A Theory of Implementation-Dependent Low-Level Software

Marius Nita, Dan Grossman, and Craig Chambers

University of Washington

Abstract. We present a theory describing implementation-dependent assumptions that a C program might make, such as the size and alignment of data. We define a static analysis to encode such assumptions in a constraint that describes language implementations (i.e., compilers and architectures) on which a program is memory-safe. The constraint produced by the analysis is a formula in a theory of first-order logic for which implementations are models. By defining an abstract machine that is parameterized by an implementation, we can prove the analysis sound. This foundation explains some common coding practices and the poorly understood assumptions they are implicitly making.

1 Introduction

In recent years, research has demonstrated many ways to improve the quality of C code by using programming-language technology. Such work detects safety violations (e.g., array-bounds errors), enforces temporal protocols, and provides new languages and compilers for reliable systems programming. The results are a practical success for programming-language theory. However, there remains a crucial and complementary set of complications that this paper begins to address:

The memory-safety of a C program often depends on assumptions that hold for some but not all compilers and machines.

Examples of assumptions include how `struct` values are layed out in memory, the size of values, and alignment restrictions on memory accesses. To our knowledge, existing work on safe low-level code (1) checks or simply assumes full portability (e.g., that the input program is unaffected by structure padding), or (2) checks the input program assuming one particular language implementation.

Demanding full portability for all code (e.g., by enforcing informally specified restrictions on C programs [14]) is too strict because low-level code often has inherently non-portable parts. It is often impractical to rewrite large legacy applications in portable languages or to use perfect libraries that abstract all the issues. Such approaches can also be a poor match for low-level code and assume that portable languages or libraries are available for an ever-increasing number of platforms.

Conversely, implicitly relying on some language-implementation assumptions can lead to pernicious defects that lie dormant until one uses an implementation violating the assumptions. Defects like dangling-pointer dereferences are

largely independent of the language implementation (so testing or verification on the "old machine" can find many of them), but defects like assuming two `struct` types have similar layouts are not. The results can be severe. Conceptually simple tasks like porting an application from a 32-bit to a 64-bit machine become expensive and error-prone. Software tested on widely available platforms can break when run on novel hardware such as embedded systems. Widely used compilers cannot change data-representation strategies without breaking legacy code that implicitly relied on "undocumented behavior."

Common practice confronts this "somewhat but not completely portable" dilemma by manually isolating implementation-dependent assumptions. For example, the Linux source code has an `arch` subdirectory; avoiding assumptions in the rest of the kernel is left to careful coding. As another example, a garbage collector for a high-level language might assume pointers are four bytes and aim to be correct for any implementation satisfying this assumption. Similarly, a run-time system accessing object headers might make assumptions about the layout of `struct` values. In all cases, the code is *semi-portable*, meaning it is—by design—correct for many but not all compilers and machines.

To help support semi-portable programming, we have begun building a semantics-based porting tool for C. When finished, it will input a C program and output a *description of a set of implementations* on which the program "makes sense." Defining the implementation-description language has forced us to give precise meaning to poorly understood platform-specific issues. The result is a foundation for any analysis seeking to account for implementation dependencies.

Many language tools have wrestled with semi-portability and implementation dependencies, but the issues have not been isolated and considered rigorously. To do so, we have built a novel model for a small-but-relevant C-level language.

1.1 Approach

The key to our formal model is isolating the idea of an *implementation*. An implementation has two roles: (1) as a parameter to the operational semantics, and (2) as something a portability constraint describes. The actual definition of an implementation includes things like determining the offset of a `struct` field and what alignment restrictions memory accesses must obey. This work fully describes the details, but the main insight is this: by parameterizing the operational semantics by an implementation, we can take a program P and state that a property, namely safety, holds for P on implementations satisfying a constraint S. That is, P is semi-portable assuming S.

To see how an implementation is a parameter to the operational semantics, consider a pointer dereference `*e`. The number of bytes accessed depends on the size of `e`'s type, and this size depends on the implementation. So the operational semantics can have the form $\mathit{impl} \vdash P \to P'$ where *impl* is an implementation and P is a program state. That way, the dereference rule can use *impl* to get the size (and become stuck if *impl* deems the access misaligned).

As for portability constraints, they are formulas in a theory of first-order logic for which implementations as models. For example, the constraint "$\mathrm{access}(4, 8) \wedge$

size(long) = 8" is modeled by any implementation in which values of type long occupy 8 bytes and 8-byte loads of 4-byte aligned data are allowed.[1]

Now given a program P we can try to find a constraint S such that if $impl \models S$, then the abstract machine does not get stuck when running P given $impl$. In particular, it will not treat an integer as a pointer, read past the end of a `struct`, perform a misaligned memory access, etc. Finding an S that describes exactly the set of "safe" implementations is trivially undecidable, so a sound approximation (all models are safe, but not all safe implementations are models) is warranted. In this work, we take a very conservative approach: A type system for source programs produces S (which one can view as an effect), using no flow-sensitivity or alias information. In practice, a code-analysis tool may use a more sophisticated analysis, but the set-up (produce an S given P) remains the same.

The key theorem states that the S generated is sound: P cannot get stuck on any $impl$ modeling S. To prove it, we define a second type system for program states. This second type system, which exists only for the proof, is parameterized by an $impl$ like the dynamic semantics. The proof then has two parts:

1. The second type system and operational semantics enjoy the conventional preservation and progress properties (not including orthogonal issues like uninitialized memory, which we could prevent via complementary techniques).
2. If the first type system produces S given P, then P type-checks in the second type system for any $impl$ such that $impl \models S$.

1.2 Outline

Section 2 presents examples of semi-portable code. Section 3 presents our core formal model, including the definition of implementations, our first-order theory, the dynamic and static semantics of our language, and our soundness theorem. Section 4 briefly discusses extensions to the model we have investigated, most importantly support for arrays. We then discuss related work and conclude.

