounded pointer coercions.

The (PTR ARITH) rule shows the checks required for pointer arithmetic. Since the new pointer has the same bounds as the original pointer, the checks simply verify that the new pointer is still within bounds. Also, we ensure that we are not incrementing a null pointer, since the result would violate our invariant.

The (PTR DEREF) rule verifies a pointer dereference by checking that a pointer is non-null and that it does not point to the end of the region. As argued above, these two conditions suffice to ensure that the pointer points to a valid object of type τ. The (PTR WRITE) rule has the same requirements and adds the same run-time checks to verify that it is safe to update the value to which e_1 points.

The (PTR ALLOC) rule shows how allocation is checked. The only run-time check required is to verify that we are allocating at least one element; however, we must also verify that the base type τ type checks in the empty environment, which ensures that it has no external dependencies.

Figure 3.3 shows the typing rules for coercions to and from our bounded pointer

type. The (INT COERCION), (NULL COERCION), and (PTR COERCION) rules allow coercions to and from pointer types. The first rule, (INT COERCION), is trivial: any pointer can be safely considered an integer. The second rule, (NULL COERCION), allows conversion between different pointer bounds if the pointer is null, since a null pointer satisfies its invariant regardless of the bounds. Note that the null check is performed at run time, which is far more flexible than a static check for a null pointer (and may still be eliminated by the optimizer). Also note that a similar rule could be added for converting zero integers to a pointer type with arbitrary bounds. Finally, the (PTR COERCION) rule allows coercion between different bounds as long as the new bounds are contained within the old bounds and are still aligned appropriately.

Notice that there are two ways to coerce a pointer value to a pointer type with different bounds: (NULL COERCION) or (PTR COERCION). In general, there is no way to decide statically which rule to apply, so our implementation combines them into a more general rule that says you can convert e from one pointer type to another if either $e = 0$ or the conditions in the (PTR COERCION) rule hold.

To see how these rules work in practice, consider the following example:

$$
\begin{array}{l}
\text{let } s : \{\ buf : \text{ptr int } buf\ (buf \oplus len);\ len : \text{int}\ \} = \\
\quad \{\ buf = 0;\ len = 0\ \} \\
\text{in} \\
\text{let } n : \text{int} = 10 \text{ in} \\
\text{let } p = \text{new int}[n](0) \text{ in} \\
s.len := n; \\
s.buf := p; \\
*(s.buf \oplus 5) := 42
\end{array}
$$

When allocating the pointer p, we use the (PTR ALLOC) rule, whose premises are

satisfied trivially. This rule will insert the assertion $n > 0$ immediately before the allocation; however, this assertion will be eliminated by the optimizer, which can see that $n = 10$.

In the next line, we assign n to $s.len$. Following the (STRUCT WRITE) rule, we perform the appropriate substitutions on every field of the structure, resulting in one nontrivial premise: $\Gamma \vdash buf$: ptr int buf $(buf \oplus n) \Rightarrow \gamma_{buf}$. This premise can be derived using either (PTR COERCION) or (NULL COERCION). As described above, we cannot choose statically between these options, so we emit the disjunction of the two assertions: $(s.buf = 0) \vee (buf \leq buf \leq buf \leq (buf \oplus n) \leq (buf \oplus len))$ (alignment checks omitted). Of course, the null check will suffice at run time. Note that because $s.buf = 0$, we can assign whatever value we like to $s.len$; for this reason, it is often convenient to assign bounds before assigning the pointer.

Next, we assign p to $s.buf$. Once again, we use the (STRUCT WRITE) rule, which yields the premise $\Gamma \vdash p$: ptr int p $(p \oplus s.len) \Rightarrow \gamma_{buf}$. Once again, we use the coercion rules, which yields the assertion $p = 0 \vee (p \leq p \leq p \leq (p \oplus s.len) \leq (p \oplus n))$. Since $p \neq 0$, we actually use the bounds check this time, and since $s.len = n$ (due to the previous assignment), these checks are satisfied.

Finally, we write to the fifth element of $s.buf$. To do so, we first compute $s.buf \oplus 5$, which uses the (PTR ARITH) rule; this rule yields the assertion $(s.buf \neq 0) \wedge (s.buf \leq (s.buf \oplus 5) \leq (s.buf \oplus s.len))$. We then apply the (PTR WRITE) rule, which yields the assertion $(s.buf \oplus 5) \neq (s.buf \oplus 10) \wedge (s.buf \oplus 5) \neq 0$, which can be optimized away based on the arithmetic checks. A reasonably sophisticated optimizer will also realize that $s.buf \neq 0$ and that $s.len = 10$, which allows all of these checks to be eliminated statically.

In our implementation, this pointer type is written as "`t * BOUND(lo, hi)`" for base type `t` and bounds `lo` and `hi`. The keyword `__this` can be used within bounds expressions to refer to the pointer itself, and the keyword `__auto` can be used in place of a bounds expression to request automatic dependencies. We provide shorthands such as `COUNT(n)`, which is equivalent to `BOUND(__this, __this + n)`, and `SAFE`, which is equivalent to `COUNT(1)`.

In our experience, `SAFE` is by far the most common pointer type, and is assumed on any pointers that are unannotated and that are not eligible for automatic dependencies. `COUNT(n)`, however, is the most common programmer-supplied annotation. That said, the full generality of this annotation is often needed. For example, the Linux `sk_buff` structure, defined in `include/linux/skbuff.h`, contains one structure field that was annotated as follows:

```
struct sk_buff { ...
    unsigned char * BOUND(head, end) head,
                  * BOUND(head, end) data,
                  * BOUND(head, end) tail,
                  * COUNT(sizeof(struct skb_shared_info)) end;
};
```

This declaration allows us to state that `data` and `tail` remain between `head` and `end`, and that there is additional space after `end` to store data from another structure.

Generally speaking, we have found that this dependent pointer type is flexible enough to express a wide variety of common C idioms for tracking pointer bounds. In Chapter 4, we will discuss our experience using this dependent type to annotate real-world code.

3.1.3 Inference

The type rules presented in the previous section expect all pointer types to be labeled with a base pointer and an end pointer, as if the programmer had already provided type annotations for every pointer in the program. However, inserting these annotations into an existing program would be both tedious and messy; therefore, we must provide some inference mechanism to simplify the process of converting existing code.

A key tool for this inference phase is the automatic dependency mechanism described in the previous chapter. Automatic dependencies allow us to insert fresh variables to track dependencies in the case of local variables. However, we must still deal with pointer types that are not associated with local variables (e.g., function parameters or base types of other pointer types), and for performance reasons, it is beneficial to reduce the number of cases in which automatic dependencies are introduced.

To this end, we use an inference mechanism based on CCured's pointer graph [NCH+05].[1] CCured's pointer graph assigns a unique identifier to every syntactic pointer in the program, which includes every "$*$" in the program text as well as every variable whose address could potentially be taken. The edges in this graph indicate several possible relationships: *points-to* edges indicate that one node may point to another, *cast* edges indicate that a cast assignment moves data from one node to another at some point in the program, and *compat* edges indicate that two nodes must have identical types. Note that *points-to* edges and *cast* edges are directed and are derived directly from the program text, whereas *compat* edges are undirected, and they arise between the *points-to* targets of any

[1] Many thanks to Matt Harren for his work on this feature.

```
match node flags with
    case no arithmetic ⟹ ptr τ p (p ⊕ 1)
    case forward arithmetic only ⟹
        if top-level type of local variable then
            ptr τ p
        else
            warn user; ptr τ p (p ⊕ 1)
    case forward and backward arithmetic ⟹
        if top-level type of local variable then
            ptr τ
        else
            warn user; ptr τ p (p ⊕ 1)
```

Figure 3.4: Algorithm for inferring types based on CCured inference results.

nodes that have a *cast* edge between them.

Each node in this graph has flags indicating whether a given node holds data that may flow to an arithmetic operation at some point in the future. CCured provides a solver that sets these flags at the relevant arithmetic operations and then flows them across *cast* and *compat* edges as relevant.

Deputy generates the pointer graph and runs the solver immediately before automatic dependencies are processed. For pointer p with base type τ and no programmer-supplied annotations, the inferred type is determined using the algorithm in Figure 3.4.

The first case in this algorithm uses a `SAFE` pointer wherever possible. The second and third cases may return an incomplete type, which will prompt the automatic dependency phase to insert a suitable fresh variable for the missing bounds expressions. Thus, the CCured-based inference phase guides the automatic dependency phase by indicating where

automatic dependencies are necessary and where they can be omitted entirely. Note also that we warn the user when arithmetic is applied to an unannotated non-local variable (e.g., a function parameter or a global), which typically indicates that an annotation is missing.

For example, consider the following code, which determines whether an array of zeros contains an integer:

```
int findzero(int * buf, int * end) {
    int * cur = buf;
    while (cur < end) {
        if (*cur == 0) return 1;
        cur++;
    }
    return 0;
}
```

There are three syntactic pointers in this example. Since forward pointer arithmetic is applied to `cur` and since `cur` is a local variable, we assign it the type "ptr int *cur*", or `BOUND(cur, __auto)` in the concrete syntax. Since `cur`'s value comes from `buf`, `buf` also has forward arithmetic; however, since `buf` is not a local variable, we cannot use automatic bounds. Thus, we warn the user that bounds information is probably required, and we give `buf` the type "ptr int buf ($buf \oplus 1$)", or `SAFE`. Finally, since no pointer arithmetic is applied to `end`, it gets the type "ptr int end ($end \oplus 1$)", or `SAFE`.

Unfortunately, in this case the default type assigned to `end` is not quite correct. Since `end` points to the end of the array, there are probably no more elements after it, so it should ideally have the type "ptr int *end end*", or `COUNT(0)`. Callers will encounter errors if they attempt to pass a `COUNT(0)` pointer when a `COUNT(1)` pointer is expected. Thus, the programmer must explicitly provide the correct annotation for `end`.

3.1.4 Optimization

Because Deputy generates a large number of run-time checks, many of which are trivial or unnecessary, optimization is essential. In particular, it is important to provide an optimizer that is aware of the most common checks emitted by Deputy; since the type checker relies on the presence of an optimizer, it is not sufficient to rely on the back end compiler to eliminate unnecessary checks.

The most important optimization is a flow-insensitive optimization that looks for assertions for the form $e + c_1 \leq e + c_2$ for constants c_1 and c_2 and for comparators $\leq$, $\geq$, $=$, and $\neq$. Essentially, if we have two syntactically equal expressions e with different constant offsets, we can simply compare constants c_1 and c_2 to determine whether the assertion holds. (Note that expressions emitted by the CIL front-end are free from side-effects, so we may safely assume that two syntactically identical subexpressions evaluate to the same value.)

One reason why this optimization arises frequently is that the most common pointer type is "ptr τ p $(p \oplus 1)$", indicating a pointer to a single object of type τ. In this case, dereferencing the pointer yields the check $p \neq p \oplus 1$, which can be eliminated using the logic described above.

In addition to this important flow-insensitive optimization, Deputy's pointer bounds optimizer has several flow-sensitive optimizations.[2] First, there is a forward substitution pass that replaces variables with the expressions used to define them, wherever possible. This standard dataflow algorithm is quite useful because it pushes more information into the checks themselves, making the aforementioned flow-insensitive optimization more effective.

[2]Thanks to Zach Anderson for his work on Deputy's flow-sensitive optimizer.

Finally, Deputy provides a dataflow analysis that tracks which pointers are null or non-null, using information from conditionals. Based on this information, we can eliminate unnecessary null checks.

Further details on Deputy's flow-sensitive optimizations can be found in Zach Anderson's masters report [And07].

3.2 Null-Terminated Arrays

Our bounded pointer type covers many common cases involving pointer bounds in existing C programs. However, it does not cover the case of null-terminated pointers, which is another extremely common idiom in C programs.

Informally, a null-terminated pointer is one in which the upper bound of the memory area to which it points is indicated by the presence of a null element as opposed to an explicitly-tracked pointer to the end of the array. This idiom requires slightly different reasoning from the bounded pointers discussed previously. In particular, we must ensure that a pointer is never incremented beyond the null element, and we must ensure that the null element is never overwritten with a non-null element.

The main challenge in designing a usable null-terminated pointer type is that null-terminated pointers and bounded pointers must be able to interact in useful ways. Consider, for example, the following function:

```
int inputlen() {
    char buf[100];
    char *p = buf;
    buf[0] = 0;
    readline(p, 100);
    return strlen(p);
}
```

In this example, we need to decide what type to assign to the variable `p`. If we assign it the type "ptr char p ($p \oplus 100$)", or `char * COUNT(100)`, using the bounded pointer from the previous section, then we would have difficulty calling `strlen` on this pointer, since `strlen` requires a null-terminated pointer. However, if we were to declare that the pointer `p` is a null-terminated pointer (with no indication of the true extent of the buffer), then we would not be able to call `readline` on `p`, since `readline` requires a buffer of 100 elements, and `p`'s null terminator is at index 0. Thus, we cannot assign a reasonable type to `p` if we must choose between explicit bounds and null-termination; rather, we must be able to talk about both invariants simultaneously.

