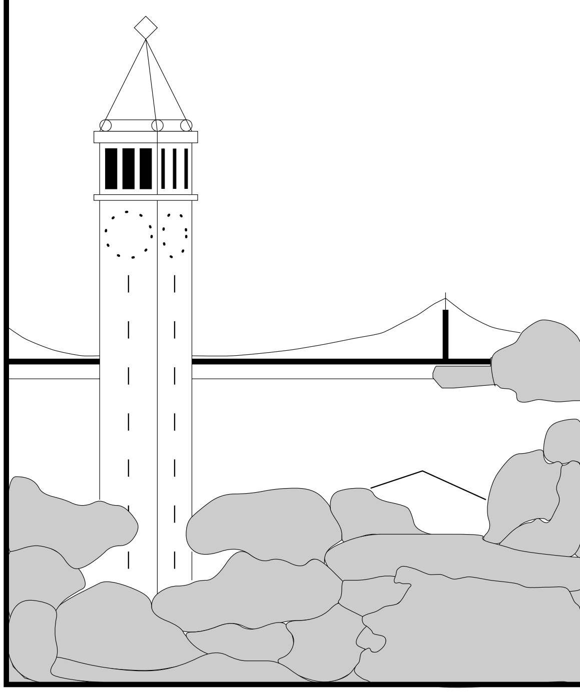
The CCured Type System and Type Inference

Westley Weimer
University of California, Berkeley
weimer@cs.berkeley.edu



Report No. UCB/CSD-03-1247

December 14, 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

The CCured Type System and Type Inference

by Westley Weimer

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor George C. Necula
Research Advisor

(Date)

* * * * * * *

Professor Alex Aiken
Second Reader

(Date)

The CCured Type System and Type Inference

Westley Weimer
University of California, Berkeley
weimer@cs.berkeley.edu

December 14, 2003

Abstract

We present CCured, a type and run-time check system that bring safety to the C programming language. CCured includes a type system for C programs that classifies pointers according to their usage and instructs a source-to-source translator to extend the program with run-time checks in order to guarantee memory safety. We show that the type system is sound in the presence of these run-time checks. CCured can be used on existing C programs thanks to a simple pointer-kind inferencer; on many programs this inferencer discovers that over 80% of the pointers are type-safe.

A significant contribution of our work with CCured is a notion of physical subtyping that is expressive enough to capture common C programming paradigms, is sound in the presence of pointer arithmetic and is suited for simple type inference.

This report formalizes the semantics of CCured and presents experimental evidence that such a combination of static analysis and run-time checking for C can make real system software like Apache modules, Linux device drivers, and network server software memory-safe with a reasonable performance cost and can find programming errors in instances where some existing tools like Purify cannot.

1 Introduction

Memory safety is the most important partial-correctness property in extensible or security-critical systems. A program is memory-safe when it accesses only memory addresses within the bounds of the objects it has allocated or been granted access to. Lack of memory safety is almost always indicative of a programming error and is especially dangerous in security-critical software that manipulates untrusted data. Memory safety is also highly desirable for extensible component-based systems to prevent inadvertent interference between components.

Two examples of mission-critical extensible systems are the Linux kernel and the Apache web server, both of which can be extended with dynamically loadable modules. Extensibility is critical to the success of these systems, since it allows modules to be written by people with domain expertise but with only minimal knowledge of the core system. However, extensibility has a steep price if memory safety cannot be guaranteed: the whole system or some module may crash due to an error in some other module. This weakness is exacerbated by the fast pace of development in the module space, and the lack of detailed system knowledge by module developers.

Security offers an even more compelling case for memory safety: more than half of all recent CERT security incidents arose due to memory safety bugs in network service software [WFBA00]. A first step towards ending widespread vulnerability is ensuring memory safety both for systems being developed and for those currently in use. One way to achieve this is to write programs in a strongly typed programming language. But for the majority of systems that are already in use and for those situations in which C is still the language of choice, it is important to be able to execute C programs safely.

Most C programs are in fact already memory-safe, although proving this fact is beyond the capability of a C type checker. This raises the question whether an alternate *sound* type system could be designed for C such that most C programs would be typeable. In this paper we describe the CCured type system that attempts to achieve just that. CCured keeps track of pointer usage and often discovers that most pointers in a program are used safely, requiring only a null check before dereference. It can also detect when pointers are involved in pointer arithmetic and thus require bounds checks before dereference. For some pointers CCured is not able to keep track statically of the type of referenced data and it adds both bounds checking and run-time type checking. A surprising result is that even with CCured's simple and intuitive type system, for many benchmark and real-world programs we can reduce the static count of pointers type-checked dynamically to under 1% of all the pointers in the program, and the number of those that require bounds checks to under 10%. Furthermore, CCured has a linear-time whole-program pointer-kind inference which makes it easily applicable to large, existing C programs.

Programs made type-safe by CCured typically perform within a factor of two of the original program, depending on the effectiveness of the inference. We believe this performance penalty is within tolerable limits, and for many programs, especially mission-critical ones, a fair price to pay for increased reliability and security.

CCured improves on previous attempts at making C safe [ABS94, JK97, KLP88, PF97] by recognizing that most pointers do not need to be checked extensively at run time. Since CCured does a significant part of the checking statically, it is often able to achieve an *order of magnitude* better performance on the processed code when compared with purely run-time approaches. It also exhibits fewer incompatibilities between processed code and unprocessed libraries than many prior approaches to checking C at run time. This is what enables us to use CCured on systems code with a negligible performance cost and makes us hopeful that a similar approach could be used in deployed code and not just during testing. In fact, CCured is not only more efficient than testing tools like Purify [HJ91] but also more effective in finding errors, a fact demonstrated by a number of errors that Purify misses and CCured catches in the SPECINT95 programs `go`, `ijpeg` and `compress`.

The CCured type system is inspired on one hand by work on dynamic types [ACPP91, Hen92] and on the other hand by work on physical subtyping [CR99, SCB$^+$99] (a form of subtyping for structured types based on the physical layout of the type, not the type structure). In this respect, a novel contribution of this paper is the integration of physical subtyping and pointer arithmetic. Both of these features must be present in a type system for C, yet their sound combination is subtle and has not been previously explored.

In a previous paper [NMW02a], we formalized and proved type soundness for a combination of statically typed and dynamically typed pointers for a small language of integers and pointers. In this paper we describe the theoretical changes that must be made to both the type system and also the type inference algorithm in order to handle a more realistic language with pointer arithmetic, nested aggregate types and function pointers. Section 2 describes the static and operational semantics of a C-like language that includes CCured pointer kinds. We prove that the type system is sound in Section 3. An inference algorithm that allows existing C programs to reap the benefits of CCured is described in Section 4. Additional C features that our implementation supports are covered in Section 5. We then describe a number experiments using CCured for a variety of programs including SPECINT95 benchmarks, extension modules for the Apache web server and the Linux kernel, and a File Transfer Protocol (FTP) server in Section 6. We discuss in more detail the connections with related work in Section 7.

2 C With Pointer Annotations

This section presents the static and operational semantics for a slight abstraction of the C language that has been extended to include pointer annotations.

2.1 The Language

Figure 1 presents a simplified version of the CCured type system that includes pointer-kind annotations. The `int` type is used for all of the scalar types in C (e.g., `float`, `short`, `char`). $\tau *_{\mathtt{q}}$ is the type of pointers with kind q that point to objects of type τ. The kind q associated with a pointer determines its capabilities and the sorts of run-time checks that must be performed to ensure safety. $\mathtt{void}*_{\mathtt{q}}$ behaves as it does in C: it is a pointer that cannot be dereferenced and must be cast to another type before being used. In CCured such pointers also have associated kinds q that determine their capabilities. The pointer kinds q associated with functions and structure or union fields are associated with pointers created by taking the address of such an object. Note that structures and unions have the same qualifier q associated with all fields. From the perspective of CCured pointer kinds, the address of the aggregate and the address of its elements all have the same kind and capabilities.

The only difference between CCured types and standard C types is the addition of the pointer kinds q. All of the other types presented can be found in existing C programs. The system as presented does not allow for recursive types (because structure and unions are not given names) or function return values. Recursive types do

CCured Types:	τ	::=	`int`	a scalar that is the same size as a pointer
		|	$\tau *_{\mathtt{q}}$	a pointer to type τ with kind q
		|	$\mathtt{void}*_{\mathtt{q}}$	a pointer with kind q that cannot be dereferenced
		|	$\tau[n]$	an array of n contiguous τs
		|	$(\tau_1, ..., \tau_m)*_{\mathtt{q}}$	a pointer to a function with m arguments and kind q
		|	$\mathtt{struct}\{f_1 : \tau_1 \;\; q, ..., f_n : \tau_n \;\; q\}$	a structure with n components
		|	$\mathtt{union}\{f_1 : \tau_1 \;\; q, ..., f_n : \tau_n \;\; q\}$	a union with n disjuncts
Pointer Kinds:	q	::=	`SAFE`	a null pointer or valid pointer to its declared type
		|	`SEQ`	a pointer with bounds information
		|	`FSEQ`	a pointer with upper bound information
		|	`DYN`	a pointer with dynamic type information

Figure 1: A simplified version τ of **the CCured type system** extended to include pointer kind annotations q.

Pointer Kind	**Arithmetic**	**Cast or Assign To**	**Check On Dereference**
`SAFE`	no	matching `SAFE`, `SEQ` or `FSEQ` pointers	null check
`SEQ`	yes, any	matching `SAFE`, `SEQ` or `FSEQ` pointers	both bounds
`FSEQ`	yes, increments only	matching `SAFE`, `SEQ` or `FSEQ` pointers	one bound, null check
`DYN`	yes, any	any `DYN` pointer	both bounds, run-time type check

Figure 2: A **summary of CCured pointer kinds** q and their capabilities.

not complicate the type or inference systems we will describe and were omitted for simplicity and brevity. Function return values can be simulated by passing a pointer as an extra argument and are likewise omitted.

Figure 2 shows a summary of CCured pointer kinds and their capabilities. `SAFE` pointers are either null or valid pointers to an object described by their static type. `SEQ` pointers carry memory bound information with them and may be subject to pointer arithmetic. `FSEQ` pointers carry upper-bound information and may be incremented via pointer arithmetic. If a `SEQ` or `FSEQ` pointer is within bounds then it points to an object of its static type. `DYN` pointers carry special meta-data information. Their static types cannot be trusted and run-time checks and tags are used to distinguish pointer and scalar values written through `DYN` pointers. `DYN` pointer kinds typically arise when a pointer is declared with a certain static type in the program but is later cast to an incompatible type.

Figure 3 shows an imperative language with declarations, functions, statements, expressions and simplified C-style lvalues. A normal C program can be converted to this language by introducing well-typed temporaries to hold the results of intermediate memory references inside expressions [NMW02b] and passing extra parameters to simulate function return values. Since our analyses are flow-insensitive, control-flow beyond function calls is not modeled. It is important to note that the language presented in Figure 3 is truly a subset of C. Some features, like local variables and function return values, have been removed; no features have been added. The treatment of lvalues used here, for example, does not change the underlying language and is merely a way of representing structural information in C.

A program is a list of declarations followed by an initial statement. A declaration list contains variable declarations and function declarations. A statement either assigns an expression value to an lvalue or calls a function through a function pointer. Expressions may be integers, compound expressions (notably including pointer arithmetic), casts of other expressions (where both the old type and the new type are explicit in the syntax), lvalues (where the type is again explicit in the syntax) or the addresses of lvalues. An lvalue refers to region of storage [KR88, NMW02b]. We model lvalues as host-offset pairs where the host denotes a large region of storage and the offset moves within that region. In the C expression `var.f1.f2`, `var` is the host and the fields `.f1` and `.f2` are offsets within it.

A well-formedness condition on types requires that for all $\tau *_{\mathtt{DYN}}$ or $v : \tau \;\; \mathtt{DYN}$, every kind q mentioned inside τ is also `DYN`. This condition is used to enforce type-safey in CCured. Since `DYN` pointers are dynamically-typed, it does not make sense to have a `DYN` reference to another pointer that is statically typed (e.g., a `SAFE` pointer). In CCured, statically typed pointers carry strong invariants about the values of their referents. A `DYN` pointer could be used (through a series of casts) to change any value within the bounds of an object it can point to. In particular,

Program:	P	$::=$	(D, s)	a program is a list of declarations and a statement
Declarations:	D	$::=$	$(v : \tau \;\; q) \; ; D$	declares v to have type τ, the pointer $\&v$ has kind q
		$\mid$	$(fun_i(v_1 : \tau_1 \;\; q_1, ...) \;\; q = s) \; ; D$	declares the function fun_i, the function pointer has kind q
		$\mid$	nil	
Statements:	s	$::=$	$s_1 \; ; s_2$	combines two statements in sequence
		$\mid$	$l =_\tau e$	an assignment statement copying a value of type τ
		$\mid$	$(*e_p)(e_1, ..., e_n)$	a function call through a function pointer
Expressions:	e	$::=$	n	an integer constant
		$\mid$	$e_1 + e_2$	pointer or scalar arithmetic
		$\mid$	$(\tau_1 \leftarrow \tau_2)e$	casts e from type τ_2 into type τ_1
		$\mid$	l_τ	treat the lvalue l as an expression with type τ
		$\mid$	$\&l$	treat the address of the lvalue l as a pointer expression
Lvalue:	l	$::=$	(h, o)	a host and offset pair defines an lvalue
Hosts:	h	$::=$	v	the lvalue lives inside a declared variable
		$\mid$	$*v$	the lvalue lives in memory
Offsets:	o	$::=$	$._s f \; . \; o$	accesses a structure field
		$\mid$	$._u f \; . \; o$	accesses a union disjunct (syntactic sugar for a cast)
		$\mid$	$[e] \; . \; o$	indexes into an array
		$\mid$	nil	

Figure 3: An **imperative language** that is a a slight abstraction of C.

it could change any statically-typed pointer it could reach, resulting in that statically-typed pointer referencing an invalid address. This would break the invariant on the statically-typed pointer and is thus disallowed. The world of DYN pointers and the world of statically-typed pointers are forever separated in CCured: DYN pointers may not point to statically-typed pointers.

In addition, function types are constrainted so that for all $(\tau_1, ..., \tau_m)*_{\mathtt{q}}$, either $q = \mathtt{SAFE}$ or $q = \mathtt{DYN}$ and $\tau_i = \mathtt{int}*_{\mathtt{DYN}}$. The $q = \mathtt{SAFE}$ or $q = \mathtt{DYN}$ condition captures the idea that pointer arithmetic does not make sense on function pointers, so we need only keep track of type information. The bounds information stored with a SEQ or FSEQ pointer would not be helpful. The condition on DYN function pointers that $\tau_i = \mathtt{int}*_{\mathtt{DYN}}$ simplifies the presentation of function arguments and is not necessary in our implementation. We are asking the programmer in our simplified language to cast DYN function arguments to pointers even if they would have been scalars and then cast them back inside the function body. Since a DYN pointer carries enough run-time type information to remember if an integer was stored inside it, this allows us to to simplify our presentation of the typing rules for function calls. In our implementation, this treatment of DYN function arguments happens automatically.

Finally, the kind associated with an array, structure or union type must be either DYN or SAFE. Such kinds are usually associated with the pointer formed by taking the address of an object. In CCured, the address of an object never has pointer kind SEQ or FSEQ. Instead, the address of an array of τs is typically treated as a SAFE pointer to an array of τs (i.e., $\tau[n]*_{\mathtt{SAFE}}$) which can then be cast to a SEQ or FSEQ pointer that ranges over the individual τs within the array (i.e., $\tau*_{\mathtt{SEQ}}$).

Now that we have outlined our simplified C-like language we will discuss the typing rules and static semantics for it, paying special attention to the role played by CCured's pointer kinds.

