Published in the Proceedings of the *Principles of Programming Languages, 2002*, pages ??–??

CCured: Type-Safe Retrofitting of Legacy Code

George C. Necula Scott McPeak Westley Weimer
University of California, Berkeley
{necula,smcpeak,weimer}@cs.berkeley.edu

Abstract

In this paper we propose a scheme that combines type inference and run-time checking to make existing C programs type safe. We describe the CCured type system, which extends that of C by separating pointer types according to their usage. This type system allows both pointers whose usage can be verified statically to be type safe, and pointers whose safety must be checked at run time. We prove a type soundness result and then we present a surprisingly simple type inference algorithm that is able to infer the appropriate pointer kinds for existing C programs.

Our experience with the CCured system shows that the inference is very effective for many C programs, as it is able to infer that most or all of the pointers are statically verifiable to be type safe. The remaining pointers are instrumented with efficient run-time checks to ensure that they are used safely. The resulting performance loss due to run-time checks is 0–150%, which is several times better than comparable approaches that use only dynamic checking. Using CCured we have discovered programming bugs in established C programs such as several SPECINT95 benchmarks.

1 Introduction

The C programming language provides programmers with a great deal of flexibility in the representation of data and the use of pointers. These features make C the language of choice for systems programming. Unfortunately, the cost is a weak type system and consequently a great deal of "flexibility" in introducing subtle bugs in programs.

This research was supported in part by the National Science Foundation Career Grant No. CCR-9875171, and ITR Grants No. CCR-0085949 and No. CCR-0081588, and gifts from AT&T Research and Microsoft Research. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

POPL '02, Jan. 16-18, 2002 Portland, OR USA
(c) 2002 ACM ISBN 1-58113-450-9/02/01...$5.00

While in the 1970s sacrificing type safety for flexibility and performance might have been a sensible language design choice, today there are more and more situations in which type safety is just as important as, if not more important than, performance. Errors like array out-of-bounds accesses lead both to painful debugging sessions chasing inadvertent memory updates and to malicious attacks exploiting buffer overrun errors in security-critical code. (Almost 50% of recent CERT advisories result from security violations of this kind [29].) Type safety is desirable for isolating program components in a large or extensible system, without the loss of performance of separate address spaces. It is also valuable for inter-operation with systems written in type-safe languages (such as type-safe Java native methods, for example). Since a great deal of useful code is already written or being written in C, it would be useful to have a practical scheme to bring type safety to these programs.

The work described in this paper is based on two main premises. First, we believe that even in programs written in unsafe languages like C, a large part of the program can be verified statically to be type safe. Then the remaining part of the program can be instrumented with run-time checks to ensure that the execution is memory safe. The second premise of our work is that in many applications, some loss of performance due to run-time checks is an acceptable price for type safety, especially when compared to the alternative cost of reprogramming the system in a type-safe language.

The main contribution of this paper is the CCured type system, an extension of the C type system with explicit types for pointers into arrays, and with dynamic types. It extends previous work on adding dynamic types to statically typed languages: types and capabilities for the statically-typed elements are known at compile time, while the dynamically-typed elements are guarded by run-time checks. Our type system is inspired by common C usage, and includes support for physical type equivalence [8] and special "sequence" pointers for accessing arrays. The second contribution of the paper is a simple yet effective type-inference algorithm that can translate ordinary C programs into CCured mostly automatically and in a matter of seconds even for 30,000-line programs. We have used this inference algorithm to produce type-safe versions of several C programs for which we observed a slowdown of 0–150%. In the process, we have also found programming bugs in the analyzed code, the most

```
1  int *_1 *_2 a;                      // array
2  int i;                              // index
3  int acc;                            // accumulator
4  int *_3 *_4 p;                      // elem ptr
5  int *_5 e;                          // unboxer
6  acc = 0;
7  for (i=0; i<100; i++) {
8     p = a + i;                       // ptr arith
9     e = *p;                          // read elem
10    while ((int) e % 2 == 0) {       // check tag
11       e = * (int *_6 *_7) e;        // unbox
12    }
13    acc += ((int)e >> 1);            // strip tag
14 }
```

Figure 1: A short C program fragment demonstrating safe and unsafe use of pointers.

surprising being several array out-of-bounds errors in the SPECINT95 `compress`, `go` and `ijpeg` benchmarks.

We continue in Section 2 with an informal overview of the system in the context of a small example program. Then in Section 3 we present a simple language of pointers, with its type system (in Section 4) and operational semantics (in Section 5), followed by a discussion of the type safety guarantees of CCured programs. In Section 6 we present a simple constraint-based type inference algorithm for CCured. We discuss informally the extension of the language presented in this paper to the whole C programming language in Section 7, necessary source code changes in Section 8, and in Section 9 we relate our experience with a prototype implementation.

2 Overview of the Approach

To ensure memory safety, for each pointer we must keep track of certain properties of the memory area to which it is supposed to point. Such properties include the area's size and the types of the values it contains. For some pointers this information can be computed precisely at compile time and for others we must compute it at run time, in which case we have to insert run-time safety checks.

These two kinds of pointers appear in the example C program shown in Figure 1. The program operates on a hypothetical disjoint union datatype we call "boxed integer" that has been efficiently implemented in C as follows: if a boxed integer value is odd then it represents a 31-bit integer in the most significant bits along with a least significant tag bit equal to one, otherwise it represents a pointer to another "boxed integer". We use the C datatype `int *` to represent boxed integers. The variable `a` is a pointer to an array of boxed integers. The purpose of the function is to accumulate in the variable `acc` the sum of the first 100 boxed integers in the array. In line 8 we compute the address of a boxed integer and in line 9 we fetch the boxed integer. The loop in lines 10–12 unboxes the integer. The subscripts on the pointer type constructors "`*`" have been added to simplify cross-referencing from the text.

By inspection of the program we observe that the values of the variables `a` and `p` are supposed to point into the same array. Neither of these variables is subject to casts (and they have no other aliases) and thus we know that the type of the values they point to is indeed "`int *`". Furthermore, we observe that while the pointer `a` is subject to pointer arithmetic, the pointer `p` is not. This means that we must check uses of "`a`" for array out-of-bounds errors but we do not need to do so for the uses of "`p`" (assuming that a check is performed in line 8 where "`p`" is initialized). In this paper we refer to "`p`" as a *safe* pointer and to "`a`" as a *sequence* pointer. To be more precise we associate this information with the pointer type constructors $*_4$ and $*_2$ respectively. Safe and sequence pointers have only aliases that agree on the type of the value pointed to and thus point to memory areas whose contents is statically typed.

Now we turn our attention to the pointer values of "`e`". These values are used with two incompatible types "`int *`" and "`int * *`". This means that we cannot count on the static type of "`e`" as being an accurate description of its values. In our type system we say that "`e`" has a *dynamic* pointer type and we associate this information with the pointer type constructors $*_5$ and $*_7$. Dynamic pointers always point into memory areas whose contents do not have a reliable static type and must therefore store extra information to classify their contents as pointers or integers. Correspondingly, the aliases of dynamic pointers can only be other dynamic pointers; otherwise a safe pointer's static type assumptions could be violated after a memory write through a dynamic pointer alias. This means that the type constructors $*_1$, $*_3$ and $*_6$ must also be classified as dynamic pointers. In this example program, we have a mixture of pointers whose static type can be relied upon and thus require little or no access checks (the safe and sequence pointers), and also pointers whose static type is unreliable and thus require more extensive checking.

Motivated by this and similar examples, the CCured language is essentially the union of two languages: a strongly typed language (containing safe and sequence pointers), and an untyped language for which the type information is maintained and checked at run time.

