Dependent Types for Low-Level Programming

Jeremy Paul Condit
Matthew Thomas Harren
Zachary Ryan Anderson
David Gay
George Necula

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-129
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-129.html

October 13, 2006

Acknowledgement

Many thanks to Feng Zhou, Ilya Bagrak, Bill McCloskey, Rob Ennals, and Eric Brewer for their contributions to Deputy. This material is based upon work 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 Low-Level Programming

Jeremy Condit Matthew Harren Zachary Anderson David Gay[†] George C. Necula

University of California, Berkeley
{jcondit, matth, zra, necula}@cs.berkeley.edu

[†]Intel Research, Berkeley
david.e.gay@intel.com

Abstract

We describe the key principles of a flexible dependent type system for low-level imperative languages. Two major contributions are (1) a new typing rule for handling mutation that follows the model of Hoare's axiom for assignment and (2) a technique for automatically inferring dependent types for local variables. This type system is more expressive than previous dependent type systems because types can now depend on mutable variables; in addition, it improves ease of use by inferring dependent type annotations for local variables. Decidability is addressed by emitting run-time checks for those conditions that cannot be checked statically.

Using these principles, we have designed Deputy, a dependent type system for C whose types allow the programmer to describe several common idioms for the safe use of pointers and tagged unions. We have used Deputy to enforce memory safety properties in a number of common benchmark suites as well as in Linux device drivers and TinyOS [16] components. These experiments show that Deputy's dependent types are useful in a wide range of C programs and that they have a relatively low annotation burden and performance cost.

1. Introduction

Types provide a convenient and accessible mechanism for specifying program invariants. Dependent types extend simple types with the ability to express invariants relating multiple state elements. While such dependencies likely exist in all programs, they play a fundamental role in low-level programming. 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 variant type (as in a Pascal variant, or a C union) along with a tag that identifies the active variant. If we ignore such dependencies we cannot even prove the memory safety of most low-level programs.

In this paper we present a few general principles that enable the convenient use of dependent types in low-level programs. Specifically we consider and contribute solutions to the following challenges:

- Soundness: Mutation of variables or heap locations, used heavily low-level programs, might invalidate the types of some state elements. Previous dependent type systems include restrictions to ensure that mutation never affects types [1, 28, 29]. 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.
- Decidability: Dependent type checking involves reasoning about the run-time values of expressions. In most previous dependent type systems, dependencies are restricted to the point where all checking can be done statically. Instead we allow most dependencies that can be expressed in the expression language, and we propose the use of run-time checks where static checking is not sufficient. This hybrid type-checking strategy has also been proposed recently by Flanagan [10].
- Usability: Writing complete dependent type declarations can be a considerable burden. We describe a technique for automatic dependency inference for local variables, starting from existing declarations for global variables, data structures, and functions.

We have applied these general principles for low-level dependent types to create the Deputy type system for the C programming language. Deputy provides a set of dependent types that allow programmers to safely express common C programming idioms, most notably those involving pointer arithmetic and union types. Previous approaches to safe C involved significant changes in the program's data representation in order to add metadata for checking purposes: certain pointers were given a "fat" representation that includes the pointer and its bounds, and tags were added to union values [17, 21]. These data representation changes can introduce new race conditions, and they make it very hard to interface with external libraries or to process one function or module at a time. Until now there was no satisfactory solution to this issue, although partial solutions have been proposed [7, 18, 24]. To solve this problem, we observe that most of the metadata required for checking purposes already exists in the program. Using Deputy's dependent types, the programmer can identify this preexisting metadata, allowing the compiler to check the code without changing data representations. In this paper, we show experimental evidence that this approach is applicable to a wide range of C programs, with relatively low annotation burden and performance penalty.

In Section 2, we present a high-level preview of the main stages in the Deputy system for safe low-level programming, using a simple example. Section 3 contains the technical core of the paper, describing our dependent types and our inference technique for a core imperative language with references. Of particular importance are Section 3.2, which presents the general principles for handling mutation, and Section 3.5, which describes our scheme for inferring dependencies automatically. Then, Section 4 shows how these general principles can be applied to provide safe handling of C's pointer arithmetic and union types. In Section 5 we describe our experience using Deputy for a range of C programs. Finally, we discuss related work in Section 6.

2. Deputy Overview

In order to provide an intuition for the formal development starting in the next section, we present here an example showing the operation of Deputy on a simple C program. The goal is to demonstrate the capabilities of Deputy while deferring the details of how it works to later sections.

In Figure 1, we show the source code for a program that returns the sum of the elements between `buf` and `end` in an array of integers. Underlined and italicized code indicates annotations and

```
1  int sum (int * count(end - buf) buf, int * end) {
2    int sum = 0;
3    while(buf < end) {
4      assert(0 < end - buf);
5      sum += * buf;
6
7      int tmplen = (end - buf) - 1;
8      assert(0 <= 1 <= end - buf);
9      int * count(tmplen) tmp = buf + 1;
10
11     assert(0 <= end - tmp <= tmplen);
12     buf = tmp;
13   }
14   return sum;
15 }
```

Figure 1. A simple Deputy program, along with the instructions and annotations added during automatic dependency inference (underlined) and the assertions added during type checking (in italics).

run-time checks inserted automatically by Deputy. The body of the loop is an expanded version of the more compact statement "`sum += * buf ++`". We have introduced the `tmp` temporary variable so that we can consider separately the increment of `buf` and the assignment of the result back to `buf`; however, this temporary variable is not required.

The example input is the program shown in Figure 1 without any of the statements shown underlined or in italics. This program is standard C, with one programmer-supplied annotation for the type of the `buf` formal argument. The annotated type `int * count(end - buf)` describes a pointer into an array of at least `end - buf` integers. Implicitly, this annotation also specifies that `end - buf` is non-negative.

Deputy operates in several passes over the program:

Pass 1: Automatic dependency generation. In order to reduce the annotation burden, Deputy requires fully-specified types only for globals, data structures, and function types. In this first pass, Deputy creates new variables for the missing dependencies in the types of local variables. In Figure 1, the underlined code is the code added in this pass; here, Deputy has created the `tmplen` local variable in order to complete the type of the `tmp` variable declared in line 9. The newly created variables are updated whenever the variable for which they were created is updated; thus, the initialization of `tmplen` in line 7 corresponds to the initialization of `tmp` in line 9. The value used to initialize `tmplen` reflects the fact that the array "`buf + 1`" (the initial value of `tmp`) has one fewer element than `buf`. Section 3.5 gives the precise description of this pass, independent of the actual dependent types used.