2.2 Static Semantics

The static semantics for our C-like language with pointer kinds is very similar to the static semantics for C but with special attention given to casts and pointer arithmetic. Pointer kinds and the physical layout of the underlying types in memory restrict casts, and pointer kinds restrict pointer arithmetic.

The type-checking judgments will make use of an environment Γ mapping variables to types and qualifiers. The type is the type of the variable and the qualifier is associated with the pointer formed by taking the address of the variable. We write $\Gamma[{}^{v}\!/_{\tau \;\; q}]$ to mean Γ modified so that v maps to $\tau \;\; q$. In addition, we will make use of a mapping V

Programs and Global Declarations:

$$\frac{\emptyset \vdash D : \Gamma \quad V, \Gamma \vdash s}{V \vdash (D, s)} \ \textit{program}$$

$$\frac{}{\Gamma \vdash nil : \Gamma} \ \textit{nil-decl} \qquad \frac{\Gamma_1[{}^{v}\!/\!{}_{\tau \ \ q}] \vdash D : \Gamma_2 \quad q \in \{\texttt{SAFE}, \texttt{DYN}\}}{\Gamma_1 \vdash (v : \tau \ \ q)\ ;\ D : \Gamma_2} \ \textit{var-decl}$$

$$\frac{\Gamma[{}^{v_1}\!/\!{}_{\tau_1 \ \ q_1}]...[{}^{v_n}\!/\!{}_{\tau_n \ \ q_n}] \vdash s \quad \Gamma_1[{}^{fun_i}\!/\!{}_{(\tau_1, ..., \tau_n)*_q \ \ q}] \vdash D : \Gamma_2 \quad q \in \{\texttt{SAFE}, \texttt{DYN}\}}{\Gamma_1 \vdash (fun_i(v_1 : \tau_1 \ \ q_1, ..., v_n : \tau_n \ \ q_n) \ \ q = s)\ ;\ D : \Gamma_2} \ \textit{fun-decl}$$

Statements:

$$\frac{V, \Gamma \vdash s_1 \quad V, \Gamma \vdash s_2}{V, \Gamma \vdash s_1\ ;\ s_2} \ \textit{seq} \qquad \frac{V, \Gamma \vdash l_\tau : \tau \quad V, \Gamma \vdash e : \tau \quad \tau \in \{\texttt{int}, \tau' *_q, \texttt{void}*_q, (\tau_1, ..., \tau_m)*_q\}}{V, \Gamma \vdash l =_\tau e} \ \textit{assign}$$

$$\frac{V, \Gamma \vdash e_p : (\tau_1, ..., \tau_n)*_q \quad V, \Gamma \vdash e_i : \tau_i \ \ (1 \le i \le n)}{V, \Gamma \vdash (*e_p)(e_1, ..., e_n)} \ \textit{call}$$

Expressions:

$$\frac{V, \Gamma \vdash e_1 : \tau_1 \quad V, \Gamma \vdash e_2 : \texttt{int} \quad \tau_1 \in \{\texttt{int}, \ \tau_2 *_q \ (q \neq \texttt{SAFE})\}}{V, \Gamma \vdash e_1 + e_2 : \tau_1} \ \textit{op} \qquad \frac{V, \Gamma \vdash e : \tau_2 \quad V \vdash \alpha(\tau_2) \le \alpha(\tau_1)}{V, \Gamma \vdash (\tau_1 \leftarrow \tau_2)e : \tau_1} \ \textit{cast}$$

$$\frac{}{V, \Gamma \vdash n : \texttt{int}} \ \textit{int} \qquad \frac{\Gamma \vdash h : \tau_1, q_1 \quad V, \Gamma, \tau_1, q_1 \vdash o : \tau_2, q_2}{V, \Gamma \vdash (h, o)_{\tau_2} : \tau_2} \ \textit{lvalue} \qquad \frac{\Gamma \vdash h : \tau_1, q_1 \quad V, \Gamma, \tau_1, q_1 \vdash o : \tau_2, q_2}{V, \Gamma \vdash \&(h, o) : \tau_2 *_{q_2}} \ \&\textit{lvalue}$$

Lvalue hosts:

$$\frac{\Gamma(v) = \tau \ \ q}{\Gamma \vdash v : \tau, q} \ \textit{varhost} \qquad \frac{\Gamma(v) = \tau *_{q_1} \ \ q_2}{\Gamma \vdash *v : \tau, q_1} \ \textit{memhost}$$

Lvalue offsets:

$$\frac{}{V, \Gamma, \tau, q \vdash nil : \tau, q} \ \textit{nil-off} \qquad \frac{\tau_1 = \tau_2[n] \quad V, \Gamma \vdash e : \texttt{int} \quad V, \Gamma, \tau_2, q_1 \vdash o : \tau_3, q_2}{V, \Gamma, \tau_1, q_1 \vdash ([e]\ .\ o) : \tau_3, q_2} \ \textit{index}$$

$$\frac{\tau_1 = \texttt{struct}\{..., f \ \ \tau_2 \ \ q_2, ...\} \quad V, \Gamma, \tau_2, q_2 \vdash o : \tau_3, q_3}{V, \Gamma, \tau_1, q_1 \vdash (._s f\ .\ o) : \tau_3, q_3} \ \textit{sfield}$$

$$\frac{\tau_1 = \texttt{union}\{..., f \ \ \tau_2 \ \ q_2, ...\} \quad V, \Gamma, \tau_2, q_2 \vdash o : \tau_3, q_3}{V, \Gamma, \tau_1, q_1 \vdash (._u f\ .\ o) : \tau_3, q_3} \ \textit{ufield}$$

Figure 4: **Typing rules** for the imperative language. V is a global description of `void*` equivalence classes (see Section 2.3). α, V and $\le$ determine the physical subtyping relationship detailed in Section 2.3.

from instances of the type `void*` to other concrete types in the program. V can be viewed as a global description of `void*` equivalence classes that is used to reason about subtyping in the presence of the otherwise-opaque type `void*`. We view each appearance of `void*` in the C program as a type variable. This is important because C programs often cast τ`*` to `void*` and then back again (e.g., when placing an object in a generic container). If `void*` were treated as an actual empty type, then the cast from `void*` back to τ`*` would look like an unsafe downcast. Treating `void*` as a type variable (i.e., equating it with τ`*`) allows us to recognize additional cases when such casts are safe.

The first typing judgment is for whole programs: $V \vdash (D, s)$. The judgment is true if the externally-provided `void*` mapping can be used to type-check the program (D, s), which is a list of declarations and an initial statement. Our typing judgment for declarations is of the form: $\Gamma_1 \vdash D : \Gamma_2$. The input environment is represented by Γ_1 and Γ_2 is the output environment. D is the declaration under consideration (information about which is typically added to Γ_1 to form Γ_2). Our typing judgment for statements has the form $V, \Gamma \vdash s$.

Expressions have a typing judgment of the form $V, \Gamma \vdash e : \tau$, where e is the expression under consideration and τ is its type. Lvalue hosts (which evaluate to the beginning of a region of storage) have a typing judgment of the form $\Gamma \vdash h : \tau, q$, where τ is the type of the object referenced by the host h and q is the qualifier associated with the pointer $\&(h, nil)$. Finally, lvalue offsets have a judgment of the form $V, \Gamma, \tau_1, q_1 \vdash o : \tau_2, q_2$. In that judgment, τ and q_1 are the type and qualifier associated with the surrounding lvalue host (and the pointer formed by taking its address) and τ_2 and q_2 are the resulting type and qualifier when that host pointer is adjusted by the offset o.

Figure 4 gives typing rules for the imperative language in Figure 3. The form of the typing derivation for an entire program is shown below:

$$\frac{\emptyset \vdash D : \Gamma \quad V, \Gamma \vdash s}{V \vdash (D, s)} \; \textit{program}$$

(D, s) is the program, comprised of declarations and statements. $\emptyset$ is an empty environment. Γ is formed from the declarations in D and is then used to type-check s.

The derivation for a variable declaration is shown below:

$$\frac{\Gamma_1[{}^{v}\!/\!_{\tau \;\; q}] \vdash D : \Gamma_2 \quad q \in \{\texttt{SAFE}, \texttt{DYN}\}}{\Gamma_1 \vdash (v : \tau \;\; q)\; ; D : \Gamma_2} \; \textit{var-decl}$$

The environment Γ_1 is extended to contain a mapping from v to the type τ and the pointer kind q. The kind q is associated with the pointer $\&v$ and is stored in the environment so that address-of expressions can be type-checked. In CCured, the kind associated with the address of a variable must be either `SAFE` or `DYN`. This is because pointer arithmetic cannot be safely applied to the address of a variable without a cast: the pointer after the arithmetic will point to another location on the stack or the data segment.[1] In the case of arrays, we note that C implicitly converts between the name (and thus the address) of an array and a pointer to its first element. CCured makes this cast explicit, turning for example a `SAFE` pointer to the address of an array into a `SEQ` pointer to its first element. The `SEQ` pointer can then be used to walk over the elements of the array. Thus the address of a variable is always either `SAFE` (if it is always cast in a way that can be verified statically) or `DYN` (if its type must be checked at run-time). Since all of the global declarations occur before any of the statements, the typing derivations for statements and expressions never yields a new V or Γ.

The typing derivation for statements verifies that a statement is well-typed. The assignment derivation is shown below:

$$\frac{V, \Gamma \vdash l : \tau \quad V, \Gamma \vdash e : \tau \quad \tau \in \{\texttt{int}, \tau' *_q, \texttt{void}*_q, (\tau_1, ..., \tau_m)*_q\}}{V, \Gamma \vdash l =_\tau e} \; \textit{assign}$$

The $\tau \in \{\texttt{int}, \tau' *_q, \texttt{void}*_q, (\tau_1, ..., \tau_m)*_q\}$ restriction limits τ to a subset of the types that can be directly assigned in C. In C, arrays may not be assigned directly and structures and unions may be assigned directly. We forbid assignments at the structure or union level, requiring the programmer to copy the fields pointwise. This allows us to check the validity of each individual assignment. This restriction is based on the C language.

[1] For example, the buggy C program fragment { `int a,b,c; * ((int *)(&b) + 1) = 5;` } will (if GCC or MSVC is used) have the officially-undefined and compiler-dependent effect of assigning 5 to either `a` or `c` depending on the stack layout. If this example is run through CCured with the standard pointer-kind inference, the `(int *)` cast will be given the kind `FSEQ`. The cast from the `SAFE` pointer `&b` to $\texttt{int}*_{\texttt{FSEQ}}$ will result in an `FSEQ` pointer with length $|\texttt{int}|$. The write through the pointer will trigger a CCured run-time exception because the `FSEQ` pointer is out of bounds after the arithmetic. The `go` SPEC95 benchmark exhibits errors of this form; they are caught by CCured but missed by Purify. See Section 6 for more information.

The typing derivation for an expression verifies that the expression is well-typed and gives its type. For example:

$$\frac{V,\Gamma \vdash e : \tau_2 \quad V \vdash \alpha(\tau_2) \leq \alpha(\tau_1)}{V,\Gamma \vdash (\tau_1 \leftarrow \tau_2)e : \tau_1}\ \textit{cast}$$

The $V \vdash \alpha(\tau_2) \leq \alpha(\tau_1)$ judgment checks a subtyping relationship between τ_1 and τ_2 and is detailed in Section 2.3. Intuitively, $V \vdash \alpha(\tau_2) \leq \alpha(\tau_1)$ holds if it is safe to cast an object of type τ_2 into type τ_1. The `void*` equivalence classes in V may be required to check that $\alpha(\tau_2) \leq \alpha(\tau_1)$ if either τ_1 or τ_2 contains or mentions `void*`.

The typing derivation for lvalue hosts yields the type of the object stored in that host and the pointer kind associated with the address of that object. The pointer kind associated with the address of the lvalue is necessary to type-check expressions of the form $\&l$. The *memhost* lvalue host judgment is shown below:

$$\frac{\Gamma(v) = \tau{*}_{q_1}\ \ q_2}{\Gamma \vdash *v : \tau, q_1}\ \textit{memhost}$$

Note that in the *memhost* example the qualifier q_2 associated with $\&v$ is not necessary when we are considering the host $*v$. In C the expressions $*\&v$ and v are the same, so when the expression $*v$ is considered, its address is v, so the qualifier associated with its address is the qualifier q_1 associated with v. Also notice that the form of the *memhost* rule eliminates the possibility of referencing through a (statically typed) `void*` or function pointer, since those two types do not match the template $\tau{*}_{q_1}$. In C it is not possible to dereference a `void*` or a function pointer.

The lvalue offset typing derivation takes the type of the current host and the kind associated with the address of the current host as assumptions and yields a new type-kind pair. Lvalue offsets are used to select subregions within one large region of storage:

$$\frac{\tau_1 = \texttt{struct}\{...,f\ \ \tau_2\ \ q_2,...\} \quad V,\Gamma,\tau_2,q_2 \vdash o : \tau_3,q_3}{V,\Gamma,\tau_1,q_1 \vdash (._s f\ .\ o) : \tau_3,q_3}\ \textit{sfield}$$

In the case of a structure field selection the type τ_2 and kind q_2 associated with the structure field f are taken as the new host and any remaining offsets o are considered in turn.

Notice that assignments and parameter passing may only occur between equal types. All type manipulations must go through explicit casts which are governed by a special subtyping relation (defined in Section 2.3). Even before formally specifying an operational semantics, it is important to note that well-typed programs can go wrong, just as they can in C. The program $(x : \texttt{int}{*}_{\texttt{DYN}}\ \texttt{DYN}\ ,\ x =_{\texttt{int}{*}_{\texttt{DYN}}} (\texttt{int}{*}_{\texttt{DYN}} \leftarrow \texttt{int})0\ ;\ *x =_{\texttt{int}} 5)$ type-checks but fails at run-time under a C-like model because it dereferences a null pointer.

2.3 Physical Subtyping

The previous section gave the standard static semantics for our C-like language in terms of a special physical subtyping relationship $\leq$. The syntactic structure of types in C is insufficient to capture all relationships between them from the perspective of memory safety. For example, the following two C types can be viewed as physically equivalent: `struct A {short a; short b; int *c;}` and `struct B { char d[4]; int *e; }`. A pointer to an object of type `struct A` can be cast to a pointer to an object of `struct B` safely. The two `shorts` and the array of four `chars` both cause the first four bytes after the beginning of the struct to be viewed as scalars, and in both types the next four bytes are pointers to integers. In the presence of C's unions and `typedefs` it is clear that the declared textual structure of the type cannot be used directly. Instead we interpret the type as a C compiler would when laying out space on the stack or in memory for a variable of that type. When we are comparing two types from the perspective of memory safety, we must make sure that any pointers that occur within them "line up" and occur at the same offsets with the same base types. This sort of reasoning is necessary to prove that casts in C programs are in fact safe: C programmers are notorious for their use of this sort of reasoning to guide the casts in their programs [CR99, SCB+99].