All values and memory areas in the system are either part of the safe/sequence world, or part of the dynamic world. The only place these worlds touch is when a typed memory area contains a pointer to untyped memory. Untyped memory cannot contain safe or sequence pointers because we cannot assign a reliable static type to the contents of dynamic areas. We shall formalize these invariants starting in Section 4. Before then, in order to provide some intuition for the formal development, we summarize in Figure 2 the capabilities and invariants of various pointer kinds. Since in C the null pointer is used frequently, we allow the safe pointers to be null. Similarly, we allow arbitrary integers to be "disguised" as sequence and dynamic pointers, but not as safe pointers.

Kind	Invariants maintained	Capabilities	Access checks required
Safe pointer to τ	• Either 0 or a valid address containing a value of type τ. • Aliases are either safe or sequence pointers of base type τ.	• Cast from sequence pointer of base type τ. • Set to 0. • Cast to integer.	• Null-pointer check when dereferenced.
Sequence pointer to τ	• Knows at run-time if it is an integer, and if not, knows the memory area (containing a number of values of type τ) to which it points. • Aliases are safe and sequence pointers of base type τ.	• Cast to safe pointer of base type τ. • Cast from integer. • Cast to integer. • Perform pointer arithmetic.	• Non-pointer check (subsumes null-pointer check). • Bounds check when dereferenced or cast to `SAFE`.
Dynamic pointer	• Knows at run-time if it is an integer, and if not, knows the memory area (containing a number of integer or dynamic pointer values) to which it points. • The memory area pointed to maintains tags distinguishing integers from pointers. • Aliases are dynamic pointers.	• Cast to and from any dynamic pointer type. • Cast from integer. • Cast to integer. • Perform pointer arithmetic.	• Non-pointer check. • Bounds check when dereferenced. • Maintain the tags in the pointed area when reading and writing.

Figure 2: Summary of the properties and capabilities of various kinds of pointers.

$$
\begin{array}{llcl}
\text{Types:} & \tau & ::= & \texttt{int} \mid \tau\ \texttt{ref SAFE} \mid \tau\ \texttt{ref SEQ} \\
& & \mid & \texttt{DYNAMIC} \\
\text{Expressions:} & e & ::= & x \mid n \mid e_1\ \texttt{op}\ e_2 \mid (\tau)e \\
& & \mid & e_1 \oplus e_2 \mid\ !e \\
\text{Commands:} & c & ::= & \texttt{skip} \mid c_1; c_2 \mid e_1 := e_2
\end{array}
$$

Figure 3: The syntax of a simple language with pointers, pointer arithmetic and casts.

3 A Language of Pointers

There are many constructs in the C programming language that can be misused to violate memory safety and type safety. Among them are type casts, pointer arithmetic, arrays, union types, the address-of operator, and explicit deallocation. To simplify the presentation of the key ideas behind our approach we describe it formally for a small language containing only pointers with casts and pointer arithmetic, and then we discuss informally in Section 7 how we extend the approach to handle the remaining C constructs.

Figure 3 presents the syntax of types, expressions and commands for a simple programming language that serves as the vehicle for formalizing CCured. At the level of types we have retained only the integers and the pointer types. In C the symbol "`*`" is used in various syntactic roles in conjunction with pointer types; to avoid confusion we have instead adopted the syntax of ML references for our modeling language. We have three flavors of pointer types corresponding to safe, sequence and dynamic pointers respectively. The type `DYNAMIC` is a pointer type that does not carry with it the type of the values pointed to. This is indicative of the fact that we cannot count on the referenced type of a dynamic pointer.

Among expressions we have integer literals and an assortment of binary integer operations, such as the arithmetic and relational operations, written generically as `op`. Relational operations on pointers are done after casting the pointers to integers. The binary operation $\oplus$ denotes pointer arithmetic and the notation $!e$ denotes the result of reading from the memory location denoted by e (like `*`e in C).

The language of commands is greatly simplified. The only notable form of commands is memory update through a pointer (`p :=` e is like `*p =` e in C). Control flow operations are not interesting because our approach is flow insensitive. Function calls are omitted for simplification; instead we discuss briefly in Section 7 how we handle function calls and function pointers. Among other notable omissions are variable updates and the address-of operator on variables. Instead, to simplify the formal presentation, we consider that a variable that is updated or has its address taken in C would be allocated on the heap and operated upon through its address (which is an immutable pointer variable) in our language. Finally, we ignore here the allocation and deallocation of memory (including that of stack frames). Even though the resulting language appears to be much simpler than C, it allows us to expose formally and in a succinct way the major ideas behind our type system, the type inference algorithm and the implementation. The implementation itself handles the entire C language.

Our example program from Figure 1 can be transcribed in this language (with the use of variable declarations and a few additional control-flow constructs) as shown in Figure 4. The major change in this version is that we have replaced the variables `i`, `acc`, `p` and `e` by pointers, and the accesses to those variables by memory operations. (The lines 1–5 are technically not representable in our language. We show them only to provide a context for the rest of the example. We also ignore the initialization of these variables.) All the newly introduced pointer type constructors are `SAFE` since the corresponding pointers are used only for reading and writing. Another change is that one or more nested dynamic pointer type constructors are collapsed into the `DYNAMIC` type.

```
DYNAMIC ref SEQ a;                   // array
int ref SAFE p_i;                    // index
int ref SAFE p_acc;                  // accumulator
DYNAMIC ref SAFE ref SAFE p_p;       // elem ptr
DYNAMIC ref SAFE p_e;                // unboxer
p_acc := 0;
for (p_i := 0; !p_i < 100; p_i := !p_i + 1) {
   p_p := (DYNAMIC ref SAFE)(a ⊕ !p_i);
   p_e := !!p_p;
   while ((int) !p_e % 2 == 0) {
      p_e := !!p_e;
   }
   p_acc := !p_acc + ((int)!p_e >> 1);
}
```

Figure 4: The program from Figure 1, translated to CCured.

4 The Type System

In this section we describe the CCured type system for the language introduced in the previous section. The purpose of this type system is to maintain the separation between the statically typed and the untyped worlds, and to ensure that all well-typed programs can be made to run safely with the addition of appropriate run-time checks. The run-time checks are described as part of the operational semantics in Section 5. For now we concentrate on type checking, and we shall assume that the program contains complete pointer kind information.

The type system is expressed by means of the following three judgments:

Expression typing:	$\Gamma \vdash e : \tau$
Command typing:	$\Gamma \vdash c$
Convertibility:	$\tau \leq \tau'$

In these judgments Γ denotes a typing environment mapping variable names to types. Since we do not have declarations in our language, the typing environment is assumed to be provided externally. The derivation rules for the typing judgments are shown in Figure 5.

Observe in the typing rules that we check casts with respect to a convertibility relation on types, whose rules are defined at the bottom of the figure. We also have a special typing rule for creating a safe null pointer. Pointer arithmetic can be done on sequence and dynamic pointers. Memory operations are legal for safe and dynamic pointers. A dereference operation on a sequence pointer can be performed after the pointer is cast to a safe one. Notice also how the type `DYNAMIC` is used both for pointers into untyped areas and for the values stored into those areas.

The type convertibility relation captures the situations in which a cast or coercion is legal in CCured. Notice in the rules that any type can be converted to an integer but integers can be converted only to sequence or to dynamic pointers. However, whenever we convert an integer into a sequence or a dynamic pointer we obtain pointers for which dereferences are prevented by run-time checks. The last conversion rule is used for converting sequence pointers into safe pointers, in which case the referenced type cannot change. If this last conversion rule is used then our operational semantics inserts a run-time check to verify that the pointer being cast is within the bounds of its home area.