In order to design a type that incorporates both bounds information and null-termination, we must consider two important issues. First, we need to be able to cast this pointer to a "normal" bounded pointer for usability purposes, but we need to ensure that the null-terminator falls *outside* the "normal" pointer's bounds so that the null element cannot be overwritten. Second, we must allow null elements within the "known" bounds of a null-terminated pointer, in order to handle cases like the initialization of the first element in our example above.

Our solution to this problem is to extend the bounded pointer type from the

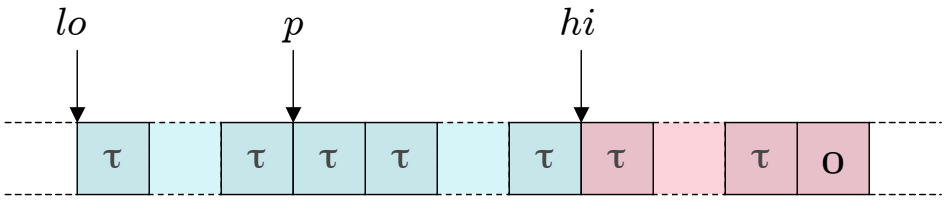

Figure 3.5: Memory layout for a pointer p of type "ntptr τ *lo hi*".

previous section. Informally, we say that a null-terminated pointer, written "ntptr", is the same as a bounded pointer except that it also contains a null-terminated sequence of elements starting at the upper bound. That is, the explicit bounds designate some known region of allocated objects, and the memory region extends beyond this area until a null element is reached. This new invariant is illustrated in Figure 3.5.

In the example above, we would assign `p` the type "ntptr char p $(p \oplus 99)$", which is written `NT COUNT(99)` in our concrete syntax. This type indicates that `p` points to a region with 99 unrestricted characters followed by a null-terminated sequence. Assuming that all stack buffers are initialized to zero on function entry, this null-terminated sequence will have length zero (i.e., it consists of a single null element). Now we can allow `readline` to verify the length of `p` despite the null element at index 0, and we can allow `strlen` to verify that the pointer is null-terminated.

This feature is particularly useful in the case of a function such as `strlcpy`, which might be defined as follows:

```
size_t strlcpy(char * NT COUNT(size - 1) dst,
               const char * NT COUNT(0) src,
               size_t size) {
    char * dst_cur = dst;
    char * src_cur = src;
    int i = 0;
    while (*src_cur && i < size - 1) {
        *dst_cur++ = *src_cur++;
        i++;
    }
    *dst_cur = 0;
    return i;
}
```

Note that our null-terminated pointers allow us to specify very precisely the invariant on `dst`. Specifically, it is a pointer to a null-terminated sequence of characters that contains at least `size - 1` characters, followed by a null-terminated sequence. Thus, callers may only pass buffers that meet the size requirements when calling this function, but we also preserve the invariant that these pointers are null-terminated. Similarly, `src` indicates that the size of the "known" region is zero, so it represents a sequence of zero or more characters followed by a null element.

In this example, we copy `dst` and `src` to local variables in order to take advantage of automatic dependencies; otherwise, we would not be able to increment `dst` itself.

3.2.1 Definition

Formally, our notion of null termination is defined as follows:

Definition 2 (Null-terminated pointer types) *A value v of type "ntptr τ lo hi" satisfies the requirements in Definition 1. In addition, if $v \neq 0$, then there exists an entirely zero element in the interval $[hi..hi')$, where hi' is the true end of the allocation area.*

Thus, a null-terminated array has two components: an initial segment with "known" bounds *lo* (inclusive) and *hi* (non-inclusive), and a variable-sized segment beginning at *hi* and ending with a null terminator.

One important special case of this pointer type (given a pointer variable named p) is the type "ntptr τ p p". Since the lower and upper bounds are identical, the "known" region is empty, which means that p is a null-terminated sequence of elements of type τ with no minimum length. Strings in C programs are often assigned the type "ntptr char p p", which can be written as `NT COUNT(0)` or the syntactic sugar `NTS`.

3.2.2 Type Rules

Figure 3.6 and Figure 3.7 show the type rules for this new pointer type. These rules are derived from the rules for the original bounded pointer type and thus share many similarities. However, there are a number of important differences to highlight.

First, note that the (NTPTR ARITH) has a slightly different bounds check. Instead of verifying that the new pointer is less than *hi*, we verify that it is less than $findzero(hi, |\tau|)$, where $findzero$ is a function that returns the address of the first null element found starting at address *hi*. (This function requires the size of the base type τ.) Thus, arithmetic can move past the upper bound if the null-termination allows it. Note that the upper bound of the derived pointer is extended in this case in order to preserve

(LOCAL NTPTR ARITH)

$$\frac{\Gamma \vdash_L e_1 : \text{ntptr } \tau \; lo \; hi \qquad \Gamma \vdash_L e_2 : \text{int}}{\Gamma \vdash_L e_1 \oplus e_2 : \text{ntptr } \tau \; lo \; hi}$$

(LOCAL NTPTR INT COERCION)

$$\frac{\Gamma \vdash_L e : \text{ntptr } \tau \; lo \; hi}{\Gamma \vdash_L e : \text{int}}$$

(NTPTR ARITH)

$$\frac{\Gamma \vdash e_1 : \text{ntptr } \tau \; lo \; hi \Rightarrow \gamma_1 \qquad \Gamma \vdash e_2 : \text{int} \Rightarrow \gamma_2}{\Gamma \vdash e_1 \oplus e_2 : \text{ntptr } \tau \; lo \; max(hi, e_1 \oplus e_2) \;\Rightarrow\; \gamma_1 \wedge \gamma_2 \wedge (e_1 \neq 0) \wedge (lo \leq (e_1 \oplus e_2) \leq findzero(hi, |\tau|))}$$

(NTPTR DEREF)

$$\frac{\Gamma \vdash e : \text{ntptr } \tau \; lo \; hi \Rightarrow \gamma}{\Gamma \vdash *e : \tau \Rightarrow \gamma \wedge (e \neq 0)}$$

(NTPTR WRITE)

$$\frac{\Gamma \vdash e_1 : \text{ntptr } \tau \; lo \; hi \Rightarrow \gamma_1 \qquad \Gamma \vdash e_2 : \tau \Rightarrow \gamma_2}{\Gamma \vdash *e_1 := e_2 \Rightarrow \text{assert}(\gamma_1 \wedge \gamma_2 \wedge ((e_1 \neq hi) \vee (e_2 = 0) \vee (*e_1 \neq 0)) \wedge (e_1 \neq 0)); *e_1 := e_2}$$

(NTPTR ALLOC)

$$\frac{x \notin \text{Dom}(\Gamma) \qquad \emptyset \vdash_L \tau :: \text{type} \qquad \Gamma \vdash e_{init} : \tau \Rightarrow \gamma_{init} \qquad \Gamma \vdash_L e_{len} : \text{int} \qquad \Gamma, x : \text{ntptr } \tau \; x \; (x \oplus (e_{len} - 1)) \vdash c \Rightarrow c'}{\Gamma \vdash \text{let } x = \text{new } \tau[e_{len}](e_{init}) \text{ in } c \;\Rightarrow\; \text{assert}(\gamma_{init} \wedge (e_{len} > 0)); \text{let } x = \text{new } \tau[e_{len}](e_{init}) \text{ in } c'}$$

Figure 3.6: Type rules for null-terminated pointer operations.

$$\frac{\Gamma \vdash e : \text{ntptr } \tau \; lo \; hi \Rightarrow \gamma}{\Gamma \vdash e : \text{int} \Rightarrow \gamma} \; (\textsc{ntptr int coercion}) \qquad \frac{\Gamma \vdash e : \text{ntptr } \tau \; lo \; hi \Rightarrow \gamma}{\Gamma \vdash e : \text{ntptr } \tau \; lo' \; hi' \Rightarrow \gamma \wedge (e = 0)} \; (\textsc{ntptr null coercion})$$

$$\frac{\Gamma \vdash e : \text{ntptr } \tau \; lo \; hi \Rightarrow \gamma}{\begin{array}{l}\Gamma \vdash e : \text{ntptr } \tau \; lo' \; hi' \quad \Rightarrow \\ \quad \gamma \wedge (lo \leq lo' \leq e \leq hi' \leq findzero(hi, |\tau|)) \\ \quad\quad \wedge ((lo' - lo) \bmod |\tau| = 0) \wedge ((hi - hi') \bmod |\tau| = 0)\end{array}} \; (\textsc{ntptr ntptr coercion})$$

$$\frac{\Gamma \vdash e : \text{ntptr } \tau \; lo \; hi \Rightarrow \gamma}{\Gamma \vdash e : \text{ptr } \tau \; lo \; hi \Rightarrow \gamma} \; (\textsc{ntptr ptr coercion})$$

Figure 3.7: Type rules for null-terminated pointer coercions.

the invariant that a pointer is always within its bounds.

The (NTPTR DEREF) and (NTPTR WRITE) rules are modified to take into account the new restrictions. The dereference restrictions are actually weaker that they were for bounded pointers; since there is at the very least a null element at the *hi* pointer, all dereferences of non-null null-terminated pointers are safe. The restrictions for pointer writes are also weakened somewhat; specifically, we allow writes to the *hi* pointer as long as we are not overwriting a null element with a non-null element. It is important to allow null terminators to be overwritten with new null terminators, since this behavior occurs in many C programs.

We also have a new allocation rule, (NTPTR ALLOC), which differs only in that the upper bound is reduced by one element. The final allocated element is left outside of the

"known" bounds and is assumed to be set to zero by the runtime system.

Finally, we have a number of coercions. The integer and null coercions are identical to the bounded pointer versions. The (NTPTR NTPTR COERCION) rule differs from the original only in that we use $findzero$ once again to extend the upper bound as necessary. Finally, the (NTPTR PTR COERCION) indicates that we can convert a null-terminated pointer to a normal bounded pointer with the identical bounds and base type; since the null element always falls outside the "known" bounds, it is impossible for this new alias to overwrite the null pointer. Note that the reverse coercion is disallowed; the only way to create a null-terminated pointer is to allocate one.

For example, consider the following code, which is derived from the earlier `inputlen` example:

$$
\begin{aligned}
&\text{let } buf = \text{new char}[100](0) \text{ in}\\
&*buf := 0;\\
&\text{let } m = readline(buf, 100) \text{ in}\\
&\text{let } n = strlen(buf) \text{ in}\\
&\ldots
\end{aligned}
$$

When we allocate buf, we use the (NTPTR ALLOC) rule, which assigns buf the type "ntptr char buf $(buf \oplus 99)$". When writing to buf, we invoke the (NTPTR WRITE) rule, and the checks emitted will verify that $buf \neq (buf \oplus 99)$, likely at compile time.

For the two function calls, we assume that the function $readline$ has type "$(a$: ptr char a $(a \oplus (n-1)), n$: int$) \rightarrow$ int" and that $strlen$ has type "$(s$: ntptr char s $s) \rightarrow$ int". When calling $readline$, note that we use the (NTPTR PTR COERCION) rule in order to convert to a non-null-terminated pointer; because $readline$ only requires a buffer of length $n-1$, or 99, this conversion is safe and will not allow the final zero to be overwritten (or even

accessed) by *readline*. For the second call, we use the (NTPTR NTPTR COERCION), which converts *buf* to type "ntptr char *buf buf*". Because the new upper bound is smaller than the old upper bound, this check can be performed without even searching for the null element in *buf*; most likely, all checks emitted by the coercion will be optimized away.

Our approach to handling null termination is a key contribution of our work. When designing CCured [NCH+05], we struggled with the issue of null termination; in particular, it was unclear whether null elements should fall inside or outside the pointer metadata. Deputy's approach solves this problem neatly by splitting the area into two regions: one with "known" bounds and no restrictions, and one with null-terminated data where care must be taken to avoid overwriting the null element with a non-null value. The key insight is that explicit bounds and null termination must be handled simultaneously, by a single type constructor, rather than separating the invariant into two distinct types.

3.2.3 Inference

Inference of null-terminated pointers is particularly important. In many cases, intermediate expressions or types must be flagged as a null-terminated pointer instead of a bounded pointer in order to avoid coercing a null-terminated pointer to a bounded pointer prematurely. For example, our infrastructure turns the statement `str = cond ? "foo" : NULL` into the following code:

```
char *tmp;
if (cond) {
    tmp = "foo";
} else {
    tmp = NULL;
}
str = tmp;
```

Since both `"foo"` and `str` must be null-terminated, we must make sure that `tmp` is null-terminated as well. But since `tmp` is added by our infrastructure, this task *must* be done by the inference algorithm.

Inferring null-terminated types is a simple extension to our inference procedure for bounded pointer types. Once again, we use CCured's pointer graph, and we add a new flag indicating whether a pointer is null-terminated. This flag is propagated in both directions across *cast* edges and *compat* edges, which means that null-terminated pointers spread as far as possible in both directions.

However, this inference procedure is only invoked at function granularity, which means that the flag indicating a null-terminated pointer does not propagate outside a function to callers or callees. We have found that this approach simplifies the programmer's task significantly, since propagation outside a given function tends to result in unhelpful or bizarre errors. In particular, when this flag is propagated through a commonly-used function like `memset()`, errors involving null termination will appear in completely unrelated parts of the code.

Programmers can indicate that a pointer is not null-terminated by explicitly annotating it as a normal bounded pointer. The flag indicating null-termination will not flow

through such nodes, so a coercion from null-terminated to bounded pointers will occur at that location.