As a result, before determining our subtyping relationship we first convert C types into memory layouts, which can be viewed as flattened forms of C types. Arrays are flattened lazily so that the flattening process takes time proportional to the syntactic size of a type rather than its dynamic extent at run-time. Figure 5 shows our language of memory layouts. A partial function $\alpha : \tau \to \mathcal{L}$ maps C types to layout lists:

Layouts:	σ	$::=$	$\mathsf{Int}(n)$	a contiguous series of scalars of total size n
		$\mid$	$\sigma[n]$	an array of n contiguous copies of σ
		$\mid$	$\mathsf{FunPtr}(\tau_1, ..., \tau_n, q)$	a function pointer with n arguments and kind q
		$\mid$	$\mathsf{Ptr}(\tau)$	a pointer to a CCured type
		$\mid$	v_i	a variable representing a `void*` type
Layout List:	$\mathcal{L}$	$::=$	$[]$	an empty layout list
		$\mid$	$\sigma :: \mathcal{L}$	a layout list with head σ and tail $\mathcal{L}$

Figure 5: A description of **memory layouts** σ.

$$
\begin{array}{lcl}
\alpha(\texttt{int}) & = & \mathsf{Int}(1) :: [] \\
\alpha(\tau *_{\mathtt{q}}) & = & \mathsf{Ptr}(\tau * \mathtt{q}) :: [] \\
\alpha(\texttt{void}*_{\mathtt{q}}) & = & v_i :: [] \\
\alpha(\tau[n]) & = & (\alpha(\tau))[n] :: [] \\
\alpha((\tau_1, ..., \tau_n)*_{\mathtt{q}}) & = & \mathsf{FunPtr}(\tau_1, ..., \tau_n, q) :: [] \\
\alpha(\texttt{struct}\ \{f_1 : \tau_1\ \ q, ..., f_n : \tau_n\ \ q\}) & = & \alpha(\tau_1) :: ... :: \alpha(\tau_n) :: [] \\
\alpha(\texttt{union}\ \{f_1 : \tau_1\ \ q, ..., f_n : \tau_n\ \ q\}) & = & \text{see text}
\end{array}
$$

We used a refined version of the function α where $\mathsf{Int}(i) :: \mathsf{Int}(j) :: \mathcal{L}$ is always reduced to $\mathsf{Int}(i+j) :: \mathcal{L}$. Our implementation also expands structure tag names and `typedefs` (which are present in C but not in this presentation) and handles padding and alignment (which can be viewed as introducing extra $\mathsf{Int}(i)$'s not explicitly mentioned in the type). Unions are the only special case in the mapping. In C, a union is given enough space in memory to store its largest member and a union access allows the value stored there to be viewed using one of many types.[2] We compute $\alpha(\texttt{union}\{\tau_1 ...\})$ as follows. Let τ_M be the type member of the union with maximal size (this must be computed inductively because of nested unions). For every other τ_i in the union, let $\mathcal{L}_M$ be the prefix of $\alpha(\tau_M)$ that has the same size as $\alpha(\tau_i)$. If $V \vdash \mathcal{L}_M = \alpha(\tau_i)$ for all i, then the union is safe and $\alpha(\tau_M)$ should be returned. This means that τ_M is physically equal to the prefices of all the other member types, so no matter how their elements are used to obtain a "free cast," the program remains safe. The suffix of τ_M that starts just after the size of all other τ_i can only be reached through τ_M, so it need not be compared against anything else for consistency. If these tests fails, the union is inherently unsafe and no associated layout can be provided: the address of the union will be a `DYN` pointer. One could use tagged unions that record the last type written to the union, but this does not avoid altering the layout of the union. These layouts are a simplifying intermediate representation for physical subtyping [CR99, SCB+99] queries on actual C types.

We will define a physical equality judgment $V \vdash A = B$ and a physical subtyping judgment on $V \vdash A \leq B$ on layouts and layout lists. Intuitively, $V \vdash A = B$ means that A and B would have the same compiler-generated memory layout with respect to scalars and that their pointers are properly aligned. The judgment $V \vdash A \leq B$ means that an object with layout A may be safely cast to an object with layout B. We will use $\leq$ to reason about the top-level portion of objects being cast in the program and $=$ to reason about memory areas that must be equal and invariant (typically because they are one level of indirection below a pointer that is being cast at the top level).

It is important to note that $V \vdash A \leq B$ and $V \vdash B \leq A$ do not imply that $V \vdash A = B$. The reason for this is that memory layouts must be invariant when dereferencing matching pointers. A `DYN` pointer can be cast to an integer ($V \vdash \mathsf{Ptr}(\tau *_{\mathtt{DYN}}) \leq \mathsf{Int}(1)$) and an integer may be cast to a `DYN` pointer ($\vdash \mathsf{Int}(1) \leq \mathsf{Ptr}(\tau *_{\mathtt{DYN}})$). However, a `SAFE` pointer to a structure containing an integer may not be cast to a `SAFE` pointer to a structure containing a `DYN` pointer. If it could, the first `SAFE` pointer might be dereferenced and its integer field might be modified, changing the value of the `DYN` pointer (e.g., changing it from a valid pointer to an ordinary integer) without updating CCured's run-time type information. If the top-level `SAFE` pointers in that example were replaced with `DYN` pointers, however, the cast would be fine. The subtyping relationship between two `DYN` pointers is given by the *ptr*-`DYN` rule without inspecting what they point to.

[2] The C standard actually provides a weaker guarantee, but C programmers typically expect the behavior described above. The C standard only guarantees that the first atomic type in the first member of each union will share space with the first atomic type in every other member. In practice, programmers expect the type elements in each union member to line up contiguously and for all members to share space.

Physical Equality:

$$\frac{}{V \vdash A = A}\ \textit{reflex} \qquad \frac{V \vdash B = A}{V \vdash A = B}\ \textit{sym} \qquad \frac{V \vdash A = B \quad V \vdash B = C}{V \vdash A = C}\ \textit{trans}$$

$$\frac{V \vdash \sigma_1 = \sigma_2 \quad V \vdash \mathcal{L}_1 = \mathcal{L}_2}{V \vdash \sigma_1 :: \mathcal{L}_1 = \sigma_2 :: \mathcal{L}_2}\ \textit{eqlist} \qquad \frac{V(v_i) = \sigma}{V \vdash v_i = \sigma}\ \textit{eqvoid}$$

$$\frac{V \vdash \mathcal{L}_1 = \mathcal{L}_2}{V \vdash \mathcal{L}_1 = \mathsf{Int}(0) :: \mathcal{L}_2}\ \textit{scalar-0} \qquad \frac{V \vdash \mathcal{L}_1 = \mathcal{L}_2}{V \vdash \mathsf{Int}(i) :: \mathsf{Int}(j) :: \mathcal{L}_1 = \mathsf{Int}(i+j) :: \mathcal{L}_2}\ \textit{scalar-i}$$

$$\frac{V \vdash \mathcal{L}_1 = \mathcal{L}_2}{V \vdash \sigma_1[1] :: \mathcal{L}_1 = \sigma_1 :: \mathcal{L}_2}\ \textit{array-1} \qquad \frac{V \vdash \mathcal{L}_1 = \mathcal{L}_2 \quad n > 1}{V \vdash \sigma_1[n] :: \mathcal{L}_1 = \sigma_1 :: \sigma_1[n-1] :: \mathcal{L}_2}\ \textit{array-n}$$

$$\frac{V \vdash \alpha(\tau_1) = \alpha(\tau_2)}{V \vdash \mathsf{Ptr}(\tau_1 *_{\mathtt{SAFE}}) = \mathsf{Ptr}(\tau_2 *_{\mathtt{SAFE}})}\ \textit{ptr}\text{-}\mathtt{SAFE}\text{-}\mathtt{SAFE} \qquad \frac{\sigma_1, \sigma_2 \in \{\mathsf{Ptr}(\tau_1 *_{\mathtt{DYN}}), \mathsf{FunPtr}(\tau_1, ..., \mathtt{DYN})\}}{V \vdash \sigma_1 = \sigma_2}\ \textit{ptr}\text{-}\mathtt{DYN}$$

$$\frac{V \vdash \alpha(\tau_i') = \alpha(\tau_i)}{V \vdash \mathsf{FunPtr}(\tau_1, ..., \tau_n, \mathtt{SAFE}) = \mathsf{FunPtr}(\tau_1', ..., \tau_n', \mathtt{SAFE})}\ \textit{funptr}\text{-}\mathtt{SAFE}$$

Physical Subtyping:

$$\frac{}{V \vdash \mathcal{L} \leq []}\ \textit{width} \qquad \frac{V \vdash A = B}{V \vdash A \leq B}\ \textit{eq-leq} \qquad \frac{V \vdash \sigma_1 \leq \sigma_2 \quad V \vdash \mathcal{L}_1 \leq \mathcal{L}_2}{V \vdash \sigma_1 :: \mathcal{L}_1 \leq \sigma_2 :: \mathcal{L}_2}\ \textit{subt}$$

$$\frac{\sigma \in \{\mathsf{Ptr}(\tau *_q), \mathsf{FunPtr}(\tau, ..., q)\}}{V \vdash \sigma \leq \mathsf{Int}(1)}\ \textit{ptr-int} \qquad \frac{\sigma \in \{\mathsf{Ptr}(\tau_1 *_{q_1}), \mathsf{FunPtr}(\tau_1, ..., q_1)\} \quad q_1 \neq \mathtt{SAFE}}{V \vdash \mathsf{Int}(1) \leq \sigma}\ \textit{int-ptr}$$

$$\frac{V \vdash \alpha(\tau_1) = \alpha(\tau_2)[n] \quad q \in \{\mathtt{SEQ}, \mathtt{FSEQ}\}}{V \vdash \mathsf{Ptr}(\tau_1 *_{\mathtt{SAFE}}) \leq \mathsf{Ptr}(\tau_2 *_{\mathtt{q}})}\ \textit{ptr}\text{-}\mathtt{SAFE}\text{-}\mathtt{SEQ} \qquad \frac{V \vdash \alpha(\tau_1) = \alpha(\tau_2)[n] \quad q \neq \mathtt{DYN}}{V \vdash \mathsf{Ptr}(\tau_1 *_{\mathtt{q}}) \leq \mathsf{Ptr}(\tau_2 *_{\mathtt{SAFE}})}\ \textit{ptr-to}\text{-}\mathtt{SAFE}$$

$$\frac{V \vdash \alpha(\tau_1)[n] = \alpha(\tau_2)[m] \quad q_1, q_2 \in \{\mathtt{SEQ}, \mathtt{FSEQ}\}}{V \vdash \mathsf{Ptr}(\tau_1 *_{\mathtt{q_1}}) \leq \mathsf{Ptr}(\tau_2 *_{\mathtt{q_2}})}\ \textit{ptr}\text{-}\mathtt{SEQ}\text{-}\mathtt{SEQ}$$

Figure 6: **Physical subtyping** judgments.

The physical equality and physical subtyping derivation rules are shown in Figure 6. The structure a physical equality derivation is demonstrated by the *eqlist* rule:

$$\frac{V \vdash \sigma_1 = \sigma_2 \quad V \vdash \mathcal{L}_1 = \mathcal{L}_2}{V \vdash \sigma_1 :: \mathcal{L}_1 = \sigma_2 :: \mathcal{L}_2} \; \textit{eqlist}$$

Two layout lists represent physically equal types if their heads are physically equal and their tails are physically equal. The *eqvoid* derivation rule makes use of the mapping V to determine a more concrete base type for each occurrence of `void*`.

The *array-1* and *array-n* rules allow arrays to be unrolled on demand and equate an array of length one with its element. `SAFE` pointers and `SAFE` function pointers are only physically equal when their underlying types are physically equal as well.

The *ptr*-`DYN` rule relies on run-time type-checking: the base types of `DYN` pointers and `DYN` function pointers are not even considered. Normal pointers and function pointers may even be cast interchangeably: `DYN` pointers have all the capabilities of normal C pointers.

Physical subtyping is primarily based on width: the source type may be larger than the destination type provided that their common prefices are physically equal. This is the same as traditional object-oriented width subtyping. Physical subtyping is only relevant at the top level of a cast. After the first level of indirection through a non-`DYN` pointer, all types must be physically equal. The *ptr-int* rule allows any pointer value to be treated as an integer. The *int-ptr* rule allows an integer to be disguised as a non-`SAFE` pointer, although CCured's run-time checks will prevent it from being dereferenced later. The *ptr*-`SAFE`-`SEQ` allows a `SAFE` pointer to a large object (typically an array) to be cast to a `SEQ` or `FSEQ` pointer to smaller objects (typically the array components) provided that they tile perfectly. The *ptr-to*-`SAFE` rule allows a `SAFE`, `FSEQ` or `SEQ` pointer to be treated as a `SAFE` pointer to something that is a subtype of its base type. For example, a `SEQ` pointer to an integer might be cast to a `SAFE` pointer to an integer. At run-time such a cast requires a bounds check. Finally, the *ptr*-`SEQ`-`SEQ` rule allows to sequence pointers to be cast provided that there exists a tiling such that they line up perfectly. For example, a `SEQ` pointer to a structure containing two integers might be cast to an `FSEQ` pointer to a structure containing three integers because it is possible to tile both into layouts of length six that are physically equal. This rule provides great flexibility but it is hard to use automatically in practice.

Note that such the subtyping judgment gives more freedom than C itself allows. For example, we can conclude that `struct {int a; int b;}` is a subtype of `int`, but in C it is not possible to cast a structure to an integer. In C all such casts take place with an addition level of indirection: a pointer to the structure might be cast to a pointer to the integer. In CCured such a cast could not be made between two `SAFE` pointers because `struct {int a; int b;}` is not physically equal to `int`.

Our intuitive notion of physical subtyping for pointers is different in four main ways from that of previous work. First, we treat `void*`s as type variables that stand for concrete types. Second, we require the strict equality of the common prefix of two aggregates. This is in contrast to [CR99, SCB+99] where `void*` is allowed in the smaller aggregate in positions where a regular pointer is present in the larger one. Third, previous work did not consider physical subtyping in the presence of pointer arithmetic.

The fourth and most important difference between our subtyping relation and previous formulations is that we do not require complete static correctness. This is an essential part of CCured's combination of static and dynamic checks. Our static subtyping judgment allows casts from one type to another provided that type-safety is guaranteed by run-time checks. This can be seen most clearly in the *ptr*-`DYN` rule in Figure 6, where no restriction is placed on the base types of the dynamic pointers. Our subtyping relationship also extends previous work by handling all C types including unions and function pointers.

2.4 Operational Semantics

Now that we have described the static semantics for CCured programs we will explain the operational semantics and the run-time checks. Whenever the type-safety of an expression cannot be guaranteed statically, a run-time check must be inserted. This can be as simple as a null check for a `SAFE` pointer or as complicated as a meta-data check, bounds check and run-time type check when a value is read through a `DYN` pointer.

Our operational semantics derivations make use of a memory μ that models the store. The memory μ is a partial function mapping memory addresses to values. Variable declarations extend the domain of memory. Part of

Values and States:

Expression Values:	V	$::=$	n	scalar
		$\mid$	$\mathsf{Safe}(p)$	a possibly-null pointer
		$\mid$	$\mathsf{Seq}(p,l,u)$	pointer, lower bound, upper bound
		$\mid$	$\mathsf{FSeq}(p,u)$	pointer, upper bound
		$\mid$	$\mathsf{Dyn}(p,m)$	pointer, meta-data pointer
Stored Values:	m	$::=$	$Data(V)$	data under user control
		$\mid$	$Meta(l,u)$	lower bound, upper bound of `DYN` region
		$\mid$	$FunMeta(p,n_a)$	dynamic function pointer and argument count
		$\mid$	$Code(v_1,...,v_n,s)$	formal parameters and function body
Memory:	μ	$:$	$N \rightarrow m$	partial function mapping addresses to values
Environment:	S	$:$	$v \rightarrow V$	partial function mapping variables names to addresses

Programs:

$$\frac{\emptyset,\emptyset \vdash D \Downarrow \mu_1, S \quad \mu_1, S \vdash s \Downarrow \mu_2}{\vdash (D,s) \Downarrow}\ \textit{prog}$$

Declarations:

$$\frac{}{\mu, S \vdash \textit{nil} \Downarrow \mu, S}\ \textit{nil-decl}$$

$$\frac{a, \mu_2 = \mathsf{NewVar}(\tau, \mu_1) \quad S_2 = S_1[{}^{v}\!/_{\mathsf{Safe}(a)}] \quad \mu_2, S_2 \vdash D \Downarrow \mu_3, S_3}{\mu_1, S_1 \vdash v : \tau\ \ \texttt{SAFE}\ ; D \Downarrow \mu_3, S_3}\ \textit{decl-}\texttt{SAFE}$$

Statements:

$$\frac{\mu_1, S \vdash s_1 \Downarrow \mu_2 \quad \mu_2, S \vdash s_2 \Downarrow \mu_3}{\mu_1, S \vdash s_1\ ;\ s_2 \Downarrow \mu_3}\ \textit{seq} \qquad \frac{\mu_1, S \vdash \&l \Downarrow V_l \quad \mu_1, S \vdash e \Downarrow V_e \quad \mu_1 \vdash Write(V_l, \tau, V_e) \Downarrow \mu_2}{\mu_1, S \vdash l =_\tau e \Downarrow \mu_2, S}\ \textit{assign}$$

Figure 7: Operational semantics of **programs, declarations and statements** for the `SAFE` subset of the imperative language. NewVar produces a fresh stack address and a new memory in which that address is is zero-filled with an element of the given type.