It is important to point out that in most cases casts act as conversions between different representations of values and in some cases they are also accompanied by run-time checks. The run-time manipulations that accompany casts make convertibility different from subtyping in several respects. First, convertibility extended with transitivity and viewed as a coercion-based subtyping relation [4], would be incoherent. For example, the series of coercions corresponding to $\texttt{DYNAMIC} \leq \texttt{int} \leq \texttt{DYNAMIC}$ have a different effect from the identity since even if we start with a perfectly usable pointer we end up with a pointer that has lost its capability to perform memory operations. Because of lack of coherence we cannot allow a general subsumption rule and instead we let the program control the use of conversions, through casts. Consequently, we do not have a transitivity rule and we rely instead on the programmer to obtain the same effect by using a sequence of casts.

5 Operational Semantics

In this section we describe the run-time checks that are necessary for CCured programs to run safely. We do this in the form of an operational semantics for CCured.

The execution environment consists of a mapping Σ from variable names to values, a set of allocated memory areas H (which we call *homes*), and a mapping M (the memory) from addresses within the homes to values. The mapping Σ is assumed to be provided externally just like the similar mapping Γ from the typing rules. In our language only the memory changes during the execution. In order to better expose the precise costs of using each kind of pointer we use a low-level representation of addresses as natural numbers. A home is represented by its starting address ($H \subseteq \mathbb{N}$) and for all homes we define a function $\texttt{size} : H \to \mathbb{N}$ whose value is the size of the home. We require the following properties of the set H and the function `size`:

- NULL: $0 \in H$ and $\texttt{size}(0) = 1$.
- DISJOINT: $\forall h, h', i, i'.\ (h \neq h' \ \wedge\ 0 \leq i < \texttt{size}(h) \ \wedge\ 0 \leq i' < \texttt{size}(h')) \ \Rightarrow\ h + i \neq h' + i'$.

We choose the size of the null home to be 1 in order to ensure that only the null pointer belongs to the null home. We write H^* for the set $H \setminus \{0\}$. A memory corresponding to a set of allocated homes is a mapping of addresses to values $M_H : \mathbb{N} \to \mathit{Values}$ such that its domain consists exactly of the addresses contained in the non-null homes:

$$\mathit{Dom}(M_H) = \{\, h + i \mid h \in H^* \wedge 0 \leq i < \texttt{size}(h) \,\}$$

Since the set of homes H does not change during the evaluation we often omit the subscript H from the memory. We define two operations on memory. We write $M(n)$ to denote the contents of the memory address n, and we write $M[{}^{v}\!/\!_{n}]$ to denote a new memory state obtained from M by storing the value v at address n. Both of these operations are defined only when n is a valid address ($n \in \mathit{Dom}(M)$).

Expressions:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \ \texttt{op}\ e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash (\tau) e : \tau} \qquad \frac{}{\Gamma \vdash (\tau\ \texttt{ref SAFE})0 : \tau\ \texttt{ref SAFE}}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{ref SEQ} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \tau\ \texttt{ref SEQ}} \qquad \frac{\Gamma \vdash e_1 : \texttt{DYNAMIC} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{DYNAMIC}} \qquad \frac{\Gamma \vdash e : \tau\ \texttt{ref SAFE}}{\Gamma \vdash\ !e : \tau} \qquad \frac{\Gamma \vdash e : \texttt{DYNAMIC}}{\Gamma \vdash\ !e : \texttt{DYNAMIC}}$$

Commands:

$$\frac{}{\Gamma \vdash \texttt{skip}} \qquad \frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1 ; c_2} \qquad \frac{\Gamma \vdash e : \tau\ \texttt{ref SAFE} \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e := e'} \qquad \frac{\Gamma \vdash e : \texttt{DYNAMIC} \quad \Gamma \vdash e' : \texttt{DYNAMIC}}{\Gamma \vdash e := e'}$$

Convertibility:

$$\frac{}{\tau \leq \tau} \qquad \frac{}{\tau \leq \texttt{int}} \qquad \frac{}{\texttt{int} \leq \tau\ \texttt{ref SEQ}} \qquad \frac{}{\texttt{int} \leq \texttt{DYNAMIC}} \qquad \frac{}{\tau\ \texttt{ref SEQ} \leq \tau\ \texttt{ref SAFE}}$$

Figure 5: The typing judgments.

Expressions:

$$\frac{}{\Sigma, M \vdash n \Downarrow n}\ \text{INT} \qquad \frac{\Sigma(x) = v}{\Sigma, M \vdash x \Downarrow v}\ \text{VAR} \qquad \frac{\Sigma, M \vdash e_1 \Downarrow n_1 \quad \Sigma, M \vdash e_2 \Downarrow n_2}{\Sigma, M \vdash e_1\ \texttt{op}\ e_2 \Downarrow n_1\ \texttt{op}\ n_2}\ \text{OP}$$

Casts:

$$\frac{\Sigma, M \vdash e \Downarrow n}{\Sigma, M \vdash (\texttt{int}) e \Downarrow n}\ \text{C1} \qquad \frac{\Sigma, M \vdash e \Downarrow \langle h, n \rangle}{\Sigma, M \vdash (\texttt{int}) e \Downarrow h + n}\ \text{C2}$$

$$\frac{\Sigma, M \vdash e \Downarrow n}{\Sigma, M \vdash (\tau\ \texttt{ref SEQ}) e \Downarrow \langle 0, n \rangle}\ \text{C3} \qquad \frac{\Sigma, M \vdash e \Downarrow \langle h, n \rangle}{\Sigma, M \vdash (\tau\ \texttt{ref SEQ}) e \Downarrow \langle h, n \rangle}\ \text{C4}$$

$$\frac{\Sigma, M \vdash e \Downarrow n}{\Sigma, M \vdash (\texttt{DYNAMIC}) e \Downarrow \langle 0, n \rangle}\ \text{C5} \qquad \frac{\Sigma, M \vdash e \Downarrow \langle h, n \rangle}{\Sigma, M \vdash (\texttt{DYNAMIC}) e \Downarrow \langle h, n \rangle}\ \text{C6}$$

$$\frac{\Sigma, M \vdash e \Downarrow n}{\Sigma, M \vdash (\tau\ \texttt{ref SAFE}) e \Downarrow n}\ \text{C7} \qquad \frac{\Sigma, M \vdash e \Downarrow \langle h, n \rangle \quad \boxed{0 \leq n < \texttt{size}(h)}}{\Sigma, M \vdash (\tau\ \texttt{ref SAFE}) e \Downarrow h + n}\ \text{C8}$$

Pointer arithmetic:

$$\frac{\Sigma, M \vdash e_1 \Downarrow \langle h, n_1 \rangle \quad \Sigma, M \vdash e_2 \Downarrow n_2}{\Sigma, M \vdash e_1 \oplus e_2 \Downarrow \langle h, n_1 + n_2 \rangle}\ \text{ARITH}$$

Memory reads:

$$\frac{\Sigma, M \vdash e \Downarrow n \quad \boxed{n \neq 0}}{\Sigma, M \vdash\ !e \Downarrow M(n)}\ \text{SAFERD} \qquad \frac{\Sigma, M \vdash e \Downarrow \langle h, n \rangle \quad \boxed{h \neq 0} \quad \boxed{0 \leq n < \text{size}(h)}}{\Sigma, M \vdash\ !e \Downarrow M(h + n)}\ \text{DYNRD}$$

Commands:

$$\frac{}{\Sigma, M \vdash \texttt{skip} \Longrightarrow M}\ \text{SKIP} \qquad \frac{\Sigma, M \vdash c_1 \Longrightarrow M' \quad \Sigma, M' \vdash c_2 \Longrightarrow M''}{\Sigma, M \vdash c_1 ; c_2 \Longrightarrow M''}\ \text{CHAIN}$$

$$\frac{\Sigma, M \vdash e_1 \Downarrow n \quad \boxed{n \neq 0} \quad \Sigma, M \vdash e_2 \Downarrow v_2}{\Sigma, M \vdash e_1 := e_2 \Longrightarrow M[{}^{v_2}\!/\!_{n}]}\ \text{SAFEWR} \qquad \frac{\Sigma, M \vdash e_1 \Downarrow \langle h, n \rangle \quad \boxed{h \neq 0} \quad \boxed{0 \leq n < \text{size}(h)} \quad \Sigma, M \vdash e_2 \Downarrow v_2}{\Sigma, M \vdash e_1 := e_2 \Longrightarrow M[{}^{v_2}\!/\!_{h+n}]}\ \text{DYNWR}$$

Figure 6: The operational semantics. The boxed premises are the run-time checks that CCured uses.