Finally, if the flag for null-termination escapes to an unannotated function parameter or global variable, Deputy issues a warning to the user. Much like in the case of arithmetic flags that flow to unannotated parameters or globals, Deputy prefers to warn the user about potential errors at interfaces instead of making possibly unintended changes to those interfaces.

3.2.4 Optimization

Null-terminated pointers also merit some additional optimizations for particular special cases.

First, it is important to take care when implementing the run-time checks to ensure that the `findzero()` function is only invoked when needed. For example, when accessing the third element of a pointer of type `NT COUNT(5)`, there is no need to search for the null terminator, since the element being accessed is clearly in bounds.

In addition, null-terminated pointers are often used by walking through the pointer one element at a time, as in the following code:

```
char * nt count(0) p = ...;
while (*p > 0) {
    ...;
    p++;
}
```

In this case, it is important for our dataflow analyzer to track the fact that $*p > 0$ and use it to eliminate the corresponding bounds check.

3.3 Dependent Union Types

To ensure that C unions are used correctly, programmers often provide a "tag" that indicates which union field is currently in use; however, the conventions for how this tag is used vary from program to program. Our type system provides dependent type annotations that allow the programmer to specify for each union field the condition that must hold when that field is in use.

To introduce unions, we add a family of new type constructors called "union$_n$", where n indicates the number of fields in the union. This constructor takes n type arguments indicating the types of each field of the union as well as n integer arguments indicating whether the corresponding field of the union is currently active. Thus, we write a union type as "union$_n$ τ_1 ... τ_n e_1 ... e_n", where τ_i are the field types and e_i are *selector expressions*. If selector e_j is nonzero, then the corresponding field with type τ_j is the active field of the union. As usual, the selectors are local expressions, so they can depend on other values in the current environment just as pointer bounds do.

In the case of unions, we do not add any new syntax aside from the "union$_n$" constructor. However, we reuse much of the syntax added for structures; in particular, we reuse the structure literal and field reference syntax when introducing and eliminating union types. As with bounded pointers, we add type rules for the relevant operations on this new type constructor.

An example union type is as follows:

$$x : \text{struct } \{\ tag : \text{int};\ u : \text{union}_2 \text{ int (ref int) } (tag \geq 2)\ (tag = 1)\ \}$$

Here, we have a structure containing a union and its associated tag, which is a

common idiom found in C programs. The union $x.u$ contains two fields: an integer and a reference to an integer. The selector expressions indicate that the union contains an integer when $tag \geq 2$ and that it contains an integer reference when $tag = 1$. Note that these selector expressions must be mutually exclusive.

3.3.1 Definition

The following invariant describes how unions are handled in Deputy.

Definition 3 (Union types) *If a value v has type*

$union_n\ \tau_1\ \ldots\ \tau_n\ e_1\ \ldots\ e_n$, then these two conditions hold:

- *at most one of the selector values is non-zero, and*
- *if e_i is non-zero, then v is a value of type τ_i.*

3.3.2 Type Rules

In Figure 3.8, we show the rules for type checking dependent unions.

The (UNION LITERAL) rule shows how union literals are coerced to a certain union type. The guard condition verifies that the selector expression for the ith field is nonzero and that the selector expressions for the remaining fields are zero. The (UNION ACCESS) rule shows that a field can be accessed if its selector is nonzero. Finally, the (TAG COERCION) rule allows us to make changes to the tag field as long as the value of the selector fields remains unchanged; this feature accommodates cases where more than one tag value corresponds to a single field.

(UNION LITERAL)

$$\frac{\Gamma \vdash e : \tau_i \Rightarrow \gamma \qquad \text{for all } j,\ \Gamma \vdash e_j : \text{int} \Rightarrow \gamma_j \qquad \gamma' = \gamma \wedge (\bigwedge_{1 \le j \le n} \gamma_j) \wedge (e_i \neq 0) \wedge (\bigwedge_{1 \le j \le n, j \neq i} (e_j = 0))}{\Gamma \vdash \{\ f_i = e\ \} : \text{union}_n\ \tau_1\ \ldots\ \tau_n\ e_1\ \ldots\ e_n \Rightarrow \gamma'}$$

(UNION ACCESS)

$$\frac{\Gamma \vdash \ell : \text{union}_n\ \tau_1\ \ldots\ \tau_n\ e_1\ \ldots\ e_n \Rightarrow \gamma_\ell}{\Gamma \vdash \ell.f_i : \tau_i \Rightarrow \gamma_\ell \wedge (e_i \neq 0)}$$

(TAG COERCION)

$$\frac{\Gamma \vdash e : \text{union}_n\ \tau_1\ \ldots\ \tau_n\ e_1\ \ldots\ e_n \Rightarrow \gamma \qquad \Gamma \vdash e_i' : \text{int} \Rightarrow \gamma_i}{\Gamma \vdash e : \text{union}_n\ \tau_1\ \ldots\ \tau_n\ e_1'\ \ldots\ e_n' \Rightarrow \gamma \wedge \bigwedge_{1 \le i \le n} (\gamma_i \wedge ((e_i = 0) \Leftrightarrow (e_i' = 0)))}$$

Figure 3.8: Typing rules for tagged unions.

Note that we do not have a rule for writing directly to the field of a union. Rather, when updating a union, we use the standard $l := e$ syntax where e will usually be a union literal. Since the tag and the union must often be set simultaneously, we must make use of parallel assignment to process the tag update and the union update simultaneously. Although this approach requires a bit more programmer effort, we believe that the additional cost in programmer effort is relatively small.

The primary alternative to this approach is to have an explicit rule for writing to a union field along with a coercion rule that allows arbitrary tags as long as the union's data is all zero. Then, much as in the example with bounded pointers, we can update a union by zeroing it, setting its tag, and then setting its data. Unfortunately, this approach typically requires the programmer to identify this code anyway, because existing code that uses unions rarely zeros a union before updating it. Also, this approach imposes the additional restriction that all union fields must have types for which zero is a valid value.

Note that it is also possible to use automatic dependencies to generate appropriate tag fields for an untagged union. However, since it is inefficient to generate n fresh variables to hold the value of each selector when a single integer field would suffice, it is possible to write a variant of this union type that uses a single selector expression. This variant is less useful for existing tags, since it is unlikely that existing tags a predictable scheme for mapping tag values to fields; however, this variant can be useful when automatic dependencies are needed. The rules for this variant are a straightforward modification of the rules presented in this section.

To see how these rules work in practice, consider the following example:

$$\begin{array}{l}\text{let } tag : \text{int} = 2 \text{ in} \\ \text{let } u : \text{union}_2 \text{ int (ref int) } (tag \geq 2) \ (tag = 1) = \{\ f_1 = 7\ \} \text{ in} \\ \text{let } p = \text{new int}(42) \text{ in} \\ tag, u := 1, \{\ f_2 = p\ \};\end{array}$$

In this example, we initialize tag to 2 and u to an integer value using a union literal. Since $tag = 2$, the assertion generated by the (UNION LITERAL) rule holds. Next, we update this tag and union using parallel assignment. When we apply our (PARALLEL VAR WRITE) rule, we get the following premises after substitution:

$$\begin{array}{l}\Gamma \vdash 1 : \text{int} \Rightarrow \gamma_{tag} \\ \Gamma \vdash \{\ f_2 = p\ \} : \text{union}_2 \text{ int (ref int) } (1 \geq 2) \ (1 = 1) \Rightarrow \gamma_u\end{array}$$

The first premise is obviously derivable, and the second premise is once again derivable via the (UNION LITERAL) rule.

In practice, we have found these union types to be useful in several locations. For example, in the Linux `e1000` driver, we were able to provide the following annotations:

```
struct e1000_option {
    enum { enable_option, range_option, list_option } type;
    union {
        struct { ...  } r WHEN(type == range_option ||
                               type == enable_option);
        struct { ...  } l WHEN(type == list_option);
    } arg;
};
```

In the concrete syntax, the `WHEN` annotations are used to specify the condition for each union field. Note that we were able to use an existing tag field, and the ability to specify an arbitrary predicate was quite useful.

However, we have generally found the union types to be less effective for annotations than the bounded or null-terminated pointer types. The biggest problem encountered when adding these annotations is that C programmers often define union types separately from the location where they are used, and the tag information can vary significantly from location to location. In addition, union tag idioms are less ingrained in C programmers than pointer bound idioms, so there is greater variation in the techniques used to disambiguate unions.

Nevertheless, we believe that this union type provides an effective starting point for improving safety in real C code. Indeed, by providing a standard method for annotating union tags, we can hopefully provide a useful standard for tagging and checking union types.

Chapter 4

Applications

In this chapter, we present a number of case studies to demonstrate how Deputy can be used on real-world C programs. These case studies include a number of smaller benchmarks taken from preexisting benchmark suites, several components for the TinyOS operating system, and most importantly, a significant fraction of the Linux kernel.

4.1 Implementation

We implemented Deputy using the CIL infrastructure [NMW02].[1] Our implementation is 21,500 lines of OCaml code in addition to the CIL front-end itself. This code includes about 6,500 lines for the core type checker, 8,500 lines for the inference component (which is derived from CCured's code), and about 6,000 lines for the optimizer.

Given an annotated C program, our implementation proceeds through the following phases. First, we run the inference engine, which adds annotations where necessary,

[1] This implementation is available at `http://deputy.cs.berkeley.edu/`.

including a tag indicating locations where automatic bounds are useful. Second, the automatic bounds inference executes, which replaces automatic bounds tags provided by the programmer or the inference engine with fresh variables, as described in the previous chapter. Third, we run the core type checker, which inserts run-time checks according to the type rules. Fourth, we run the optimizer, which eliminates run-time checks where necessary. Finally, we emit the resulting program as C code for compilation with GCC.

Users invoke Deputy by running the `deputy` command in place of `gcc` as their compiler, either via `make` or the command line. The `deputy` executable takes the same command-line arguments as `gcc`, so this switch is mostly transparent to the programmer.

Once the build process has been set up, using Deputy is similar to the edit-compile-debug cycle of a standard compiler. When running Deputy on existing unannotated code, the inference stage and type checker will typically report numerous warnings and errors indicating places where additional annotations or code changes are required. For example, warnings and errors might look like the following:

```
fs/exec.c:204: Warning: Type "char * COUNT(0) SNT * COUNT(0) SNT"
in formal "argv" of copy_strings should be annotated NT.

fs/exec.c:1271: Warning: Type "char * NT" in formal "corename" of
format_corename needs a bound annotation.

fs/super.c:660: Error: Type mismatch in coercion:
  from: void * BOUND(...)
    to: struct block_device * BOUND(...)
   exp: data
```

Once the programmer has made the changes necessary to eliminate compile-time errors and assertions, the programmer must run the actual program with representative

test cases in order to eliminate run-time assertions. A typical assertion might look like the following:

```
drivers/ide/ide-cd.c:2183: cdrom_read_tocentry: Assertion failed
in upper bound coercion: buf == 0 || buf + buflen <= buf + 1
```

These assertions are designed to be similar to existing compiler errors in order to provide a familiar edit-compile-debug cycle.

4.1.1 C Features Supported

Our implementation covers most of C's features, thanks to the CIL front-end, which simplifies complex C code to a manageable form. In addition to the C language features and dependent type constructors presented in the preceding chapters, we also support:

- *Address-of.* We allow the use of the address-of operator on local variables and structure fields. In all cases, the target of the address-of operator must not be involved in any dependencies—that is, its type must be well-formed in the empty environment, and no other types may depend on it. With these restrictions, we can ensure that taking the address of a variable or field does not interfere with the corresponding update rules.

- *Polymorphism.* We support parametric polymorphism at the function and structure level as long as all type variables are instantiated with types that have a one-word run-time representation and that do not have any dependencies—that is, they must be well-formed in the empty environment. This approach ensures that the polymorphic

code need only be compiled once and that it does not interfere with dependencies. Programmers typically use this feature in place of C's `void*` type; to do so, they must explicitly replace `void*` with a type variable using new syntax provided by Deputy.

- *Non-null pointers.* Programmers can allocate pointers as `NONNULL`, which allows the optimizer to eliminate many more null pointer checks. Since these checks are perhaps the most common and most expensive checks, this annotation is very useful in reducing run-time overhead.

- *Allocators.* The programmer can mark certain functions such as `malloc()` and derivative functions as allocators. Such annotations tell Deputy which function calls should be treated like the "new" syntax in the language of our dependent type framework. Likewise, the programmer may mark deallocators such as `free()`; however, the use of these functions is not currently checked (see below).

- *Special C functions.* Deputy allows programmers to identify certain common C functions that must be recognized in order to ensure type safety. For example, `memset`, `memcpy`, and `memcmp` all have special-case handling that is critical for ensuring type safety. In addition, we support limited checking for `printf`-style variable argument functions.

4.1.2 Sources of Unsoundness

Despite the annotations and additional C features supported by our implementation, there are still several important sources of unsoundness in Deputy, including the following:

- *Deallocation.* Currently, we trust that Deputy programs will handle deallocation properly; that is, we trust that every value obtained by `malloc` will be passed to `free` exactly once, and that no other non-null values will be passed to `free`. Similarly, we allow programs to take the address of stack-allocated variables, but we do not prevent programs from accessing such objects after the stack frame has been deallocated. This problem is orthogonal to the problem of checking bounded pointers and tagged unions, and addressing it is significantly more complex, since it involves reasoning about inherently global invariants. Some possible solutions to this problem include using a garbage collector [Boe93], automatic pool allocation [DKAL05], or a reference-counting solution such as CCount [ABC+07].