Pass 2: Flow-insensitive type checking and instrumentation. Once all the variables have complete types, Deputy checks each instruction separately. Most of the type checking is done statically, but all of the checks that involve reasoning about run-time values of expressions are emitted as run-time assertions. In Figure 1, we show in italics the assertions added during type checking for our example. The assertion in line 4 ensures that the `buf` array is nonempty and can therefore be safely dereferenced. The assertion in line 8 ensures that the array is being incremented by a positive number that is less than or equal to the size of the array.

The assignment in line 12 exposes 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. The intuition behind the check in line 11 is that we must ensure that *after* the assignment `buf` points to at least `end - buf` elements (as required by its type), yet we know that before the assignment `tmp` points to at least `tmplen` elements. Section 3.2 describes the type checking and instrumentation pass.

Pass 3: Flow-sensitive optimization of checks. Because our flow-insensitive type checker emits many redundant checks, we follow our flow-insensitive type checker with a flow-sensitive data-flow analysis for detecting trivial and redundant checks. We do not describe in this paper any particular analysis. In our implementation we use a global data-flow analysis for copy propagation and linear machine arithmetic. In the example shown in Figure 1 even a simple analysis can eliminate all the assertions introduced during type checking. A good analysis during optimization not only results in faster code but also allows us to detect statically checks that are guaranteed to fail. Most previous dependent type systems restrict the dependencies such that all assertions fall within theories with simple decision procedures, rejecting programs with assertions that cannot be discharged statically. In Deputy we expect that some assertions will remain in the program even after optimization, either because they use operators for which we do not have efficient decision procedures or because the programmer did not fully specify all the necessary invariants.

We can also use this example program to contrast Deputy with safe C type systems based on fat pointers [17, 21]. In such systems, the formal argument `buf` and the local `tmp` might be represented as two-word structures carrying the pointer and its length. In contrast, Deputy requires no changes to the representation of pointers and can therefore process the function `sum` without processing the call sites. This approach represents a crucial advantage over other tools, because it allows the programmer to use an incremental and modular strategy for porting existing code, starting with the more error-prone modules first. Another advantage of the dependent type strategy is that the metadata used in assertions is written in terms of the data that the program itself manipulates, thus allowing the optimizer to take advantage of the checks already present in the original program. In contrast, if we use a fat pointer representation for `buf` in our example, the optimizer would not be able to take advantage of the conditional in line 3 to eliminate the assertion for the dereference of `buf`. These benefits have allowed us to apply Deputy incrementally to modular software such as Linux device drivers and TinyOS components, as described in Section 5.

Interestingly, the automatic dependency generation feature of Deputy introduces a fat representation for locals, whereas for data structures, function parameters, and globals, the programmer must specify how to compute the metadata from data already existing in the program. Only if this metadata is absent, or not present in a fashion usable by Deputy, must the programmer change the original code. Using dependent types for globals along with automatic dependency generation for locals appears to be a good compromise in terms of usability and annotation burden, as we discuss further in our experimental results (Section 5).

3. Dependent Type Framework

In this section, we present the key principles of our dependent type system in terms of a small imperative language. Specifically, we show how to handle mutation in the presence of dependent types and how we use hybrid type checking to allow more expressive dependent types. Finally, we show how dependent types can be generated automatically. None of this material is specific to C.

We present our type system in several stages. We start with a simple language that demonstrates the mutation rule alone, and we state our main soundness theorem for this core language. Then we extend the language and type system with a rule for parallel assignment, which in turn allows us to present our approach to automatically generating the dependent types for local variables.

Constructors	C	::=	int | ref | ...
Types	τ	::=	$C \mid \tau_1\ \tau_2 \mid \tau\ e$
Kinds	κ	::=	type $\mid$ type $\to \kappa \mid \tau \to \kappa$
L-expressions	ℓ	::=	$x \mid *e$
Expressions	e	::=	$n \mid \ell \mid e_1$ op e_2
Commands	c	::=	skip $\mid c_1; c_2 \mid \ell := e \mid$ assert(γ) $\mid$ let $x : \tau = e$ in $c \mid$ let $x =$ new $\tau(e)$ in c
Predicates	γ	::=	e_1 comp $e_2 \mid$ true $\mid \gamma_1 \wedge \gamma_2$

$x, y \in$ Variables $\quad n \in$ Integer constants

op $\in$ Binary operators $\quad$ comp $\in$ Comparison operators

Figure 2. The grammar for a simple dependently-typed language.

Finally, we extend the type system with support for dependently-typed structures and function calls.

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. In this language, types are specified using type constructors applied to some number of other types or expressions. Constructors can be viewed as type families indexed by other 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 types, or to expressions of a certain type, to eventually form complete types. The kind of each constructor is given by the mapping "kind"; so far, this mapping is defined as follows:

$$\begin{array}{rcl} \text{kind(int)} & = & \text{type} \\ \text{kind(ref)} & = & \text{type} \to \text{type} \end{array}$$

To show how this system can be extended with additional type constructors, consider the `count` annotation used in Figure 1. To represent this type, we can introduce the constructor "array" with kind type $\to$ int $\to$ type, such that "array τ e_{len}" is the type of arrays of elements of type τ and length e_{len}. In the concrete syntax this type is written as "τ `* 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 to Deputy. Finally, note that we omit loops and conditionals, which are irrelevant to our flow-insensitive type system.

3.2 Type Rules

In this section, we present the type rules for the core language. Figure 3 shows these rules and summarizes the judgment forms involved.

Our strategy for handling mutation in the presence of dependent types relies on 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 might be affected by each mutation operation. We restrict types to contain only expressions formed us-

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

(TYPE CTOR)
$$\frac{}{\Gamma \vdash_L C :: \text{kind}(C)}$$

(TYPE EXP)
$$\frac{\Gamma \vdash_L \tau :: (\tau' \to \kappa) \qquad \Gamma \vdash_L e : \tau'}{\Gamma \vdash_L \tau\ e :: \kappa}$$

(TYPE TYPE)
$$\frac{\Gamma \vdash_L \tau_1 :: (\text{type} \to \kappa) \qquad \emptyset \vdash_L \tau_2 :: \text{type}}{\Gamma \vdash_L \tau_1\ \tau_2 :: \kappa}$$

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

(LOCAL NAME)
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau}$$

(LOCAL NUM)
$$\frac{}{\Gamma \vdash_L n : \text{int}}$$

(LOCAL INT ARITH)
$$\frac{\Gamma \vdash_L e_1 : \text{int} \qquad \Gamma \vdash_L e_2 : \text{int}}{\Gamma \vdash_L e_1 \text{ op } e_2 : \text{int}}$$