2 Examples

Section 2.1 presents tiny code examplese to explain issues of semi-portability and relevant implementation constraints. Section 2.2 complements this "tutorial" with actual systems and coping strategies related to these concepts.

2.1 Small Code Fragments

Example 0: Accessing Memory A memory operation such as `(*e).f` accesses n bytes at an alignment m. If `e` has type `struct T*` and the `f` field has type τ, then n is the τ's *size* and m is the greatest common divisor of the *alignment* of `struct T*` and the *offset* of `f`.

Implementations choose sizes, alignments, and offsets such that cast-free programs do not fail. For example, if a machine prohibits 8-byte accesses on 4-byte

[1] This example constraint is slightly simplified; see Section 3.4.

alignments, a compiler might put pad bytes before `f` fields or break 8-byte accesses into two 4-byte accesses. In this paper, we describe implementations abstractly with an *access* function of type Int → Int → Bool, as well as *size* and *alignment* functions mapping types to integers and alignments, respectively.[2]

The code `(*e).f` therefore needs the constraint access(n, m) with m and n as defined above. However, this assumes `e` will evaluate to a pointer with alignment m and a τ at the right offset. The constraints for casts must ensure that.

Example 1: Prefix

```
struct S1 { int* f1; int* f2; int* f3; };
struct D1 { int* g1; int* g2; };
```

A cast from `struct S1*` to `struct D1*` requires that `struct D1` has a less stringent alignment than `struct S1`, and for each field in `struct D1` there is a field of compatible type in `struct S1` at the same offset. In this case, the C standard requires every implementation to meet these constraints (and for `g1` and `f1` to have offset 0 and `g2` to have the same offset as `f2`), but our purpose is to capture these and less portable notions precisely.

Example 2: Flattening and Alignment

```
struct S2 { int* f1; struct {int* f2; double f3;} f4; };
struct D2 { int* g1; int* g2; };
```

A cast from `struct S2*` to `struct D2*` has similar constraints as in Example 1, but this time the C standard provides no guarantee. In fact, some implementations put pad bytes before `f4` because of alignment constraints and an assumption that all `struct` types are defined at top-level. Our system generates a constraint for this cast preventing such a representation mismatch.

Example 3: Suffix

```
struct S3 { int* f1; int* f2; double f3; };
struct D3 { int* g1; double g2; };
struct S3* x = ...;
struct D3* y = (struct D3 *)(&(x->f2));
```

The cast in the initializer for `y` above is a situation where the source and destination types both point to an `int*` followed by a `double`. However, an implementation with 4-byte pointers, 8-byte doubles, and 8-byte alignment of doubles cannot support this cast because `struct D3` has more padding. Implementations without padding can allow this cast, even though `&x->f2` has type `int**` in C.

[2] As explained later, an alignment is an offset and a modulus.

Example 4: Arrays and Prefixes

```
struct S1 * x = ...;
struct D1 * y = ((struct D1 *)x)[7];
```

Prior examples implicitly assumed the cast-target was not used as an array (i.e., it was used as a pointer to only one element). This issue is orthogonal to array-bounds violations; we must reject the cast in Example 4 even if `x` points to more than 7 elements. Section 4 discusses extending our model to support arrays.

Example 5: Safe-But-Inequivalent Implementations

```
struct S5 { long f1; };
struct D5 { short g1; short g2; };
```

Assuming there is no padding, `sizeof(long)==2*sizeof(short)`, and that there are no misaligned accesses, a cast from `struct S5*` to `struct D5*` is safe. However, endianness lets different implementations behave differently. We leave such notions of equivalence to future work, focusing here only on safety, which is still incredibly useful in writing and porting semi-portable code. In particular, our approach detects many no-padding assumptions.

2.2 Practical Scenarios

The extent of the portability problem is not precisely known because defects lie dormant until one changes hardware or compiler.

The LinuxARM project, a port of Linux to the ARM embedded processor, provides compelling evidence that defects are subtle and widespread. The ARM compiler uses at least 4-byte alignment for all structs whereas gcc for the x86 uses less alignment for structs containing only `short` and `char` fields. To quote [2]:

> At this point, several years of fixing alignment defects in Linux packages have reduced the problems in the most common packages. Packages known to have had alignment defects are: Linux kernel; binutils; cpio; RPM; Orbit (part of Gnome); X Windows. This list is *very* incomplete.

They also note that defects sometimes lead to alignment traps, but sometimes lead to silent data corruption. Kernel developers are told to, "be careful" [17].

Ports to 64-bit platforms are also revealing. The state-of-the-art appears to be lint-like technology, such as gcc's `-Wpadded` flag, which reports all uses of padding. However, aggressive compiler warnings can produce so much information for legacy code that some people suggest using multiple independent compilers and looking only at lines for which they all produce warnings [18].

3 Core Language

This section develops a formal model that can explain examples 0–3 above. We define a core language, a definition of *implementation*, a dynamic semantics, a

a logic that constrains implementations, a type-and-effect system to produce constraints, and a safety theorem. For tractability, we consider just a small expression language inspired by C and missing orthogonal language features such as functions. For simplicity, we also assume all pointers have the same size.

3.1 Idealized Syntax

$$\begin{array}{ll} \tau ::= \text{short} \mid \text{long} \mid \tau* \mid N & e ::= s \mid l \mid x \mid e = e \mid e.f \mid (\tau*)e \mid *(\tau*)(e) \mid (\tau*)\&e{\rightarrow}f \\ t ::= N\{\overline{\tau\ f}\} & \quad \mid \ \mathsf{new}\ \tau \mid e; e \mid \mathsf{if}\ e\ e\ e \mid \tau\ x;\ e \end{array}$$

Fig. 1. Source-Language Syntax

Our source programs (Figure 1) are a small subset of C with some small, convenient changes. Most notably, all terms are expressions. A program is a list of struct definitions ($\overline{t}$) and an expression (e). (We write $\overline{x}$ for a sequence of elements drawn from x and $\cdot$ for the empty sequence. We write x^i for a length-i sequence.) Struct definitions have global scope to allow mutually recursive types.