Expressions:

$$\frac{}{\mu, S \vdash n \Downarrow n}\ int \qquad \frac{\mu, S \vdash e_1 \Downarrow V_1 \quad \mu, S \vdash e_2 \Downarrow n \quad Add(V_1, n) = V_2}{\mu, S \vdash e_1 + e_2 \Downarrow V_2}\ op \qquad \frac{\mu, S \vdash e \Downarrow V_1 \quad Conv(V_1, \tau_2 \leftarrow \tau_1) = V_2}{\mu, S \vdash (\tau_2 \leftarrow \tau_1)e \Downarrow V_2}\ cast$$

$$\frac{\mu, S \vdash h \Downarrow V_1 \quad \mu, S, V_1 \vdash o \Downarrow V_2 \quad \mu \vdash Read(V_2, \tau) \Downarrow V_3}{\mu, S \vdash (h, o)_\tau \Downarrow V_3}\ lvalue \qquad \frac{\mu, S \vdash h \Downarrow V_1 \quad \mu, S, V_1 \vdash o \Downarrow V_2}{\mu, S \vdash \&(h, o) \Downarrow V_2}\ \&lvalue$$

Lvalue hosts:

$$\frac{S(v) = V}{\mu, S \vdash v \Downarrow V}\ varhost \qquad \frac{S(v) = a_1 \quad \mu(a_1) = Data(V)}{\mu, S \vdash *v \Downarrow V}\ memhost$$

Lvalue offsets:

$$\frac{n_f = \mathsf{OffsetOf}(f) \quad V_2 = Add(V_1, n_f) \quad \mu, S, V_2 \vdash o \Downarrow V_3}{\mu, S, V_1 \vdash (._s f\ .\ o) \Downarrow V_3}\ sfield \qquad \frac{}{\mu, S, V \vdash nil \Downarrow V}\ nil\text{-}off$$

$$\frac{\mu, S \vdash e \Downarrow n_e \quad V_2 = Add(V_1, n_e) \quad \mu, S, V_2 \vdash o \Downarrow V_3}{\mu, S, V_1 \vdash ([e]\ .\ o) \Downarrow V_3}\ index \qquad \frac{\mu, S, V_1 \vdash o \Downarrow V_2}{\mu, S, V_1 \vdash (._u f\ .\ o) \Downarrow V_2}\ ufield$$

Figure 8: Operational semantics of **expressions** in the `SAFE` and `SEQ` subset of the imperative language. OffsetOf gives the numerical pointer offset associated with a field of a structure.

our notion of safety requires that the program never try to read or write from an invalid address. In addition, an environment S mapping variable names to their addresses is maintained for all of the variables in scope. Expression values range over scalars and pointers. `SAFE`, `SEQ`, `FSEQ` and `DYN` pointers all have disjoint values with multiple parts. The presentation is meant to be evocative of the CCured implementation where multiple words are used to store extra pointer information (e.g., the value $\mathsf{FSeq}(p, u)$ would be stored using two words while the value $\mathsf{Safe}(p)$ would be stored using one word, just like a normal C pointer). To simplify this presentation, all values are assumed to be the same size. Program expression values may be stored in *Data* areas within memory. CCured-controlled `DYN`-pointer meta-data can also be found in memory. Finally, function bodies are stored in memory but safe programs will never read or write function code, merely invoke it. Figure 7 describes our model of program values and gives the operational semantics for programs, declarations and statements.

The operational semantics judgment for programs has the form $\vdash (D, s) \Downarrow$ which means that the program executes without any memory-safety violations or CCured run-time exceptions. The judgment for declarations has the form $\mu_1, S_1 \vdash D \Downarrow \mu_2, S_2$ where μ_1 and S_1 are the state of memory and the environment before the declaration is considered and μ_2 and S_2 are the resulting memory and environment. Variable declarations extend the domain of memory and the environment by allocating zero-filled space to store the declared variable. The judgment for statements has the form $\mu_1, S \vdash s \Downarrow \mu_2$, where μ_1 and S are the values of memory and the environment before executing the statement s and μ_2 is the resulting value of memory. In general μ_2 is not the same as μ_1 if the statement has side-effects. However, since we do not consider dynamic allocation in this simplified language the domains of μ_1 and μ_2 are always identical. The judgment for expressions has the form $\mu_1, S \vdash e \Downarrow V$, where μ_1 and S are the input memory and environment, e is the pure expression under consideration and V is its value. We evaluate lvalue hosts usings judgments of the form $\mu, S \vdash h \Downarrow V$, where μ and S are the input memory and environment, h is the host under consideration and V is a pointer to the start of that host. Lvalue offsets, which adjust pointers within hosts, are evaluated using judgments of the form $\mu, S, V_1 \vdash o \Downarrow V_2$, where μ and S are the input environment, V_1 is the current pointer within a host, o is the offset by which it is being adjusted, and V_2 is the resulting pointer.

In addition to judgments for evaluating statements and expressions, we introduce ancillary judgments for memory reads, writes and accesses. The judgment $\mu_1 \vdash Write(V_l, \tau, V_e) \Downarrow \mu_2$ means that it is safe to write the value V_e of type τ to the address V_l in the input memory μ_2 and that the resulting memory is μ_2. The judgment $\mu \vdash Read(V_l, \tau) \Downarrow V_e$ means that it is safe to read a value of type τ from the address V_l in the input memory μ and that the resulting value is V_e. The judgment $\mu \vdash Access(V_l, \tau)$ means that it is safe to access an object of type τ at address V_l in memory μ and that a *Data* object is currently stored there (e.g., the address V_l does not point to function code). The boxed

Pointer Arithmetic:

$$\frac{}{Add(n_1,n_2) = n_1 + n_2} \qquad \frac{}{Add(\mathsf{Seq}(n_1,n_2,n_3),n_4) = \mathsf{Seq}(n_1+n_4,n_2,n_3)} \qquad \frac{\boxed{\boxed{n_1+n_3 \geq n_1}}}{Add(\mathsf{FSeq}(n_1,n_2),n_3) = \mathsf{FSeq}(n_1+n_3,n_2)}$$

Casts to `int`:

$$\frac{}{Conv(\mathsf{Safe}(n), \texttt{int} \leftarrow \tau) = n} \quad \frac{}{Conv(\mathsf{Seq}(n_1,n_2,n_3), \texttt{int} \leftarrow \tau) = n_1} \quad \frac{}{Conv(\mathsf{FSeq}(n_1,n_2), \texttt{int} \leftarrow \tau) = n_1}$$

Casts to `SAFE`:

$$\frac{}{Conv(\mathsf{Safe}(n), \tau_2{*}_{\texttt{SAFE}} \leftarrow \tau_1) = \mathsf{Safe}(n)} \quad \frac{\boxed{\boxed{u \neq 0}} \quad \boxed{\boxed{l \leq p}} \quad \boxed{\boxed{p + |\tau_2| \leq u}}}{Conv(\mathsf{Seq}(p,l,u), \tau_2{*}_{\texttt{SAFE}} \leftarrow \tau_1) = \mathsf{Safe}(p)}$$

$$\frac{\boxed{\boxed{u \neq 0}} \quad \boxed{\boxed{p + |\tau_2| \leq u}}}{Conv(\mathsf{FSeq}(p,u), \tau_2{*}_{\texttt{SAFE}} \leftarrow \tau_1) = \mathsf{Safe}(p)}$$

Casts to `SEQ`:

$$\frac{}{Conv(n, \tau{*}_{\texttt{SEQ}} \leftarrow \texttt{int}) = \mathsf{Seq}(n,0,0)} \qquad \frac{}{Conv(\mathsf{FSeq}(p,u), \tau_2{*}_{\texttt{SEQ}} \leftarrow \tau_1) = \mathsf{Seq}(p,p,u)}$$

$$\frac{}{Conv(\mathsf{Seq}(p,l,u), \tau_2{*}_{\texttt{SEQ}} \leftarrow \tau_1) = \mathsf{Seq}(p,l,u)} \qquad \frac{\boxed{\boxed{p \neq 0}}}{Conv(\mathsf{Safe}(p), \tau_2{*}_{\texttt{SEQ}} \leftarrow \tau_1{*}_{\texttt{SAFE}}) = \mathsf{Seq}(p,p,p+|\tau_1|)}$$

Casts to `FSEQ`:

$$\frac{Conv(V, \tau_2{*}_{\texttt{SEQ}} \leftarrow \tau_1) = \mathsf{Seq}(p,l,u) \quad \boxed{\boxed{u = 0 \textit{ or } p \geq l}}}{Conv(V, \tau_2{*}_{\texttt{FSEQ}} \leftarrow \tau_1) = \mathsf{FSeq}(p,u)}$$

Memory Accesses:

$$\frac{\boxed{\boxed{p \neq 0}} \quad \boxed{\{i | p \leq i < p + |\tau|\} \subseteq Dom(\mu)} \quad \boxed{\mu(p) = Data(V)}}{\mu \vdash Access(\mathsf{Safe}(p), \tau)}$$

Memory Reads:

$$\frac{\mu \vdash Access(\mathsf{Safe}(p), \tau) \quad \boxed{\mu(p) = Data(V)}}{\mu \vdash Read(\mathsf{Safe}(p), \tau) \Downarrow V}$$

$$\frac{Conv(\mathsf{Seq}(p,l,u), \tau{*}_{\texttt{SAFE}} \leftarrow \tau{*}_{\texttt{SEQ}}) = \mathsf{Safe}(n) \quad \mu \vdash Read(\mathsf{Safe}(n), \tau) \Downarrow V}{\mu \vdash Read(\mathsf{Seq}(p,l,u), \tau) \Downarrow V}$$

$$\frac{Conv(\mathsf{FSeq}(p,u), \tau{*}_{\texttt{SAFE}} \leftarrow \tau{*}_{\texttt{FSEQ}}) = \mathsf{Safe}(n) \quad \mu \vdash Read(\mathsf{Safe}(n), \tau) \Downarrow V}{\mu \vdash Read(\mathsf{FSeq}(p,u), \tau) \Downarrow V}$$

Memory Writes:

$$\frac{\mu_1 \vdash Access(\mathsf{Safe}(p), \tau) \quad \mu_2 = \mu_1[{}^{p}\!/\!_{Data(V)}]}{\mu_1 \vdash Write(\mathsf{Safe}(p), \tau, V) \Downarrow \mu_2} \qquad \frac{Conv(\mathsf{FSeq}(p,u), \tau{*}_{\texttt{SAFE}} \leftarrow \tau{*}_{\texttt{FSEQ}}) = \mathsf{Safe}(n) \quad \mu_1 \vdash Write(\mathsf{Safe}(n), \tau, V_1) \Downarrow \mu_2}{\mu_1 \vdash Write(\mathsf{FSeq}(p,u), \tau, V_1) \Downarrow \mu_2}$$

$$\frac{Conv(\mathsf{Seq}(p,l,u), \tau{*}_{\texttt{SAFE}} \leftarrow \tau{*}_{\texttt{SEQ}}) = \mathsf{Safe}(n) \quad \mu_1 \vdash Write(\mathsf{Safe}(n), \tau, V_1) \Downarrow \mu_2}{\mu_1 \vdash Write(\mathsf{Seq}(p,l,u), \tau, V_1) \Downarrow \mu_2}$$

Figure 9: Operational semantics for **pointer arithmetic**, **casts**, **memory reads** and **memory writes**.

checks involving the domain of μ associated with *Access* derivations model page-table checks that are carried out by memory-protection hardware. They are not meant to model any software bounds checks.

Pointer arithmetic and casts are of extreme importance to the safety of our system and we introduce special judgments for them. The judgment $Add(V_1, n) = V_2$ means that the result of adding n to V_1 is V_2. Typically V_1 is a pointer and n represents an offset from pointer arithmetic or array indexing. In the case of `FSEQ` pointers we must ensure that the pointer always advances positively. The judgment $Conv(V_1, \tau_{new} \leftarrow \tau_{old}) = V_2$ means that when a value V_1 of type τ_{old} is cast to type τ_{new}, the resulting value is V_2. In C the values V_1 and V_2 are almost always identical (but casts from `int` to `float` may change the underlying bit pattern, for example). In CCured pointers carry special meta-data and bounds information that must be manipulated during casts. Such casts between pointer kinds are better viewed as conversions or coercions.

Finally, we introduce two special judgments to abstract away underlying machine and language complexity. The $a, \mu_{new} = \mathsf{NewVar}(\tau, \mu_{old})$ function selects a new address a such that a through $a + |\tau|$ are unused in μ_{old}. It returns that address a and an updated μ_{new} that maps a up to $a+|\tau|$ to a zero-filled object of type τ. In particular, all pointers inside τ are set to the null value appropriate for their pointer kind. This initialization to null is used to establish CCured pointer kind invariants, which are usually of the form "either the pointer is null or ..." Our implementation inserts initialization code to set all local and global variables when entering a new scope before executing any user code. In addition, we write $n = \mathsf{OffsetOf}(f)$ to say that n is the numerical address offset of the beginning of the field f from the beginning of its enclosing structure. Either assume that static type information specifies the enclosing structure or that the program has been alpha-converted to make all structure field names unique. In our implementation the true difficulty in computing $\mathsf{OffsetOf}$ is handling alignment and padding. $\mathsf{OffsetOf}$ and α, the function that converts types to layouts, must agree with each other and with the underlying C compiler on the layout of structure fields.[3]

We present the operational semantics for the subset of the language without function calls or `DYN` values first in Figure 7. The structure of the derivation rule for programs is shown below:

$$\frac{\emptyset, \emptyset \vdash D \Downarrow \mu_1, S \quad \mu_1, S \vdash s \Downarrow \mu_2}{\vdash (D, s) \Downarrow} \; prog$$

μ represents the state of memory and S represents the environment. The declarations are evaluated, populating the initial environment with zero-filled regions of storage for the declared global variables. The resulting memory and environment are used to evaluate the program itself.

The derivation rule for declarations produces a new memory and stack extended to hold the declared variable:

$$\frac{a, \mu_2 = \mathsf{NewVar}(\tau, \mu_1) \quad S_2 = S_1[{}^{v}/_{\mathsf{Safe}(a)}] \quad \mu_2, S_2 \vdash D \Downarrow \mu_3, S_3}{\mu_1, S_1 \vdash v : \tau \;\; \mathtt{SAFE}\; ; D \Downarrow \mu_3, S_3} \; decl\text{-}\mathtt{SAFE}$$

In the *decl*-`SAFE` rule, S is extended to record the fact that the address of the variable v is associated with a `SAFE` pointer to the new zero-filled address a.

Evaluating a statement can change the contents of memory but cannot change the address of a variable. The *assign* derivation rule is shown below:

$$\frac{\mu_1, S \vdash \&l \Downarrow V_l \quad \mu_1, S \vdash e \Downarrow V_e \quad \mu_1 \vdash Write(V_l, \tau, V_e) \Downarrow \mu_2}{\mu_1, S \vdash l =_\tau e \Downarrow \mu_2, S} \; assign$$

The lvalue is evaluated to obtain the address it references and the expression is evaluated to obtain its value. The judgment $\mu_1 \vdash Write(V_1, \tau, V_e) \Downarrow \mu_2$ performs all of the run-time checks required to write V_e, an object of type τ, to the address V_1. If everything succeeds the returned memory μ_2 is μ_1 updated with the new value.