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

(VAR)
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \Rightarrow \text{true}}$$

(NUM)
$$\frac{}{\Gamma \vdash n : \text{int} \Rightarrow \text{true}}$$

(INT ARITH)
$$\frac{\Gamma \vdash e_1 : \text{int} \Rightarrow \gamma_1 \qquad \Gamma \vdash e_2 : \text{int} \Rightarrow \gamma_2}{\Gamma \vdash e_1 \text{ op } e_2 : \text{int} \Rightarrow \gamma_1 \wedge \gamma_2}$$

(DEREF)
$$\frac{\Gamma \vdash e : \text{ref } \tau \Rightarrow \gamma}{\Gamma \vdash *e : \tau \Rightarrow \gamma}$$

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

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

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

Figure 3. The four judgments used by Deputy's type system and the core type checking rules for each.

ing constants, local variables, and arbitrary arithmetic operators. In other words, we do not allow memory dereferences in types.[1] 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.

First, we consider the well-formedness rules for types. Let Γ be 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, and we write $\Gamma \vdash_L \tau :: \kappa$ to indicate that in Γ, τ is a local, well-formed type with kind κ. The type rules for this judgment are shown at the top of Figure 3 and are mostly standard. One notable requirement is that type arguments must be well-formed in the empty environment, as shown in rule (TYPE TYPE), whereas expression arguments must be well-typed in Γ, as shown in rule (TYPE EXP). This conservative 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 also provide a set of rules for local expressions. The judgment $\Gamma \vdash_L e : \tau$ says that in environment Γ, e is a local, well-typed expression with type τ. The rules for local expressions are standard.

We have a separate judgment for general (i.e., non-local) expressions. This judgment, written $\Gamma \vdash e : \tau \Rightarrow \gamma$, says that in environment Γ, if γ is satisfied at run time, then e is a well-typed expression with type τ. The condition γ is a boolean predicate that must hold in order for the judgment to be valid. This predicate is generated during type checking and will be inserted in the instrumented program as run-time checks unless it can be discharged statically.

The rules presented in Figure 3 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 before, we might add new typing rules for pointer arithmetic and dereference:

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

These rules are responsible for the assertions generated in line 4 and line 8 in Figure 1. 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. We might also add a coercion rule, allowing long arrays to be used where shorter arrays are expected:

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

In our implementation, we ensure that type checking is syntax-directed by invoking the coercion rules only from the rules for commands, which are discussed below.

The judgment for checking commands is responsible for inserting into the program the guard conditions generated by the expression rules. This judgment, written $\Gamma \vdash c \Rightarrow c'$, says that in environment Γ, command c is compiled to command c'. These two commands have identical semantics; however, c' contains assertions with the guard conditions necessary to execute safely.

The first typing rule for commands, (VAR WRITE), is responsible for updates to variables in the presence of dependent types and is one of the main contributions 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 upon the Hoare axiom for assignment, which says that for a given predicate ϕ, the weakest precondition of the command $x := e$ with respect to ϕ is $\phi[^{e}/_{x}]$. If we view the type environment Γ as a predicate on the state of the program, the (VAR WRITE) rule states that the predicate "Γ" implies the weakest precondition of $x := e$ with respect to "Γ". Section 3.3 makes this intuition more precise.

To understand this rule in more detail, consider the array example once again. Suppose we have the following code:

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

In this example, we have an integer variable n and an array variable a with length n. Decrementing n should be safe as long as $n > 0$, 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 $\Gamma \vdash n[^{n-1}/_{n}] : \text{int}[^{n-1}/_{n}] \Rightarrow \gamma_n$ and $\Gamma \vdash a[^{n-1}/_{n}] : (\text{array int } n)[^{n-1}/_{n}] \Rightarrow \gamma_a$. 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 $\gamma_a = 0 \le n-1 \le n$. After static optimization, this check can be reduced to $0 \le n - 1$, which is precisely the check we expected in the informal discussion above.[2]

As a second example, consider line 12 in Figure 1. Here `buf` has type "array int (`end` − `buf`)", so the (VAR WRITE) rule requires $\Gamma \vdash \texttt{buf}[^{\texttt{tmp}}/_{\texttt{buf}}] : \text{array int } (\texttt{end} - \texttt{buf})[^{\texttt{tmp}}/_{\texttt{buf}}] \Rightarrow \gamma_{\texttt{buf}}$, or $\Gamma \vdash \texttt{tmp} : \text{array int } (\texttt{end} - \texttt{tmp}) \Rightarrow \gamma_{\texttt{buf}}$. Applying the (ARRAY COERCE) rule gives $\gamma_{\texttt{buf}}$ = "$0 \le \texttt{end} - \texttt{tmp} \le \texttt{tmplen}$", as seen on line 11.

Generally speaking, the (VAR WRITE) rule allows us to verify that dependencies in the local environment have not been broken, and the local-type restriction on base types of pointers ensures that there are no dependencies from the heap. In short, 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 comparatively simple. The (MEM WRITE) rule requires no reasoning about dependencies because the well-formedness rule for reference types requires that the contents of a reference be independent of its environment. The (LET) rule is standard except for the substitution when checking the expression e; this substitution follows similar logic to the assignment rule, allowing for the possibility of self-dependencies. (Note that we need not check the rest of the environment, since none of the variables in Γ can depend on the new variable x.) Finally, the (ALLOC) rule resembles the (LET) rule but with no substitution required thanks to the restrictions on reference types.

[1] In the full version of Deputy for C, local expressions also exclude function calls, references to fields of other structures, and variables whose address is taken.

[2] We take care to account for possible overflow of machine arithmetic. However, this is not hard when reasoning about array indices that must be bound by the length of an array.

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). 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:

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

In particular, note that each constructor C must have some meaning given by $[\![C]\!]\rho$. If additional constructors are added, the proof requires that their meanings 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 allows us to view the type environment as a predicate on the state of the program, which is essential for the adequacy of using Hoare's assignment axiom for type checking.

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

We say that $\rho \models \Gamma$ if ρ and Γ are well-formed and if for all $x \in \text{Dom}(\Gamma)$, $[\![x]\!]\rho \in [\![\Gamma(x)]\!]\rho$. 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. A sketch of the proof is shown 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* $[\![c]\!]\rho = \rho'$.

3.4 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 the command $x_1, \ldots, x_n := e_1, \ldots, e_n$. 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 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)

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

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.