The values of integer and safe pointer expressions are plain integers (without any representation overhead over C), while the values of sequence and dynamic pointer expressions is of the form $\langle h, n\rangle$:

$$\textit{Values} \quad v ::= n \quad | \quad \langle h, n\rangle$$

The latter kind of pointer value carries with it its "identity" (represented by its home). The home is used both to check if the pointer is actually an integer converted to a pointer (when the home is 0), or otherwise to retrieve the size of the home while performing a bounds check.

The operational semantics is defined by means of two judgments. We write $\Sigma, M \vdash e \Downarrow v$ to say that in the environment Σ and in the memory state denoted by M the expression e evaluates to value v. For commands we use a similar judgment $\Sigma, M \vdash c \Longrightarrow M'$ but in this case the result is a new memory state. The derivation rules for these two judgments are given in Figure 6.

Notice that we have eight rules for casts, one rule for each combination of destination type and form of the value being cast. The rules `C3` and `C5` show that an integer is converted to a sequence or to a dynamic pointer by using a null home. The rule `C7` applies when we cast the integer 0 or a safe pointer to another safe pointer, while the rule `C8` applies for casts from sequence pointers to safe pointers. In this latter case we must perform a bounds check. Here and in the rules to follow we mark such run-time checks with a box around them. Other instances of run-time checks are for memory operations. If the memory operation uses a safe pointer then only a null-pointer check must be done, otherwise a non-pointer check and a bounds check must be done.

The typing rules from Figure 5 suggest that we can perform a sequence of conversions τ `ref SEQ` $\leq$ `int` $\leq \tau$ `ref SEQ` or even a similar one where the destination type is τ' `ref SEQ`. This is indeed legal in CCured but the operational rules show that when starting with a pointer value $\langle h, n\rangle$ we end up after these two conversions with the value $\langle 0, h{+}n\rangle$, which is a pointer value that cannot be used for reading and writing. This property is quite important in practice: programs that cast pointers into integers and then back to pointers will not be able to use the resulting pointers as memory addresses. We discuss this issue further in Section 8.

5.1 Type Safety

The type system described in Section 4 enforces the separation between the typed and the untyped worlds. The operational semantics of Section 5 describes the run-time checks we perform for each operation on various pointer kinds. In this section we formalize and outline a proof of the resulting safety guarantees we obtain for CCured programs.

For each *non-null home* we define its `kind` as either $\textit{Typed}(\tau)$, meaning that it contains a number of values of type τ and has only safe and sequence pointers of base type τ pointing to it, or as *Untyped*, meaning that it contains a number of values of type `DYNAMIC` and has only pointers of type `DYNAMIC` pointing to it.

Then we define for each type τ the set $\|\tau\|_H$ of valid values of that type. As the notation suggests, this set depends on the current set of homes:

$$\begin{array}{lcl}
\|\texttt{int}\|_H & = & \mathbb{N} \\
\|\texttt{DYNAMIC}\|_H & = & \{\langle h, n\rangle \mid h \in H \land \\
 & & \qquad (h = 0 \lor \texttt{kind}(h) = \textit{Untyped})\} \\
\|\tau\ \texttt{ref SEQ}\|_H & = & \{\langle h, n\rangle \mid h \in H \land \\
 & & \qquad (h = 0 \lor \texttt{kind}(h) = \textit{Typed}(\tau))\} \\
\|\tau\ \texttt{ref SAFE}\|_H & = & \{h + i \mid h \in H \land 0 \leq i < \texttt{size}(h) \land \\
 & & \qquad (h = 0 \lor \texttt{kind}(h) = \textit{Typed}(\tau))\}
\end{array}$$

We extend the notation $v \in \|\tau\|_H$ element-wise to the corresponding notation for environments $\Sigma \in \|\Gamma\|_H$ (meaning $\forall x \in \textit{Dom}(\Sigma).\ \Sigma(x) \in \|\Gamma(x)\|_H$).

At all times during the execution, the contents of each memory address must correspond to the typing constraints of the home to which it belongs. We say that such a memory is well-formed (written $\textit{WF}(M_H)$), a property defined as follows:

$$\begin{array}{l}
\textit{WF}(M_H) \stackrel{def}{=} \\
\quad \forall h \in H^*.\ \forall i \in \mathbb{N}. \\
\qquad 0 \leq i < \texttt{size}(h) \Rightarrow \\
\qquad\quad (\texttt{kind}(h) = \textit{Untyped} \Rightarrow M(h+i) \in \|\texttt{DYNAMIC}\|_H \\
\qquad\quad \land\ \texttt{kind}(h) = \textit{Typed}(\tau) \Rightarrow M(h+i) \in \|\tau\|_H)
\end{array}$$

There are several reasons why the evaluation of an expression or a command can fail. The most obvious is that a boxed run-time check can fail. We actually consider this to be safe behavior. Another reason for failure is that operands can evaluate to unexpected values, such as if the second operand of $\oplus$ evaluates to a value of the form $\langle h, n\rangle$. The third reason is that the operations on memory are undefined if they involve invalid addresses. We state below two theorems saying essentially that the last two reasons for failure cannot happen in well-typed CCured programs.

In order to state a progress theorem we want to distinguish between executions that stop because memory safety is violated (i.e. trying to access an invalid memory location) and executions that stop because of a failed run-time check (the boxed hypotheses in the rules of Figure 6). We accomplish this by introducing a new possible outcome of evaluation. We say that $\Sigma, M \vdash e \Downarrow \textit{CheckFailed}$ when one of the run-time checks fails during the evaluation of the expression e. Similarly, we say that $\Sigma, M \vdash c \Longrightarrow \textit{CheckFailed}$ when the execution of the command c results in a failed run-time check. Technically, this means that we add derivation rules that initiate the *CheckFailed* result when one of the run-time check fails and also rules that propagate the *CheckFailed* outcome from the subexpressions to the enclosing expression.

Theorem 1 (Progress and type preservation) *If* $\Gamma \vdash e : \tau$ *and* $\Sigma \in \|\Gamma\|_H$ *and* $\textit{WF}(M_H)$ *then either* $\Sigma, M_H \vdash e \Downarrow \textit{CheckFailed}$ *or else* $\Sigma, M_H \vdash e \Downarrow v$ *and* $v \in \|\tau\|_H$*.*

Theorem 2 (Progress for commands) *If* $\Gamma \vdash c$ *and* $\Sigma \in \|\Gamma\|_H$ *and* $\textit{WF}(M_H)$ *then either* $\Sigma, M_H \vdash c \Longrightarrow \textit{CheckFailed}$ *or else* $\Sigma, M_H \vdash c \Longrightarrow M'_H$ *and* $\textit{WF}(M'_H)$*.*

The proofs of these theorems are fairly straightforward inductions on the structure of the typing derivations. Note that the progress theorems state more than just memory

Expressions and commands:

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{ref}\ q \mapsto C_1 \quad \Gamma \vdash e_2 : \texttt{int} \mapsto C_2}{\Gamma \vdash e_1 \oplus e_2 : \tau\ \texttt{ref}\ q \mapsto C_1 \cup C_2 \cup \{q \neq \texttt{SAFE}\}} \qquad \frac{\Gamma \vdash e : \tau' \mapsto C \quad \tau' \leq \tau \mapsto C'}{\Gamma \vdash (\tau)e : \tau \mapsto C \cup C'} \qquad \frac{}{\Gamma \vdash (\tau\ \texttt{ref}\ q)0 : \tau\ \texttt{ref}\ q \mapsto \emptyset}$$

$$\frac{\Gamma \vdash e : \tau\ \texttt{ref}\ q \mapsto C}{\Gamma \vdash !e : \tau \mapsto C} \qquad \frac{\Gamma \vdash e_1 : \tau\ \texttt{ref}\ q \mapsto C_1 \quad \Gamma \vdash e_2 : \tau_2 \mapsto C_2 \quad \tau_2 \leq \tau \mapsto C_3}{\Gamma \vdash e_1 := e_2 \mapsto C_1 \cup C_2 \cup C_3}$$

Convertibility:

$$\frac{}{\tau \leq \texttt{int} \mapsto \emptyset} \qquad \frac{}{\texttt{int} \leq \tau\ \texttt{ref}\ q \mapsto \{q \neq \texttt{SAFE}\}} \qquad \frac{}{\tau_1\ \texttt{ref}\ q_1 \leq \tau_2\ \texttt{ref}\ q_2 \mapsto \{q_1 \preceq q_2\} \cup \{q_1 = q_2 = \texttt{DYNQ} \vee \tau_1 \approx \tau_2\}}$$

Figure 7: Constraint generation rules.

safety. They also imply that well-typed computations of non-dynamic type are type preserving, similar to corresponding results for a type-safe language. This means that CCured is memory safe and is also type safe for the non-dynamic fragment.

6 Type Inference

So far we have considered the case of a program that is written using the CCured type system. Our implementation does allow the programmer to write such programs directly in C with the pointer kinds specified using the `__attribute__` keyword of GCC. But our main goal is to be able to use CCured with existing, un-annotated C programs. For this purpose we have designed and implemented a type inference algorithm that, given a C program, constructs a set of pointer-kind qualifiers that make the program well-typed in the CCured type system.

Our inference algorithm can operate either on the whole program, or on modules whose interfaces have been annotated with pointer-kind qualifiers. We rely on the fact that the C program already uses types of the form "τ `ref`". All we need is to discover for each occurrence of the pointer type constructor whether it should be safe, sequence or dynamic. To describe the inference algorithm we extend the CCured type language with the pointer type "τ `ref` q", where q is a qualifier variable ranging over the set of qualifier values $\{\texttt{SAFE}, \texttt{SEQ}, \texttt{DYNQ}\}$ (where `DYNQ` is the qualifier associated with the `DYNAMIC` type).

The inference algorithm starts by introducing a qualifier variable for each syntactic occurrence of the pointer type constructor in the C program. We then scan the program and collect a set of constraints on these qualifier variables. Next we solve the system of constraints to produce a substitution $\mathcal{S}$ of qualifier variables with qualifier values and finally we apply the substitution to the types in the C program to produce a CCured program.

The substitution $\mathcal{S}$ is applied to types using the following rules:

$$\begin{array}{lcl} \mathcal{S}(\texttt{int}) & = & \texttt{int} \\ \mathcal{S}(\tau\ \texttt{ref}\ q) & = & \begin{cases} \texttt{DYNAMIC} & \text{if } \mathcal{S}(q) = \texttt{DYNQ} \\ \mathcal{S}(\tau)\ \texttt{ref}\ \mathcal{S}(q) & \text{otherwise} \end{cases} \end{array}$$

Note that when the qualifier q is substituted with `DYNQ` we ignore the referenced type (τ) of the pointer, which is consistent with the idea that for the dynamic pointers we should not count on the declared referenced type. `DYNAMIC` pointers never point to typed areas and thus the inference algorithm is designed to infer only `DYNQ` qualifiers in the referenced types of `DYNQ` pointers.

The overall strategy of inference is to find as many `SAFE` and `SEQ` pointers as possible. Simply making all qualifiers `DYNQ` yields always a well-typed solution, but `SAFE` and `SEQ` pointers are preferred.

1. Constraint Collection. We collect constraints using a modified typing judgment written $\Gamma \vdash e : \tau \mapsto C$ and meaning that by scanning the expression e in context Γ we inferred type τ along with a set of constraints C. We also use the auxiliary judgments $\tau \leq \tau' \mapsto C$ to collect constraints corresponding to the convertibility relation, and $\Gamma \vdash c \mapsto C$ to express constraint collection from commands. The intent is that a solution to the set of constraints C, when applied as a substitution to the elements appearing before the symbol $\mapsto$, yields a valid typing judgment of the corresponding syntactic form in CCured. The rules for the constraint collection judgments are shown in Figure 7.

The constraints for pointer arithmetic are fairly straightforward and those for casts are expressed as a separate convertibility judgment. For memory reads and writes, we must bridge the gap between the rules of C and the rules of CCured. Specifically, we allow memory access through `SEQ` (not just `SAFE`) pointers, and we allow `int`s to be read or written through `DYNAMIC` pointers. In both cases, an implicit cast is inserted to yield a valid CCured program. In a memory write we allow for a conversion of the value being written to the type of the referenced type.

To express the convertibility constraints in a concise way we introduce a convertibility relation on qualifier values, which essentially says `SEQ` can be cast to `SAFE`:

$$q \preceq q' \stackrel{def}{=} q = q' \ \vee\ (q = \texttt{SEQ} \wedge q' = \texttt{SAFE})$$

Finally, to capture the requirement that all `DYNAMIC` pointers point only to dynamically typed areas, for each type of the form τ `ref` q' `ref` q we collect a `POINTSTO` constraint $q = \texttt{DYNQ} \Rightarrow q' = \texttt{DYNQ}$.

After constraint generation we end up with a set containing the following four kinds of constraints:

$$\begin{array}{ll} \texttt{ARITH:} & q \neq \texttt{SAFE} \\ \texttt{CONV:} & q \preceq q' \\ \texttt{POINTSTO:} & q = \texttt{DYNQ} \Rightarrow q' = \texttt{DYNQ} \\ \texttt{TYPEEQ:} & q = q' = \texttt{DYNQ} \vee \tau_1 \approx \tau_2 \end{array}$$

The constraint $\tau_1 \approx \tau_2$ requires that a valid solution is a substitution $\mathcal{S}$ that makes the types $\mathcal{S}(\tau_1)$ and $\mathcal{S}(\tau_2)$ identical. This notion is made more precise below.

2. Constraint Normalization. The next step is to normalize the generated constraints into a simpler form. Notice that the system of constraints we have generated so far has conditional constraints. The `POINTSTO` constraints are easy to handle because we can ignore them as long as the qualifier q on the left is unknown, and if it becomes `DYNQ`, we add the constraints "$q' = \texttt{DYNQ}$" to the system of constraints. If q remains unknown at the end of the normalization process we will make it `SAFE` or `SEQ`.