Types τ include short and long, pointers ($\tau*$), and struct types (N rather than the more verbose `struct` N). As in C, all pointers are explicit. A struct definition (t) names the type and gives a sequence of fields. For simplicity, we assume all field names in a program are disjoint. Several expression forms are identical to C, including short and long constants (s and l), variables (x), assignments ($e = e$), field access ($e.f$), and pointer casts ($(\tau*)e$).

For pointer dereference ($*(\tau*)(e)$) and pointing to a field ($(\tau*)\&e \rightarrow f$), it is a technical convenience to require a type annotation in the syntax. After all, dereference in C *is* type-directed (if e has type $\tau*$, then $*e$ reads `sizeof(`τ`)` bytes). The cast in $(\tau*)\&e{\rightarrow}f$ helps with "suffix casts" as in Example 3.

The remaining expression forms are for memory allocation or control flow. $\mathsf{new}\ \tau$ allocates uninitialized space to hold a τ. $e; e$ is sequence. $\mathsf{if}\ e\ e\ e$ is a conditional (branching on whether the first expression is 0). Finally, $\tau\ x;\ e$ binds a local variable x of type τ in e. Memory management is not our concern, so the dynamic semantics uses heap allocation for local variables.

As defined in Section 3.3, program evaluation depends on an implementation *impl* and modifies a heap H. We write $\mathit{impl}; \overline{t} \vdash H; e \rightarrow H'; e'$ for one evaluation step. Rather than define a *translation* to a lower-level implementation-dependent language, we *extend* e with new lower-level forms. This equivalent approach of consulting the implementation at run-time simplifies the metatheory while fully exposing the intricacies of implementation dependencies.

Figure 2 defines the syntactic extensions for run-time expressions and heaps. A value v is a sequence of "small values" w, which can be initialized bytes b, uninitialized bytes uninit, or pointers $\ell{+}i$. A pointer is a label ℓ and an offset i (Pointers to the middle of data have non-zero offsets.) This approach is higher level than assembly language but low enough for middle pointers, suffix casts, etc., making it "just right" for modeling semi-portable C.

$$
\begin{array}{lll}
 & b ::= 0 \mid 1 \mid \ldots \mid 255 & v ::= \overline{w} \\
i, a, o \in \mathbb{N} & w ::= b \mid \mathsf{uninit} \mid \ell{+}i & \alpha ::= [a, o] \\
 & e ::= \ldots \mid \overline{w} & H ::= \cdot \mid H, \ell \mapsto v, \alpha
\end{array}
$$

Fig. 2. Syntax extensions for run-time behavior

Heaps map labels to values and alignments $[a, o]$; the latter means address ℓ is o mod a. (Typically o is zero.) As Section 3.2 shows, an implementation specifies an alignment for each type; allocating an object of type τ produces a fresh heap location with this alignment.

3.2 Implementations

$$
\sigma \quad ::= \quad \text{byte} \mid \text{pad}[i] \mid \text{ptr}_\alpha(\overline{\sigma}) \mid \text{ptr}_\alpha(N)
$$

$$
\begin{array}{ll}
\mathit{impl.xtype}(\overline{t}, \tau) = \overline{\sigma} & \mathit{impl.ptrsize} = i \\
\mathit{impl.align}(\overline{t}, \tau) = \alpha & \mathit{impl.xliteral}(s) = \overline{b} \\
\mathit{impl.offset}(\overline{t}, f) = i & \mathit{impl.xliteral}(l) = \overline{b} \\
\mathit{impl.access}(\alpha, i) = \{\text{true}, \text{false}\} &
\end{array}
$$

Fig. 3. Implementations and low-level types

The components of an implementation (*impl*) guide the dynamic semantics. Figure 3 summarizes these components:

- A translation of types into a lower-level representation ($\overline{\sigma}$), described below. We write $\mathit{impl.xtype}(\overline{t}, \tau)$ for the $\overline{\sigma}$ corresponding to the translation of a type τ (assuming type definitions $\overline{t}$).
- An *alignment* function (*impl.align*) returns the alignment α used to allocate space for a τ.
- An *offset* function (*impl.offset*) takes a field f and returns the number of bytes from the beginning of the nearest enclosing struct to the field f.
- An *access* function takes an alignment and a size and returns true if accessing size-bytes of memory at the alignment is not an error. We write $\mathit{impl.access}(\alpha, i)$ if this function returns true.
- The size of pointers (*impl.ptrsize*) is a constant i. This is a slight simplification since a C implementation could use different sizes for different pointers.
- A *literal* function (*impl.xliteral*) translates integer literals into byte sequences.

The access function is typically associated with hardware and the other components with compilers. A "sensible" implementation cannot define its components in isolation (e.g., the type translation must mind the access function); our constraint language will let us define such restrictions.

Low-level types (the target of *impl.xtype*) are $\overline{\sigma}$, a sequence of σ. For example, if long is four bytes, the translation is byte byte byte byte. The type pad[i] represents i bytes of padding (data of unknown type but known size). The type

$\text{ptr}_\alpha(\overline{\sigma})$ describes pointers to data described by $\overline{\sigma}$ at alignment α (i.e., α is the alignment of the pointed-to data). As a technical point, we disallow the type N for low-level types except for the form $\text{ptr}_\alpha(N)$. This restriction simplifies type equalities without restricting implementations or disallowing recursive types.