Although parallel assignment syntax is unavailable in C, we can gain some of the benefit of this construct by automatically grouping certain adjacent assignments into parallel assignments, provided that the proper independence relationships hold. In addition, parallel assignments are frequently introduced by the automatic dependency transformation, which is discussed in the following section.

3.5 Automatic Dependencies

Until now, we have presented the Deputy 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. As described in Section 2, this feature operates as a preprocessing step before type checking.

We allow local variables to omit expressions in their dependent types. For example, a variable 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, in Figure 1, we added `tmplen` to track the length of `tmp`, updating it as appropriate. Note that dependent types with expressions omitted will have an *incomplete type* whose kind is $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow$ type for $n \geq 1$.

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 define 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' computes the same result as c.

The interesting rules for deriving this judgment are given in Figure 4. In the (AUTO LET) rule, we add new variables to track any missing dependencies for x. We initialize these fresh variables using the dependencies of the initial value e, and we record in Δ the fact that variables x_1 through x_n store x's dependencies. To determine the number and types of the automatic variables

(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 4. Rules for automatic dependencies.

x_1 through x_n, we look at the kind of the incomplete type. For example, the incomplete type "array τ" has kind int $\to$ type, so for arrays we will add a single automatic variable of type int.

In the (AUTO VAR WRITE) rule, we handle assignment to a variable that may have automatic dependencies. Using parallel assignment, we update all of the automatic variables associated with x. Note that we invoke the type checking judgment only for the purpose of recovering the expressions appearing in the type of the right-hand side of the assignment.

For example, consider the following code fragment in which x has an incomplete type:

$$\begin{array}{l} \text{let } a1 : \text{array int } n1 = \ldots \text{ in} \\ \text{let } a2 : \text{array int } n2 = \ldots \text{ in} \\ \text{let } x : \text{array int } \ = a1 \text{ in} \\ \text{if } (\ldots) \text{ then } x := a2; \\ *(x + 3) := 0; \end{array}$$

On the last line, Deputy needs to know the length of array x so that it can insert the appropriate check. One solution would be a static analysis to determine the length of x, but this analysis is difficult since x could point to either $a1$ or $a2$. Instead, our preprocessor inserts the new variable nx to track x's length dynamically. Here is the segment after processing by the automatic dependency step, with the new code underlined:

$$\begin{array}{l} \text{let } a1 : \text{array int } n1 = \ldots \text{ in} \\ \text{let } a2 : \text{array int } n2 = \ldots \text{ in} \\ \underline{\text{let } nx : \text{int} = n1 \text{ in}} \\ \text{let } x : \text{array int } \underline{nx} = a1 \text{ in} \\ \text{if } (\ldots) \text{ then } x,\ \underline{nx} := a2,\ \underline{n2}; \\ *(x + 3) := 0; \end{array}$$

Now the assertion needed on the last line is simply $0 \leq 3 < nx$.

Although this transformation can potentially insert many new variables into the program, these variables can often be eliminated by the optimizer. Copy propagation is a particularly useful optimization here, since it can eliminate automatic variables in the common case when they are only assigned once. At the same time, Deputy's automatic dependencies are general enough to handle situations such as the example above, where static analysis would be hard. Separating preprocessing from optimization allows us to keep the automatic dependency transformation simple and powerful, while still taking advantage of static analysis where possible.

This transformation also 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.

(TYPE STRUCT)

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

(STRUCT LITERAL)

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

(STRUCT READ)

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

(STRUCT WRITE)

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

Figure 5. Structure type checking rules.

3.6 Structures

So far, we have presented a type system that supports dependencies among mutable local variables. Our type annotations also provide a natural way to express dependencies among structure fields. The strategy used here is the same as the strategy used with local variables: by restricting the scope of dependencies, we ensure that when a value is modified, the compiler can enumerate all of the locations that might depend on that value.

To add structures to our language, we extend the grammar for types, l-expressions, and expressions:

$$\begin{array}{lll} \tau & ::= & \ldots \mid \text{struct } \{f_1 : \tau_1;\ \ldots\ f_n : \tau_n\} \\ \ell & ::= & \ldots \mid \ell.f \\ e & ::= & \ldots \mid \{f_1 = e_1; \ldots; f_n = e_n\} \end{array}$$

The new type "struct $\{f_1 : \tau_1;\ \ldots\ f_n : \tau_n\}$" defines a mutable record type in which the i^{th} field has label f_i and type τ_i. The l-expression $\ell.f$ accesses a structure field with name f. The expression $\{f_1 = e_1; \ldots; f_n = e_n\}$ is a structure literal that initializes field f_i to expression e_i.

Field types can depend on other fields of the same structure, but we maintain the invariant that structure fields do not depend on, and are not depended on by, any value outside of the structure. We use the field names as variables to express dependencies between fields, as in the following declaration:

$$y\ :\ \text{struct } \{f_1 : \text{array int } (f_2 + 1);\ f_2 : \text{int}\}$$

Here the field f_1 contains an array whose length is one greater than the value of field f_2.

Figure 5 shows the rules for type checking structures. The (TYPE STRUCT) rule describes well-formed structure types. When checking that field types are well-formed, the environment contains the names and types of the fields in the current structure. Thus, field types can depend only on other fields in the same structure.

The (STRUCT LITERAL) rule shows how to check a structure literal, and the (STRUCT READ) rule shows how to type check field reads. Whenever a field name appears in the type being read, we replace it with an expression denoting the corresponding field of the structure being accessed. Using the example above, a read from $y.f_1$ would have type "array int $(y.f_2 + 1)$".

(FUNCTION DEFINITION)

$$\frac{\begin{array}{c}\text{for all } 1 \le i \le n,\ \ (x_1 : \tau_1,\ \ldots,\ x_n : \tau_n) \vdash_L \tau_i :: \text{type} \\ (x_1 : \tau_1,\ \ldots,\ x_n : \tau_n) \vdash_L \tau_{ret} :: \text{type} \\ \text{all paths in } c \text{ end with a return} \qquad x_1', \ldots, x_n' \text{ not used in } c \\ \Gamma = (x_1 : \tau_1,\ \ldots,\ x_n : \tau_n) \qquad F = (x_1 : x_1',\ \ldots,\ x_n : x_n') \\ \Gamma; F \vdash c \Rightarrow c'\end{array}}{\begin{array}{l}\vdash fn(x_1 : \tau_1,\ \ldots,\ x_n : \tau_n) : \tau_{ret} = c \ \Rightarrow \\ \quad fn(x_1 : \tau_1,\ \ldots,\ x_n : \tau_n) : \tau_{ret} = \\ \quad\ \text{let } x_1' : \tau_1 = x_1 \text{ in } \ldots \text{let } x_n' : \tau_n = x_n \text{ in } c'\end{array}}$$