- *Concurrency.* Deputy makes no attempt to reason about concurrency. Among other things, it is possible to have race conditions wherein one thread alters a pointer variable between the safety check and use performed by another thread. However, if the original program contains no race conditions, Deputy's checks will be safe and will not introduce any new race conditions.

- *Non-*`printf`*-style variable-argument functions.* We do not check variable-argument function calls unless they use the `printf` convention. Even then, we cannot process the body of `printf`-style functions correctly. (There is, however, no fundamental technical issue preventing such a feature.)

- *Inline assembly.* Deputy ignores inline assembly. Much like code that resides in other modules, Deputy trusts that this code maintains the invariants of the dependent types in the program. Future versions might be able to perform some analysis of

inline assembly, since typical inline assembly should be irrelevant to Deputy's safety properties.

- *Trusted code.* Deputy's type checker will skip any code that has been annotated as trusted. This annotation is used, for example, when a program performs a cast that Deputy cannot verify as safe. When we present our experiments, we will indicate how often we needed to use this escape hatch.

4.2 Small Benchmarks

To test Deputy, we annotated a number of standard benchmark suites, including Olden [Car96], Ptrdist [ABS94], and selected tests from the SPEC CPU95 [SPE95], SPEC CPU00 [SPE00], and MediaBench [LPMS97] suites.[2] In addition, Deputy has been run on three programs for version 2 of the TinyOS [HSW+00] sensor network operating system.[3]

The majority of these benchmarks have also been used in the CCured project [NCH+05], so we can compare Deputy's overhead to that of CCured. We ran Deputy and CCured using simple error messages (which permits more compiler optimizations), and we disabled CCured's garbage collector and stack checks. Thus, these experiments are head-to-head comparisons of the execution time overhead for the type safety and spatial memory safety checks inserted by both tools.

In the case of TinyOS, the benchmarks were written in a dialect of C called nesC [GLvB+03], which is compiled to C and then passed to GCC or Deputy. The generated C code contains no race conditions, no dynamic memory allocation, and no variable-

[2]Thanks to Matt Harren and George Necula for their work on porting this code.
[3]Thanks to David Gay for his work on porting this code and running tests on it.

argument functions, so inline assembly and trusted code are the only sources of unsoundness.

Results for these experiments are shown in Table 4.1. The first and second columns identify the benchmark, and the third and fourth columns indicate the number of modifications required to run each benchmark with Deputy. Modifications are indicated by the number of lines added and deleted; a changed line is counted in both columns. We conservatively estimate the number of lines changed by summing the number of lines added and removed and dividing by the total number of lines.

In the remaining columns, we present the execution time for each benchmark using Deputy and CCured. For each tool, we present the execution time for the code with and without the tool enabled, and we present the ratio of these times as well. All tests except the TinyOS tests were performed on a 2 GHz dual-core Intel Xeon processor running Linux 2.6.17. For each test, we measured the user time billed to each process using the `time` utility, and we report the average over five runs (standard deviations were negligible in all cases). In some cases, the "original" time for the two tools differs because significantly different methods were used when porting these programs to their respective tools; however, in most cases, the results show that porting makes little difference in overall performance.

In all experiments, we changed less than 30% of the lines of code in the program, and in most cases, this number was within 10%. Note, of course, that this number is somewhat conservative due to the way changed lines are double-counted and due to the small size of these benchmarks. Also, because Deputy does not change any data representations, these changes can be made incrementally, as time allows. We required 20 trusted annotations in order to port these benchmarks.

Suite	Benchmark	Lines	Lines Changed	Deputy			CCured		
				Orig.	New	Ratio	Orig.	New	Ratio
SPEC	`go`	29339	+130 -37 (0.6%)	0.65	0.73	**1.12**	0.65	0.69	**1.06**
	`gzip`	8678	+183 -124 (3.5%)	6.21	6.97	**1.12**	-	-	**-**
	`li`	7431	+435 -243 (9.1%)	0.06	0.09	**1.47**	0.06	0.09	**1.45**
Olden	`bh`	1907	+242 -330 (30.0%)	1.50	1.63	**1.09**	1.52	1.89	**1.25**
	`bisort`	679	+28 -66 (13.8%)	1.32	1.25	**0.95**	0.97	0.95	**0.98**
	`em3d`	358	+55 -13 (19.0%)	0.36	0.55	**1.53**	0.36	0.70	**1.95**
	`health`	605	+17 -10 (4.5%)	0.15	0.18	**1.21**	0.15	0.16	**1.04**
	`mst`	417	+46 -16 (14.9%)	0.56	0.73	**1.31**	0.63	0.63	**1.00**
	`perimeter`	395	+5 -0 (1.3%)	0.02	0.92	**39.80**	0.02	0.89	**42.80**
	`power`	768	+21 -10 (4.0%)	1.22	1.24	**1.02**	1.61	3.28	**2.03**
	`treeadd`	127	+8 -6 (11.0%)	0.56	0.99	**1.79**	0.56	0.62	**1.11**
	`tsp`	565	+7 -3 (1.8%)	1.69	1.74	**1.03**	1.68	1.73	**1.03**
Ptrdist	`anagram`	695	+67 -19 (12.4%)	2.43	2.41	**0.99**	2.44	3.63	**1.49**
	`bc`	7257	+195 -115 (4.3%)	0.81	1.11	**1.37**	0.80	0.99	**1.24**
	`ft`	2175	+48 -29 (3.5%)	0.52	0.59	**1.15**	0.52	0.54	**1.03**
	`ks`	792	+16 -6 (2.8%)	1.50	1.76	**1.18**	1.50	1.83	**1.22**
	`yacr2`	3976	+211 -214 (10.7%)	0.06	0.10	**1.81**	0.05	0.11	**2.11**
MediaBench	`adpcm`	387	+24 -6 (7.8%)	1.83	2.12	**1.15**	-	-	**-**
	`epic`	3413	+281 -104 (11.3%)	0.02	0.03	**1.56**	-	-	**-**
TinyOS	`Blink`	74	0 (0%)	-	-	**1.04**	-	-	**-**
	`BaseStation`	282	0 (0%)	-	-	**1.17**	-	-	**-**
	`Oscilloscope`	149	±3 (2.0%)	-	-	**1.13**	-	-	**-**
	OS components	11698	±48 (0.4%)	-	-	**-**	-	-	**-**

Table 4.1: Deputy benchmarks for small and medium-sized programs.

The slowdown observed with Deputy ranged from nothing to a factor of 1.81, with a single outlier at 39.80 on the `perimeter` test. This outlier is the result of a very small program with many pointer dereferences whose null checks are difficult to optimize away; however, if we remove null checks and leave them to the hardware, this slowdown is reduced to a modest 1.15. CCured exhibits similar behavior, ranging from zero overhead to a factor of 2.11, with a similar outlier on the `perimeter` test.

Comparing the execution time ratios for Deputy and CCured reveals that these tools are both competitive in terms of run-time overhead. Deputy prevails in 8 of the 16 tests where data for both tools is available, and CCured prevails in 7 of the 16. Both tools prevail by a significant margin in some subset of these tests. One important conclusion from this comparison is that Deputy's split architecture, which consists of a naive flow-insensitive type checker plus a flow-sensitive optimizer, is competitive with CCured's more direct approach.

If null checks are disabled, leaving such checks to the virtual memory system, Deputy's numbers improve noticeably. The worst case is a factor of 1.86 with `yacr2`, and all remaining tests have a slowdown that is less than 1.40. CCured fares slightly worse with the same optimization; there are three tests with slowdowns of 1.89, 2.01, and 2.06, with the rest below 1.40.

In several cases, we observe speedups when running Deputy. We believe that these speedups are a result of transformations made by the CIL infrastructure or optimizations that Deputy applied to the original code as well as to the generated checks.

The TinyOS tests were compiled with nesC 1.2.7 and GCC 3.4.3 and were executed

on a `mica2` mote, which has a 7.37 MHz Atmel ATmega128 microcontroller. The results table shows the number of changes required in the code itself as well as the number of changes in TinyOS code outside of these programs. Since these tests perform periodic tasks, these tests measured the percent of CPU usage consumed during one minute; they averaged five runs, and the standard deviation was within 2% of CPU usage. Deputy's overhead was within 17% on all three tests. Code size expansion, which is important for embedded systems with limited code space, was below 16% on all TinyOS programs.

Further work on TinyOS using Deputy has recently been reported by Cooprider et al. [CAE+07]. In this work, Deputy was applied to 37,921 lines of TinyOS code with annotations added to 0.76% of the code, in addition to a few pervasive code changes and slightly more than 30 trusted lines. They report negative RAM overhead, 9.5% ROM overhead, and a 6.5% slowdown, which suggests that Deputy is quite suitable for enforcing safety in embedded systems.

During these tests Deputy found several bugs. A run-time failure in a Deputy-inserted check exposed a bug in TinyOS's radio stack (some packets with invalid lengths were not being properly filtered). In `epic` we found an array bounds violation and a call to `close` that should have been a call to `fclose`. We also caught several bugs that we were previously aware of: `ks` has two type errors in arguments to `fprintf`, and `go` has six array bounds violations.

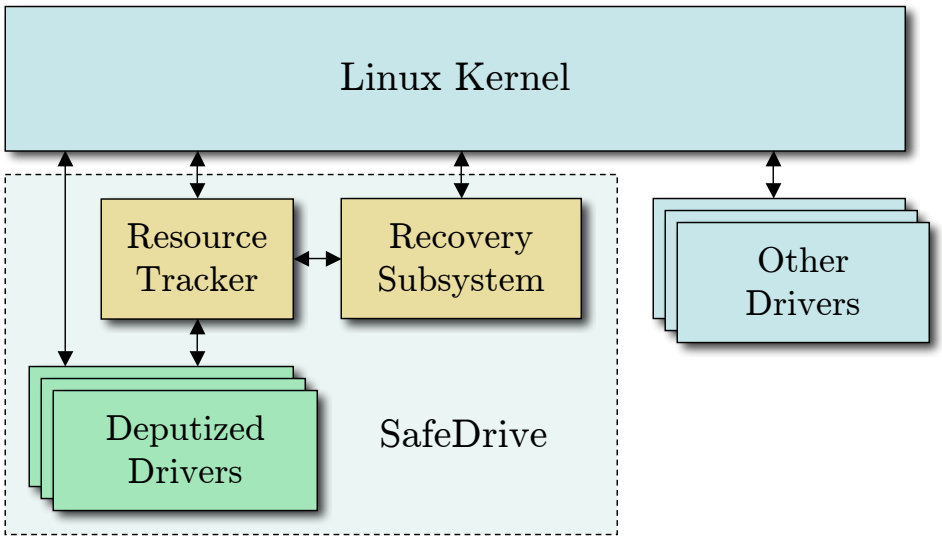

Figure 4.1: SafeDrive's architecture.

4.3 Linux Device Drivers

SafeDrive [ZCA+06] is a system created by the Ivy research group at Berkeley for detecting and recovering from failures in Linux device drivers. In this system, Deputy is used on device drivers, independent of the kernel, and failures detected by Deputy trigger a recovery mechanism. Because Deputy can be applied to existing code without changing the representation of key data structures or the device driver API, the implementation of the recovery mechanism is both simpler and faster than previous systems such as Nooks [SBL05].

SafeDrive's architecture is illustrated in Figure 4.1. As shown here, we apply Deputy to some subset of the drivers used by the kernel, and we interpose a resource tracker and a recovery subsystem.[4] The resource tracker is invoked by kernel API calls that

[4]These components were designed and implemented by Feng Zhou.

Driver	Type	Hardware
`e1000`	Network	Intel PRO/1000
`tg3`	Network	Broadcom Tigon3
`usb-storage`	Storage	USB mass-storage
`intel8x0`	Sound	Intel 8x0
`emu10k1`	Sound	Creative Audigy 2
`nvidia`	Video	NVidia cards

Table 4.2: Drivers used in experiments.

acquire or release kernel resources on behalf of the driver. This tracker maintains a list of kernel resources that the driver has acquired, such as kernel memory and locks. Each tracked object is associated with a *compensation action* that indicates how the resource should be freed; for example, a memory allocation's compensation is a call to `kfree()`, and a lock's compensation action releases the lock. If the resource is released normally, the object is removed from the tracker.

When a fault occurs, SafeDrive immediately returns from the driver with an appropriate error code, which the kernel handles gracefully. The kernel then invokes the recovery subsystem, which executes compensation actions in order to clean up the driver's state. The driver is then restarted, and its initialization routines are expected to reset the device from an arbitrary state.

We ran SafeDrive on the drivers listed in Table 4.2.[5] All of these drivers were taken from Linux 2.6.15.5 in their original form. We modified the kernel build process to use Deputy when compiling these drivers; however, the remainder of the kernel is still built using GCC, as usual.

A key feature of SafeDrive that is enabled by Deputy is the fact that we can link

[5]Thanks to Feng Zhou, Matt Harren, Ilya Bagrak, and Bill McCloskey for porting drivers.