3.3 Dynamic Semantics

The dynamic semantics is a small-step rewrite system defined using evaluation contexts and parameterized by an implementation and type definitions. Figure 4 holds the full definition for $impl; \bar{t} \vdash H; e \rightarrow H'; e'$. As in C, the left side of assignments ("left-expressions") are evaluated differently from other expressions ("right-expressions"). Therefore, we have two sorts of contexts (L and R) defined by mutual induction and a different sort of primitive reduction ($\xrightarrow{\text{l}}$ and $\xrightarrow{\text{r}}$) for each sort of context. In particular, $\mathsf{R}[e]_\text{r}$ is a right-context R containing a right-hole filled by e and $\mathsf{R}[e]_\text{l}$ is a right-context R containing a left-hole filled by e. Each context contains one right-hole or one left-hole (not both).

The primitive reductions that do not use *impl*. (D-Cast, D-Seq, D-IfF, D-IfT) are straightforward. Note casts have no run-time effect. D-Short and D-Long use *impl*, but only to translate literals to byte-sequences.

D-New extends the heap with a new label holding uninitialized data. *impl* determines the alignment and size of the new space, with the latter computed by applying the auxiliary size function to the translation of the allocated type. The resulting value is the pointer ℓ+0. Note the type system does not prevent getting stuck due to uninitialized data. D-Let is much like D-New (it has identical hypotheses) because we heap-allocate local variables. The resulting expression is the substitution of $*(\tau*)(\ell\text{+}0)$ for x.

D-Deref gets data from the heap by extracting a sequence "from the middle" of $H(\ell)$. The sequence is from offset j (where the evaluated expression is $*(\tau*)(\ell + j)$) to $j + k$ (where k is the size of τ's translation). If one cannot "carve up" $H(\ell)$ in this way, the rule does not apply (so the machine is stuck). As expected, we use *impl.access* to enforce alignment on the memory access.

D-Assign has hypotheses identical to D-Deref plus the requirement that the right-hand side be a value equal in size to the value being replaced in the heap. The resulting heap differs only from offset j to offset $j + k$ of $H(\ell)$.

D-Faddr takes a pointer value and increases its offset by the offset of the field f. D-FetchL, the one primitive reduction in left contexts, is similar, but we also change a type to reflect that $e.f$ refers to less memory than e. A "left-value" (i.e., a terminal left-expression) looks like $*(\tau*)(\ell\text{+}j)$.

D-Fetch uses offset and size information from *impl* to get a subsequence of a value. It does not use the access function because it does not access the heap.[3]

The functions $\text{size}(impl, \overline{\sigma})$ and $\text{size}(impl, \overline{w})$, used in several rules, return byte-counts. Their definitions are easy but omitted due to space constraints.

[3] On real machines, large values do not fit in registers. We could model this by treating field access as an address-of-field computation followed by a dereference.

$$\mathsf{R} ::= [\cdot]_{\mathsf{r}} \mid \mathsf{L} = e \mid *(\tau*)(\ell{+}i) = \mathsf{R} \mid \mathsf{R}.f \mid *(\tau*)(\mathsf{R}) \mid (\tau*)\mathsf{R} \mid \mathsf{R}; e \mid (\tau*)\&\mathsf{R}{\rightarrow}f \mid \mathsf{if}\ \mathsf{R}\ e\ e$$
$$\mathsf{L} ::= [\cdot]_{\mathsf{l}} \mid \mathsf{L}.f \mid *(\tau*)(\mathsf{R})$$

$$\frac{D \vdash H; e \xrightarrow{\mathsf{r}} H'; e'}{D \vdash H; \mathsf{R}[e]_{\mathsf{r}} \rightarrow H'; \mathsf{R}[e']_{\mathsf{r}}} \qquad \frac{D \vdash H; e \xrightarrow{\mathsf{l}} H'; e'}{D \vdash H; \mathsf{R}[e]_{\mathsf{l}} \rightarrow H'; \mathsf{R}[e']_{\mathsf{l}}}$$

D-Cast $\dfrac{}{D \vdash H; (\tau*)\overline{w} \xrightarrow{\mathsf{r}} H; \overline{w}}$ D-Seq $\dfrac{}{D \vdash H; (v; e) \xrightarrow{\mathsf{r}} H; e}$ D-IfF $\dfrac{}{D \vdash H; \mathsf{if}\ 0^i\ e_1\ e_2 \xrightarrow{\mathsf{r}} H; e_2}$

D-IfT $\dfrac{b_1 \ldots b_i \neq 0^i}{D \vdash H; \mathsf{if}\ (b_1 \ldots b_i)\ e_1\ e_2 \xrightarrow{\mathsf{r}} H; e_1}$ D-Short $\dfrac{impl.\mathit{xliteral}(s) = \overline{b}}{impl; \overline{t} \vdash H; s \xrightarrow{\mathsf{r}} H; \overline{b}}$ D-Long $\dfrac{impl.\mathit{xliteral}(l) = \overline{b}}{impl; \overline{t} \vdash H; l \xrightarrow{\mathsf{r}} H; \overline{b}}$

D-New $$\frac{\ell \notin \mathrm{Dom}(H) \quad impl.\mathit{align}(\overline{t}, \tau) = \alpha \quad impl.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma} \quad \mathrm{size}(impl, \overline{\sigma}) = i}{impl; \overline{t} \vdash H; \mathsf{new}\ \tau \xrightarrow{\mathsf{r}} (H, \ell \mapsto \mathsf{uninit}^i, \alpha); \ell{+}0}$$