(FUNCTION RETURN)

$$\frac{\begin{array}{c}\text{the current function has signature } (x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau_{ret} \\ \Gamma \vdash e : \tau_{ret} \left[F(x_i) / x_i \right] \Rightarrow \gamma\end{array}}{\Gamma; F \vdash \text{return } e \ \Rightarrow\ \text{assert}(\gamma);\ \text{return } e}$$

(FUNCTION CALL)

$$\frac{\begin{array}{c}fn \text{ has signature } (x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau_{ret} \\ \text{for all } 1 \le i \le n,\ \Gamma \vdash e_i : \tau_i \left[e_j / x_j \right]_{1 \le j \le n} \Rightarrow \gamma_i \\ x \notin \text{Dom}(\Gamma) \qquad \tau_x = \tau_{ret} \left[e_j / x_j \right]_{1 \le j \le n} \\ \Gamma \vdash_L \tau_x :: \text{type} \qquad \Gamma, x : \tau_x; F \vdash c \Rightarrow c'\end{array}}{\begin{array}{l}\Gamma; F \vdash \text{let } x = fn(e_1, \ldots, e_n) \text{ in } c \ \Rightarrow \\ \quad \text{assert}(\bigwedge_{1 \le i \le n} \gamma_i);\ \text{let } x = fn(e_1, \ldots, e_n) \text{ in } c'\end{array}}$$

Figure 6. Rules for type checking function definitions and calls.

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.

3.7 Function Calls

Deputy also supports dependent function types. As with local variables, the types of formal parameters can depend on other formals. The return type is also allowed to depend on the formals. To support functions calls we introduce the command "let $x = fn(e_1, \ldots, e_n)$ in c" for function call and "return e" for function return.

Figure 6 extends our type system with functions. Consider the (FUNCTION DEFINITION) rule, which shows the compilation of a function named fn. Return types can depend on the initial values of the parameters, so we must save a copy of these values before executing the function body, which could modify the formal variables. We change the environment of our command judgment to add a map F from formal variables to the initial values of those formals. This map is only used by the return command; all other commands pass it along unchanged.

The (FUNCTION CALL) rule shows how to check a function call. The arguments to the function are type checked according to the specified type signature, and the return type depends on the actual values of the arguments. Notice the similarity between type checking a function call and type checking a parallel assignment.

We require that the result of the function call be placed in a fresh variable x, rather than in an existing variable. We do not consider function calls to be expressions because the (VAR WRITE) rule requires that expressions be pure. In our implementation, we automatically create a fresh temporary to store the result of each function call.

4. Dependent Types for C

We have applied the techniques described in the previous section to check various type safety properties of annotated C programs. Section 4.1 presents a dependent type constructor that assures pointer arithmetic is done correctly. This type constructor is a generalization of the array example we presented earlier. Section 4.2 discusses how Deputy can enforce safe usage of unions in C using the same techniques, and Section 4.3 reviews other issue related to the C language.

(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)

$$\frac{\Gamma \vdash e_1 : \text{ptr } \tau\ lo\ hi \Rightarrow \gamma_1 \qquad \Gamma \vdash e_2 : \text{int} \Rightarrow \gamma_2}{\begin{array}{l}\Gamma \vdash e_1 \oplus e_2 : \text{ptr } \tau\ lo\ hi \ \Rightarrow \\ \quad \gamma_1 \wedge \gamma_2 \wedge (e_1 \neq 0) \wedge (lo \le (e_1 \oplus e_2) \le hi)\end{array}}$$

(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)

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

(PTR ALLOC)

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

(INT COERCION)

$$\frac{\Gamma \vdash e : \text{ptr } \tau\ lo\ hi \Rightarrow \gamma}{\Gamma \vdash e : \text{int} \Rightarrow \gamma}$$

(NULL COERCION)

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

(PTR COERCION)

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

Figure 7. Bounded pointer type checking rules.

4.1 Pointer Bounds

This section describes the type constructor that Deputy uses to support pointer arithmetic. We generalize the array example in the previous section by using dependent types to denote the lower and upper bounds of a memory region and by allowing arbitrary pointer arithmetic within this region.

We add to our language a constructor "ptr", where kind(ptr) = type $\to$ int $\to$ int $\to$ type. The type "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. We also 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.

$$e ::= \ldots \mid e_1 \oplus e_2$$

The $\oplus$ operator may be used in local expressions.

Using the ptr type, we can write the following examples of Deputy's bounded pointer types:

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

In our actual implementation, Deputy represents these types using C's pointer syntax, with the pointer bounds appearing as annotations on pointer types. Deputy uses the type "`int * bound(a,b)`" to mean "ptr int a b". Also, we provide syntactic sugar for common annotations; for example, one could write the second declaration above as "`int * count(n) x`". (This `count` annotation is a slight modification of the one presented in the overview.) For unannotated pointers, we use the automatic dependency technique described in Section 3.5.

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 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. Such pointers are needed for several common programming idioms.

Figure 7 show 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.

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 is trivial, since the first premise invokes the (PTR DEREF) rule to verify that e_1 is safe to dereference.

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.

The (INT COERCION), (NULL COERCION), and (PTR COERCION) rules allow coercions to and from pointer types. The first rule is trivial: any pointer can be safely considered an integer. The second rule 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. Also note that a similar rule could be added for converting zero integers to a pointer type with arbitrary bounds. Finally, the pointer 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's 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.

This dependent pointer type is flexible enough to express a wide variety of common C idioms for tracking pointer bounds. In Section 5, we will discuss our experience using this dependent type to annotate real-world code.

One common idiom not covered by this type is a null-terminated pointer. Although we omit the details from this paper for reasons of space, our implementation provides a variant of this bounded pointer that allows for null-terminated pointers. These pointers have a null-terminated sequence of elements *above* the upper bound, and the rules for these pointers are updated to allow access to these elements. Coercions are provided to convert between the two bounded pointer types when possible.

4.2 Dependent Union Types

Deputy's dependent type system can also be used to ensure safety for C union types. Unions are used frequently in C code to customize the use of memory areas. To ensure that 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. Union fields are accessed using the $u.f_i$ syntax, where f_i indicates the i^{th} field of the union. We also add a union literal $\{f_i = e\}$, which initializes the field f_i of the union to expression e. For simplicity, we do not have special syntax for writing to a field of a union; rather, unions are updated by overwriting the union with a new union literal.

An example of a union type is as follows:

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

In this example, 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.

The following invariant describes how unions are handled in Deputy.

