Dependent Types for Low-Level Programming

Jeremy Condit[1], Matthew Harren[1], Zachary Anderson[1],
David Gay[2], and George C. Necula[1]

[1] University of California, Berkeley
[2] Intel Research, Berkeley

Abstract. In this paper, we describe the key principles of a dependent type system for low-level imperative languages. The major contributions of this work are (1) a sound type system that combines dependent types and mutation for variables and for heap-allocated structures in a more flexible way than before and (2) a technique for automatically inferring dependent types for local variables. We have applied these general principles to design Deputy, a dependent type system for C that allows the user to describe bounded pointers and tagged unions. Deputy has been used to annotate and check a number of real-world C programs.

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 cannot describe such dependencies we cannot prove even the memory safety of most low-level programs.

In this paper, we consider the main obstacles that limit the convenient use of dependent types in low-level programs:

- *Soundness:* Mutation of variables or heap locations, used heavily in low-level programs, might invalidate the types of some state elements. Previous dependent type systems are of limited usefulness because they contain restrictions that preclude the use of mutable variables in dependent types [2,19,20]. Instead, 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.
- *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.

R. De Nicola (Ed.): ESOP 2007, LNCS 4421, pp. 520–535, 2007.

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 [7], is essential for handling real-world code.
- *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's dependent types allow programmers to specify common C programming idioms 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 [11,14]. Instead, Deputy allows programmers to specify relationships between existing data elements, which in turn allows the compiler to check the safety of most pointer and union operations without changing program data structures. This approach enables users to apply Deputy modularly and incrementally to system components, which is especially important for very large systems or in the presence of external libraries.

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 for a core imperative language with references and structures, and Section 4 presents our automatic dependency inference technique. Then, Section 5 shows how this dependent type framework can be instantiated to provide safe handling of pointer arithmetic and union types in C. Finally, we discuss related work in Section 6.

2 Overview

In order to provide an intuition for the general principles described in this paper, we discuss here how these principles are instantiated for enforcing memory safety with the Deputy type system. Consider the sample code shown in Figure 1 without any of the underlined or italicized statements. 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 to an array of at least `end - buf` integers.

Deputy processes this program in three passes:

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 Figure 1, this inference pass adds the underlined code; specifically, it introduces the variable `tmplen`, which is updated to store the length of the array pointed to by `tmp`. We describe this algorithm in Section 4.

```
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      int tmplen = (end - buf) - 1;
7      assert(0 <= 1 <= end - buf);
8      int * count(tmplen) tmp = buf + 1;
9      assert(0 <= end - tmp <= tmplen);
10     buf = tmp;
11   }
12   return sum;
13 }
```

Fig. 1. A Deputy program, along with the code added during automatic dependency inference (underlined) and the assertions added during type checking (in italics). The temporary variable is shown to better demonstrate Deputy features but is not required.

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 Figure 1, the italicized code shows the assertions that have been added in this stage. For example, the assertion in line 4 ensures that the `buf` array is nonempty and can therefore be safely dereferenced.

The check on line 9 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. Here, we ensure that `tmp` has at least `end - tmp` elements and thus will satisfy `buf`'s type invariant when assigned to `buf`. Such self-dependencies are particularly useful when designing flexible types for low-level code. The rules for type checking and for inserting run-time checks are described in Section 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. Using standard data-flow techniques, we can eliminate a large number of the unnecessary checks in the program, and we can also identify checks that are guaranteed to fail. In Figure 1, all checks could reasonably be eliminated by the optimizer. 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. Our current optimizer uses standard data-flow techniques and is discussed in detail in a separate technical report [1]. However, it is worth pointing out that the amount of static memory safety enforcement depends directly on the quality of the optimizer.