D-Let $$\frac{\ell \notin \mathrm{Dom}(H) \quad impl.\mathit{align}(\overline{t}, \tau) = \alpha \quad impl.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma} \quad \mathrm{size}(impl, \overline{\sigma}) = i}{impl; \overline{t} \vdash H; \tau\ x;\ e \xrightarrow{\mathsf{r}} (H, \ell \mapsto \mathsf{uninit}^i, \alpha); e\{*(\tau*)(\ell{+}0)/x\}}$$

D-Deref $$\frac{\begin{array}{ll} H(\ell) = \overline{w}_1\overline{w}_2\overline{w}_3, [a, o] & \mathrm{size}(impl, \overline{w}_1) = j \qquad impl.\mathit{access}([a, o+j], k) \\ impl.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma} & \mathrm{size}(impl, \overline{w}_2) = \mathrm{size}(impl, \overline{\sigma}) = k \end{array}}{impl; \overline{t} \vdash H; *(\tau*)(\ell{+}j) \xrightarrow{\mathsf{r}} H; \overline{w}_2}$$

D-Assign $$\frac{\begin{array}{ll} H(\ell) = \overline{w}_1\overline{w}_2\overline{w}_3, [a, o] & \mathrm{size}(impl, \overline{w}_1) = j \qquad impl.\mathit{access}([a, o+j], k) \\ impl.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma} & \mathrm{size}(impl, \overline{w}_2) = \mathrm{size}(impl, \overline{\sigma}) = \mathrm{size}(impl, \overline{w}) = k \end{array}}{impl; \overline{t} \vdash H; (*(\tau*)(\ell{+}j)) = \overline{w} \xrightarrow{\mathsf{r}} (H, \ell \mapsto \overline{w}_1\overline{w}\,\overline{w}_3, [a, o]); \overline{w}}$$

D-Faddr $$\frac{impl.\mathrm{offset}(f) = j'}{impl; \overline{t} \vdash H; (\tau)\&\ell{+}j{\rightarrow}f \xrightarrow{\mathsf{r}} H; \ell{+}(j + j')}$$

D-FetchL $$\frac{impl.\mathrm{offset}(f) = j' \qquad N\{\ldots \tau_2\ f \ldots\} \in \overline{t}}{impl; \overline{t} \vdash H; (*(\tau_1*)(\ell{+}j)).f \xrightarrow{\mathsf{l}} H; *(\tau_2*)(\ell{+}(j + j'))}$$

D-Fetch $$\frac{\begin{array}{ll} N\{\ldots \tau\ f \ldots\} \in \overline{t} & impl.\mathit{offset}(\overline{t}, f) = \mathrm{size}(impl, \overline{w}_1) \\ impl.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma} & \mathrm{size}(impl, \overline{\sigma}) = \mathrm{size}(impl, \overline{w}_2) \end{array}}{impl; \overline{t} \vdash H; \overline{w}_1\overline{w}_2\overline{w}_3.f \xrightarrow{\mathsf{r}} H; \overline{w}_2}$$

Fig. 4. Dynamic Semantics (letting $D ::= impl; \overline{t}$)

3.4 First-Order Formulas

To define a sound type system for our language, we need to limit what implementations we consider. That is, "P does not get stuck" makes no sense, but "P run on implementation *impl* does not get stuck" does. We choose to use first-order logic to give a syntactic representation to a set of implementations; a formula S represents the implementations that model it, i.e., the set $\{impl \mid impl \models S\}$.

The syntax for formulas S is first-order logic with (1) sorts for aspects of our language (fields f, types τ, low-level types $\overline{\sigma}$, alignments α, etc.), (2) arithmetic,

syntax	interpretation under *impl*	defined in
$\text{xtype}(\bar{t}, \tau)$	$\mathit{impl.xtype}(\bar{t}, \tau)$	Figure 3
$\text{align}(\bar{t}, \tau)$	$\mathit{impl.align}(\bar{t}, \tau)$	
$\text{offset}(\bar{t}, f)$	$\mathit{impl.offset}(\bar{t}, f)$	
$\text{access}(\alpha, i)$	$\mathit{impl.access}(\alpha, i)$	
$\text{xliteral}(s), \text{xliteral}(l)$	$\mathit{impl.xliteral}(s), \mathit{impl.xliteral}(l)$	
$\text{size}(\overline{\sigma}), \text{size}(\overline{w})$	$\text{size}(\mathit{impl}, \overline{\sigma}), \text{size}(\mathit{impl}, \overline{w})$	omitted (straightforward)
$\text{subtype}(\bar{t}, \overline{\sigma}_1, \overline{\sigma}_2)$	$\mathit{impl}; \bar{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$	Figure 6
$\text{subalign}(\alpha_1, \alpha_2)$	$\vdash \alpha_1 \leq \alpha_2$	

Fig. 5. Function Symbols for the First-Order Theory

and (3) function symbols relevant to implementation-dependencies. Figure 5 defines these function symbols and their interpretation. These interpretations induce the full definition of $\mathit{impl} \models S$ as usual (e.g., $\mathit{impl} \models S_1 \wedge S_2$ if $\mathit{impl} \models S_1$ and $\mathit{impl} \models S_2$). Consider two example formulas:

- $\forall \tau, \bar{t}.\ \text{access}(\text{align}(\bar{t}, \tau), \text{size}(\text{xtype}(\bar{t}, \tau)))$
- Let $\bar{t}_0$ abbreviate: $N_1\{\text{short } f_1 \text{ short } f_2 \text{ short } f_3\}\ N_2\{\text{short } g_1 \text{ short } g_2\}$ in the formula: $\text{subtype}(\bar{t}_0, \text{xtype}(\bar{t}_0, N_1*), \text{xtype}(\bar{t}_0, N_2*))$.