However, the same is not true of the `TYPEEQ` constraints. If we postpone such constraints and the qualifiers involved remain unconstrained we would like to make them both `SAFE` or `SEQ` (to mimimize the number of `DYNAMIC` pointers). But to do that we must introduce in the system the type equality constraint, which might lead to contradictions that require backtracking. Fortunately there is a simple solution to this problem. We start by simplifying the `TYPEEEQ` constraint based on the possible forms of the types τ_1 and τ_2:

$$\begin{array}{lll} q = q' = \texttt{DYNQ} \vee \texttt{int} \approx \texttt{int} & \mapsto & \emptyset \\ q = q' = \texttt{DYNQ} \vee \texttt{int} \approx \tau_2 \; \texttt{ref} \; q_2 & \mapsto & \{q = \texttt{DYNQ}, \\ & & \; q' = \texttt{DYNQ}\} \\ q = q' = \texttt{DYNQ} \vee \tau_1 \; \texttt{ref} \; q_1 \approx \texttt{int} & \mapsto & \{q = \texttt{DYNQ}, \\ & & \; q' = \texttt{DYNQ}\} \\ q = q' = \texttt{DYNQ} \vee \tau_1 \; \texttt{ref} \; q_1 \approx \tau_2 \; \texttt{ref} \; q_2 & \mapsto & \{q = q'\} \cup C \\ \multicolumn{3}{c}{\text{where } q_1 = q_2 = \texttt{DYNQ} \vee \tau_1 \approx \tau_2 \mapsto C} \end{array}$$

The only subtlety is in the last rule. We observe that a constraint of the form "$q = q' = \texttt{DYNQ} \vee \tau_1 \; \texttt{ref} \; q_1 \approx \tau_2 \; \texttt{ref} \; q_2$" arises only when the types "$\tau_1 \; \texttt{ref} \; q_1 \; \texttt{ref} \; q$" and "$\tau_2 \; \texttt{ref} \; q_2 \; \texttt{ref} \; q'$" appear in the program. This means that the following `POINTSTO` constraints also exist:

$$\begin{array}{c} q = \texttt{DYNQ} \Rightarrow q_1 = \texttt{DYNQ} \\ q' = \texttt{DYNQ} \Rightarrow q_2 = \texttt{DYNQ} \end{array}$$

This in turn means that the disjunct $q = q' = \texttt{DYNQ}$ in the last reduction rule is redundant and can be eliminated.

After simplifying all `TYPEQ` constraints, the normalized system has only the following kinds of constraints:

$$\begin{array}{ll} \texttt{ARITH:} & q \neq \texttt{SAFE} \\ \texttt{CONV:} & q \preceq q' \\ \texttt{POINTSTO:} & q = \texttt{DYNQ} \Rightarrow q' = \texttt{DYNQ} \\ \texttt{ISDYN:} & q = \texttt{DYNQ} \\ \texttt{EQ:} & q = q' \end{array}$$

3. Constraint Solving. The final step in our algorithm is to solve the remaining set of constraints. The algorithm is quite simple:

3.1 Propagate the `ISDYN` constraints using the constraints `EQ`, `CONV`, and `POINTSTO`. After this is done all the other qualifier variables can be made `SEQ` or `SAFE`, as follows:

3.2 All qualifier variables involved in `ARITH` constraints are set to `SEQ` and this information is propagated using the constraints `EQ` and `CONV` (in this latter case the `SEQ` information is propagated only from q' to q, or against the direction of the cast).

3.3 We make all the other variables `SAFE`.

Essentially, we find first the minimum number of `DYNQ` qualifiers. Among the remaining qualifiers we find those on which pointer arithmetic is performed and we make them `SEQ`, and the remaining qualifiers are `SAFE`. This solution is the best one possible in terms of maximizing the number of `SAFE` and `SEQ` pointers.

The whole type inference process is linear in the size of the program. A linear number of qualifier variables is introduced (one for each syntactic occurrence of a pointer type constructor), then a linear number of constraints is created (one for each cast or memory read or write in the program). During the simplification of the `TYPEQ` constraints the number of constraints can get multiplied by the maximum nesting depth of a qualifier in a type. Finally, constraint solving is linear in the number of constraints.

7 Handling the Rest of C

In the interest of clarity we have formalized in this paper only a small subset of the CCured dialect of C. Our implementation handles the entire C programming language along with most of the extensions in the GNU C dialect. In this section we discuss informally how we handle the rest of the C programming language. The full details are presented in a forthcoming paper.

In the `DYNAMIC` world, structures and arrays are simply alternative notations for saying how many bytes of storage to allocate. In the `SAFE` world, structures accesses are required to respect the types of all fields. For example, it is possible to have a `SAFE` pointer to a structure field, but you cannot perform arithmetic on such a pointer. We treat unions as syntactic sugar for casts.

Explicit deallocation is currently ignored, and the Boehm-Weiser conservative garbage collector [3] is used to reclaim storage. However, the CCured system maintains enough type information to allow the use of a precise collector; this may be future work.

The address-of operator in C can yield a pointer to a stack-allocated variable. The variable to which the pointer points may be inferred to live in the `DYNAMIC` or `SAFE` worlds, depending on how the pointer is used. The only difficulty is that the storage will be deallocated when the function returns, so the CCured run-time checks ensure no stack pointer ever gets written into the heap or globals. This restriction allows the common use of address-of to implement call-by-reference; for other uses, the storage in question may have to be allocated on the heap instead.

DYNAMIC function pointers and variable-argument functions are also handled in CCured, by passing a hidden argument which specifies the types of all arguments passed. The hidden argument is then checked in the callee, and parameters interpreted accordingly. Among other things, this level of checking is sufficient to detect format string errors.

Certain C library functions must be handled specially. Several functions (of which `malloc` is the most important) are treated polymorphically, lest all dynamically-allocated data be marked DYNAMIC. A few others impose constraints on argument qualifiers: e.g., `memcpy` internally does pointer arithmetic, and hence cannot accept SAFE pointers.

8 Source Changes

The CCured type system and inference algorithm are designed to minimize the amount of source changes required to conform to its restrictions. However, there are still a few cases in which legal C programs will stop with a failed run-time check. In those cases manual intervention is necessary.

One common situation is when the program stores a pointer in a variable declared to hold an integer, then casts it back to a pointer and dereferences the pointer. In some cases, it suffices to change the variable's declaration from from (say) `unsigned long` to `void*`. This type will certainly be marked DYNAMIC, but it will work. For other programs, we may be able to replace casts with pointer arithmetic. For example, if e is a sequence or dynamic pointer expression then the legal CCured expression "$e \oplus (n - (int)e)$" is effectively a cast of the integer n to a pointer (with the same home as e). As a last resort, it is possible to query the garbage collector at run time to find the home and type of any pointer, but so far this has not been necessary.

Another problem in otherwise legal C code is the interaction between `sizeof` and our fat pointers: one must change occurrences of `sizeof(`*type*`)` to `sizeof(`*expression*`)`, whenever *type* contains pointers. A typical example, allocating an array of 5 integer pointers, is

```
int **p = (int**)malloc(5 * sizeof(int*));
```

This code will always allocate space for 5 SAFE pointers, even if `p` is inferred to point to SEQ or DYNAMIC pointers. This code must be changed to

```
int **p = (int**)malloc(5 * sizeof(*p));
```

so the size passed to `malloc` is related to the size of `*p`.

While most uses of address-of are to implement call-by-reference, some programs attempt to store stack pointers into memory. Among the programs we have compiled with CCured, only two (the SPECINT95 benchmarks `li` and `ijpeg`) do this. The solution is to annotate certain local variables with a qualifier that causes them to be allocated on the heap. For `li`, which makes fairly extensive use of this feature, this results in a performance penalty of about 25%.

When CCured changes the representation of pointers, this can lead to problems when calling functions in libraries that were not compiled with CCured. The typical solution is to write wrapper functions which translate between two-word and one-word arguments and return values. The wrapper must do the run-time checks assocated with the pointers, before passing them to the underlying library.

The wrapper solution works well for the standard C library. However, we expect to encounter difficulties when interoperating with third-party libraries whose interface involves passing pointers to large structures which themselves contain pointers. We are experimenting with an alternative implementation scheme in which the bookkeeping information for sequence and dynamic pointers that escape the CCured world are kept in a global table so that we do not have to change the representation of exported data structures.