Driver	Original LOC	Deputy Changes				
		Modified	Bounds	Nullterm	Unions	Trusted
e1000	17011	260	146	15	2	47
tg3	13270	359	78	9	0	64
usb-storage	13252	136	16	11	0	21
intel8x0	2897	124	31	2	0	8
emu10k1	11080	441	66	11	0	23
nvidia	10126	224	42	35	0	27

Table 4.3: Annotation burden for each driver.

Deputy-compiled drivers directly with a GCC-compiled kernel. Previous memory safety tools would have required significant changes to the binary interface between the drivers and the kernel, which would have significantly complicated the process of converting this code, not to mention the code for recovering from errors. Deputy's dependent types allow us to focus our efforts on the most vulnerable and least-tested code first without converting the entire kernel or writing expensive wrappers to convert between data formats.

For example, `include/linux/skbuff.h` contains a definition of the `sk_buff` structure, which is used to pass network data between the driver and the kernel. This structure contains a number of pointers to data buffers that are manipulated directly by both the kernel and the driver. With the annotations we provide (which were shown in Section 3.1), we can safely access the data in this structure without changing its layout. Previous tools would have needed to modify this structure, making it impossible to link a transformed driver with a preexisting kernel.

Fault Category	Code Transformation
Loop fault	Make loop count larger by 1 with 50% chance, 2–1024 with 44% chance, and 2K–4K with 6% chance
Scan overrun	Make size parameter to `memset`-like functions larger with the probabilities above
Off-by-one	Change < to <= in boolean expressions
Flipped condition	Negate conditions in `if` statements
Missing assignment	Remove assignments and initialization of locals
Corrupt parameter	Replace a pointer parameter with null or a numeric parameter with a random number
Missing call	Remove calls to functions and return a random result as in the line above

Table 4.4: Categories of faults injected in recovery experiments.

4.3.1 Annotation Burden

In order to compile these drivers with Deputy, we needed to add annotations to the Linux headers and to the drivers themselves, and we also needed to make modifications to the driver code in order to ensure that the code respects the annotations.

The annotation burden for this task as reported by our research group members is shown in Table 4.3. This table indicates the number of source code annotations that were performed in each case, and it also shows the number of each type of annotation that was inserted, including annotations for bounded pointers, null-terminated pointers, tagged unions, and trusted code. The overall number of modifications typically falls within 1-4% for each driver. Qualitatively, converting a single driver takes about 3-5 person-hours, for a person who is familiar with exactly one of Deputy and Linux.

		SafeDrive off			Total
		Crashes	Test Failed	Test Succeeded	
SafeDrive on	Static Deputy error	10	0	3	13
	Dynamic Deputy error	34	2	5	41
	No error detected	0	19	67	86
Total		44	21	75	140

Table 4.5: Breakdown for the 54 cases in which SafeDrive detected errors.

4.3.2 Fault Injection

In order to evaluate the effectiveness of Deputy and SafeDrive, the `e1000` driver was tested both with and without a number of artificially-injected faults.[6] The faults injected were from the categories listed in Table 4.4. In these tests, 20 randomly-chosen faults from each of these categories were injected at compile time, and the `e1000` driver was executed both with and without SafeDrive enabled.

Table 4.5 shows the results of these experiments as they relate to Deputy. The vertical columns indicate the results when SafeDrive is off, and the horizontal rows indicate the results when SafeDrive is on, so the numbers in each box show which kinds of errors Deputy did and did not detect. In particular, each column provides a breakdown of how the use of SafeDrive and Deputy altered the behavior of a given set of test cases. In the first column, we see that 44 of the injected errors caused the driver to crash without Deputy, and Deputy was able to catch all of these errors, 10 at compile time and 34 at run time. In the second column, we see that without Deputy, 21 of the errors resulted in a failure of the driver but not a crash of the operating system; in this case, only 2 of the errors were caught by Deputy. Finally, in the third column, there are 75 cases where the injected bug

[6] These fault-injection tests were designed and run by Feng Zhou.

did not manifest itself during the test when running without Deputy; here, Deputy found 8 of these latent bugs but did not find the remaining 67.

The first conclusion to draw from these results is that Deputy is very effective at finding crashing errors. Since memory corruption is a frequent cause of these outright crashes, this result is expected. Deputy finds fewer of the errors that did not cause a crash, often because these errors fall outside Deputy's definition of safety. For example, perhaps the driver passes an incorrect parameter to the kernel, but that parameter has nothing to do with type or memory safety. However, Deputy does manage to find some of these bugs, including eight bugs that would otherwise not have been noticed during this test. It is also important to note that Deputy's checks lay the foundation for higher-level program analyses that can find higher-level program flaws, such as the 86 undetected errors.

4.4 Linux Kernel

In addition to the SafeDrive experiments described above, we have also converted a stripped-down version of the Linux kernel to use Deputy. The purpose of this experiment is to demonstrate that it is feasible to apply Deputy to an operating system in its entirety and to measure the performance impact of Deputy's checks when they are placed throughout the code.

To create this kernel, we took the unmodified Linux 2.6.15.5 kernel, and we created a configuration that included all basic operating system services, including the core kernel services, the virtual memory manager, the virtual file system layer, the TCP/IP network stack, the `ext2` file system, and the drivers required to run on our test machine and on a

VMware virtual machine. The unmodified kernel consists of approximately 1,200,000 lines of code, and our stripped-down version contains approximately 435,000 lines of code. Thus, it is a significant fraction of the entire kernel distribution, and indeed, it is the most vital fraction of the kernel code, since most of the code we have included is included in every configuration of the kernel.

4.4.1 Annotation Burden

In order to compile and run the 435,000-line kernel with Deputy, we needed to add Deputy annotations to approximately 2627 lines (about 0.6% of the code), and we needed to trust approximately 3273 lines (about 0.8% of the code).[7] Although these numbers seem relatively small, it is difficult to judge the significance of a single line of trusted code by counting it at compile time, since the impact of this trust varies greatly depending on how the code is used within the program and how often it is executed. Nevertheless, these numbers provide an estimate of our code coverage and the amount of code that must be reviewed carefully by a programmer in order to enforce safety.

The annotation process took approximately 7 person-weeks of effort, with the combined effort of four different contributors. The rate of progress increased significantly as we became more familiar with the kernel and with idioms that were useful for annotating kernel code. Since this work has covered at least a third of the full Linux distribution without the benefit of expertise in the Linux kernel itself, it seems quite conceivable that a team of knowledgeable kernel developers could use annotate and modify the entire kernel distribution with a few months of work.

[7] Thanks to Feng Zhou, Rob Ennals, and Zach Anderson for their work on porting Linux code.

One prominent idiom not currently covered by Deputy's annotations (and therefore responsible for a nontrivial fraction of the trusted code) is upcasts and downcasts. For example, the `sk_buff` structure contains a pointer to an object of type `dst_entry`, which represents part of a protocol-independent routing destination cache. However, when these `sk_buff` objects are used with the IPv4 protocol, the `dst_entry` pointer actually points to a structure of type `rtable`, whose first elements match those of `dst_entry`. The IPv4 code frequently casts from `dst_entry` to `rtable` in these cases.

The difficulty in annotating such code is that the metadata indicating the true type of this pointer is not always nearby or stored in a consistent manner, as it often is with pointer bounds. In this case, the indicative field is actually the `input` field of the structure `dst_entry`, which contains a function pointer that handles IP delivery. From Deputy's point of view, the value of this field describes the true type of the containing structure. Unfortunately, though, such downcasts are not always tracked in the same manner in other structures.

The ideal solution to this problem is to have the program add run-time type information in a consistent manner. Although this change involves modifying existing data structures, this change takes place at the programmer's level as opposed to the compiler's level, making it more palatable than data structure changes made by previous memory safety tools like CCured [NCH+05].

A second kernel idiom that Deputy has difficulty with is page-size buffers containing null-terminated data. The programmer is tempted to annotate such buffers as `char * NT COUNT(PAGE_SIZE - 1)`; however, the kernel does not always force the last element to

Bandwidth	Deputy (F)	Deputy (V)
bzero	0.99	1.01
file_rd	0.98	0.99
mem_cp	0.98	1.00
mem_rd	0.99	1.01
mem_wr	0.99	1.00
mmap_rd	0.87	0.91
pipe	0.98	0.97
tcp	0.92	0.84

Latency	Deputy (F)	Deputy (V)
connect	1.03	1.12
ctx	1.08	1.16
ctx2	1.01	1.08
fs	1.17	1.06
fslayer	1.02	1.07
mmap	1.51	1.51
pipe	1.16	1.29
proc	1.00	1.05
rpc	1.27	1.41
sig	1.33	1.36
syscall	1.04	1.04
tcp	1.20	1.59
udp	1.29	1.41

Table 4.6: Performance of the Deputized Linux kernel on the HBench-OS benchmark suite.

be zero. In other scenarios, Deputy is happy to force the final element to be zero, but with such delicate constraints on allocation and page data, we are reluctant to change the behavior of the kernel in such cases. Hopefully, further review of the kernel code will determine an appropriate change to either the annotation or the code.

4.4.2 Performance: HBench-OS

In order to measure the performance impact of Deputy's run-time checks, we ran the HBench-OS suite of kernel benchmarks [BS97]. These benchmarks test the bandwidth and latency for a number of kernel primitives, including memory reads and writes, simple system calls and file system operations, and simple network operations. We ran these tests on the same kernel compiled with three different tools: GCC 4.1.3, Deputy with fast checks, and Deputy with verbose checks. In the "fast" version, an assertion failure prints a generic message and causes a kernel panic, whereas in the "verbose" version, an assertion failure prints a specific error message and has the option to continue executing. The former version

is more amenable to compiler optimizations and has less static data, so it is expected to perform better; however, it is less useful for diagnosing assertion failures. These tests were performed on a 2.33 GHz Intel Xeon processor with a 4 MB cache and 1 GB of RAM.[8]

The results of these performance tests are shown in Table 4.6. The "Deputy (F)" column presents the ratio of the "fast" Deputy kernel's performance to the GCC kernel, and the "Deputy (V)" column presents the ratios for the "verbose" kernel. Of course, higher numbers are better in the case of bandwidth tests, and lower numbers are better in the case of latency tests.

In the bandwidth tests, the file and memory tests show very little (if any) overhead for both kernels. Since these tests perform simple reads and writes to memory, they exercise very little kernel code. The "mmap", "pipe", and "tcp" tests show more noticeable performance degradation, since they involve additional kernel code; the fast kernel is slowed down by up to 13% whereas the verbose kernel's slowdown was as much as 17%.

The latency tests show a more noticeable performance impact, particularly in operations that exercise a fair amount of kernel code, such as file system and network tests. Most tests for the fast kernel were within 20-30% overhead, with 51% in the worst case, whereas the verbose kernel suffered 30-40% overhead in most cases, with 59% in the worst case. The verbose kernel routinely suffers a performance penalty that is 2-3 times that of the fast kernel.

Although we hope that further optimization and porting effort can reduce these overheads even further, we also believe that these performance penalties could be acceptable in cases where reliability is favored over performance. Also, it is important to note that these

[8]Thanks to Feng Zhou and David Gay for their assistance with the HBench test setup.

	GCC	Deputy (F)	Deputy (V)
Kernel Build			
Real Time (s)	161.4	162.4	174.7
User Time (s)	129.6	128.7	126.8
System Time (s)	16.0	16.1	23.5
SPECweb99			
Latency (ms/op)	315.2	321.0	323.5
Client Bandwidth (Kbits/s)	380.1	374.7	371.9
WebStone (50 clients)			
Latency (ms/op)	61.5	61.5	62.0
Server Throughput (Mbits/s)	129.8	129.9	128.9
WebStone (100 clients)			
Latency (ms/op)	119.4	119.2	119.4
Server Throughput (Mbits/s)	134.3	134.7	134.5

Table 4.7: End-to-end benchmark results for the Deputized Linux kernel.

measurements show penalties for very specific kernel operations, but the total time spent executing kernel code is typically dwarfed by time spent running application code; thanks to Amdahl's Law, these penalties should be less noticeable on end-to-end benchmarks.

4.4.3 Performance: Kernel Build

In addition to the microbenchmarks discussed in the previous section, we also ran two end-to-end benchmarks that indicate overall application performance when using Deputy on the kernel.

The first such end-to-end benchmark is a kernel build. In this test, we ran GCC 4.1.3 (not Deputy) to build the Deputy-annotated kernel itself on the same machine that we used for the previous benchmarks. Note that no Deputized code is running at user level; all overhead should be incurred at the kernel level. We ran this test using the same set of kernels, which are built with GCC, Deputy with fast checks, and Deputy with verbose

checks. For each kernel used in our test, we measured the amount of user, system, and real (wall-clock) time required to execute this benchmark, and we report the average of five runs (all standard deviations were within 1 second). Each test was run after a fresh reboot and a single iteration of the benchmark in order to warm up the cache.

The topmost section of Table 4.7 shows the results of these experiments. The user time reported for these tests is within 3 seconds for all these tests, which is as expected—since Deputy was not run on user-level code, there should be no difference in the time consumed here. The system time shows an overhead of 47% in the case of the Deputized kernel with verbose checks, but there is almost no discernible overhead for the version with fast checks. In fact, the difference between the GCC test and the fast Deputy test is only a small fraction of a standard deviation. The wall-clock time shows this same trend: the kernel with verbose checks shows noticeable overhead (about 8%), but the version with fast checks shows negligible overhead.