(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)

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

Figure 8. Typing rules for tagged unions.

DEFINITION 2 (Union types). *If a value v has type $\text{union}_n\ \tau_1\ \ldots\ \tau_n\ e_1\ \ldots\ e_n$, 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.*

In Figure 8, we show the rules for type checking dependent unions. In this case, we have only two rules, one governing union literals and one governing union access. Note that well-formedness of union types is handled by the general rules given in the core type system.

The (UNION LITERAL) rule shows how union literals are coerced to a certain union type. The guard condition verifies that the selector expressions have been set properly for the desired union field. The (UNION ACCESS) rule shows that a field can be accessed if its selector is nonzero.

When updating a union, we use the standard $l := e$ syntax where e will usually be a union literal. Parallel assignment is often helpful in this case for setting the tag of a union along with the value of the union itself.

4.3 Other C Language Constructs

In addition to the dependent type constructors for bounded pointers and tagged unions, we support null-terminated arrays, taking the address of structure fields and variables (so long as there are no dependencies between those fields or variables and other nearby fields or variables), and limited use of `printf`-style variable argument functions.

The following sources of unsoundness are not currently handled by Deputy:

- *Deallocation.* Currently, we trust that Deputy programs will use `malloc` and `free` correctly. This problem is orthogonal to the problem of checking bounded pointers and tagged unions, and it could be solved by use of a garbage collector [4] or other methods [9]. 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.
- *Non-`printf`-style variable-argument functions.* We do not check variable-argument function calls unless they use the `printf` convention.
- *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.
- *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.

Deputy does not provide special support for multithreaded code; however, if all values with dependent types are correctly synchronized in the original code, the resulting program will be thread-safe as well.

Deputy supports separate compilation, and the object files it produces can be linked with object files produced by other C compilers. As mentioned earlier, this feature makes it easy to apply Deputy to a program incrementally by annotating one file at a time.

5. Experiments

We implemented Deputy using the CIL infrastructure [22]. Our implementation is 20,587 lines of OCaml code, not counting the CIL front-end. Our implementation uses the type system described in this paper to ensure partial safety for C programs. Given an annotated program, Deputy creates automatic bounds variables where needed, adds assertions as dictated by the type rules, and uses several flow-sensitive, intraprocedural optimization passes to remove those assertions. The resulting program is then emitted as C code and compiled by `gcc`.

To test Deputy we annotated a number of standard benchmarks, including Olden, Ptrdist, and selected tests from the SPEC CPU and MediaBench suites [2, 5, 19, 25], as seen in Table 1. For each test, we replaced `gcc` with Deputy when compiling the program. Deputy emits errors whenever the program has not been correctly annotated; these annotations must be added by hand. Once the program compiles, the programmer typically needs to add or modify annotations in order to eliminate run-time errors that are due to insufficient or incorrect annotations.

We also used Deputy to enforce type safety in version 2 of the TinyOS [16] sensor network operating system. We tested three simple demo applications: periodic LED blinking (Blink), forwarding radio packets to and from a PC (BaseStation), and simple periodic data acquisition (Oscilloscope). TinyOS is written in a C dialect (nesC [13]) that is normally compiled to C and then fed to `gcc`. As with the other benchmarks, we replace this last step by Deputy. TinyOS is concurrent, but the nesC compiler enforces the use of atomic statements to protect shared data. Additionally, TinyOS includes no dynamic memory allocation or variable-argument functions. Thus, with the exception of inline assembly and trusted code, Deputy enforces full memory safety for TinyOS programs.

Finally, we used Deputy on a number of Linux device drivers as part of the SafeDrive project [30]. The drivers converted include `e1000`, `tg3`, `usb-storage`, `intel8x0`, `emu10k1`, and `nvidia`, totaling over 65,000 lines of code. We changed less than 4% of these lines in order to use Deputy and the SafeDrive recovery mechanism, and we observed a 4-23% increase in kernel CPU utilization. Because Deputy allows us to annotate the driver API without changing the binary API, drivers compiled with Deputy can be loaded by a kernel compiled with `gcc`, which is a significant advantage over earlier tools such as CCured [21]. In addition, Deputy's language-based approach to the problem of driver isolation and recovery provides better performance and finer-grained isolation than previous hardware-based approaches such as Nooks [26] and Xen [3]. The remainder of this section will focus on the benchmarks and TinyOS experiments; detailed performance results for Linux device drivers are available in the SafeDrive paper [30].

5.1 Annotation Burden

Deputy will add automatic dependencies to all local variables and cast expressions. Pointer types in any other locations (including function types, global variables, nested pointers, and structure fields) require bounds annotations if they could point to an array. Most of these annotations use the syntactic sugar "`count(n)`" mentioned in Section 4.1, but a few will use the full generality of the "ptr" constructor, with both a lower and upper bound. If an annotation is missing on a pointer type that requires one, we assume it points to a single object and has the annotation `count(1)`.

Benchmark	Lines	Lines Changed	Exec. Time Ratio
SPEC			
`go`	29722	80 (0.3%)	1.11
`gzip`	8673	149 (1.7%)	1.23
`li`	9636	319 (3.3%)	1.50
Olden			
`bh`	1907	139 (7.3%)	1.21
`bisort`	684	24 (3.5%)	1.01
`em3d`	585	45 (7.7%)	1.56
`health`	717	15 (2.1%)	1.02
`mst`	606	66 (10.9%)	1.02
`perimeter`	395	3 (0.8%)	0.98
`power`	768	20 (2.6%)	1.00
`treeadd`	377	40 (10.6%)	0.94
`tsp`	565	4 (0.7%)	1.02
Ptrdist			
`anagram`	635	36 (5.7%)	1.40
`bc`	7395	191 (2.6%)	1.30
`ft`	1904	58 (3.0%)	1.03
`ks`	792	16 (2.0%)	1.10
`yacr2`	3976	181 (4.6%)	1.98
MediaBench I			
`adpcm`	387	15 (3.9%)	1.02
`epic`	3469	240 (6.9%)	1.79
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 1. Deputy benchmarks. For each test, we show the size of the benchmark including comments, the number of lines we changed in order to use Deputy, and the ratio of the execution time under Deputy to the original execution time. "OS components" are the parts of TinyOS used by the three TinyOS programs.

Table 1 summarizes the results of Deputy on our benchmarks. The first column indicates the size of each benchmark, and the second column indicates the number of lines that we needed to modify or add, including both annotations and changes to the code itself. The third column shows the overhead of Deputy (discussed further below).