Figure 8 shows the rules for expressions and lvalues. The judgments for expressions evaluate to a value but do not change the program state. The *cast* derivation rule is highlighted below:

$$\frac{\mu, S \vdash e \Downarrow V_1 \quad Conv(V_1, \tau_2 \leftarrow \tau_1) = V_2}{\mu, S \vdash (\tau_2 \leftarrow \tau_1) e \Downarrow V_2} \; cast$$

[3]Agreeing with the compiler is more difficult in practice than in theory; `gcc` and Microsoft Visual C do not even agree with each other on all alignment issues and the C standard leaves many details up to the compiler.

Note that in CCured a cast may actually require a value conversion, similar to the transformation that occurs in C when a `float` is cast to an `int`. Casting between pointer kinds may cause additional meta-data to be created or discarded, and may even cause a run-time exception if the meta-data for the incoming pointer value indicate that it cannot be safely cast to the outgoing pointer kind.

An lvalue hosts evaluates to the address of the beginning of that region of storage. The *varhost* rule is indicative and uses the environment S to map a variable to its address:

$$\frac{S(v) = V}{\mu, S \vdash v \Downarrow V}\ varhost$$

Lvalue offset judgments consider memory, the stack and the address associated with their current host and evaluate to a new address within that host. The *sfield* derivation rule is shown below as an example:

$$\frac{n_f = \mathsf{OffsetOf}(f) \quad V_2 = Add(V_1, n_f) \quad \mu, S, V_2 \vdash o \Downarrow V_3}{\mu, S, V_1 \vdash (._s f\ .\ o) \Downarrow V_3}\ sfield$$

The $\mathsf{OffsetOf}(field)$ operator returns the numerical offset of a field from the beginning of its structure. At run-time, accessing a structure field behaves just like pointer arithmetic.

Figure 9 shows the rules for pointer arithmetic, casts and memory operations. Terms in $\boxed{\boxed{\text{double boxes}}}$ are run-time checks added by CCured. Terms in $\boxed{\text{single boxes}}$ are run-time checks enforced by the memory management hardware or operating system and represent a conventional notion of memory safety. In a well-typed program, the hardware safety checks will never fail: a CCured run-time check will always fail first. Thus the single-boxed checks need not be present and the program can still be trusted to run safely.

Pointer arithmetic (Add) adds an integer to the pointer part of a value and is not defined for `SAFE` pointers. `FSEQ` pointer arithmetic must not decrease the pointer value. The simpler check $n_3 \geq 0$ would suffice in this model but fails in the presence of wrap-around 32-bit arithmetic. Note that we need not check $n_1 + n_3 \leq n_2$ at this time: that check occurs when the pointer is dereferenced. Casts to `int` extract the pointer part of a value. Since `SAFE` pointers are always in bounds or null (and thus can be accessed without a bounds check), casting an `FSEQ` or `SEQ` pointer to a `SAFE` pointer requires a bounds check. Integers may be disguised as `SEQ` or `FSEQ` pointers. In such cases the upper-bound component is set to 0. The cast from `SAFE` to `SEQ` requires that the original pointer be non-null. We could also use a cast rule that takes $\mathsf{Safe}(0)$ to $\mathsf{Seq}(0, 0, 0)$ as a special case. Casts to `FSEQ` are handled by casting to `SEQ` and then throwing away the lower bound. Since an `FSEQ` pointer must either be an integer disguised as a pointer or have its pointer below its upper bound, we explicitly check the upper bound field when such casts are made. Reading data through a `SAFE` pointer requires that the pointer not be null and that the address range (given by the pointer and the size of the type) be valid. In addition, the pointer must point to the data segment and not to CCured-controlled meta-data or a function body. These properties are checked using the *Access* judgment. `SEQ` and `FSEQ` pointer reads are handled by first casting to `SAFE` (which performs any required bounds-checks) and then reading through the resulting `SAFE` pointer. Memory writes of `SAFE` pointers first check that it is valid to access that pointer and then update memory. Memory writes of `SEQ` and `FSEQ` values cast to `SAFE` (performing and bounds checks) and then write through the resulting `SAFE` pointer.

The checks presented in Figure 9 are assumed to use pure arithmetic with no wraparound. When implementing these checks using arithmetic modulo 2^{32}, special care must be taken. Despite this, it is possible to implement these checks very efficiently. For example, when casting from a `SEQ` to a `SAFE` pointer we require that $u \neq 0$, $l \leq p$ and $p + |\tau_2| \leq u$. Our implementation first checks that $u \neq 0$. If it is not, the single check $p - l < u - l$ suffices when $-$ is machine arithmetic (i.e., module 2^32). The reasoning is by cases. If $p \geq l$ then $p - l$ is positive, so $p - l < u - l$ is true exactly when $p < u$. If we maintain the further implementation invariant that $u - pmod|\tau_2| = 0$ for all `SEQ` and `FSEQ` pointers, no other checks are needed. On the other hand, if $p < l$ then the check is the same as $2^{32} + p - l < u - l$, which is the same as $2^{32} + p < u$, which is never true, so the check will fail (which is what we want if $p < l$: the pointer is out of bounds). Casts from `FSEQ` to `SAFE` are handled similarly.

2.5 Function Semantics

So far we have described the operational semantics and run-time checks associated with the subset of the language that does not include `DYN` or function pointers. We will now introduce our handling of function calls and function pointers.

Function Declaration:

$$\frac{p, \mu_2 = \mathsf{NewAddr}(Code(v_1, ..., v_n, s), \mu_1) \quad S_2 = S_1[{}^{fun_i}/_{\mathsf{Safe}(p)}] \quad \mu_2, S_2 \vdash D \Downarrow \mu_3, S_3}{\mu_1, S_1 \vdash fun_i(v_1 : \tau_1 \;\; q_1, ..., v_n : \tau_n \;\; q_n) \;\; \mathtt{SAFE} = s \; ; D \Downarrow \mu_3, S_3} \; \textit{fundecl-}\mathtt{SAFE}$$

Function Calls:

$$\frac{\begin{array}{c} \mu_1, S_1 \vdash e_p \Downarrow \mathsf{Safe}(n_p) \\ \boxed{\boxed{n_p \neq 0}} \\ \boxed{n_p \in Dom(\mu_1)} \\ \boxed{\mu_1(n_p) = Code(v_1, ..., v_m, s)} \\ \boxed{n = m} \end{array} \qquad \begin{array}{c} \mu_1, S_1 \vdash e_i \Downarrow V_i \quad (1 \leq i \leq n) \\ a_i, \mu_{i+1} = \mathsf{NewAddr}(V_i, \mu_i) \quad (1 \leq i \leq n) \\ S_2 = S_1[{}^{v_1}/_{a_1}]...[{}^{v_n}/_{a_n}] \\ \mu_{n+1}, S_2 \vdash s \Downarrow \mu_{n+2} \\ \mu_{n+4} = \mathsf{Remove}(\mu_{n+3}, a_1, ..., a_n) \end{array}}{\mu_1, S_1 \vdash (*e_p)(e_1, ..., e_n) \Downarrow \mu_{n+4}} \; \textit{ptrcall-}\mathtt{SAFE}$$

Figure 10: Operational semantics for **functions**.

Function calls allocate new space for their parameters on the stack and remove this space when they return. We introduce the $p, \mu_2 = \mathsf{NewAddr}(V, \mu_1)$ function to handle this sort of allocation and the allocation done by CCured when it lays out meta-data. It returns a new address p that was unused in μ_1 and an updated memory μ_2 such that $\mu_2(p) = V$. Allocation and initialization are performed atomically from the perspective of the program, so if the new value being created is a pointer value there is no time at which it does not follow CCured's invariants. In addition, we introduce the $\mu_{new} = \mathsf{Remove}(\mu_{old}, a_1, ..., a_n)$ function which removes the addresses a_1 through a_n from the domain of the input memory μ_{old} and returns the result as μ_{new}.

Figure 10 describes the operational semantics for function declaration and function calls. In a function declaration, the function value itself is a pieces of code in memory and the address of the function yields a `SAFE` function pointer to that code.

A `SAFE` function call requires that the function pointer be non-null, that the pointer be in the domain of memory, that it truly point to a *Code* object and that the number of actual arguments be equal to the number of formal parameters. The arguments are then evaluated and the environment is extended to contain new locations initialized with their actual values. Once the function body has been evaluated, the stack frame is popped using the special Remove function.

2.6 The Dynamically-Checked Fragment

Thus far we have discussed the operational semantics and run-time checks for the subset of the imperative language without any `DYN` pointers. `DYN` pointers keep additional information so that their types can be checked at run-time. Once a `DYN` pointer is created its physical extent remains constant. However, the program may treat the values reachable by a `DYN` pointer as either pointers or scalars depending on the casts used. Since those casts could not be verified statically we must keep extra status bits associated with each word reachable through a `DYN` pointer. These status bits indicate whether the last value written there was a scalar or a pointer. In this simplified model, all stored `DYN` values carry such status bits: the pointer $\mathsf{Dyn}(n_p, n_m)$ is distinguishable from the scalar $\mathsf{Dyn}(n, 0)$ by its non-null meta-data pointer component. In our implementation additional storage is reserved adjacent to each `DYN` area. This storage is used to hold bitfields that store tags for each word in the area.

`DYN` function pointers expect all of their arguments to be `DYN` pointers as well (scalars may be disguised as `DYN` pointers if necessary) and may be called with additional actual arguments. Each function with a `DYN` function pointer is given a special piece of $FunMeta(p, n)$ meta-data that records the start of the function p and its argument count n.

Figure 11 shows the `DYN` fragment of the language. A `DYN` variable declaration (one where the address of the variable is a `DYN` pointer) creates storage for the variable itself and for the meta-data. Pointer arithmetic and casts from `DYN` to `int` work as in the `SEQ` case. Disguising an `int` as a `DYN` pointer yields a `DYN` pointer with a null meta-

DYN Variable Declarations:

$$\frac{a_v, \mu_2 = \mathsf{NewVar}(\tau, \mu_1) \quad a_m, \mu_3 = \mathsf{NewAddr}(Meta(a_v, a_v + |\tau|), \mu_2) \quad S_2 = S_1[{}^{v}\!/_{\mathsf{Dyn}(a_v, a_m)}] \quad \mu_3, S_2 \vdash D \Downarrow \mu_4, S_3}{\mu_1, S_1 \vdash v : \tau \ \ \mathtt{DYN}\ ;\ D \Downarrow \mu_4, S_3}\ \textit{decl-}\mathtt{DYN}$$

DYN Arithmetic and Casts to `int`:

$$\frac{}{Add(\mathsf{Dyn}(n_1, n_2), n_3) = \mathsf{Dyn}(n_1 + n_3, n_2)} \qquad \frac{}{Conv(\mathsf{Dyn}(n_1, n_2), \mathtt{int} \leftarrow \tau) = n_1}$$

Casts to DYN:

$$\frac{}{Conv(n, \tau{*}_{\mathtt{DYN}} \leftarrow \mathtt{int}) = \mathsf{Dyn}(n, 0)} \qquad \frac{}{Conv(\mathsf{Dyn}(n_1, n_2), \tau_2{*}_{\mathtt{DYN}} \leftarrow \tau_1{*}_{\mathtt{DYN}}) = \mathsf{Dyn}(n_1, n_2)}$$

DYN Memory Accesses:

$$\frac{\begin{array}{c}\boxed{\boxed{n_m \neq 0}}\\ \boxed{n_m \in Dom(\mu)}\\ \boxed{\boxed{\mu(n_m) = Meta(n_l, n_u)}}\end{array} \qquad \begin{array}{c}\boxed{\boxed{n_l \leq p}} \qquad \boxed{\boxed{p + |\tau| < n_u}}\\ \boxed{\{i | p \leq i < p + |\tau|\} \subset Dom(\mu)}\\ \boxed{\mu(p) = Data(\mathsf{Dyn}(n_p, n_m))}\end{array}}{\mu \vdash Access(\mathsf{Dyn}(p, n_m), \tau{*}_{\mathtt{DYN}})}$$

DYN Memory Reads:

$$\frac{\mu \vdash Access(\mathsf{Dyn}(p, n_m), \tau{*}_{\mathtt{DYN}}) \quad \boxed{\mu(p) = Data(\mathsf{Dyn}(n_p, n_m))}}{\mu \vdash Read(\mathsf{Dyn}(p, n_m), \tau{*}_{\mathtt{DYN}}) \Downarrow \mathsf{Dyn}(n_p, n_m)} \qquad \frac{\mu \vdash Read(\mathsf{Dyn}(n_1, n_2), \mathtt{int}{*}_{\mathtt{DYN}}) \Downarrow \mathsf{Dyn}(n_p, n_m)}{\mu \vdash Read(\mathsf{Dyn}(n_1, n_2), \mathtt{int}) \Downarrow n_p}$$

DYN Memory Writes:

$$\frac{\mu_1 \vdash Access(\mathsf{Dyn}(p, n_m), \tau{*}_{\mathtt{DYN}}) \quad \mu_2 = \mu_2[{}^{p}\!/_{Data(V)}]}{\mu_1 \vdash Write(\mathsf{Dyn}(p, n_m), \tau{*}_{\mathtt{DYN}}, V) \Downarrow \mu_2} \qquad \frac{\mu_1 \vdash Write(\mathsf{Dyn}(p, n_m), \mathtt{int}{*}_{\mathtt{DYN}}, \mathsf{Dyn}(n, 0)) \Downarrow \mu_2}{\mu_1 \vdash Write(\mathsf{Dyn}(p, n_m), \mathtt{int}, n) \Downarrow \mu_2}$$

DYN Function Declarations:

$$\frac{\begin{array}{c}n_c, \mu_2 = \mathsf{NewAddr}(Code(v_1, ..., v_n, s), \mu_1)\\ n_m, \mu_3 = \mathsf{NewAddr}(FunMeta(n_c, n), \mu_2)\\ a, \mu_4 = \mathsf{NewAddr}(\mathsf{Dyn}(n_c, n_m), \mu_3)\end{array} \quad S_2 = S_1[{}^{fun_i}\!/_{a}] \quad \mu_4, S_2 \vdash D \Downarrow \mu_5, S_3}{\mu_1, S \vdash fun_i(v_1 : \tau_1 \ \ q_1, ..., v_n : \tau_n \ \ q_2) \ \ \mathtt{DYN} = s\ ;\ D \Downarrow \mu_5, S_3}\ \textit{fundecl-}\mathtt{DYN}$$

DYN Function Calls:

$$\frac{\begin{array}{c}\mu_1, S_1 \vdash e_p \Downarrow \mathsf{Dyn}(n_p, n_m)\\ \boxed{\boxed{n_m \neq 0}}\\ \boxed{n_m \in Dom(\mu_1)}\\ \boxed{\boxed{\mu_1(n_m) = FunMeta(n_p, m)}}\\ \boxed{\boxed{n \geq m}}\end{array} \quad \begin{array}{c}\boxed{\mu_1(n_p) = Code(v_1, ..., v_m, s)}\\ \mu_1, S_1 \vdash e_i \Downarrow V_i \ \ (1 \leq i \leq n)\\ a_i, \mu_{i+1} = \mathsf{NewAddr}(V_i, \mu_i) \ \ (1 \leq i \leq m)\\ S_2 = S_1[{}^{V_1}\!/_{a_1}]...[{}^{V_n}\!/_{a_m}]\\ \mu_{n+1}, S_2 \vdash s \Downarrow \mu_{n+2}\\ \mu_{n+4} = \mathsf{Remove}(\mu_{n+3}, a_1, ..., a_m)\end{array}}{\mu_1, S_1 \vdash (*e_p)(e_1, ..., e_n) \Downarrow \mu_{n+4}}\ \textit{ptrcall-}\mathtt{DYN}$$

Figure 11: Operational semantics for DYN **values**.

data field. Casts between DYN pointers of different static types are no-ops (since the static type is not trusted), but there are no valid casts between DYN pointers and pointers with other kinds. The DYN and SAFE worlds are separate, although SAFE pointers may point to DYN pointers.