As a result, we can conclude that Deputy's checks have relatively little overall impact on system performance for a large, multi-process job. Even with verbose debugging information, the overall performance degradation can be as low as 8%.

4.4.4 Performance: Apache Web Server

Finally, we tested end-to-end performance for a network server running on our Deputized kernel by using the SPECweb99 [SPE99] and WebStone 2.5 [Min02] benchmark suites on the Apache 2.2.3 web server. For this test, we ran the web server on the same machine described in the previous kernel benchmarks and with the same set of kernels as the previous benchmarks. We used two client machines on the same subnet to generate

requests; these machines and the network were otherwise quiescent while the tests were taking place.

For the SPECweb99 tests, we generated 100 simultaneous requests throttled at 400 Kbits/s. Each client generates requests from a preset distribution involving 70% static requests and 30% dynamic requests, where dynamic a mix of `GET`, `POST`, and server-side CGI and ad-generation requests. For each test, we ran a 10-minute warm-up period followed by three 10-minute measurement intervals.

In the second section of Table 4.7, we show the bandwidth and latency measured by these tests. The latency tests measure milliseconds per operation and had standard deviations between 2.2 and 2.5 ms/op. These tests show a 1.8% slowdown with the fast Deputy kernel and a 2.6% slowdown with the verbose Deputy kernel. The bandwidth tests measure the bandwidth in Kbits per second achieved by each client and had standard deviations between 2.4 and 3.4 Kbits/s. The fast Deputy tests showed a 1.4% slowdown, and the verbose tests showed a 2.2% slowdown.

For the WebStone tests, the clients issued requests for five files with sizes ranging from 0.5 KB to 5 MB according to a preset distribution. We measured results for 50 and 100 simultaneous clients, with three 10-minute measurements each.

The third and fourth sections of Table 4.7 shows the results of these measurements. We report the latency and server throughput observed during these tests. Standard deviations for latency were within 0.17ms except for the 50-client verbose test, which was 0.84ms. Standard deviations for throughput were within 1.8 Mbit/s. For all tests, we observe a very slight advantage in favor of the fast Deputy kernel and a disadvantage in the case of the

verbose Deputy kernel; however, in all cases, the difference between the Deputy and GCC results is well within one standard deviation. In other words, there is essentially no observable difference between these tests when run with a standard kernel versus a Deputized kernel.

We believe that the reason for these results, when compared with the SPECweb99 results, is the small set of test files used by WebStone. WebStone's test data totals less than 6 MB, whereas SPECweb99's requests are distributed over a 450 MB worth of data. Thus, most requests in the WebStone test are going to the cache, whereas many requests in the SPECweb99 test are going to the filesystem and therefore invoking more kernel code.

Overall, these numbers suggest that there is relatively little performance impact when running applications on a Deputy-enhanced kernel. Although our microbenchmarks show slowdowns of more than a factor of two in some tests, the overall impact for a typical application is quite modest—as little as 2.2%, even with verbose checks enabled.

Chapter 5

Related Work

There are a number of areas of related work that will be discussed in this chapter. These include other techniques for ensuring safety in existing imperative programs, other languages and systems using dependent types, hybrid type checkers, and type qualifiers.

5.1 Safety for Imperative Programs

Enforcing safety in existing code—particularly in languages like C—has been a central problem in computer science research for several decades. Generally speaking, this work can be split into several categories: (a) language-based solutions, which are applied at the source code level, (b) binary instrumentation techniques, which are applied to the executable itself, and (c) hardware-based solutions, which use memory protection to rule out errors.

5.1.1 Language-Based Safety

CCured [NCH+05] analyzes a whole program in order to instrument pointers with checkable bounds information. CCured classifies pointers according to their usage and then inserts appropriate checks. Although CCured was used on many large server programs, its use of fat pointers presents a significant obstacle to usability, which Deputy's dependent types address. Whereas CCured requires the programmer to make global changes to a program before testing or debugging it, Deputy allows incremental porting that uses existing program metadata.

One attempt to address these issues led to the development of data slicing [CN05], a technique for separating CCured's metadata from the original program's data. This technique allows CCured to link with external libraries without converting between two data layouts, but it does not account for changes or updates made to this data by external code. For example, if external code replaces a pointer in a data structure shared with CCured's code, CCured must somehow reconcile this change with its own metadata. With Deputy, this problem does not exist, because assumptions about where metadata is stored and how it must be maintained are made explicit at the programmer's level.

Cyclone [JMG+02] is a type-safe variant of C that incorporates many modern language features. Like CCured, it allows programmers to use fat pointers, so the resulting programs are incompatible with existing libraries. Cyclone allows some dependent type annotations; for example, the programmer can annotate a pointer with the number of elements it points to. However, Deputy provides more general pointer bound support (in particular, self-dependencies make Deputy's bounded pointers much more flexible) as well

as support for null-terminated bounded pointers and dependent union types.

Dhurjati and Adve developed SAFECode [DA06], which uses run-time checks to ensure that C programs access objects within their allocated bounds. This technique is derived from earlier work by Jones and Kelly that stored bounds information in a splay tree [JK97, RL04]. SAFECode has overhead comparable to that of CCured on a set of small to medium-size programs, but it does not ensure full type safety, and because it uses automatic pool allocation [DKAL05], it requires programmers to cede control of the allocator.

5.1.2 Binary Instrumentation

An alternative to language-based safety tools is binary instrumentation. The main advantage of this approach is that source code is not required; thus, it can be applied to executables or libraries provided by a third party. This approach also avoids the problems of library compatibility that arise when fat pointers are used, since any data structure transformation can be applied equally well to all parts of the program. However, because these tools do not have the benefit of source-level information, they are generally a bit less efficient than language-based safety tools. In addition, it is more difficult for such tools to enforce higher-level properties such as type safety; many focus on memory safety only.

One of the earliest binary instrumentation tools was software-based fault isolation, or SFI [WLAG94]. This tool allowed a process to load some untrusted code and modify that code such that untrusted code could only jump to other untrusted code or write to untrusted data. Assuming that this code is stored in a region of memory whose size is a power of two, it is possible to determine whether an address lies within this region by checking the topmost bits of a given address. Alternatively, SFI provides the option to simply *set* the top

bits of each memory address to the appropriate segment number, thus forcing all memory references into a given region; this approach improves run-time performance but is less useful for debugging. In general, the authors point out that they are sacrificing common case performance to find uncommon errors, which is a somewhat counterintuitive strategy; however, this strategy can be a win overall in cases where there are frequent cross-domain calls. The same approach is taken by Deputy and SafeDrive; even though Deputy checks all accesses, it makes up for this penalty in the ease of conversion and the simplicity of cross-domain calls.

One challenge in designing an efficient SFI implementation is dealing with control-flow integrity. Malicious code could attempt to jump past the inserted checks, and on architectures such as x86, which have variable-length instructions, it is possible to jump into the middle of an existing instruction. Dealing with these possibilities can be quite costly in terms of performance. Recent work on control-flow integrity, or CFI [ABEL05], has shown an alternate and more efficient approach to this problem: insert identifiers at sources and destinations of jump instructions in order to verify that jump instructions can only jump to addresses that are expected by the compiler.

More recently, XFI [EAV+06] has built on CFI in order to provide memory integrity guarantees for untrusted code. As with SFI, XFI offers somewhat weaker guarantees than Deputy (e.g., it does not check type safety) in return for the flexibility of being able to apply XFI to arbitrary binaries. In addition to the efficiency provided by the CFI framework, XFI also has the advantage of being split into an instrumentation component and a verifier. Most of the complexity is in the instrumentation component, which means

that the trusted verifier can be kept relatively small. Although Deputy makes no such distinction between instrumentation and verification, recent work has shown that it is possible to design an assembly-level verifier for Deputy-compiled code, building on related work for CCured [HN05].

An extreme approach to binary instrumentation is represented by Purify [HJ92], which tracks the allocation status of each byte in the heap in order to verify correct usage. Purify also leaves unallocated bytes before and after each allocation in order to improve its ability to detect memory errors such as buffer overruns. As with previous binary instrumentation techniques, the checks performed are relatively coarse-grained and do not verify correct usage of memory above the level of the allocator. In the case of Purify, we trade extremely costly checks (unusable in production code) for the increased chance to find bugs during program development.

5.1.3 Hardware-Based Safety

Finally, there are a number of approaches to this problem that do not attempt to modify the code, either at the source or the binary level. Instead, these techniques take advantage of the virtual memory hardware to find bugs or to isolate untrusted code. These techniques tend to be even less intrusive than binary instrumentation techniques, since they are implemented using the MMU. However, they are also very coarse-grained; typically, they cannot detect memory corruption within addresses belonging to a module (or even an entire application).

One example of this approach is Nooks [SBL05], a system for isolating Linux device drivers, much like SafeDrive. In Nooks, Linux device drivers occupy the same address

space as the kernel, but memory access privileges are altered when executing driver code in order to prevent errant drivers from corrupting kernel memory. This approach provides more coarse-grained checks, and the cross-domain calls between the kernel and the driver are both expensive and complicated. However, Nooks has the benefit of working on unmodified drivers, whereas Deputy and SafeDrive require the programmer to annotate driver code. Also, Nooks has a different set of vulnerabilities—whereas Nooks is incapable of catching memory corruption within a driver's address space (and may corrupt data structures that will later be copied out), Deputy is vulnerable to any code marked as trusted by the programmer, which often happens when Deputy is unable to provide suitable annotations for the code in question.

Another approach in this same vein is the Xen VMM's Safe Hardware Interface [FHN+04]. In this approach, all drivers execute in their own separate virtual machine, using a simple interface to send data to and from other virtual machines. As with Nooks, the basic mechanisms for enforcing isolation are all hardware-based, which provides coarse-grained but simple and widely-applicable fault isolation. Although this approach requires more programmer effort than the Nooks approach, it applies more broadly and has better performance overall. A similar approach based on L4 Linux [LUSG04] runs device drivers in a virtual machine, but without attempting to impose a common hardware interface. These authors claim that they are better able to deal with malicious code than Nooks (and presumably SafeDrive).

A similar approach has been taken by microkernel operating systems for many decades. Systems such as Mach [BRS+85] and Exokernel [EKJ95] put as much code as

possible into user space in order to prevent crashes or corruption caused by buggy or malicious code. Our work focuses on providing similar guarantees in the context of existing monolithic systems, and ideally at a fraction of the performance cost.

5.2 Dependent Types

Dependent types [Pie02, Pie05] are a widely-studied concept in the field of programming languages. Much of the original work on dependent types arose from the Curry-Howard correspondence, which establishes a mapping between types and logical propositions; in this framework, dependent product types and dependent sum types correspond to universal and existential quantification [ML84]. Further uses of dependent types in logical frameworks include the Edinburgh Logical Framework [HHP87] and the Calculus of Constructions [CH88], which form the basis of many modern interactive theorem provers.

More recently, several languages have attempted to incorporate dependent types into a practical, general purpose programming language. The earliest of these languages is probably Cayenne [Aug98], which is a general-purpose functional programming language that uses dependent types. Cayenne allows dependent types to refer to arbitrary expressions from the source language, which means that its type system is undecidable. Cayenne makes no attempt to skirt around the issue of decidability; such issues are left to the programmer. In addition, Cayenne is a pure functional language, so it does not need to reason about mutation as it relates to dependent types.

Another approach to using dependent types in a practical programming language is demonstrated by Dependent ML [XP99] and Xanadu [Xi00], which are functional and

imperative versions of the same core idea. In these languages, expressions appearing in dependent types are different from program expressions, and any constraints involving these expressions must be decidable at compile time. The language of index expressions (which appear in types) is frequently connected to the language of program expressions through the use of *singleton types*—that is, types that are inhabited by only one value. For example, a variable `n` might have the DML type `int(m)`, which means "all integers with value `m`". This type connects the program variable `n` with the index variable `m`. This approach simplifies type checking by separating the world of program expressions and index expressions; however, it is correspondingly more difficult to describe dependencies succinctly. As with Cayenne, DML and Xanadu do not allow mutation of variables involved in dependencies.

In contrast to these three languages, Deputy aims for a middle ground, allowing expressive annotations in the original source language, while using run-time checks to keep the type checker simple and decidable. We also allow mutation of expressions in dependent types, unlike these other systems. The main disadvantage of Deputy's approach is that many errors will be delayed until run time (or perhaps never caught) due to our use of run-time checks. In addition, Deputy requires a knowledgeable designer to craft appropriate type constructors that correspond to common C idioms; this approach makes Deputy more useful for existing code but somewhat less general.

Another approach to reasoning about mutation in dependently-typed programs is Hoare Type Theory [NM05], which uses a monadic type constructor based on Hoare triples to isolate and reason about code with side-effects. This approach grew out of Cyclone [JMG+02] much as Deputy grew out of CCured [NCH+05]; in both cases, the problem of

compatibility required some form of dependent types in the context of an imperative program. Deputy differs from Hoare Type Theory in that it attempts to assign flow-insensitive types to each program variable instead of trying to track the effects of a statement via Hoare triples. This approach simplifies the type system significantly, both for the designer of new type constructors and for the end user. Deputy also uses run-time checks and automatic dependencies to make its types more usable and more scalable.