We can use this example to contrast our approach with safe C type systems that use fat pointers [11,14]. With these systems, the pointer `buf` might be

$$
\begin{array}{lll}
\text{Ctors} & C ::= \text{int} \mid \text{ref} \mid \ldots & \\
\text{Types} & \tau ::= C \mid \tau_1\ \tau_2 \mid \tau\ e & \\
\text{Kinds} & \kappa ::= \text{type} \mid \text{type} \to \kappa \mid \tau \to \kappa & \\
\text{L-exprs} & \ell ::= x \mid *e & \\
\text{Exprs} & e ::= n \mid \ell \mid e_1\ \text{op}\ e_2 & \\
\text{Cmds} & c ::= \text{skip} \mid c_1; c_2 \mid & \\
 & \quad \ell := e \mid \text{assert}(\gamma) \mid & \\
 & \quad \text{let } x : \tau = e \text{ in } c \mid & \\
 & \quad \text{let } x = \text{new } \tau(e) \text{ in } c & \\
\text{Preds} & \gamma ::= e_1\ \text{comp}\ e_2 \mid \text{true} \mid \gamma_1 \wedge \gamma_2 &
\end{array}
$$

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

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

Fig. 2. The grammar for a simple dependently-typed imperative language

stored as a two-word pointer, which means that all callers of this function must be instrumented as well. In contrast, Deputy's annotations require no changes outside this function, which is a crucial advantage over existing tools. Also, since Deputy's checks refer to existing program data, our optimizer can take advantage of existing checks such as the conditional in line 3. These benefits have allowed us to apply Deputy incrementally to modular software such as Linux device drivers and TinyOS components, as described in Section 5.3.

3 Dependent Type Framework

This section presents the key components of our dependent type framework. Our full type system supports dependencies between, and mutation of, local variables, formal parameters, global variables, and structure fields. In this section, we start with a system that includes only local variables, and then we extend it with heap-allocated structures. The remaining features are not discussed in this paper, but further details are available in a companion technical report [5].

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, 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. For the two constructors we have seen so far, the kind of "int" is "type", and the kind of "ref" is "type → type".

To show how this system can be extended with additional type constructors, consider the `count` annotation used in Figure 1. 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 "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 program. Finally, note that we omit loops and conditionals, which are irrelevant to our flow-insensitive type system, and we omit function calls, which can be added later as an extension [5].

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 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. 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.[1]

We now consider the well-formedness rules for types, shown at the top of Figure 3. 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. Note 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 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.

[1] 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.

$\Gamma \vdash_L \tau :: \kappa$ 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.

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

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

$$\frac{}{\Gamma \vdash \mathrm{skip} \Rightarrow \mathrm{skip}}\ (\textsc{skip}) \qquad \frac{\Gamma \vdash c_1 \Rightarrow c_1' \qquad \Gamma \vdash c_2 \Rightarrow c_2'}{\Gamma \vdash c_1; c_2 \Rightarrow c_1'; c_2'}\ (\textsc{seq})$$

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

$$\frac{\Gamma \vdash e_1 : \mathrm{ref}\ \tau \Rightarrow \gamma_1 \qquad \Gamma \vdash e_2 : \tau \Rightarrow \gamma_2}{\Gamma \vdash *e_1 := e_2 \ \Rightarrow\ \mathrm{assert}(\gamma_1 \wedge \gamma_2);\ *e_1 := e_2}\ (\textsc{mem write})$$

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

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

Fig. 3. The four judgments used by our type system and the core type checking rules for each. Additional rules (with nontrivial γ predicates) will be added later.

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 earlier, we might add new typing rules for pointer arithmetic and dereference:

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

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

These rules are responsible for the assertions generated in line 4 and line 7 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:

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

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

The judgment for checking commands, 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 have identical semantics if no assertion in c' fails.

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 and only if one can prove that $\phi \implies \phi[{}^{e}\!/_{x}]$. 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. Section 3.4 makes this intuition more precise.

To understand this rule in more detail, consider 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, 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 $\Gamma \vdash n[^{n-1}/_{n}] : \text{int}[^{n-1}/_{n}] \Rightarrow \gamma_n$ 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 $\gamma_a = 0 \leq n-1 \leq n$. After static optimization, this check can be reduced to $0 \leq n-1$, which is precisely the check we expected.[2]

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 largely straightforward. Note that 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) and (ALLOC) rules require a substitution when checking e; however, since we are introducing a new variable, we need not check the rest of the environment as in the (VAR WRITE) rule.