9 Experiments

We ran though our translator several C programs ranging in size from 400 (`treeadd`) to 30,000 (`ijpeg`) lines of code (including whitespace and comments), with several goals. First, we wanted to measure the performance impact of the run-time checks introduced by our translator. Second, we wanted to see how effective our inference system is at eliminating these checks. Finally, we investigated what changes to the program source are required to make the program run under the CCured restrictions.

We used several test cases, some from SPECINT95 [26]: `compress` is LZW data compression; `go` plays the board game Go; `ijpeg` compresses image files; `li` is a Lisp interpreter; and some from the Olden benchmark suite [6], a collection of small, compute-intensive kernels: `bh` is an n-body simulator; `bisort` is a sorting algorithm; `em3d` solves an elecromagnetism problem; `health` simulates Colombia's health care system; `mst` computes minimum spanning trees; `perimeter` computes perimeters of regions in images; `power` simulates power market prices; `treeadd` simply builds a binary tree; `tsp` uses a greedy algorithm to approximately solve random Traveling Salesman Problem instances; and `voronoi` constructs Voronoi diagrams.

Most of the source changes needed to run these benchmarks were simple syntactic adjustments, such as adding (or correcting) prototypes and marking `printf`-like functions. A few benchmarks required changing `sizeof` (prevalent in `ijpeg`) or moving locals into the heap (for `li`). No program required changes to the data structures or other basic design elements. A number of remaining bugs in our implementation prevents us from applying CCured to the other benchmarks in the SPECINT95 suite.

The running time (median of five) of each of the benchmarks is shown in Figure 8. The measurements were made on an otherwise quiescent 1GHz AMD Athlon, 768MB Linux machine, using the `gcc-2.95.3` compiler with `-O2` or `-O3` optimization level (depending on benchmark size).

In all cases the pointer kind inference was performed over the whole program. However, because inference time is linear in the size of the program (as argued in Section 6), we have not observed scalability problems with our whole-program approach. In fact, our biggest scalability problem is with the optimizer in the C compiler that consumes our output (presented as a single, large C source file).

Most of the benchmarks have between 30% and 150% slowdown. To measure the effectiveness of our inference al-

Name	Lines of code	Orig. time	CCured sf/sq/d	**ratio**	Purify ratio
SPECINT95					
compress	1590	9.586s	87/12/0	**1.25**	28
go	29315	1.191s	96/4/0	**2.01**	51
ijpeg	31371	0.963s	36/1/62	**2.15**	30
li	7761	0.176s	93/6/0	**1.86**	50
Olden					
bh	2053	2.992s	80/18/0	**1.53**	94
bisort	707	1.696s	90/10/0	**1.03**	42
em3d	557	0.371s	85/15/0	**2.44**	7
health	725	2.769s	93/7/0	**0.94**	25
mst	617	0.720s	87/10/0	**2.05**	5
perimeter	395	4.711s	96/4/0	**1.07**	544
power	763	1.647s	95/6/0	**1.31**	53
treeadd	385	0.613s	85/15/0	**1.47**	500
tsp	561	3.093s	97/4/0	**1.15**	66

Figure 8: CCured versus original performance. The measurements are presented as ratios, where 2.00 means the program takes twice as long to run when instrumented with CCured. The "sf/sq/d" column show the percentage of (static) pointer declarations which were inferred `SAFE`, `SEQ` and `DYNAMIC`, respectively.

gorithm we used CCured with a naive inference algorithm that makes all pointers `DYNAMIC`. The slowdown in this case is more significant (6 to 20 times slower) and it approaches that reported by other researchers [2, 13, 18, 19] who tried an all-run-time-checks approach to memory safety for C. For example, the most pointer-intensive benchmark is `li`, which runs 16 times slower if all pointers are blindly marked `DYNAMIC`; however, once the inference discovers that *all* the pointers are `SAFE` or `SEQ`, it is only twice as slow.

Program size has a big influence on how many of the pointers can be statically verified. Small programs like the Olden benchmark suite tend to have few data types, and they are used in straightforward ways. Large programs, especially those designed to be extended in the future, use pointers in many ways. In the case of `ijpeg`, it uses object-oriented downcasts throughout, and thus a large number of the pointers become `DYNAMIC`.

We discovered and fixed several bugs in the SPECINT95 benchmarks: `compress` and `ijpeg` each contain one array bounds violation, and `go` has (at least) eight array bounds violations and one use of an uninitialized integer variable as an array index. In each case we verified that fixing the bug did not change the program's eventual output (for the test vectors considered), which partially explains how these bugs survived for so long in otherwise well-tested programs.

Most of the bugs in `go` involved erroneous index arithmetic within large, multi-dimensional arrays. Finding these bugs demonstrates an advantage of our type-sensitive approach. If we simply marked all home areas as untyped, and only checked for errors when a pointer strayed out of its home area, we would miss errors that happen to stay within the intended home area. While we were originally motivated by performance to discover safe pointers, we found that doing so enhanced our bug-finding ability too.

The last column in Figure 8 shows the slowdown of these programs when instrumented with Purify (version 2001A) [10], a tool that instruments existing C binaries to detect memory errors and leaks by keeping two bits of storage for each byte in the heap (unallocated, uninitialized and initialized). However, Purify does not catch pointer arithmetic that yields a pointer to a separate valid region [13], a property that Fischer and Patil [20] show to be important. Purify tends to slow programs down by a factor of 10 or more, much more than CCured. Of course, Purify does not require source code, so may be applicable in more situations. Purify did find the uninitialized variable in `go`, but none of the other bugs, because the accesses in question did not stray far enough to be noticed.

10 Related Work

Abadi et al. [1] study the theoretical aspects of adding a `Dynamic` type to the simply-typed λ-calculus and discuss extensions to polymorphism and abstract data types. Thatte [28] extends their system to replace the `typecase` expressions with implicit casts. Their system does not handle reference types or memory updates and `Dynamic` types are introduced to add flexibility to the language. In contrast our system was designed to handle memory reads and writes, allows `DYNAMIC` values to be manipulated (e.g., via pointer arithmetic) without checking their tags, and uses `DYNAMIC` types to guarantee the safety of code that cannot be statically verified.

Chandra and Reps [8] present a method for physical type checking of C programs based on structure layout in the presence of casts. Their inference method can reason about casts between various structure types by considering the physical layout of memory. Our example in Section 2 would fail to type check in their system for the same reason that we must mark some of the pointers `DYNAMIC`: its safety cannot be guaranteed at compile time. Siff et al. [24] identify that many casts in C programs are safe upcasts and present a tool to check such casts.

The programming languages CLU [17], Cedar/Mesa [16] and Modula-{2+,3} [5] include similar notions of a dynamic type and a typecase statement. This idea can also be seen in CAML's exception type [22].

Other related work in this area falls into three broad categories: (1) extensions to C's type system, (2) adding run-time checks to C, and (3) removing run-time checks from LISP.

Previous efforts to extend C's type system usually deal with polymorphism. Smith et al. [25] present a polymorphic and provably type-safe dialect of C that includes most of C's features (and higher-order functions, which our current system handles weakly) but lacks casts and structures. Evans [9] describes a system in which programmer-inserted annotations and static checking techniques can find errors and anomalies in large programs. Ramalingam et al. [21] have presented an algorithm for finding the coarsest acceptable type for structures in C programs. Most such type systems and inference methods are presented as sources of information. In this paper we present a type and inference system with the goal of making programs safe.