The number of changed lines varied widely by program, depending on the quality of the original code and how extensively it uses arrays and unions. Many of the changes involved refactoring the code to avoid unsafe constructs such as bad casts or variable argument functions. The statistics here do not include annotations added to the system-wide header files. In some cases we were unable to fix the unsafe code, so we added "trusted" annotations to tell Deputy's type checker to ignore the bad code. Among the 19 non-TinyOS programs, we added 27 such annotations.

The TinyOS changes from Table 1 are mostly in OS components (scheduler, sensors, timers, radio, and serial port). There were four trusted casts to allow access to memory-mapped I/O locations, and there were eight `count` annotations. The small number of annotations is due in part to the use of Java-like interfaces to specify TinyOS's APIs; as a result one annotation is automatically applied to many functions. One API change was necessary in TinyOS, contributing 32 changed lines.

5.2 Performance

For the non-TinyOS programs, we measured the performance in terms of execution time. The tests were run on a machine with two Intel Xeon 2.40GHz CPUs running Linux 2.6.15.2. All tests were run five times and averaged; in all cases, the standard deviation was negligible. We ran these tests with `gcc` 4.0.3 and with Deputy. The "Exec. Time Ratio" column of Table 1 shows the execution time of each program when using Deputy relative to the execution time without Deputy.

The slowdown exhibited by these benchmarks was within 25% in at least half of the tests, with 98% overhead in the worst case. With the sole exception of `yacr2`, Deputy's performance improves on the performance reported for CCured on the SPEC, Olden, and Ptrdist benchmarks [21]. However, CCured is checking stack overflow and running a garbage collector, whereas Deputy is not. Nevertheless, these numbers show that Deputy's run-time checks have a relatively low performance penalty that is competitive with other memory safety tools. These numbers also indicate that splitting the type checker from the optimizer is a viable implementation for a hybrid type system.

In two cases, `treeadd` and `perimeter`, the Deputy version ran faster than the original version. For `treeadd`, the reason is that Deputy's optimizer improves the original code beyond what `gcc` would ordinarily do; if we disable our optimizer, this test yields a ratio of 1.29. For `perimeter`, disabling the optimizer makes no change; we believe that the speedup is a result of some of the simple transformations made by our compiler infrastructure.

The TinyOS programs were compiled with nesC 1.2.7 and `gcc` 3.4.3 and ran on a `mica2` mote, which has a 7.37MHz Atmel ATmega128 microcontroller. These programs are performing periodic tasks, so performance is measured by counting the number of CPU cycles used during one minute of program execution. These tests were also run five times, with a standard deviation of at most 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.

5.3 Bugs Found

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.

6. Related Work

The first piece of related work is a companion paper in which we describe SafeDrive [30], a system for safe and recoverable device drivers for Linux. SafeDrive uses Deputy to enforce isolation of device drivers without resorting to hardware-based approaches, such as Xen [3] and Nooks [26], or binary-instrumentation approaches, such as SFI [27]. The SafeDrive paper contains a high-level description of Deputy from the C programmer's perspective, essentially at the level of the overview in Section 2. In contrast, this current paper presents in detail the principles behind our type system, including our techniques for handling mutation and automatic dependencies. We also show the details of how this type system is instantiated in the case of C programs with pointer arithmetic and tagged unions. Aside from our references to SafeDrive's Linux experiments, the two papers discuss disjoint sets of experiments.

The remaining related work falls into several categories: dependent types, hybrid type checking, type qualifiers, and safety for imperative programs.

Dependent types. DML [29] and Xanadu [28] are two previous systems that used dependent types in functional and imperative lan-

guages, respectively. Both systems add dependent types to existing languages in a practical manner, refining existing type information. Unlike Deputy, the expressions appearing in dependent types are separate from program expressions themselves, and since these expressions are pure, these systems need not reason about dependencies on mutable state. Also, these systems require that type checking be decidable at compile time, further restricting the kinds of dependencies that can be expressed. Our type system allows dependencies on mutable state; in combination with hybrid type checking, this approach allows us to describe more complex program invariants using our dependent types.

Hoare Type Theory [20] combines dependent type theory and Hoare logic in order to reason about dependently-typed imperative programs. This approach uses a monadic type constructor based on Hoare triples to isolate and reason about mutation. In contrast, our type system assigns flow-insensitive types to each program variable, using run-time checks and automatic dependencies to address decidability and usability.

Cayenne [1] is a dependently-typed variant of Haskell. Although its type system is undecidable, it offers very flexible dependent types and does not separate index expressions from program expressions as in DML and Xanadu. Deputy avoids the problem of undecidability by emitting run-time checks; furthermore, it allows updates to mutable state.

Harren and Necula [15] developed a dependent type system for verifying the assembly-level output of CCured. Their system allows dependencies on mutable data, but it requires programs to be statically verifiable. In contrast, Deputy verifies programs at source level and uses run-time checks to enforce dependencies.

Microsoft's SAL annotation language [14] provides dependent type annotations for C programs that are similar to the bounded pointer annotations provided by Deputy. These annotations are viewed as preconditions and postconditions as opposed to Deputy's simpler flow-insensitive types. Also, Deputy's dependent type framework makes it easy to add new dependent type constructors to describe additional C idioms. Microsoft's ESPX checker attempts to check all code statically, whereas Deputy is designed to emit run-time checks for additional flexibility.

Hybrid type checking. Hybrid type checking [10] and the Hoop language [11] are similar to Deputy in that they both use run-time checks to enforce type rules when static verification fails. However, Deputy's separation between the flow-insensitive type system and the flow-sensitive optimizer represents a different approach to the design of a hybrid type system. Also, Deputy offers automatic dependency generation, and incorporates these features into the C language, demonstrating their usefulness as a tool for retrofitting existing C code.

Ou et al. [23] 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, Deputy focuses on handling mutation of local variables and structure fields in the presence of dependencies.

Type qualifiers. CQual [12] 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. More recently, Chin et al. presented semantic type qualifiers [6], which can be used to extend a type system with qualifiers that can be proved sound in isolation. Deputy's dependent types provide more expressive qualifiers; however, it is correspondingly more difficult to prove these extensions sound in isolation.

Safety for imperative programs. CCured [21] analyzes a whole program in order to instrument pointers with checkable bounds information. Unfortunately, this instrumentation makes "cured" applications incompatible with existing libraries. Furthermore, CCured's analysis was designed to instrument a whole program at once, rather than instrumenting one module at a time, incrementally. Deputy's approach, which does not change data structures and which instruments one module at a time, is more practical for large software systems.

Cyclone [17] is a type-safe variant of C that incorporates many modern language features. 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 as well as support for dependent union types.