3.3 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 add several new syntactic constructs. We add the 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, and we add the l-expression $\ell.f$, which accesses a field with name f. We also add the expression $\{f_1 = e_1; \ldots; f_n = e_n\}$, which is a structure literal that initializes field f_i to expression e_i. For example, we could declare a structure with two fields such that field f_1 is an array whose length is one greater than the value in field f_2:

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

Note that it is legal to apply the "ref" constructor to a structure type whose fields depend on one another, because all of the structure type's dependencies are self-contained. Pointers to structures with internal dependencies are quite common in C programs.

Figure 4 shows the rules for type checking structures. The (TYPE STRUCT) rule ensures that field types depend only on other fields in the *same* structure. The (STRUCT READ) rule substitutes these field names with the appropriate

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

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

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

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

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

Fig. 4. Structure type checking rules

expressions; for example, using the declaration above, a read from $y.f_1$ would have type "array int $(y.f_2 + 1)$". 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.

In the technical report [5], we present a similar extension that allows us to type check calls to functions whose arguments depend on one another.

3.4 Soundness

We have proved the soundness of the core type system of Section 3.2. We omit the details of this proof for space reasons, but we present here the formal requirements on the framework for ensuring sound handling of mutation in presence of dependent types. Full details can be found in the technical report [5].

We define the state of execution, ρ, to be a tuple containing, among other things, a mapping ρ_A from addresses to types representing the allocation state. We define $[\![e]\!]\rho$ to be the value $v \in \text{Val}$ of expression e in state ρ.

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:

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

In particular, 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, and in some cases, these definitions may require an augmented notion of state (e.g., a constructor characterizing lock state may require a history

of locking operations). 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.

3.5 Limitations

One limitation of this type system is its flow-insensitivity. For example, incrementing an array before decrementing its length would result in an error even though these two operations are safe when taken together. One way to overcome this limitation is to use automatic dependencies to generate fresh dependencies for local variables, as discussed in Section 4. Another alternative is to use an extended (VAR WRITE) rule that handles several statements at once.

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. In these cases, 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.

4 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. 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.

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' computes the same result as c.

The interesting rules for deriving this judgment are given in Figure 5. In the (AUTO LET) rule, we add new variables to track any missing dependencies for x. These variables are initialized using expressions from the type of e (by using the type checking judgment). Note that γ is unused in this rule; however, it will be checked appropriately during the type checking phase. In the (AUTO VAR WRITE) rule, we update all of the automatic variables associated with x using a similar approach. For the purposes of this rule, we add syntax for parallel assignment, written $x_1, \ldots, x_n := e_1, \ldots, e_n$, where all expressions e_i are evaluated before assignments take place. The type checking rule for parallel assignment is

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

$$\frac{\begin{array}{c}\text{(AUTO VAR WRITE)}\\ \Gamma(x) = \tau\ x_1 \ldots x_n \qquad \Delta(x) = (x_1, \ldots, x_n) \\ \Gamma \vdash e : \tau\ e_1 \ldots e_n \Rightarrow \gamma\end{array}}{\Gamma; \Delta \vdash x := e \ \rightsquigarrow\ x,\ x_1, \ldots, x_n := e,\ e_1, \ldots, e_n}$$

Fig. 5. Rules for automatic dependencies

a straightforward extension of the (VAR WRITE) rule. Note that this technique is independent of the actual dependent types in use.

In the following example, the underlined code can be inferred using this technique:

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

By using automatic dependencies, we ensure that nx contains the number of elements in x regardless of which branch of the conditional was taken. Inferring a similar result with a purely static analysis would be much more difficult. Note, however, that in cases where static analysis would suffice, our optimizer can eliminate variables and assignments that were introduced by this transformation.