There have been many attempts to bring some measure of safety to C in the past by trading space and speed for security. Previous techniques have been concerned with spatial access errors (array bounds checks and pointer arithmetic) and temporal access errors (touching memory that has been freed) but none of them use a static analysis of the form presented here. Kaufer et al. [14] present an interpretive scheme called Saber-C that can detect a rich class of errors (including uninitialized reads and dynamic type mismatches but not all temporal access errors) but runs about 200 times slower than normal. Austin et al. [2] store extra information with each pointer and achieve safety at the cost of a large (up to 540% speed and 100% space) overhead and a lack of backwards compatibility. Jones and Kelly [13] store extra information for run-time checks in a splay tree, allowing safe code to work with unsafe libraries. This results in a slowdown factor of 5 to 6. Fischer and Patil have presented a system that uses a second processor to perform the bounds checks [19]. The total execution overhead of a program is typically only 5% using their technique but it requires a dedicated second processor. Loginov et al. [18] store type information with each memory location, incurring a slowdown factor of 5 to 158. This extra information allows them to perform more detailed checks and they can detect when stored types mismatch declared types or union members are accessed out of order. While their tool and ours are similar in many respects their goal is to provide rich debugging information and ours is to make C programs safe while retaining efficiency. Steffen's `rtcc` compiler [27] is portable and adds object attributes to pointers but fails to detect temporal access errors and does not perform any check optimizations. In fact, beyond array bounds check elimination, none of these techniques use type-based static analysis to aggressively reduce the overhead of the instrumented code.

Finally, much work has been done to remove dynamic checks and tagging operations from LISP-like languages. Henglein [11] details a type inference scheme to remove tagging and untagging operations in LISP-like languages. The overall structure of his algorithm is very similar to ours (simple syntax-direct constraint generation, constraint normalization, constraint solving) but the domain of discourse is quite different because his base language is dynamically typed. In Henglein's system each primitive type constructor is associated with exactly one tag, so there is no need to deal with the pointer/array ambiguity that motivates our `SEQ` pointers. In C it is sometimes necessary to allocate an object as having a certain type and later view it as having another type: Henglein's system disallows this because tags are set at object creation time (that is, true C-style casts or unions are not fully supported [12]). Henglein is also able to sidestep update and aliasing issues because tagging and untagging create a new copy of the object (to which `set!` can be applied, for example) so one never has tagged and untagged aliases for the same item. His algorithm does not consider polymorphism or module compilation [15]. The CCured system uses a form of physical subtyping for pointers to structures and it is not clear how to extend Henglein's constraint normalization procedure in such a case.

Jagannathan et al. [12] use a more expensive and more precise flow-sensitive analysis called *polymorphic splitting* to eliminate run-time checks from higher-order call-by-value programs. Shields et al. [23] present a system in which dynamic typing and staged computation (run-time code generation) coexist: all deferred computations have the same dynamic type at compile-time and can be checked precisely at run-time. Such a technique can handle persisting dynamic data, a weakness of our current system. Soft type systems [7] also infer types for procedures and data structures in dynamically-typed programs. Advanced soft type systems [30] can be based on inclusion subtyping and can handle unions, recursive types and other complex language features. Finally, [15] presents a practical ML-style type inference system for LISP. As with Henglein [11], such systems start with a dynamically typed language and thus tackle a different core problem.

11 Conclusion and Future Work

The C programming language is the language of choice for systems programming because of its flexibility and control over the layout of data structures and the use of pointers. Unfortunately, this comes at the expense of type safety. In this paper we propose a scheme that combines program analysis and run-time checking to bring type safety to existing C programs by trading off some performance.

The key insight of this work is that even in C programs most pointers are used in such a way that they can be verified to be type safe using typing rules similar to those of strongly typed languages. Furthermore the rest of the pointers can be checked at run-time to ensure that they are indeed used safely. The entire approach hinges on the ability to infer accurately which pointers need to be checked at run time and which do not. We present a surprisingly simple type inference algorithm that is able to do just that.

Perhaps the most surprising result of our experiments is that in many C programs most pointers are perfectly safe (and our inference is able to discover that), which means that those programs are just as safe as if they had been written in a type-safe language. Consequently the cost of enforcing safety for many C programs is relatively low and even with a prototype implementation we were able to achieve overheads several times smaller than those of comparable tools that rely exclusively on run-time checking.

The two flavors of typed pointers that we present in this paper cover many of the programming paradigms encountered in C programs. But there are still other operations on pointers that could be statically proven safe, which our type system fails to recognize. The most important example is tagged union types with incompatible members, which the current CCured system flags as untyped; a special case is object-oriented "downcasts," used heavily by `ijpeg`. To handle these situations without resorting to the `DYNAMIC` sledgehammer, it would be useful to have a more expressive language of pointer types than our current system provides.

12 Acknowledgments

We would like to thank Alex Aiken and Jeff Foster for useful comments on earlier drafts of this paper and also Raymond

To, Aman Bhargava, S. P. Rahul, and Danny Antonetti for helping with the implementation of the CCured system.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. *SIGPLAN Notices*, 29(6):290–301, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[3] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 27:807–820, Sept. 1997.

[4] V. Breazu-Tannen, C. A. Gunter, and A. Scedrov. Computing with coercions. In *LISP and Functional Programming*, pages 44–60, 1990.

[5] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula3 report, 1989.

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

[7] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[8] S. Chandra and T. Reps. Physical type checking for C. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, volume 24.5 of *Software Engeneering Notes (SEN)*, pages 66–75. ACM Press, Sept. 6 1999.

[9] D. Evans. Static detection of dynamic memory errors. *ACM SIGPLAN Notices*, 31(5):44–53, 1996.

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

[11] F. Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 205–215, 1992.

[12] S. Jagannathan and A. Wright. Effective flow analysis for avoiding run-time checks. In *Proceedings of the Second International Static Analysis Symposium*, volume 983, pages 207–224. Springer-Verlag, 1995.

[13] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. *AADEBUG*, 1997.

[14] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: an interpreter-based programming environment for the C language. In *Proceedings of the Summer Usenix Conference*, pages 161–171, 1988.

[15] A. Kind and H. Friedrich. A practical approach to type inference for EuLisp. *Lisp and Symbolic Computation*, 6(1/2):159–176, 1993.

[16] B. Lampson. A description of the Cedar language. Technical Report CSL-83-15, Xerox Palo Alto Research Center, 1983.

[17] B. Liskov, R. R. Atkinson, T. Bloom, E. B. Moss, R. Schaffert, and A. Snyder. *CLU Reference Manual.* Springer-Verlag, Berlin, 1981.

[18] A. Loginov, S. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *Proceedings of FASE 2001: Fundamental Approaches to Software Engineering*, Apr. 2001.

[19] H. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In *Automated and Algorithmic Debugging*, pages 119–132, 1995.

[20] H. Patil and C. N. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software—Practice and Experience*, 27(1):87–110, Jan. 1997.

[21] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Symposium on Principles of Programming Languages*, pages 119–132, Jan. 1999.

[22] D. Remy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Symposium on Principles of Programming Languages*, pages 40–53, 1997.

[23] M. Shields, T. Sheard, and S. L. P. Jones. Dynamic typing as staged type inference. In *Symposium on Principles of Programming Languages*, pages 289–302, 1998.

[24] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *1999 ACM Foundations on Software Engineering Conference (LNCS 1687)*, volume 1687 of *Lecture Notes in Computer Science*, pages 180–198. Springer-Verlag / ACM Press, September 1999.

[25] G. Smith and D. Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(1–3):49–72, 1998.

[26] SPEC 95. Standard Performance Evaluation Corportation Benchmarks. *http://www.spec.org/osg/cpu95/CINT95*, July 1995.

[27] J. L. Steffen. Adding run-time checking to the Portable C Compiler. *Software—Practice and Experience*, 22(4):305–316, Apr. 1992.

[28] S. Thatte. Quasi-static typing. In *Conference record of the 17th ACM Symposium on Principles of Programming Languages (POPL)*, pages 367–381, 1990.

[29] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step toward automated detection of buffer overrun vulnerabilities. In *Network Distributed Systems Security Symposium*, pages 1–15, Feb. 2000.

[30] A. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 1997.