DYN memory reads first verify that the meta-data pointer is not null and then use it to read the meta-data. If the pointer is within the bounds specified by the meta-data, then the pointer should also be within the domain of memory. Finally, the value read must be a data value (and not some function code, for example). DYN memory reads that wish to read a scalar first read the pointer stored there and then throw away the meta-data information.

DYN memory writes first perform all of the DYN memory access checks and then update memory to store the new value. Scalars are stored as DYN pointers with null meta-data pointers. In our implementation, tag bits associated with the meta-data are used to keep track of this run-time type information (i.e., whether the last value written to a particular spot in a DYN region was a DYN pointer or an integer).

A DYN function declaration allocates space for the function itself, the function meta-data and the DYN pointer associated with the address of the function. DYN function calls require more checks than their SAFE counterparts, since the static type of the DYN function pointer cannot be trusted. First the meta-data pointer is checked (to prevent attempts to call through an integer). If it is not null, the meta-data pointer must point to function meta-data (and not meta-data for a user data object). There must be at least as many actual arguments as formal parameters. Additional actual arguments are discarded as they are in C. The remaining evaluation is carried out as per SAFE function calls.

3 Type Soundness

A well-typed program can still go wrong during memory reads, memory writes and calls through function pointers. A well-typed program can only fail double-boxed (i.e., CCured) checks; it will never failed single-boxed checks (i.e., memory hardware). Thus a well-typed program will either run to completion or trigger a CCured run-time check; it will never access invalid memory. As a result, the single-boxed checks need not actually be implemented or included in the program: the CCured checks suffice. This is particularly desirable in embedded environments where memory management hardware may not be present, or in the case of loadable modules that run in the main program's address space, or in the case of systems composed from components.

Theorem 1 (Safety) *Given a program (D, s) and a typing environment V, if $V \vdash (D, s)$, then either:*

1. $\emptyset, \emptyset \vdash (D, s) \Downarrow$, *the program evaluates without any memory-safety violations.*

2. *No derivation* $\emptyset, \emptyset \vdash (D, s) \Downarrow$ *can be constructed because a* $\boxed{\boxed{\text{CCured}}}$ *term cannot be proved.*

CCured static and dynamic checks ensure a number of program invariants. These invariants characterize program values and are used in the proof of the Safety Theorem (which will be presented in the next section). Essentially, the invariants state that SAFE, SEQ and FSEQ pointers can be trusted to be either null, out of bounds or pointers to data objects. DYN pointers are either invalid or contain valid pointers to meta-data or a dynamic function descriptor.

Invariant 2 (SAFE Data Values) *If a program type-checks and at any point we have the judgment $V \vdash e : \tau*_{\mathtt{SAFE}}$ with an associated operational semantics judgment $\mu, S \vdash e \Downarrow V$ or a judgment $V \vdash l : \tau$ with $V \vdash \&l : \tau*_{\mathtt{SAFE}}$ and $\mu, S, \vdash \&l \Downarrow V$, then V will be either:*

1. $\mathsf{Safe}(0)$. A null pointer.

2. $\mathsf{Safe}(n)$ *with* $n \in Dom(\mu)$ *and* $\tau \neq (\tau_1, ..., \tau_m)$ *and* $\mu(n) = Data(V)$. A valid data pointer.

3. $\mathsf{Safe}(n)$ *with* $n \in Dom(\mu)$ *and* $\tau = (\tau_1, ..., \tau_m)$ *and* $\mu(n) = Code(v_1, ..., v_m, s)$. A valid function pointer.

Invariant 3 (SEQ Values) *If a program type-checks and at any point we have the judgment $V \vdash e : \tau*_{\mathtt{SEQ}}$ with an associated operational semantics judgment $\mu, S \vdash e \Downarrow V$ or a judgment $V \vdash l : \tau$ with $V \vdash \&l : \tau*_{\mathtt{SEQ}}$ and $\mu, S, \vdash \&l \Downarrow V$, then V will be either:*

1. $\mathsf{Seq}(n_1, n_2, 0)$. A null pointer or an integer disguised as a pointer.

2. $\mathsf{Seq}(p,l,u)$. *If* $l \le p < u$, *then* $p \in Dom(\mu)$ *and* $\mu(p) = Data(V)$. A valid data pointer.

Invariant 4 (FSEQ Values) *If a program type-checks and at any point we have the judgment* $V \vdash e : \tau *_{\texttt{FSEQ}}$ *with an associated operational semantics judgment* $\mu, S \vdash e \Downarrow V$ *or a judgment* $V \vdash l : \tau$ *with* $V \vdash \& l : \tau *_{\texttt{FSEQ}}$ *and* $\mu, S, \vdash \& l \Downarrow V$, *then* V *will be either:*

1. $\mathsf{FSeq}(n, 0)$. A null pointer or an integer disguised as a pointer.
2. $\mathsf{FSeq}(p, u)$. *If* $p < u$, *then* $p \in Dom(\mu)$ *and* $\mu(p) = Data(V)$. A valid data pointer.

Note that the global scope of these invariants helps to imply the type-safety of the memory referenced by `SAFE`, `FSEQ` and `SEQ` pointers. For example, consider the program fragment:

$$(\texttt{int}*_{\texttt{SAFE}} p \ \texttt{SAFE} \ ; fun_0(arg : \texttt{int}*_{\texttt{SAFE}}*_{\texttt{SAFE}}\texttt{SAFE}) = (p, nil) =_{\texttt{int}*_{\texttt{SAFE}}} *arg \ ; (*p, nil) =_{\texttt{int}} 5)$$

The function fun_0 takes a `SAFE` pointer to a `SAFE` pointer to an integer and sets that integer to 5. To prove this program safe, we need to know that the assignment to $(*p, nil)$ will really be a a valid memory write. Since the assignment statement type-checks, we know $V \vdash (*p, nil) : \texttt{int}$ and $V \vdash \&(*p, nil) : \texttt{int}*_{\texttt{SAFE}}$ in this program. We know from the *assign* rule in Figure 7 that the V in $\mu, S \vdash \&(*p, nil) \Downarrow V$ is the address written to by the final assignment statement. The `SAFE` Value Invariant provides enough information to prove this. By that invariant V is either null, a `SAFE` function pointer or a `SAFE` data pointer. Since the base type τ is `int`, we the function pointer case does not apply. Thus the pointer is either null or it points to a valid region in memory. The $\mu \vdash Access(V, \texttt{int})$ judgment in the operational semantics for assignments (see Figure 8) will perform the null-check at run-time. If the null-check passes, the invariant guarantees that all of the single-boxed checks in the *Access* derivation will pass. So the invariant was strong enough to prove the statement safe. Intuitively, the safety of the write through $(*p, nil)$ requires that the last value stored in p adhere to the invariants. The first value stored in p was 0 (by our NewVar zero-filling function) and any subsequent changes to p had to have been through well-typed assignment statements like this one. In fact, this particular value of p comes from the previous assignment statement. But since the value being stored in p is the same as the value of the pointer expression $*arg$, the invariant is upheld: the value of $*arg$ adheres to the invariants and since it is copied unchanged to p, p continues to adhere to the invariants. In this manner, pointer values start out safe (and null) and can only be changed to other safe values as the program progresses. We will present more detail on this intuition below.

Invariant 5 (DYN Values) *If a program type-checks and at any point we have the judgment* $V \vdash e : \tau *_{\texttt{DYN}}$ *with an associated operational semantics judgment* $\mu, S \vdash e \Downarrow V$ *or a judgment* $V \vdash l : \tau$ *with* $V \vdash \& l : \tau *_{\texttt{DYN}}$ *and* $\mu, S, \vdash \& l \Downarrow V$, *then* V *will be either:*

1. $\mathsf{Dyn}(n, 0)$. A null pointer or an integer disguised as a pointer.
2. $\mathsf{Dyn}(n_1, n_2)$ *with* $n_2 \in Dom(\mu)$ *and* $\mu(n_2) = Meta(n_l, n_u)$. *If* $n_l \le n_1 < n_u$, *then* $n_1 \in Dom(\mu)$ *and* $\mu(n_1) = Data(V)$. A valid data pointer.
3. $\mathsf{Dyn}(n_1, n_2)$ *with* $n_2 \in Dom(\mu)$ *and* $\mu(n_2) = FunMeta(n_f, n_a)$. *If* $n_1 = n_f$, *then* $n_1 \in Dom(\mu)$ *and* $\mu(n_1) = Code(v_1, ..., v_{n_a}, s)$. A valid function pointer.

The `DYN` Value invariant is similar to the invariants for `SAFE`, `SEQ` and `FSEQ` pointers but takes CCured meta-data into account.

Theorem 6 (Progress) *Given a program* (D, s) *and a typing environment* V, *if* $V \vdash (D, s)$, *then at all points during the evaluation of the program the above invariants will hold and all boxed checks will always be satisfied.*

The Safety Theorem above is an easy corollary of this Progress Theorem. The proof of this Theorem is by induction on the structure of the program. Assume that the invariants hold and that the boxed checks are satisfied for all structurally simpler programs. Consider the next bit of syntax in the program. Many of the cases (for example, the sequencing construct) are elided because they do not directly create, modify or use pointer values. Essentially, every time a pointer value is created, we show that it adheres to the invariants, and that information allows us to show that the boxed checks never fail.

All four of the declaration rules create values satisfying the invariants (see Figure 7). The SAFE declaration rule *decl*-SAFE creates a valid SAFE pointer referencing the variable declared. The *fundecl*-SAFE rule creates a valid function pointer (i.e., $\mu(p) = Code(...)$). The *decl*-DYN rule (see Figure 11) sets up meta-data and a DYN pointer for the address of the variable declared. The *fundecl*-DYN initializes the function code itself, the pointer and the function meta-data associated with the declared function. Finally, since $\mathsf{NewVar}(\tau, \mu)$ zero-fills the returned memory address by type, any internal pointers also adhere to the invariants. For example, in the declaration $b : \texttt{struct}\{\texttt{int} *_{\texttt{q}} \texttt{fld};\}$ SAFE, the pointer value $b._s\texttt{fld}$ is created with 0 in all components. Thus, all declarations create valid pointer values.

The statement, expression and lvalue rules do not contain checks or create new pointer values directly. However, they do return new pointer values created by pointer arithmetic ($Add(V_1, n)$), casts ($V_1 : \tau_1 \to \tau_2 = V_2$) and memory reads ($Read(V_2, \tau)$). They also perform memory writes ($Write(V_1, \tau, V_e)$). We next handle each of these cases in turn.

Pointer arithmetic is safe for SEQ and DYN values because they are allowed to go arbitrarily out-of-bounds (they will be checked when used). FSEQ pointers can only be advanced, so pointer arithmetic on them must contain a run-time check. If the check succeeds, then the new FSEQ value upholds the FSEQ invariant.

Casting to SAFE requires that the result be either 0 or a valid pointer. By induction, the pointer before the cast upholds the invariants associated with its old pointer kind. Thus a SEQ or FSEQ pointer is either null, out of bounds or valid. Run-time checks cover the null and out-of-bounds cases, so all values cast to SAFE that do not fail the run-time checks are valid SAFE pointers.

Casting to SEQ requires that the result either have a 0 upper bound, be out of bounds or be a valid pointer. Integers disguised as SEQ pointers are marked by 0 upper bounds. FSEQ pointers copy the current pointer value as the lower bound. If the FSEQ pointer had a 0 upper bound, then the new SEQ pointer has a 0 upper bound. If the FSEQ pointer was out-of-bounds, then the new SEQ pointer is out-of-bounds. If the FSEQ pointer was in bounds, then the new SEQ pointer is in bounds. Finally, casting a SAFE pointer to a SEQ pointer requires a null check. If it passes, the SAFE pointer can be considered as a SEQ pointer with one element. If not, the SAFE pointer can be cast to a null SEQ pointer, but we omit this case from our semantics for brevity. Casts to FSEQ behave identically but drop the lower-bound component. Since FSEQ pointers are always assumed to be above their lower-bound, we must first check that an incoming SEQ pointer is above its lower-bound before converting it to an FSEQ pointer. Casts between DYN values have no associated run-time checks because the static type of a DYN pointer is disregarded. Note that since the program is well-typed there will never be any casts between DYN and non-DYN pointers or non-trivial casts involving SAFE function pointers.

Memory reads from SAFE pointers require only a null check. The SAFE pointer invariant tells us that a SAFE value is either 0, a valid pointer or a valid function pointer. The null check handles the first case, and a well-typed program cannot read through a SAFE function pointer, so the boxed checks (which require that the pointer actually point to something in memory) will always be satisfied by the SAFE invariant. Reads from FSEQ and SEQ values first cast the pointer to SAFE (which performs all necessary bounds checks and null-upper-bounds checks) and then read as above.

DYN memory reads are more complicated. By the invariant, a DYN pointer may be a disguised integer. The DYN memory read rule checks for that by doing a null-check on the meta-data pointer. If meta-data pointer is not null, then by the invariant it points to something in memory. That piece of meta-data could either be a $Meta(l, u)$ object or a $FunMeta(...)$ value. We check at the time of the read that the program is not trying to read through a function pointer. If the meta-data is a $Meta(l, u)$ object, then by the invariant the pointer is valid if it is between the bounds l and u. A run-time check verifies that the pointer is in-bounds. If it is, then the invariant guarantees that it is in the domain of memory and points to a $Data$ value.

We must also verify that the value returned by the memory read upholds the invariants. This is usually true by structural induction: it was a valid value when last written there. The one exception to this is when the address of a function parameter (or a local variable in C) is stored in memory and then accessed after that function has returned. The pointer now references an invalid stack location. See Section 5 for CCured's handling of this case.

Memory writes are modeled by verifying that a memory access using that same pointer would succeed (which requires all of the checks above). If it does, then the address already points to a valid location in memory and we can update that location with the new value. By induction, the value being written adheres to the invariants. As with reads, FSEQ and SEQ writes are handled by casting the pointers to SAFE (which performs bounds and null-upper-bound checks). DYN memory writes always write DYN pointers, so an attempt to write a scalar n by the program actually writes $\mathsf{Dyn}(n, 0)$.

Function calls require that the pointer be valid and that it point to $Code(...)$. That is, the program should not jump to the data segment. In the case of SAFE pointers, by the invariant a SAFE value is either 0, a valid data pointer or a valid function pointer. If the program is well-typed, then a function call will not involve a SAFE data pointer. The *funcall*-SAFE rule makes a null check and is then assured that the boxed checks will go through by the invariant. In particular, since casts and pointer arithmetic are disallowed on SAFE function pointers, a SAFE function pointer can only have been passed around since it was created with the *fundecl*-SAFE rule.

DYN function calls are similar. A run-time check verifies that the meta-data component is not null. If it is not, then by the invariant it points to something in memory. We must verify at run-time that it points to $FunMeta$ (and not some data object meta-data, which could happen if the program casts a data pointer to a function pointer). If it does, we check that the stored pointer value in the function meta-data and the actual pointer value agree (i.e., there has been no pointer arithmetic on this function pointer). If they do, then by the invariant the function pointer actually points to a $Code$ object. Finally, since it is possible to cast between different DYN function pointers, we must check that we are calling with at least as many actual arguments as there are formal parameters. Additional arguments are evaluated and then discarded.

In summary, the Progress Theorem holds by structural induction: the value invariants are always true, and only the double-boxed checks may fail. It follows easily that the Safety Theorem holds: a well-typed program can only go wrong if the double-boxed checks fail. Thus the boxed checks need not even be present.