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. 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.

5 Dependent Types for C

We now show how our dependent type framework can be instantiated to support pointer bounds and tagged unions in C programs. Further details can be found in the SafeDrive paper [21] (see related work) and in the technical report [5].

5.1 Pointer Bounds

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. 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. The $\oplus$ operator may be used in local expressions. Finally, we add typing rules for all relevant operations on this type (e.g., dereference and arithmetic), the details of which can be found in the technical report [5]. Examples of the ptr type are as follows:

$$\begin{array}{ll} x : \text{ptr int } b\ (b \oplus 8) & \text{// 8 integer area starting at } b \\ x : \text{ptr int } x\ (x \oplus n) & \text{// } n \text{ integer area starting at } x \\ x : \text{ptr int } x\ e & \text{// from } x \text{ to } e \end{array}$$

These declarations (with syntactic sugar for common cases) offer C programmers a tractable but expressive way to declare pointer bounds without modifying existing data structures. Note that many of the uses of this type involve self-dependencies, which are made tractable by our support for mutation.

5.2 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. As with bounded pointers, we add type rules for the relevant operations on this new type constructor. For example:

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

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.

5.3 Experiments

We implemented Deputy using the CIL infrastructure [15].[3] Our implementation is 18,000 lines of OCaml code in addition to the CIL front-end itself. Given an annotated C program, our implementation adds automatic bounds variables, type checks the program (which inserts run-time checks), optimizes the inserted checks, and then emits the program as C code for compilation with `gcc`. The flow-sensitive optimizer tracks facts such as which pointers are null, and it uses forward substitution of locals plus basic arithmetic facts to eliminate inserted checks and to detect checks that will always fail [1]. To use Deputy, programmers run `deputy` in place of `gcc` as their compiler, and then they modify code or type annotations in order to eliminate the resulting compile-time and run-time errors.

Our implementation covers most of C's features, many of which are not discussed in this paper. However, we do not check inline assembly, some variable-argument functions, and code explicitly marked as trusted by the programmer. In addition, Deputy does not check memory deallocation, which is an orthogonal problem; for now, the user can choose to trust deallocations or to run a garbage collector. Aside from these caveats, Deputy ensures that the program is free of type and memory errors, including bounds violations and misuse of unions.

To test Deputy, we annotated a number of standard benchmarks, including Olden [4], Ptrdist [3], and selected tests from the SPEC CPU [18] and MediaBench [12] suites. We also used Deputy to enforce type safety in version 2 of the TinyOS [10] sensor network operating system, including three simple demo applications: periodic LED blinking (Blink), forwarding radio packets to and from a PC (BaseStation), and simple periodic data acquisition (Oscilloscope). Finally, we have applied Deputy to a number of Linux device drivers for use with the SafeDrive driver recovery system [21].

Results for these experiments are shown in Table 1. In all experiments, we changed less than 11% of the lines of code in the program; in most cases, we changed about 2-4%. We added a total of 27 trusted annotations that tell Deputy to ignore bad code. 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` (on the Ptrdist benchmarks), Deputy's performance improves on the performance reported for CCured on the SPEC, Olden, and Ptrdist benchmarks [14]. However, CCured is checking stack overflow and uses 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. Further details can be found in the accompanying technical report [5] and in the SafeDrive paper [21].

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.

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

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.

Suite	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%)	–

6 Related Work

SafeDrive. In a companion paper, we present SafeDrive [21], a system for safe and recoverable Linux device drivers that uses Deputy to detect faults. The SafeDrive paper contains a high-level description of Deputy from the C programmer's perspective, whereas this paper presents in detail the principles behind our type system, including our techniques for handling mutation and automatic dependencies.

Dependent types. DML [20], Xanadu [19], and Cayenne [2] are previous languages that use dependent types. In DML and Xanadu, expressions appearing in dependent types are different from program expressions and must be decidable at compile time. In Cayenne, arbitrary expressions from the same language are allowed, and thus the type system may be undecidable. We attempt to find a middle ground, allowing expressive annotations in the 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.