The first formula says that every type must have a size and alignment that allows memory to be accessed. Without this constraint, a program like $\tau x; x = e$ could get stuck because D-Let uses the alignment $\mathit{impl.align}(\bar{t}, \tau)$ for the space allocated for x. The second formula requires a low-level subtyping relationship between two pointer types (see Section 3.5). This is the constraint our static semantics generates for a cast like in Example 1 from Section 2.

These examples also demonstrate the two flavors of formulas that arise in practice. First, there are constraints every "sensible" implementation would satisfy. We are uninterested in other implementations, but stating the requirements syntactically is simpler than revisiting our definition of implementations. Second, there are constraints describing semi-portable assumptions, i.e., we do not expect every implementation to satisfy them. Our static semantics produces a formula describing the assumptions of this form that a particular program makes.

The "sensible" constraints we assume are easy to enumerate and justify:

1. Size and alignment allows access of all types:
   $\forall \tau, \bar{t}.\ \text{access}(\text{align}(\bar{t}, \tau), \text{size}(\text{xtype}(\bar{t}, \tau)))$
2. Translation of literals respects the translation of their types:
   $\forall s, l, \bar{t}.\ \text{size}(\text{xliteral}(s)) = \text{size}(\text{xtype}(\bar{t}, \text{short}))$
   $\wedge\ \text{size}(\text{xliteral}(l)) = \text{size}(\text{xtype}(\bar{t}, \text{long}))$
3. Greater alignment does not restrict access:
   $\forall \alpha_1, \alpha_2, i.\ (\text{access}(\alpha_1, i)\ \wedge\ \text{subalign}(\alpha_2, \alpha_1)) \Rightarrow \text{access}(\alpha_2, i)$
4. Translation of $\tau*$ respects the alignment and translation of τ:
   $\forall \tau, \bar{t}.\text{subtype}(\bar{t}, \text{ptr}_{\text{align}(\bar{t}, \tau)}(\text{xtype}(\bar{t}, \tau)), \text{xtype}(\bar{t}, \tau*))$

5. Struct translation respects the offset and alignment of each field:
$\forall \overline{t}, \tau, f, \overline{\sigma}.\ (N\{\dots \tau\ f \dots\} \in \overline{t} \wedge (\text{xtype}(\overline{t}, \tau) = \overline{\sigma}) \Rightarrow$
$(\exists \overline{\sigma}_1, \overline{\sigma}_2, a, o, o'.\quad \text{xtype}(\overline{t}, N) = \overline{\sigma}_1 \overline{\sigma} \overline{\sigma}_2 \ \wedge\ \text{size}(\overline{\sigma}_1) = \text{offset}(\overline{t}, f)) = o'$
$\wedge\ \text{align}(\overline{t}, N) = [a, o]\ \wedge\ \text{subalign}([a, o + o'], \text{align}(\overline{t}, \tau))$

These constraints are necessary for portable code; without them certain cast-free programs could get stuck. The C standard allows other assumptions that we can write in our logic but that our safety theorem need not assume. For example:

- long is at least as big as short: $\forall \overline{t}.\ \text{xtype}(\overline{t}, \text{long}) \geq \text{xtype}(\overline{t}, \text{short})$
- First fields always have offset 0: $\forall f, \overline{t}, \tau.\ (N\{\tau f \dots\} \in \overline{t}) \Rightarrow \text{offset}(\overline{t}, f) = 0$

3.5 Physical Subtyping

$$\textsc{ptr}\ \frac{\vdash \alpha_1 \leq \alpha_2}{D \vdash \text{ptr}_{\alpha_1}(\overline{\sigma}_1 \overline{\sigma}_2) \leq \text{ptr}_{\alpha_2}(\overline{\sigma}_1)} \qquad \textsc{pad}\ \frac{\text{size}(\mathit{impl}, \overline{\sigma}) = i}{\mathit{impl}; \overline{t} \vdash \overline{\sigma} \leq \text{pad}[i]} \qquad \textsc{seq}\ \frac{D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2 \quad D \vdash \overline{\sigma}_3 \leq \overline{\sigma}_4}{D \vdash \overline{\sigma}_1 \overline{\sigma}_3 \leq \overline{\sigma}_2 \overline{\sigma}_4}$$

$$\textsc{refl}\ \frac{}{D \vdash \overline{\sigma} \leq \overline{\sigma}} \qquad \textsc{trans}\ \frac{D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2 \qquad D \vdash \overline{\sigma}_2 \leq \overline{\sigma}_3}{D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_3}$$

$$\textsc{unroll}\ \frac{\mathit{impl}.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}}{\mathit{impl}; \overline{t} \vdash \text{ptr}_\alpha(N) \leq \text{ptr}_\alpha(\overline{\sigma})} \qquad \textsc{roll}\ \frac{\mathit{impl}.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}}{\mathit{impl}; \overline{t} \vdash \text{ptr}_\alpha(\overline{\sigma}) \leq \text{ptr}_\alpha(N)}$$

$$\textsc{align-base}\ \frac{a_1 = a_2 \times i}{\vdash [a_1, o] \leq [a_2, o]} \qquad \textsc{align-offset}\ \frac{o_1 \equiv o_2 \mod a}{\vdash [a, o_1] \leq [a, o_2]} \qquad \textsc{align-trans}\ \frac{\vdash \alpha_1 \leq \alpha_2 \qquad \vdash \alpha_2 \leq \alpha_3}{\vdash \alpha_1 \leq \alpha_3}$$

Fig. 6. Physical Subtyping (and Subtyping on Alignments)

We use subtyping on low-level types to formalize that data described by $\overline{\sigma}$ can be treated as a $\overline{\sigma}'$. This has been called *physical subtyping* [5, 27, 22] because it uses actual memory layouts. Figure 6 defines $D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ (recall $D ::= \mathit{impl}; \overline{t}$).