4 Pointer Kind Inference

A well-typed program with the appropriate run-time checks cannot go wrong, but existing C programs do not come with CCured pointer kind annotations. Given a program without pointer-kind annotations, can we infer a value for each pointer kind such that the program successfully type-checks? One conservative solution is to make every kind DYN. This works because DYN pointers can do anything a C pointer can do. However, this discards the static typing information in the program and defers all checks to run-time. The DYN solution is sound but overly conservative: the run-time checks associated with DYN pointers slow the program dramatically.

Our pointer kind inference algorithm is a flow-insensitive analysis that tries to minimize the number of DYN pointers. Some pointers may have to be DYN because they are involved in casts that cannot type-check otherwise. Among the remaining pointers it tries to make as few pointers SEQ as possible without causing the program to fail run-time checks more often than necessary. Some non-DYN pointers may have to be SEQ because they are involved in negative pointer arithmetic or are assigned to variables that are. Finally, the inference algorithm tries to make as few pointers FSEQ as possible (i.e., all those pointers that are involved in positive pointer arithmetic). All of the remaining pointers are made SAFE.

4.1 Inference Algorithm

Conceptually, pointer kind inference works by constraint resolution. It can be implemented as a graph algorithm that takes time proportional to the number of casts and pointer types in the program. Every pointer kind in the program is treated as a variable ranging over $\{\texttt{SAFE}, \texttt{SEQ}, \texttt{FSEQ}, \texttt{DYN}\}$. The inference algorithm must also generate a mapping V that maps each `void*` as a type variable in the input program to a concrete type.

The first step is to determine which pointer kinds must be DYN. Since DYN pointers are required only to handle casts that cannot be verified statically, every cast in the program is examined with respect to the physical subtyping relation $\leq$. Consider a cast $(\tau_1 \leftarrow \tau_2)$. For the resulting program to type-check, we must assign kinds such that $V \vdash \alpha(\tau_2) \leq \alpha(\tau_1)$. We compute $\mathcal{L}_1 = \alpha(\tau_1)$ and $\mathcal{L}_2 = \alpha(\tau_2)$. If either of these fail (e.g., because one of the types contains an unsafe union) then all kinds within τ_1 and τ_2 are set to DYN (and thus the cast will type-check using the *ptr*-DYN rule). Similarly, if $|\tau_1| > |\tau_2|$ (which usually corresponds to a downcast in the program), then all pointer kinds involved are made DYN. Otherwise, we attempt to prove $V \vdash \mathcal{L}_2 \leq \mathcal{L}_1$ recursively using the *width*, *subt*, *scalar*, *array* and *eqvoid* rules from Figure 6. If this succeeds, then this part of the program will type-check without adding any more DYN pointers. If not, the process continues below.

The second step is to generate the partition V. If the head of one layout list is v_j and the other is $\mathsf{Ptr}(\tau *_q)$ and V has no mapping for v_j, we update V so that $V(v_j) = \tau *_q$. This builds up a minimal mapping V linking all `void*` pointers that must be congruent because of subtyping requirements. If the head of one list is $\mathsf{Ptr}(\tau *_q)$ and the other is not a pointer, then q and all kinds within τ must be made DYN. The cast will then type-check using the *int-ptr* rule (if the other was a scalar of some sort) or *ptr*-DYN rule (if the other was a function pointer). If the head of one list is $\mathsf{Ptr}(\tau_1 *_{q_1})$ and the head of the other is $\mathsf{Ptr}(\tau_2 *_{q_2})$ we emit the constraint that $q_1 = q_2$ and recursively consider τ_1 and τ_2. If they are not equal under are subtyping relation then we set $q_1 = q_2 =$ DYN (and all kinds in τ_1 and τ_2 are made DYN as well).

This inference for matched pointers is a coarse approximation to the flexibility allowed by the subtyping judgment. For example, the subtyping rules allow for a SAFE pointer to a structure with four int fields to be cast to a SEQ pointer to an int. The operational semantics described in Figure 9 would allow the resulting SEQ pointer a range over all of the elements in the original structure. The inference presented above would make both pointers DYN. In essence, we infer pointer kinds so that the n and m in the *ptr*-SAFE-SEQ , *ptr-to*-SAFE , and *ptr*-SEQ-SEQ rules will always be 1. More ambitious inference algorithms that take advantage of those rules could yield fewer DYN pointers. In our experience, however, the benefit is minute: less than 1% of all casts fall into this category.

Finally, if the head of one list is a function pointer and the other is not, the function pointer (and all its arguments) must be made DYN. If both are function pointers and the arguments are physically equal then both may remain SAFE. Otherwise both must be DYN.

This recursive examination of all of the types involved in casts yields a set of pointer kinds that must be DYN, a valid V for the program, and a set of constraints $q_i = q_j$ for pointer congruence. Since DYN values can never be assigned to variables with non-DYN types, we then examine all assignment statements, cast expressions, and parameter passing in function call statements in the program. If a DYN value is involved on either side of such a transfer, then the other side must also be DYN. If we have a constraint $q_i = q_j$ and either variable is DYN, the other is made DYN as well. Finally, for every $\tau *_q$ with $q =$ DYN, we set all q inside τ to be DYN as well. This process is repeated until no new DYN pointers are added.

Once this is done we have an approximation to the smallest set of DYN values required to make the program type-check. Since DYN pointers are only required to handle certain casts and to verify well-formedness constraints, nothing else in the program will require that a pointer kind be DYN.

All of the remaining pointer kinds could be made SEQ and the program would type-check. However, that assignment would yield more run-time overhead than necessary (especially if many of the pointers are never involved in pointer arithmetic). We examine every appearance of pointer arithmetic or array indexing in the program. If we cannot verify that the increment is positive, then that pointer kind is set to SEQ. In addition, we trace back through assignments, casts, parameter passing and $q_i = q_j$ constraints and set the kind of all pointers that could flow into p to be SEQ as well. For pointers associated with the addresses of variables, this is not possible. In such cases those pointers remain SAFE and the *ptr*-SAFE-SEQ rule will be used to prove that the cast is valid. This processes is repeated with all instances of clearly positive explicit pointer arithmetic or array indexing, setting all affected pointers and their antecedents to be FSEQ. All remaining (unconstrained) pointers become SAFE.

The backwards tracing with SEQ and FSEQ pointers is to ensure that if a pointer needs an upper or lower bound, then that bound will be carried with it from its creation. Casts from SEQ or FSEQ down to SAFE lose bounds information. Although a program with a SAFE pointer in the middle of a chain of SEQ pointers might type-check it might also fail a run-time check that it would not otherwise fail if all of the pointers in the chain carried bounds information.

4.2 Inference Safety

Theorem 7 (Inference Safety) *Assuming that all parts of a program (D, s) can be type-checked without pointer kinds, if the inference algorithm produces a new program (D', s') and a* void* *assignment V, then $V \vdash (D', s')$.*

The proof of the theorem is by structural induction on the program. The only cases that require attention are the *op*, *index* and *cast* rules, as well as the requirement that pointer kinds be well-formed.

The well-formedness condition on DYN kinds is established by the step of the inference algorithm that propagates DYN. The algorithm assigns all pointers involved in pointer arithmetic or indexing non-SAFE kinds, so the *op* and *index* rules are satisfied. Only the *cast* rule requires a non-trivial argument. The new program must be annotated such that all casts type-check according to the subtyping judgment. As detailed in Section 4.1, the recursive examination of all of the types in casts does exactly that. The resulting program will type-check using a restricted subset of the rules listed in Figure 6. In practice we have implemented a CCured type-checker that is run on the results of the inference.

5 Support for Special C Features

This section describes implementation details for handling other C features that we chose not to model formally.

5.1 Variable-Argument Functions

Our CCured implementation supports variable-argument functions that use C's `<stdarg.h>` macros. An example of a `printf`-like function using that scheme is:

```
void my_printf(char *format, ...) {
  va_list args;   // list of variable args
  va_start(args); // initialize variable args
  while (1)       // loop forever
    switch (next_token(format)) {
      case "%s": char * p = va_arg(args, char *); ...
      case "%d": int i = va_arg(args, int); ...
      case "%g": double d = va_arg(args, double); ...
      case NULL: va_end(args); return;
    }
}
```

Note that `my_printf` does not know in advance the order, number or types of its arguments. The `va_arg` macro extracts the next argument from the `va_list` assuming that it has the given type (the type is used to compute the number of bytes to read from the stack). This is unsafe in C because the programmer can pass any type (and thus any size) to the `va_arg` macro even if the caller did not pass that type or even a type of that size. If the function is expecting a pointer and the caller passed an integer, the function will receive an unsafe pointer. If the caller did not pass enough arguments and the function continues to ask for them, the function will read arbitrary data from the stack and interpret it using the given type. CCured adds run-time checks to prevent this sort of behavior.

For each `va_list` we ask the programmer to define a `union` containing all possible types that might be passed in for those variable arguments. For example, for `printf`-like functions we might declare:

```
  union printf_arguments {
    int    a;
    double b;
    char * c;
  };
```

At the call-site of a variable-argument function we mark in a global structure the number of actuals and the type of each actual (as an index into that union). It is an error to call such a function with an argument of a type that does not match some element of the union.

Inside the body of the variable-argument function the contents of the global array is immediately copied to a local temporary associated with the `va_list`. Within a variable-argument function, the `va_start(va_list)` and `va_arg( va_list, expected_type)` macros are used to access the extra arguments in sequence. CCured redefines these macros. At each appearance of `va_arg()` we check that there are still arguments remaining and that the expected type is the same as the stored type of the actual argument. This method allows multiple variable argument lists to be processed in parallel and also to be stored and passed to other functions (such as the family of `vprintf`-like functions). With these modifications to the `<stdarg.h>` macros, and after the programmer declares the types of expected arguments, CCured is able to process the unmodified body of all variable-argument functions that we have encountered in our experiments, including an actual implementation of `printf`.

5.2 C Strings

CCured includes special handling for a common C idiom: null-terminated strings. In C it is common to manipulate strings that are represented as arrays of characters bounded by a terminating 0. CCured would normally view such strings as `FSEQ` pointers and keep separate length information. However, this length information is redundant given the usual C string invariant. To handle this special case, our CCured implementation has a pointer kind, STRING, that behaves like FSEQ but maintains the C string invariant. STRING pointers are stored as a single word (like SAFE pointers) and when their length is needed (e.g., for pointer arithmetic or conversion to SEQ) a call to `strlen()` is made. Advancing a STRING pointer requires checking that we do not go past the string end: `s++` requires $\boxed{s \neq 0 \qquad \mu(s) \neq Data(0)}$. Note that in many cases the C compiler will be able to remove this assertion based on a similar check existing in the program itself. The user may read through a non-null STRING pointer and may write through a non-null STRING pointer provided that it does not point to a terminator. This has the advantages of speed (over FSEQ pointers) and safe compatibility with external libraries. The disadvantage is that the program may lose capabilities over time: writing a 0 into the middle of a STRING may prevent the user from later accessing the second half of that STRING. Code that does so may use SEQ pointers, however, and this has not proved to be a problem in practice. In the Apache modules we examined, on average 8% of all pointers (which is one half of the non-SAFE pointers) were STRING. The method described so far maintains invariants about null-terminated strings. To establish those invariants we can extend $\mathsf{NewVar}(\tau, \mu)$ to allocate $|\tau| + 1$ zero-filled bytes. This last byte will then never be over-written.

5.3 Address of a Local

In C it is possible to store the address of a local variable and then return from the invoked function, thus creating a pointer that references an invalid stack frame location. Our CCured implementation handles this by checking on every memory write of a pointer value that the value written is not the address of a local variable. This is done by comparing the address being written to the stack pointer register.

If a program uses pointers to local variables in a way that violates this restriction, those variables must be allocated on the heap instead of the stack. The CCured translator will automatically do this for any variable annotated with a special annotation, "`heapify`". In future work we plan to add such annotations automatically.

6 Experimental Results

We tested our system on numerous C programs ranging in size from a few hundred to 30,000 lines of code. This allowed us to measure the performance cost of run-time checks inserted for safety and the manual intervention required to make existing C programs work with our system. In general, computationally expensive kernels like the Spec95, Olden and Ptrdist benchmarks showed the greatest slowdown (ranging from 0–200% overhead). Apache and Linux kernel modules and a complete FTP demon showed no noticeable performance penalty: the cost of run-time checks is dwarfed by the cost of inter-process communication. Our experiments allowed us to detect a number of bugs in existing programs and run safety-critical code without fear of memory-based security errors (e.g., buffer overruns or stack-smashing attacks).

Figure 12 shows test cases taken from the Spec95 [SPE95], Olden [Car96] and Ptrdist-1.1 [ABS94] benchmark suites. The Spec benchmarks involve integer computations, the Olden benchmarks are compute-intensive kernels and the Ptrdist programs are pointer-intensive data structure manipulations. The benchmarks have been used in previous safe C projects with poorer running times (e.g., at least a factor of 10 in [LYHR01]). Minor source changes (such as adding or correcting prototypes or marking `printf`-like functions) were required for some programs. A few benchmarks required changing `sizeof` or moving local variables to the heap. On average we had to change 1 in 100 lines. The execution time used to compute the slowdown is the median of five trials taken on a quiescent 1GHz AMD Athlon Linux machine. The last column shows the slowdown when the programs (but not the system libraries) are instrumented with Purify (version 2001A) [HJ91], a tool that works on C binaries and detects memory leaks and access violations by keeping two status bits per byte of allocated storage. Purify does not catch pointer arithmetic between two separate valid regions [JK97], a property that Fischer and Patil [PF97] 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 it 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. To quantify our earlier comments about the speed of inference, an unmodified `gcc` compilation of `go` takes 8.83 seconds, our inference takes 0.05 seconds, and adding the run-time checks takes 1.43 seconds. Other programs give similar timings.

In the process, we discovered a number of bugs in these benchmarks: `ks` passes a `FILE*` to `printf` where a `char*`

Name	Lines of code	% sf/sq/d	**CCured ratio**	Purify ratio
SPECINT95				
compress	1590	87/12/0	**1.20**	28
go	29315	96/04/0	**1.14**	51
ijpeg	31371	79/20/1	**1.43**	30
li	7761	93/06/0	**1.79**	50
Olden				
bh	2053	80/18/0	**1.45**	94
bisort	707	90/10/0	**1.08**	42
em3d	557	85/15/0	**1.42**	7
health	725	93/07/0	**1.08**	25
mst	617	87/10/0	**1.45**	5
perimeter	395	96/04/0	**1.08**	544
power	763	95/06/0	**1.35**	53
treeadd	385	85/15/0	**1.40**	500
tsp	561	97/04/0	**1.08**	66
Ptrdist-1.1				
anagram	661	80/12/3	**1.31**	34
bc	7323	90/10/0	**1.17**	100
ft	2194	92/06/2	**1.02**	12
ks	793	92/08/0	**1.11**	31
yacr2	3999	87/13/0	**1.66**	26

Figure 12: 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 DYN, respectively.

is expected, `compress` and `ijpeg` contain array bounds violations and `go` has eight array bounds violations and one use of an uninitialized variable as an array index. Most of the `go` bugs are involved in multi-dimensional arrays. This demonstrates an important advantage of our type-based approach: if we viewed the entire multi-dimensional array as one large object, some of those bugs would not be detected.

One `ks` error is near line 52 of `KS-1.c` which contains code like the following:

```
void ReadNetList(char *fname) {
  FILE *inFile;
  inFile = fopen(fname,"r");
  if (inFile == NULL) {
    fprintf(stderr,"unable to open input file [%s]",inFile);
    /* BUG: should be ‘fname’, not ‘inFile’ */
    exit(1);
  }
  ...
}
```

CCured's `printf` handling notices statically that the type expected by the argument string (`char *`) does not match the type of the actual argument (`FILE *`) and flags an error. Even if the format string were not available statically the CCured version of `printf` would catch the error at run-time when the type of the next expected argument and the next actual argument failed to match.