Hoare Type Theory [13] uses a monadic type constructor based on Hoare triples to isolate and reason about mutation in dependently-typed imperative programs. In contrast, we assign flow-insensitive types to each program variable, using run-time checks for decidability and automatic dependencies for usability.

Harren and Necula [9] 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.

Microsoft's SAL annotation language [8] 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.

Hybrid type checking. Our type system uses a form of hybrid type checking [7] with a flow-insensitive type system and automatic dependency generation. We demonstrate the effectiveness of this approach for low-level code.

Ou et al. [16] 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 [17] allows static and dynamic types to coexist using run-time checks, but it does not use dependent types.

Safety for imperative programs. CCured [14] analyzes a whole program in order to instrument pointers with checkable bounds information, and Cyclone [11] is a type-safe variant of C that incorporates many modern language features. Both use "fat" pointers, which make the resulting programs incompatible with existing libraries; Deputy's dependent types solve this crucial problem. Cyclone allows some dependent type annotations; for example, the programmer can annotate a pointer with the number of elements it points to. Deputy provides more general pointer bound support as well as support for dependent union types.

Dhurjati and Adve [6] use run-time checks to ensure 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

We have described a series of techniques that allow dependent types to be used in existing low-level imperative programs. Inspired by the handling of assignment in axiomatic semantics, we have designed a type rule for assignment that is simple yet powerful, allowing us to handle mutation in the presence of dependent types. We address decidability with run-time checks, and we address usability with a technique for automatic dependency generation. The result is a practical type system for annotating and checking low-level code.

Acknowledgments. Thanks to Feng Zhou, Ilya Bagrak, Bill McCloskey, Rob Ennals, and Eric Brewer for their contributions. 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. Anderson, Z. R. Static analysis of C for hybrid type checking. Tech. Rep. EECS-2007-1, UC Berkeley, 2007.
2. Augustsson, L. Cayenne—a language with dependent types. In *ICFP'98*.
3. Austin, T. M., Breach, S. E., and Sohi, G. S. Efficient detection of all pointer and array access errors. In *PLDI'94*.
4. Carlisle, M. C. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, June 1996.
5. Condit, J., Harren, M., Anderson, Z., Gay, D., and Necula, G. Dependent types for low-level programming. Tech. Rep. EECS-2006-129, UC Berkeley, 2006.
6. Dhurjati, D., and Adve, V. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE'06*.
7. Flanagan, C. Hybrid type checking. In *POPL'06*.
8. Hackett, B., Das, M., Wang, D., and Yang, Z. Modular checking for buffer overflows in the large. In *ICSE'06*.
9. Harren, M., and Necula, G. C. Using dependent types to certify the safety of assembly code. In *SAS'05*.
10. 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*.
11. 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).
12. 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).
13. Nanevski, A., and Morrisett, G. Dependent type theory of stateful higher-order functions. Tech. Rep. TR-24-05, Harvard University.
14. Necula, G. C., Condit, J., Harren, M., McPeak, S., and Weimer, W. CCured: Type-safe retrofitting of legacy software. *TOPLAS 27*, 3 (May 2005).
15. Necula, G. C., McPeak, S., and Weimer, W. CIL: Intermediate language and tools for the analysis of C programs. In *CC'02*, Grenoble, France.
16. Ou, X., Tan, G., Mandelbaum, Y., and Walker, D. Dynamic typing with dependent types. In *IFIP Conference on Theoretical Computer Science* (2004).
17. Siek, J. G., and Taha, W. Gradual typing for functional languages. In *Scheme and Functional Programming* (2006).
18. SPEC. Standard Performance Evaluation Corporation Benchmarks. *http://www.spec.org/osg/cpu95/CINT95* (July 1995).
19. Xi, H. Imperative programming with dependent types. In *LICS'00*.
20. Xi, H., and Pfenning, F. Dependent types in practical programming. In *POPL'99*.
21. 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*.