As expected in an imperative language, pointer types have invariant subtyping (rule PTR). However, we allow forgetting fields under a pointer type as this only restricts access to a prefix of the data. This encodes the core concept behind casts like Example 1 in Section 2. We also allow assuming a less restrictive alignment (via $\vdash \alpha_1 \leq \alpha_2$), which also only restricts how a pointer can be used.

We allow sequence-shortening under pointer types, but it is *not* correct to allow shortening as a subtyping rule. A supertype should have the same size as a subtype, which may seem odd to readers not used to subtyping in a language with explicit pointers. It is why C correctly disallows casts between struct types (as opposed to pointers to structs).

Rule PAD lets us forget about the form of data (not under a pointer) without forgetting its size. Rule SEQ lifts subtyping to sequences. As usual, subtyping is reflexive and transitive. Rules UNROLL and ROLL witness the equivalence between

a struct name and its definition. Recall we restrict a type N to occur under pointers, which is sufficient for translating recursive types.

As usual, subsumption (explicit or implicit) is sound for right-expressions but unsound for left-expressions. (For example, in Java, given `e1=e2`, one may use subsumption on `e2` but not on `e1`.) Therefore, casts are not left-expressions.

3.6 Static Semantics and Constraint Generation

$$\text{S-Var}\quad \frac{\Gamma(x) = \tau}{\overline{t}; \Gamma \vdash_{\mathsf{r}} x : \tau; \text{true}} \qquad \text{S-Assn}\quad \frac{\overline{t}; \Gamma \vdash_{\mathsf{l}} e_1 : \tau; S_1 \qquad \overline{t}; \Gamma \vdash_{\mathsf{r}} e_2 : \tau; S_2}{\overline{t}; \Gamma \vdash_{\mathsf{r}} e_1 = e_2 : \tau; S_1 \wedge S_2}$$

$$\text{S-Cast}\quad \frac{\overline{t}; \Gamma \vdash_{\mathsf{r}} e : \tau_1*; S_1}{\overline{t}; \Gamma \vdash_{\mathsf{r}} (\tau*)e : \tau*; S_1 \wedge \text{subtype}(\overline{t}, \text{xtype}(\overline{t}, \tau_1*), \text{xtype}(\overline{t}, \tau*))}$$

S-Faddr

$$\frac{\overline{t}; \Gamma \vdash_{\mathsf{r}} e : N*; S_1 \qquad N\{\ldots \tau_1\ f \ldots\} \in \overline{t}}{\begin{array}{l}\overline{t}; \Gamma \vdash_{\mathsf{r}} (\tau*)(\&e \to f) : \tau*; S_1 \wedge \exists \overline{\sigma}_1, \overline{\sigma}_2, a, o.\ \text{xtype}(\overline{t}, N*) = \text{ptr}_{[a,o]}(\overline{\sigma}_1 \overline{\sigma}_2) \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \text{offset}(f) = \text{size}(\overline{\sigma}_1) \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \text{subtype}(\text{ptr}_{[a,o+\text{offset}(f)]}(\overline{\sigma}_2), \text{xtype}(\overline{t}, \tau*))\end{array}}$$

Fig. 7. Static Semantics (letting $\Gamma ::= \cdot \mid \Gamma, x{:}\tau$ and omitting most of the rules)

The preceding definitions of constraints and subtyping provide what we need to define a static semantics for source programs (Figure 7). The judgments $\overline{t}; \Gamma \vdash_{\mathsf{r}} e : \tau; S$ and $\overline{t}; \Gamma \vdash_{\mathsf{l}} e : \tau; S$ (for right- and left-expressions respectively) produce types as usual, but also formulas S. This formula is just a conjunction of the semi-portable assumptions the program may be making.

The only interesting rules are S-Cast and S-Faddr because the "sensible" constraints in Section 3.4 suffice to ensure other expression forms (such as dereferences and assignments) cannot fail due to an implementation dependency. Due to space constraints, we show only two examples of other typing rules; see [23] for the complete type system. The constraints directly describe the implicit assumptions made in Examples 1–3 in Section 2. The S-Faddr constraint is much more complicated because we do not generally require every subsequence of fields to have an alignment appropriate for treating it as a type.

This type system does not support downcasts, which are important in practice. To support them safely, we could invert the direction of the subtyping constraint in S-Cast and employ existing techniques to ensure the casted value actually has the result of the cast. Techniques used in existing safe-C approaches [22, 15] include type tags, discriminated unions, and parametric polymorphism.

3.7 Metatheory

Safety: Ideally, we would claim that running a well-typed program on a "sensible" implementation that models the program's constraint would never get stuck. That is, given $\overline{t}; \cdot \vdash_{\mathsf{r}} e : \tau; S$, $\mathit{impl} \models S$ and the "sensible" constraints, and

$impl; \overline{t} \vdash \cdot; e \rightarrow^* H; e'$ (where $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$), either e' is a value or there exists H', e'' such that $impl; \overline{t} \vdash H; e' \rightarrow^* H'; e''$.

However, this claim is false due to uninit. We must relax our claim to admit e' might also be "legally stuck," which we define as expressions of the form $\mathsf{R}[stuck]_\mathsf{r}$ or $\mathsf{R}[stuck]_\mathsf{l}$ where:

$$stuck ::= \mathsf{if}\ (\overline{w}_1\ \mathsf{uninit}\ \overline{w}_2)\ e\ e \mid *(\tau*)(\mathsf{uninit}^i) \mid (\tau*)\&\mathsf{uninit}^i \rightarrow f$$

The proof [23] uses a "low-level, run-time type system" to capture relevant invariants that evaluation preserves. The main judgment is $impl; \overline{t}; \Psi; \Gamma \vdash_\mathsf{r} e : \overline{\sigma}$ where Ψ describes the heap ($\Psi ::= \cdot \mid \Psi, \ell \mapsto \overline{\sigma}, \alpha$). This type system has implicit subsumption and, like the dynamic semantics, many rules refer to *impl*.