One `go` error is near line 2241 of `g25.c` which contains code like the following:

```
  int lnbn[841];
  int so;
  for(ptr = nblbp[s]; ptr != EOL; ptr = links[ptr]){
    if(edge[list[ptr]] == edge[s])
      so = list[ptr];
    if(edge[list[ptr]] < edge[s])
      dir2 = list[ptr]-s;
  }
  if(lnbn[so] != 4) { /* BUG: ‘so’ may be uninitialized */
    ...
  }
```

CCured's pointer kind inference classifies `lnbn` as a safe pointer to an array of 841 integers. When it is used as an array it is cast to a `SEQ` pointer to an integer and CCured ensures that `&lnbn <= &lnbn[so] < &lnbn[841]`. When `so` is uninitialized it may fall out of that valid range and trigger a CCured run-time exception. Testing fails to reveal this bug as a segmentation fault if the value of `so` is small or other data structures are placed just before or after `lnbn` in memory.

As a final example, `compress` features a pointer that walks off the end of an array near line 867 of compress95.c. This particular example has been reproduced in its entirity in order to give a flavor for how subtle these bugs can be and how this bug might have escaped manual detection despite the frequent use of this code as a benchmark. The comments of the form `/* BUG:` were added.

```
code_int
getcode() {
  /*
   * On the VAX, it is important to have the register declarations
   * in exactly the order given, or the asm will break.
   */
  register code_int code;
  static int offset = 0, size = 0;
  static char_type buf[16];     /* BUG: we can walk off this array! */
  register int r_off, bits;
```

```
  register char_type *bp = buf; /* BUG: using this ‘bp’ pointer */

  if ( clear_flg > 0 || offset >= size || free_ent > maxcode ) {
      /*
       * If the next entry will be too big for the current code
       * size, then we must increase the size.  This implies reading
       * a new buffer full, too.
       */
      if ( free_ent > maxcode ) {
          n_bits++;
          if ( n_bits == maxbits )
              maxcode = maxmaxcode;   /* won’t get any bigger now */
          else
              maxcode = MAXCODE(n_bits);
      }
      if ( clear_flg > 0) {
          maxcode = MAXCODE (n_bits = INIT_BITS);
          clear_flg = 0;
      }
      size = readbytes( buf, n_bits );
      if ( size <= 0 )
          return -1;                  /* end of file */
      offset = 0;
      /* Round size down to integral number of codes */
      size = (size << 3) - (n_bits - 1);
  }
  r_off = offset;
  bits = n_bits;
  /*
   * Get to the first byte.
   */
  bp += (r_off >> 3); /* BUG: increase bp by (r_off >> 3)
                       * which is the same as (offset >> 3) */
  r_off &= 7;
  /* Get first part (low order bits) */
  code = (*bp++ >> r_off);
  bits -= (8 - r_off);
  r_off = 8 - r_off;  /* now, offset into code word */
  /* Get any 8 bit parts in the middle (<=1 for up to 16 bits). */
  if ( bits >= 8 ) {
      code |= *bp++ << r_off;
      r_off += 8;
      bits -= 8;
  }
  /* high order bits. */
  code |= (*bp & rmask[bits]) << r_off; /* BUG: ‘*bp’ out of bounds! */
  offset += n_bits;
  return code;
}
```

It turns out that the final dereference of `bp` in the code can produce an out-of-bounds error. This particular error is very difficult to catch by inspection and relies on the fact that the variable `offset` is `static` (meaning that it keeps the same value between different invocations of this procedure) and can continue to increase unless a certain condition is met (the first and second `if`s in the procedure) and it is reset to zero. If it becomes sufficiently high,

Module Name	Lines of code	% sf/sq/d	**CCured Ratio**
asis	149	72/28/0	**0.96**
expires	525	77/23/0	**1.00**
gzip	11648	85/15/0	**0.94**
headers	281	90/10/0	**1.00**
info	786	86/14/0	**1.00**
layout	309	82/18/0	**1.01**
random	131	85/15/0	**0.94**
urlcount	702	87/13/0	**1.02**
usertrack	409	81/19/0	**1.00**
WebStone	n/a	n/a	**1.04**

Figure 13: Apache Module Performance. A ratio of 1.04 means that the CCured module was 4% slower than the original. Any slowdown is within the noise.

the line `bp += (r_off >> 3)` and the successive `*bp++` expressions[4] can push `bp` beyond `buf+15`, its maximal legal value. At that point the final read from `bp` reads random data allocated next to `bp`. Both Purify and testing fail to find this memory error because the pointer does not stray sufficiently far from the original object. In CCured the assignment `bp = buf` is treated as a cast from the `SAFE` pointer to the array `buf` into the `SEQ` pointer `bp` and the bounds information (i.e., `buf` $\leq$ `bp` $<$ `buf + 16`) stays with `bp`. A run-time exception is raised at the point of the invalid dereference.

As a second experiment we used CCured to make memory-safe versions of a number of Apache 1.2.9 modules and then we compared their performance (measured as the total number of bytes transmitted by the server divided by the time to receive the last byte) to that of the originals. (Measurements of time-to-first-byte were similar). Figure 13 shows the results: any slowdown is within the noise. Each line represents 1,000 requests with file sizes of 1K, 10K and 100K. In all cases the module code made safe by CCured was executed on every request. The modules perform standard webserver duties: `asis` provides special raw file support, `expires` prepends timeout header information, `gzip` compresses the file contents, `usertrack` provides cookie management, and so on. `WebStone` is 100 iterations of the `manyfiles` WebStone 2.5 benchmark with every request affected by the `expires`, `gzip`, `headers`, `urlcount` and `usertrack` modules. To prevent physical network latency from masking the cost of run-time checks, the experiments were conducted with both the client and the server on the same Linux machine: throughput averaged 1150 K/s. Buffer overruns and other security errors with Apache modules have led to a least one remote security exploit [Sec00]. That particular bug was a format string problem that is prevented by the variable argument handling described in Section 5.

Converting Apache modules to CCured required manual intervention. We inspected the modules for `void*` casts and extended the Apache module API with CCured wrappers for Apache's array-handling functions. We marked Apache's internal alloc and free functions as `malloc`-like and noted `printf`-like debugging functions. We also constructed the appropriate unions for some variable-argument functions (e.g., `ap_strcat(...)`). Finally, we annotated data structures that are created by Apache and passed to the module so that they would be inferred as having lean pointers (e.g., `STRING` instead of `SEQ` for passed filenames).

We also used CCured to instrument two Linux kernel device drivers. `pcnet32` is a PCI Ethernet network driver and `sbull` is a ramdisk block-device driver. Both were compiled and run using Linux 2.4.5. We replaced Linux inline assembly macros with calls to wrapper functions. This had the advantage of allowing us to insert appropriate run-time checks into otherwise opaque assembly (e.g., we perform bounds-checks for the Linux internal `memcpy` routines). Some Linux macros (like `INIT_REQUEST`) were assumed to be part of the trusted interface. Finally, some low-level casts were trusted as part of the interface. Porting `sbull` involved changing about 20 lines of the driver source, `pcnet32` required only 5 changes (mostly removing casts to `void *`). The performance measurements are shown in Figure 14. `pcnet32` measures maximal throughput; "ping" measures latency. `sbull` measures blocked reads (writes and character I/O were similar); "seeks" measures the time to complete a set number of random seeks.

[4] In C the expression `*bp++` dereferences `bp` and saves the result, increments `bp` by one times the size of `bp`'s base type and then returns the saved result.

Name	Lines of code	% sf/sq/d	**CCured Ratio**
`pcnet32`	1661	92/8/0	**0.99**
ping			**1.00**
`sbull`	1013	85/15/0	**1.00**
seeks			**1.03**
`ftpd`	6553	79/12/9	**1.01**

Figure 14: Linux Device Driver and FTP server performance. A ratio of 1.03 means the CCured version is 3% slower than the original.

Finally, we ran `ftpd-BSD-0.3.2-5` through CCured. This involved writing wrappers for some networking functions, but has the strong advantage of eliminating all memory safety security errors. In fact, this version of `ftpd` has a known vulnerability (buffer overflow) in the `replydirname` function, which we verified that CCured prevents. The biggest hurdle was writing a 70-line wrapper for the `glob` function. As Figure 14 shows, we could not measure any significant performance difference between the CCured version and the original. As with the Apache modules, the client and server were run on the same machine to avoid I/O latency.

7 Related Work

An entire body of research [ACPP91, AWL94, CF91, Hen92, KF93, SSJ98, Tha90, WC97] examines the notion of a Dynamic type whose values are $\langle type, ptr \rangle$ pairs. Such a value can only be used by first extracting and checking the type. In particular, one can only write values that are consistent with the packaged type. Because the underlying value's static type is carried within the Dynamic package, and checked at every use, there is not a problem with Dynamic aliases for statically-typed data.

This is in contrast to CCured's `DYN`, which allows arbitrary interpretation of the value read (except that the tags prevent misinterpreting a pointer base field), and arbitrary types to be written. Thus, a memory word's type may change during execution. This flexibility is built into CCured because we expect some C programs to allocate large areas of memory and re-use that memory in different ways. However, its cost is that `DYN` must be a closed world, with no aliases of statically-typed data. The inference algorithm in CCured bears some resemblance to Henglein's inference algorithm [Hen92], but we consider physical subtyping, pointer arithmetic and updates. Henglein's algorithm has the nice feature that it does not require any type information to be present in the program. We believe that his algorithm does not extend to the more complex language we consider here and that existing C types contain valuable information that should be used to make inference both simpler and predictable (in terms of when a pointer will be inferred `DYN`).

Another line of research tries to find subsets of C which can be verified as type-safe at compile time. Chandra and Reps [CR99] 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. Siff et al. [SCB+99] identify that many casts in C programs are safe upcasts and present a tool to check such casts. CCured includes similar support, somewhat modified for soundness reasons and extended to handle pointer arithmetic. Smith et al. [SV98] 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. Ramalingam et al. [RFT99] have presented an algorithm for finding the coarsest acceptable type for structures in C programs. Each of these approaches requires programs to adhere to their particular subset, otherwise the program is rejected. CCured's static type system has comparable expressivity (CCured does well with arrays, some systems do better with polymorphism), but CCured can fall back on its very flexible `DYN` pointers to handle the corner cases.

A third popular approach is to add run-time checks to C programs. Kaufer et al. [KLP88] 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. [ABS94] 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 library compatibility. Jones and Kelly [JK97] 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 five to six. Fischer and Patil have presented a system that uses a second processor to perform the bounds checks [PF95]. Loginov et al. [LYHR01] store type information with each memory location, incurring a slowdown factor of five 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. The approaches of Austin et al. and Jones and Kelly are comparable to the implementation of CCured's `DYN` pointers. However, beyond array bounds check elimination, none of these techniques use type-based static analysis to aggressively reduce the overhead of the instrumented code.

8 Conclusions

Our paper presents a way to brings safety to the C programming language. Our novel finding is that most pointers in a C program are already used in a type-safe way. What is needed is a type system that can keep track of pointers. For this purpose, we describe the CCured type system, which for the first time combines dynamic types with physical subtyping and pointer arithmetic. Furthermore, we show a relatively simple inference algorithm that is able to infer that, as expected, most pointers do not need extensive run-time checking.

The CCured type system is not only expressive enough to handle many common C paradigms, but also intuitive. We found it easy to annotate programs by hand with pointer-kind information, or to predict the behavior of the inferencer and consequently to interpret any error messages. In some cases, however, we found that the type system is too simple to keep track of unusual invariants. For example, consider a word value that is to be interpreted as a pointer whenever it is a multiple of four. To handle these cases we might want to allow the program to declare new pointer kinds in such a way that the soundness of CCured is not compromised.

CCured has a number of disadvantages. When the inference algorithm infers that a pointer should be wide and that type passed to or from an external library, the program will fail to link correctly. In such cases a wrapper must be written to strip away the CCured-specific metadata before calling the external function. Work is currently underway to automate the generation of wrappers and store CCured metadata in a compatible manner.

At the moment, we are doing little in terms of optimizing the run-time checks. We remove the locally-redundant run-time checks, but we do not perform global optimizations or array-bounds checking optimizations. Since CCured is type-safe we could soundly add optimizations which would be unsound in C, such as those designed for Java. CCured is also in position to benefit from an improved garbage collector. Unlike C, in CCured we have enough information about pointers (either statically or at run-time) that we could use a copying garbage collector.

We believe CCured would be most useful either for mission-critical programs such as network services or for components in extensible system. Another potential application that we have not explored yet is to use CCured to make Java native methods or inline C code in C# provably type safe and thus first-class citizens. For those applications it would be possible and useful to produce proof-carrying code automatically from C programs, not only from Java programs as it is possible today [CLN+00].

9 Acknowledgments

We would like to thank George Necula for sage advice and generally making the CCured project possible. Thanks go to Scott McPeak, Jeremey Condit and Matthew Harren for extensive work on the implementation and systems applications of CCured and to Aman Bhargava for additional implementation assistance. We would like to thank Kiri Wagstaff, Alex Aiken and Donna Weimer for helpful comments on earlier drafts of this paper. 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 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.

References

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar 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.*

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 163–173, New York, NY, 1994.

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

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, May 2000.

[CR99] Satish Chandra and Thomas 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, September 6 1999.

[Hen92] Fritz Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 205–215, 1992.

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

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

[KF93] Andreas Kind and Horst Friedrich. A practical approach to type inference for EuLisp. *Lisp and Symbolic Computation*, 6(1/2):159–176, 1993.

[KLP88] Stephen Kaufer, Russel Lopez, and Sesha Pratap. Saber-C: an interpreter-based programming environment for the C language. In *Proceedings of the Summer Usenix Conference*, pages 161–171, 1988.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language (second edition).* Prentice-Hall, Englewood Cliffs, N. J., 1988.

[LYHR01] Alexey Loginov, Susan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. In *Proceedings of FASE 2001: Fundamental Approaches to Software Engineering*, April 2001.

[NMW02a] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *The 29th Annual ACM Symposium on Principles of Programming Languages*, pages 128–139. ACM, January 2002. Available from http://raw.cs.berkeley.edu/Papers/.

[NMW02b] George C. Necula, Scott McPeak, and Westley Weimer. CIL: Intermediate language and tools for the analysis of C programs. In *International Conference on Compiler Construction*, pages 213–228. Grenoble, France, April 2002. Available from http://raw.cs.berkeley.edu/Papers/.

[PF95] Harish Patil and Charles N. Fischer. Efficient run-time monitoring using shadow processing. In *Automated and Algorithmic Debugging*, pages 119–132, 1995.

[PF97] Harish Patil and Charles N. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software—Practice and Experience*, 27(1):87–110, January 1997.

[RFT99] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Symposium on Principles of Programming Languages*, pages 119–132, January 1999.

[SCB+99] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas 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.

[Sec00] SecuriTeam.com. PHP3 / PHP4 format string vulnerability. *http://www.securiteam.com/securitynews/6O00T0K03O.html*, December 2000.

[SPE95] SPEC 95. Standard Performance Evaluation Corportation Benchmarks. *http://www.spec.org/osg/cpu95/CINT95*, July 1995.

[SSJ98] Mark Shields, Tim Sheard, and Simon L. Peyton Jones. Dynamic typing as staged type inference. In *Symposium on Principles of Programming Languages*, pages 289–302, 1998.

[SV98] Geoffrey Smith and Dennis Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(1–3):49–72, 1998.

[Tha90] Satish R. Thatte. Quasi-static typing. In *Conference record of the 17th ACM Symposium on Principles of Programming Languages (POPL)*, pages 367–381, 1990.

[WC97] Andrew Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 1997.

[WFBA00] David Wagner, Jeff Foster, Eric Brewer, and Alexander Aiken. A first step toward automated detection of buffer overrun vulnerabilities. In *Network Distributed Systems Security Symposium*, pages 1–15, February 2000.