Harren and Necula [HN05] developed a dependent type system for verifying the assembly-level output of CCured. Their system, which is related to Hickey's Very Dependent Types [Hic96], focuses on dependencies between fields of a mutable structure, much like Deputy's. However, their goal is to verify the output of a compiler statically; in contrast, Deputy operates on the original source code and focuses on inserting the checks required to verify safety. Harren and Necula's verifier can be used on the output of either CCured or Deputy to verify that the compiled program adheres to the desired safety policy.

Finally, Microsoft's SAL annotation language [HDWY06] provides interface annotations similar to those of Deputy. These annotations are viewed as preconditions and postconditions as opposed to Deputy's simpler flow-insensitive types. Microsoft's ESPX checker attempts to check all code statically, whereas Deputy is designed to emit run-time checks for additional flexibility. In recent years, Microsoft has been quite successful in using these annotations in real code; as of several years ago, they had added over 400,000 annotations to production source code [HDWY06]. This result suggests that simple and natural annotations can be of great benefit to existing code and are actually usable by (and desired by) systems programmers.

5.3 Hybrid Type Checking

Deputy uses a hybrid type checking strategy, which means that it relies on compile-time checks where possible and run-time checks where necessary. The advantage of this approach is that it is more complete (i.e., it handles more programs), but the disadvantage is that some errors may be delayed to run-time—or never found at all.

The term "hybrid type checking" was coined by Flanagan [Fla06], who also instantiated these ideas in HOOP [FFT06], a hybrid object-oriented programming language. These systems are designed to deal with somewhat more general dependent types and refinement types; in contrast, Deputy focuses on a scalable and practical hybrid type checking framework for existing low-level code. Deputy also differs from these systems in its handling of mutation.

Ou et al. [OTMW04] present a type system that splits a program into portions that are either dependently or simply typed, using run-time checks at the boundaries. Our type system uses run-time checks for safety everywhere and relies on an optimizer to handle statically verifiable cases. Ou et al. allow coercions between simply- and dependently-typed mutable references at the cost of a complex run-time representation for such references. In contrast, we focus on handling mutation of local variables and structure fields in the presence of dependencies.

Gradual typing [ST06] allows static and dynamic types to coexist in a functional or object-oriented language, using run-time checks where necessary for dynamic types. Although this system does not focus on handling dependent types, it presents another example of how compile-time checking and run-time checking can be combined effectively.

Of course, hybrid type checking is in many ways a very old idea. For example, Java's type system relies on run-time bounds checks and null checks for soundness, which means that Java's type system is a hybrid type system as well. Nevertheless, it is useful to identify type systems as hybrid type systems where relevant, since it is helpful to be explicit about when the soundness of a type system depends upon run-time checks that must be inserted by the compiler.

5.4 Type Qualifiers

CQual [FFA99] allows programmers to add custom qualifiers to C programs. Using simple inference rules, these qualifiers can be propagated throughout the program in order to check for common errors such as `const` violations and format string bugs. Deputy's dependent type annotations are similar in that they refine existing C types to facilitate additional verification. Deputy's types are more expressive, but they are correspondingly difficult to reason about.

More recently, Chin et al. presented semantic type qualifiers [CMM05], which can be used to extend a type system with qualifiers that can be proved sound in isolation. In contrast, Deputy's dependent types encapsulate somewhat subtle proofs of soundness that would be significantly more difficult to automate in this manner. As with CQual, this approach represents another possible trade-off between soundness and expressiveness.

Chapter 6

Conclusion

We have presented Deputy, a framework for introducing dependent type annotations into existing C programs. Deputy's goal is to allow C programmers to provide more precise information about existing invariants in their programs; for example, programmers can associate pointers to arrays with the length of the array. With the help of these annotations, we can perform compile-time and run-time checks to verify that an existing program conforms to the more detailed specification provided by these types.

The major advantage of using dependent types for this purpose is that programmers can now annotate *existing* metadata in *existing* programs. Previous approaches to this problem were either too costly in terms of performance or required changes to program data structures. The latter problem is a particularly tricky stumbling block, since these data structure changes make it impossible to transition existing software modularly or incrementally. With Deputy, however, we can annotate and check code at function granularity, which allows programmers to convert existing programs gradually.

With this incremental approach to annotating and checking existing software, we can scale up to extremely large programs, such as the Linux kernel itself. We have shown that Deputy can be used effectively on a 435,000-line version of the Linux kernel, including all of the most commonly-used kernel components. The only obstacle separating us from a fully annotated and checked kernel is the manpower to process the rest of the kernel code, which scales well with the size of the overall project. Of course, these safety gains require run-time checks, which come at a performance cost of up to 50% in terms of microbenchmark latency and bandwidth; however, these costs are significantly less noticeable in end-to-end performance tests. We believe that this cost is a reasonable one to pay for the return in safety, and we believe that this cost will decrease with future tuning of the code and with additional optimization.

More generally speaking, Deputy represents a successful use of sound program analysis in the context of a large software system. The key ingredients for this success were (a) the use of annotations that describe higher-level invariants that already exist in the code and (b) the use of run-time checks to reduce the burden on the programmer and the compiler. This combination allowed us to scale up to a large, real-world system like Linux. We believe that this approach represents a promising new area of research, both for software systems and for programming languages.

Appendix A

Soundness Proof

This chapter presents the details of our language and our soundness proof from Chapter 2. We present first the denotational semantics of our simplified imperative language, and then we present the proof of our main theorems.

A.1 Semantics

Recall that the state of execution is represented by $\rho = (\rho_E, \rho_S, \rho_A)$, where ρ_E : Var $\rightarrow$ Val gives the value of each variable, ρ_S : Addr $\rightarrow$ Val represents the contents of memory (i.e., the store), and ρ_A : Addr $\rightarrow \tau$ indicates the type of each memory location (i.e., the allocation state). In addition, we define a distinguished ρ_{fail} value to be the unique state resulting from a failed assertion.

First, we present the semantics for types. In our system, each type τ corresponds

to a set of values in state ρ, as follows:

$$\begin{aligned}
[\![\text{int}]\!]\rho &= \text{Val} \\
[\![\text{ref}]\!]\rho &= \lambda\tau.\{a \in \text{Dom}(\rho_A) | \tau = \rho_A(a)\} \\
[\![\tau_1\ \tau_2]\!]\rho &= ([\![\tau_1]\!]\rho)\ \tau_2 \\
[\![\tau\ e]\!]\rho &= ([\![\tau]\!]\rho)\ ([\![e]\!]\rho)
\end{aligned}$$

Second, we present the semantics for expressions:

$$\begin{aligned}
[\![x]\!]\rho &= \rho_E(x) \\
[\![n]\!]\rho &= n \\
[\![e_1\ \text{op}\ e_2]\!]\rho &= [\![e_1]\!]\rho\ \text{op}\ [\![e_2]\!]\rho \\
[\![*e]\!]\rho &= \rho_S([\![e]\!]\rho)
\end{aligned}$$

Finally, the semantics for commands:

$$\begin{aligned}
[\![\text{skip}]\!]\rho &= \rho \\
[\![c_1; c_2]\!]\rho &= [\![c_2]\!]([\![c_1]\!]\rho) \\
[\![x := e]\!]\rho &= (\rho_E[x \mapsto [\![e]\!]\rho], \rho_S, \rho_A) \\
[\![*e_1 := e_2]\!]\rho &= (\rho_E, \rho_S[[\![e_1]\!]\rho \mapsto [\![e_2]\!]\rho], \rho_A) \\
[\![\text{let } x : \tau = e \text{ in } c]\!]\rho &= [\![c]\!](\rho_E[x \mapsto [\![e]\!]\rho], \rho_S, \rho_A) \\
[\![\text{let } x = \text{new } \tau(e) \text{ in } c]\!]\rho &= [\![c]\!](\rho_E[x \mapsto a], \rho_S[a \mapsto [\![e]\!]\rho], \rho_A[a \mapsto \tau]) \\
&\quad \text{where } a \notin \text{Dom}(\rho_S)
\end{aligned}$$

A.2 Soundness

Now we present our main theorems and their proofs. We begin with a proof of soundness for expressions.

Theorem 1 (Soundness for expressions) *If* $\rho \models \Gamma$, $\Gamma \vdash e : \tau \Rightarrow \gamma$, *and* $\rho \models \gamma$, *then* $[\![e]\!]\rho \in [\![\tau]\!]\rho$.

Proof: We proceed by induction on the structure of the derivation of $\Gamma \vdash e : \tau \Rightarrow \gamma$. The cases are:

- Case 1: e is x

  The last step in the derivation must be (VAR). Thus $\tau = \Gamma(x)$. Since ρ is well-formed, $[\![x]\!]\rho \in [\![\Gamma(x)]\!]\rho = [\![\tau]\!]\rho$.

- Case 2: e is n

  The last step in the derivation must be (NUM). Thus $\tau = \text{int}$. Hence $n \in \text{Val} = [\![\text{int}]\!]\rho = [\![\tau]\!]\rho$.

- Case 3: e is e_1 op e_2

  The last step in the derivation must be (INT ARITH). Thus $\tau = \text{int}$. Since $\rho \models \gamma_1 \wedge \gamma_2$, $\rho \models \gamma_1$ and $\rho \models \gamma_2$. By induction, $[\![e_1]\!]\rho \in \text{Val}$ and $[\![e_2]\!]\rho \in \text{Val}$. Thus $[\![e_1]\!]\rho$ op $[\![e_2]\!]\rho \in \text{Val} = [\![\text{int}]\!]\rho = [\![\tau]\!]\rho$.

- Case 4: e is $*e'$

  The last step in the derivation must be (DEREF). Let $a = [\![e']\!]\rho$. By induction, $a = [\![e']\!]\rho \in [\![\text{ref } \tau]\!]\rho = \{a \in \text{Dom}(\rho_A) | \tau = \rho_A(a)\}$. Thus, $a \in \text{Dom}(\rho_A)$ and $[\![\tau]\!]\rho = [\![\rho_A(a)]\!]\rho$. Since ρ is well-formed, we have $[\![*e']\!]\rho = \rho_S(a) \in [\![\rho_A(a)]\!]\rho = [\![\tau]\!]\rho$, as desired.

Next, we provide a sketch of the proof for soundness of commands:

Theorem 2 (Soundness for commands) *If* $\rho \models \Gamma$, $\Gamma \vdash c \Rightarrow c'$, *then* $[\![c']\!]\rho = \rho'$ *and either* $\rho' = \rho_{fail}$ *or* $\rho' \models \Gamma$.

First, we must present several lemmas, the first of which is the following substitution lemma:

Lemma 1 (Substitution)

$$\begin{aligned} [\![e[^{e_x}\!/_x]]\!]\rho &= [\![e]\!](\rho_E[x \mapsto [\![e_x]\!]\rho], \rho_S, \rho_A) \\ [\![\tau[^{e_x}\!/_x]]\!]\rho &= [\![\tau]\!](\rho_E[x \mapsto [\![e_x]\!]\rho], \rho_S, \rho_A) \end{aligned}$$

Proof sketch: Both lemmas are proved by a trivial induction on the structure of e and τ. If additional constructors are added to the system, a sufficient condition to ensure that they do not invalidate this proof is to make their meaning independent of ρ_E.

Our second lemma says that local types depend only on the allocation state and the state of local variables, not on the store:

Lemma 2 *If* $\Gamma \vdash_L e :: \tau$, $\rho \models \Gamma$, *and* ρ_E *and* ρ_A *are restrictions of* ρ'_E *and* ρ'_A, *respectively, then then* $[\![e]\!]\rho = [\![e]\!]\rho'$.

Proof sketch: This lemma is a straightforward induction on the structure of e. Note that the rules for local expressions disallow dereference, which means that ρ_S and ρ'_S are irrelevant.

Lemma 3 *If* $\Gamma \vdash_L \tau ::$ type, $\rho \models \Gamma$, *and* ρ_E *and* ρ_A *are restrictions of* ρ'_E *and* ρ'_A, *respectively, then* $[\![\tau]\!]\rho \subseteq [\![\tau]\!]\rho'$.

Proof sketch: Any complete type τ must be formed by applying a constructor C to some sequence of zero or more expressions or types. The rules for evaluating types of the form $[\![\tau_1\ \tau_2]\!]\rho$ and $[\![\tau\ e]\!]\rho$, along with Lemma 2, ensure that $[\![\tau]\!]\rho$ and $[\![\tau]\!]\rho'$ evaluate to $[\![C]\!]\rho$ and $[\![C]\!]\rho'$ applied to the same sequence of arguments. Thus, as long as the set produced by $[\![C]\!]\rho$ applied to some sequence of arguments depends only on ρ_E and ρ_A, and as long as that set grows monotonically with additions to ρ_E and ρ_A, the required result holds. Currently, this properly holds for both constructors “int” and “ref”.

Now we present a proof sketch for Theorem 2. The proof is by induction on the derivation of $\Gamma \vdash c \Rightarrow c'$. We do a case split on the command c, as follows:

- Case 1: c is skip

  The last step in the derivation must be (SKIP). Thus $c' = \text{skip}$. Thus $\rho' = [\![\text{skip}]\!]\rho = \rho$, so $\rho' \models \Gamma$.