Dhurjati and Adve [8] use run-time checks to enforce that C programs access objects within their allocated bounds. Their system has low overhead on a set of small to medium-size programs but does not ensure full type safety.

7. Conclusion

In this paper, we have described a series of techniques that allow dependent types to be incorporated into existing low-level imperative programs. Because dependent types are a step towards the expressiveness of predicate logic, it makes sense to borrow the handling of assignment from axiomatic semantics. The result is a type rule for assignment that is simple yet powerful, allowing us to handle mutation in the presence of dependent types. In addition, we address the common pitfalls of aliasing with lightweight syntactic restrictions on dependencies for pointers. Finally, we have contributed a technique for adding missing dependencies to existing programs.

Our experiments show that C programs contain a large number of implicit dependencies that can now be expressed and checked with Deputy. This approach allows pointer arithmetic and union types to be used safely, without having to make significant changes to existing programs or their data representation. We expect that we will find many other common programming idioms whose invariants can be enforced using the principles described here.

Acknowledgments

Many thanks to Feng Zhou, Ilya Bagrak, Bill McCloskey, Rob Ennals, and Eric Brewer for their contributions to Deputy. This material is based upon work supported by the National Science Foundation under Grant Nos. CCR-0326577, CCF-0524784, and CNS-0509544, as well as gifts from Intel Corporation.

References

[1] Augustsson, L. Cayenne—a language with dependent types. In *ICFP'98*.

[2] Austin, T. M., Breach, S. E., and Sohi, G. S. Efficient detection of all pointer and array access errors. In *PLDI'94*.

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *SOSP'03*.

[4] Boehm, H.-J. Space efficient conservative garbage collection. In *PLDI '93*, pp. 197–206.

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

[6] Chin, B., Markstrum, S., and Millstein, T. Semantic type qualifiers. In *PLDI'05*.

[7] CONDIT, J., AND NECULA, G. C. Data slicing: Separating the heap into independent regions. In *Proceedings of the 14th International Conference on Compiler Construction* (2005).

[8] DHURJATI, D., AND ADVE, V. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE'06*.

[9] DHURJATI, D., KOWSHIK, S., ADVE, V., AND LATTNER, C. Memory safety without garbage collection for embedded applications. *Trans. on Embedded Computing Sys. 4*, 1 (2005), 73–111.

[10] FLANAGAN, C. Hybrid type checking. In *POPL'06*.

[11] FLANAGAN, C., FREUND, S. N., AND TOMB, A. Hybrid types, invariants, and refinements for imperative objects. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL)* (2006).

[12] FOSTER, J. S., FAHNDRICH, M., AND AIKEN, A. A theory of type qualifiers. In *PLDI'99*, pp. 192–203.

[13] GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. The nesC language: A holistic approach to networked embedded systems. In *PLDI'03*.

[14] HACKETT, B., DAS, M., WANG, D., AND YANG, Z. Modular checking for buffer overflows in the large. In *ICSE'06*.

[15] HARREN, M., AND NECULA, G. C. Using dependent types to certify the safety of assembly code. In *SAS'05*.

[16] HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D. E., AND PISTER, K. S. J. System architecture directions for networked sensors. In *ASPLOS'00*.

[17] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference* (2002).

[18] JONES, R. W. M., AND KELLY, P. H. J. Backwards-compatible bounds checking for arrays and pointers in C programs. *Automated and Analysis-Driven Debugging* (1997).

[19] LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture* (1997).

[20] NANEVSKI, A., AND MORRISETT, G. Dependent type theory of stateful higher-order functions. Tech. Rep. TR-24-05, Harvard University.

[21] NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. CCured: Type-safe retrofitting of legacy software. *TOPLAS 27*, 3 (May 2005).

[22] NECULA, G. C., MCPEAK, S., AND WEIMER, W. CIL: Intermediate language and tools for the analysis of C programs. In *CC'02*, Grenoble, France.

[23] OU, X., TAN, G., MANDELBAUM, Y., AND WALKER, D. Dynamic typing with dependent types. In *IFIP Conference on Theoretical Computer Science* (2004).

[24] PATIL, H., AND FISCHER, C. N. Efficient run-time monitoring using shadow processing. In *Automated and Algorithmic Debugging* (1995), pp. 119–132.

[25] SPEC. Standard Performance Evaluation Corporation Benchmarks. *http://www.spec.org/osg/cpu95/CINT95* (July 1995).

[26] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *SOSP'03*.

[27] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *SOSP'94*, pp. 203–216.

[28] XI, H. Imperative programming with dependent types. In *LICS'00*.

[29] XI, H., AND PFENNING, F. Dependent types in practical programming. In *POPL'99*.

[30] ZHOU, F., CONDIT, J., ANDERSON, Z., BAGRAK, I., ENNALS, R., HARREN, M., NECULA, G., AND BREWER, E. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI'06*.

A. Soundness Proof

In this section, we provide a sketch of the proof for Theorem 2. Due to space constraints, we omit the proof for Theorem 1, which is a straightforward induction on the derivation of the expression e. We also omit the full definition of $[\![e]\!]\rho$ and $[\![c]\!]\rho$, which are standard.

The first lemma we need is the following substitution lemma:

LEMMA 1 (Substitution).

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

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* $\Gamma \vdash_L \tau :: \text{type}$, $\rho \models \Gamma$, *and* ρ' *differs from* ρ *only in the* ρ_S *element, then* $[\![\tau]\!]\rho = [\![\tau]\!]\rho'$.

Our third lemma says that closed local types depend only on the allocation state:

LEMMA 3. *If* $\emptyset \vdash_L \tau :: \text{type}$ *and* ρ_A *is a restriction of* ρ'_A, *then* $[\![\tau]\!]\rho \subseteq [\![\tau]\!]\rho'$.

Due to space constraints, we omit the proofs for these lemmas. Care must be taken to re-prove these lemmas when additional type constructors are added to the system.

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; here, we show only the cases for variable and memory update. The semantics of these commands are:

$$\begin{array}{rcl} [\![x := e]\!]\rho & = & (\rho_E[x \mapsto [\![e]\!]\rho], \rho_S, \rho_A) \\ [\![*e_1 := e_2]\!]\rho & = & (\rho_E, \rho_S[[\![e_1]\!]\rho \mapsto [\![e_2]\!]\rho], \rho_A) \end{array}$$

- Case 1: 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' and Lemma 3 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 2: 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 2, $[\![x]\!]\rho' = [\![x]\!]\rho \in [\![\Gamma(x)]\!]\rho = [\![\Gamma(x)]\!]\rho'$.