- Case 2: c is $c_1; c_2$

  The last step in the derivation must be (SEQ). Thus $c' = c_1'; c_2'$, where $\Gamma \vdash c_1 \Rightarrow c_1'$ and $\Gamma \vdash c_2 \Rightarrow c_2'$. Let $\rho' = [\![c_1']\!]\rho$ and let $\rho'' = [\![c_2']\!]\rho'$. By induction, $\rho' = \rho_{fail}$ or $\rho' \models \Gamma$. In the first case, $\rho'' = \rho_{fail}$ as well, and we're done. In the second case, we use induction again to get $\rho'' = \rho_{fail}$ or $\rho'' \models \Gamma$, as desired.

- Case 3: c is $x := e$

  The last step in the derivation must be (VAR WRITE). Thus $c' = \text{assert}(\wedge_{y \in \text{Dom}(\Gamma)} \gamma_y)$; $x := e$. If the assertion fails, then $[\![c']\!]\rho = \rho_{fail}$, and we're done. Thus, for the remainder of this argument, we assume that the guard condition holds in ρ and that $\rho' = [\![c]\!]\rho = (\rho_E[x \mapsto [\![e]\!]\rho], \rho_S, \rho_A)$. Since assignment does not change the store or allocation state, we obtain from the well-formedness of ρ the well-formedness of ρ'.

  Since $\rho \models \gamma_y$ for all guard conditions γ_y, we can apply Theorem 1 to the premises of the (VAR WRITE) rule. Thus for all $(y, \tau_y) \in \Gamma$, we have $[\![y[^e/_x]]\!]\rho \in [\![\tau[^e/_x]]\!]\rho$, and thus by Lemma 1 and the definition of ρ', $[\![y]\!]\rho' \in [\![\tau_y]\!]\rho'$. Therefore $\rho' \models \Gamma$.

- Case 4: c is $*e_1 := e_2$

The last step in the derivation must be (MEM WRITE). Thus $c' = \text{assert}(\gamma_1 \wedge \gamma_2); *e_1 := e_2$. If the assertion fails, then $[\![c']\!]\rho = \rho_{fail}$, so we're done. Thus, for the remainder of this argument, we assume that $\gamma_1 \wedge \gamma_2$ holds and that $\rho' = [\![c']\!]\rho = (\rho_E, \rho_S[[\![e_1]\!]\rho \mapsto [\![e_2]\!]\rho], \rho_A)$.

We now show that ρ' is well-formed. Let $a = [\![e_1]\!]\rho$. By applying Theorem 1 to the two premises of this rule, we get $a \in \text{Dom}(\rho_A)$ and $[\![e_2]\!]\rho \in [\![\rho_A(a)]\!]\rho$. Using this result and Lemma 3, we have $\rho'_S(a) = [\![e_2]\!]\rho \in [\![\rho_A(a)]\!]\rho = [\![\rho'_A(a)]\!]\rho \subseteq [\![\rho'_A(a)]\!]\rho'$. Then, using the fact that ρ is well-formed and Lemma 3, we can make a similar argument for the other elements of $\text{Dom}(\rho_A)$. Thus ρ' is well-formed.

To show that $\rho' \models \Gamma$, we must show that for all $x \in \text{Dom}(\Gamma)$, $[\![x]\!]\rho' \in [\![\Gamma(x)]\!]\rho'$. Using the fact that $\rho_E = \rho'_E$ and Lemma 3, $[\![x]\!]\rho' = [\![x]\!]\rho \in [\![\Gamma(x)]\!]\rho = [\![\Gamma(x)]\!]\rho'$.

- Case 5: c is let $x : \tau = e$ in c_1

  The last step in the derivation must be (LET). Thus $c' = \text{assert}(\gamma); \text{let } x : \tau = e \text{ in } c_1$. If the assertion fails, then $[\![c']\!]\rho = \rho_{fail}$, so we're done. Thus, for the remainder of this argument, we assume that γ holds and that $\rho'' = [\![c']\!]\rho = [\![c_1]\!]\rho'$, where $\rho' = (\rho_E[x \mapsto [\![e]\!]\rho], \rho_S, \rho_A)$.

  Since the guard γ is satisfied, we can apply Theorem 1 and Lemma 1 to get $[\![e]\!]\rho \in [\![\tau[^e/_x]]\!]\rho = [\![\tau]\!]\rho'$.

  Let $\Gamma' = \Gamma, x : \tau$. Because Γ is well-formed and $\Gamma' \vdash_L \tau :: \text{type}$, Γ' is well-formed. Also note that ρ' is well-formed, since ρ is well-formed, and ρ' differs from ρ only in a change to ρ_E.

Now we must show that $\rho' \models \Gamma'$. Since $x \notin \mathrm{Dom}(\Gamma)$ and $\rho \models \Gamma$, $\rho' \models \Gamma$. Thus it suffices to show that $[\![x]\!]\rho' \in [\![\tau]\!]\rho'$. Using the result in the previous paragraph, we have $[\![x]\!]\rho' = [\![e]\!]\rho \in [\![\tau]\!]\rho'$, as desired. Thus $\rho' \models \Gamma'$.

Since ρ' and Γ' are well-formed, and since $\rho' \models \Gamma'$, by induction we have $\rho'' \models \Gamma'$, where ρ'' was defined to be $[\![c']\!]\rho'$. Thus $\rho'' \models \Gamma$.

- Case 6: c is let $x = \text{new } \tau(e)$ in c_1

  The last step in the derivation must be (ALLOC). Thus $c' = \text{assert}(\gamma); \text{let } x = \text{new } \tau(e) \text{ in } c_1$. If the assertion fails, then $[\![c']\!]\rho = \rho_{fail}$, so we're done. Thus, for the remainder of this argument, we assume that γ holds and that $\rho'' = [\![c']\!]\rho = [\![c_1]\!]\rho'$, where $\rho' = (\rho_E[x \mapsto a], \rho_S[a \mapsto [\![e]\!]\rho], \rho_A[a \mapsto \tau])$ for some $a \notin \mathrm{Dom}(\rho_S)$.

  Since the guard γ is satisfied, we can apply Theorem 1 to get $[\![e]\!]\rho \in [\![\tau]\!]\rho$.

  Let $\Gamma' = \Gamma, x : \text{ref } \tau$. Because Γ is well-formed and $\emptyset \vdash_L \tau :: \text{type}$, Γ' is well-formed. To see that ρ' is well-formed, note that for all $b \neq a \in \mathrm{Dom}(\rho'_A)$, $\rho'_S(b) = \rho_S(b) \in [\![\rho_A(b)]\!]\rho = [\![\rho'_A(b)]\!]\rho \subseteq [\![\rho'_A(b)]\!]\rho'$ (the last step uses Lemma 3). For a itself, we have $\rho'_S(a) = [\![e]\!]\rho \in [\![\tau]\!]\rho = [\![\rho'_A(a)]\!]\rho \subseteq [\![\rho'_A(a)]\!]\rho'$ (again using Lemma 3). Thus ρ' is well-formed.

  Now we must show that $\rho' \models \Gamma'$. Since $x \notin \mathrm{Dom}(\Gamma)$ and $\rho \models \Gamma$, $\rho' \models \Gamma$. Thus it suffices to show that $[\![x]\!]\rho' \in [\![\text{ref } \tau]\!]\rho'$, which is equivalent to $a \in \{a \in \mathrm{Dom}(\rho'_A) | \tau = \rho'_A(a)\}$, which holds by definition of ρ'. Thus $\rho' \models \Gamma'$.

  Since ρ' and Γ' are well-formed, and since $\rho' \models \Gamma'$, by induction we have $\rho'' \models \Gamma'$, where ρ'' was defined to be $[\![c']\!]\rho'$. Thus $\rho'' \models \Gamma$.

Bibliography

[ABC+07] Zachary Anderson, Eric Brewer, Jeremy Condit, Rob Ennals, David Gay, Matthew Harren, George Necula, and Feng Zhou. Beyond bug-finding: Sound program analysis for Linux. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2007.

[ABEL05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control flow integrity: Principles, implementations, and applications. In *Computer and Communication Security*, 2005.

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Programming Language Design and Implementation (PLDI)*, 1994.

[And07] Zachary R. Anderson. Static analysis of C for hybrid type checking. Technical Report EECS-2007-1, UC Berkeley, 2007.

[Aug98] Lennart Augustsson. Cayenne—a language with dependent types. In *International Conference on Functional Programming (ICFP)*, 1998.

[Boe93] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Programming Language Design and Implementation (PLDI)*, 1993.

[BRS+85] Robert Baron, Richard Rashid, Ellen Siegel, Avadis Tevanian, and Michael Young. Mach-1: An operating environment for large-scale multiprocessor applications. *IEEE Software*, 2(4):65–67, July 1985.

[BS97] Aaron Brown and Margo Seltzer. Operating system benchmarking in the wake of Lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1997.

[CAE+07] Nathan Cooprider, William Archer, Eric Eide, David Gay, and John Regehr. Efficient Memory Safety for TinyOS. In *Embedded Network Sensor Systems (SenSys)*, 2007.

[Car96] Martin C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, June 1996.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In *European Symposium on Programming (ESOP)*, 2005.

[CER] CERT (United States Computer Emergency Readiness Team). Top 20 vulnerabilities by metric. `http://www.kb.cert.org/vuls/bymetric?open&start=1&count=20`.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[CHA+07] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. Dependent types for low-level programming. In *Lecture Notes in Computer Science: Programming Languages and Systems (ESOP Proceedings)*, volume 4421/2007, pages 520–535, 2007. Copyright 2007 Springer-Verlag. Portions of this publication are reproduced in this dissertation with kind permission of Springer Science and Business Media.

[Che02] Brian Chess. Improving computer security using extended static checking. In *IEEE Symposium on Security and Privacy*, 2002.

[CMM05] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Programming Language Design and Implementation (PLDI)*, 2005.

[CN05] Jeremy Condit and George C. Necula. Data slicing: Separating the heap into independent regions. In *Compiler Construction (CC)*, 2005.

[DA06] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *International Conference on Software Engineering (ICSE)*, 2006.

[DKAL05] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without garbage collection for embedded applications. *Transactions on Embedded Computing Systems*, 4(1):73–111, 2005.

[DRS03] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in c. In *Programming Language Design and Implementation (PLDI)*, 2003.

[EAV+06] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Operating System Design and Implementation (OSDI)*, Seattle, WA, 2006.

[EKJ95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating System Principles (SOSP)*, 1995.

[FFA99] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. A theory of type qualifiers. In *Programming Language Design and Implementation (PLDI)*, 1999.

[FFT06] Cormac Flanagan, Stephen N. Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL)*, 2006.

[FHN+04] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS)*, 2004.

[Fla06] Cormac Flanagan. Hybrid type checking. In *Principles of Programming Languages (POPL)*, 2006.

[GLvB+03] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, 2003.

[HDWY06] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *International Conference on Software Engineering (ICSE)*, 2006.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Logic in Computer Science (LICS)*, 1987.

[Hic96] Jason Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object-Oriented Languages (FOOL)*, 1996.

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX Winter Technical Conference*, 1992.

[HN05] Matthew Harren and George C. Necula. Using dependent types to certify the safety of assembly code. In *Static Analysis Symposium (SAS)*, 2005.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[JK97] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. *Automated and Analysis-Driven Debugging (AADEBUG)*, 1997.

[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*, 2002.

[LPMS97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, 1997.

[LUSG04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Operating System Design and Implementation (OSDI)*, 2004.

[Min02] Mindcraft, Inc. Webstone 2.5 benchmark, 2002. `http://www.mindcraft.com/webstone/`.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[NCH+05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *Transactions on Programming Languages and Systems (TOPLAS)*, 27(3), May 2005.

[NM05] Aleksandar Nanevski and Greg Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard University, 2005.

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. CIL: Intermediate language and tools for the analysis of C programs. In *Compiler Construction (CC)*, 2002.

[OTMW04] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP Conference on Theoretical Computer Science*, 2004.

[Pie02] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[Pie05] Benjamin Pierce, editor. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.

[RL04] Olatunji Ruwase and Monica Lam. A practical dynamic buffer overflow detector. In *Network and Distributed Systems Security (NDSS)*, 2004.

[SBL05] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *Transactions on Computer Systems*, 23(1):77–110, 2005.

[SPE95] SPEC (Standard Performance Evaluation Corporation). CPU95 Benchmark, 1995. `http://www.spec.org/cpu95/`.

[SPE99] SPEC (Standard Performance Evaluation Corporation). Web99 Benchmark, 1999. `http://www.spec.org/web99/`.

[SPE00] SPEC (Standard Performance Evaluation Corporation). CPU00 Benchmark, 2000. `http://www.spec.org/cpu00/`.

[ST06] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming*, 2006.

[WLAG94] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Symposium on Operating System Principles (SOSP)*, 1994.

[Xi00] Hongwei Xi. Imperative programming with dependent types. In *Logic in Computer Science (LICS)*, 2000.

[XP99] Howgwei Xi and Frank Pfenning. Dependent types in practical programming. In *Principles of Programming Languages (POPL)*, 1999.

[ZCA+06] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Operating System Design and Implementation (OSDI)*, 2006.