This key lemma connects the static semantics and the low-level type system:

> If $\overline{t}; \Gamma \vdash_\mathsf{r} e : \tau; S$, $impl \models S$ and $impl$ is sensible, and $impl.xtype(\overline{t}, \tau) = \overline{\sigma}$, then $impl; \overline{t}; \cdot; \Gamma \vdash_\mathsf{r} e : \overline{\sigma}$.

Given this lemma, preservation and progress (modulo legal stuck states) for the dynamic semantics and the low-level type system suffices to show type safety.

Cast-Free Portability: Having safety rely on a portability constraint (the S in $\overline{t}; \cdot \vdash_\mathsf{r} e : \tau; S$) can be viewed as weak, since S could be difficult to establish, even unsatisfiable. However, we can formalize the intuitive notion that only casts can threaten portability. That is, for the right definition of "cast-free," if e is cast-free and $\overline{t}; \Gamma \vdash_\mathsf{r} e : \tau; S$, then every sensible *impl* models S. It then follows from safety that e is portable (i.e., it will not get stuck on any sensible implementation). This intuition is supported by a formal statement and proof [23].

To be precise, a program $\overline{t}; e$ is cast-free if (1) All expressions of the form $(\tau*)e'$ actually have the form $*((\tau*)e'')$ or $(\tau*)\&e'' \rightarrow f$, and (2) For every expression of the form $(\tau*)\&e' \rightarrow f$, the type τ is the type of f (i.e., $N\{\ldots \tau\ f \ldots\} \in \overline{t}$).

4 Extensions

The model in the previous section is purposely tiny, but we have extended it in important directions [23]. Space constraints allow only sketching the basic ideas.

The most important addition is arrays because Example 4 showed that they must restrict subtyping. As is common in safe C-like languages [22, 15], we use a separate type $\tau*^\omega$ (and its low-level counterpart $\mathrm{ptr}^\omega_\alpha(\overline{\sigma})$) for pointers usable as arrays so that non-array pointers ($\tau*$) have the subtyping already described. Given an array-pointer x, the expression $\&x[e]$ returns a non-array pointer to one element, assuming e is in bounds (which we do not enforce statically). The key physical subtyping rule then allows array-pointer subtyping only if one element type is i adjacent copies of the other (cf. the more lenient PTR rule):

$$\frac{\overline{\sigma}_1 = \overline{\sigma}_2{}^i \qquad \vdash \alpha_1 \leq \alpha_2}{impl; \overline{t} \vdash \mathrm{ptr}^\omega_{\alpha_1}(\overline{\sigma}_1) \leq \mathrm{ptr}^\omega_{\alpha_2}(\overline{\sigma}_2)}$$

However, this rule is sound only given a new sensibility constraint that is unnecessary without arrays, that a type's size is a multiple of its alignment:

$$\forall \tau, \bar{t}. \exists i, a, o.\ \mathrm{size}(\bar{t}, \mathrm{xtype}(\bar{t}, \tau)) = i \times a \ \wedge\ \mathrm{align}(\bar{t}, \tau) = [a, o]$$

Other extensions to our model can allow more subtyping. Read-only types (like C's `const` but enforced) can allow deep subtyping. A proper treatment of recursive subtyping [3] is appropriate and synergistic with `const`.

5 Previous Work

To our knowledge, previous work considering implementation-dependent data-layout or low-level type-safety has done one of (1) assume a particular implementation, (2) assume the source program is fully portable, or (3) ignore features such as structs or dynamic memory allocation. The last approach relegates issues to our work, much as we relegate issues like array-bounds errors to others [8, 10].

5.1 Assuming an Implementation

Most closely related is the "physical type-checking" work of Chandra et al. [5, 27], which motivated our work considerably. Their tool classifies C casts as "upcasts", "downcasts", or "neither", reporting a warning for the last possibility. They take a byte-for-byte view of memory for a low-level type system, but they neither parameterize their system by an implementation nor produce descriptions of sets of implementations. Checking code for a new implementation would require reverification and changing their tool. They do not present a soundness theorem.

CCured [22], a memory-safe C implementation, uses physical type-checking to eliminate casts that otherwise need run-time checks. That is, CCured permits casts that work in practice (assuming a padding strategy typical when targeting the x86), but that are not allowed by the C standard or other architectures. The formal model establishing CCured's soundness has some similarities with our work, but it lacks a distinct notion of *implementation*, alignment constraints, memory allocation, recursive types, etc.

Work on typed assembly language and proof-carrying code [20, 19, 6, 13] also takes a low-level view of memory. In particular, work on allocation semantics [25, 1] exposes that addresses are integers, allowing pointer arithmetic to cross object boundaries. Type-checking binary code might be indirectly useful for semi-portable code, but it requires an assembly-level verifier for each architecture.

C-- [26] makes data representation and alignment explicit, but C-- is not a platform for writing semi-portable code. Rather, it is a low-level language designed as a target for compiling high-level languages. All padding decisions and alignment assumptions are explicit.[4] Incorrect alignment is an unchecked run-time error. C-- is intended to handle back-end code-generation issues; it is expected that the front-end compiler will generate different (but similar) code for each platform and provide a run-time system written in C.

[4] Syntactically, an omitted alignment is taken to be n for an n-byte access.

5.2 Safe C

Memory-safe dialects or implementations of C, such as Cyclone [15, 12, 11], CCured [22, 21, 7], and SAFECode [9], do not solve the semi-portability problem. Rather, they may reject (at compile-time) or terminate (at run-time) programs attempting implementation-dependent operations, or they may support only certain implementations (e.g., certain C compilers as back-ends). Furthermore, these systems all include run-time systems (e.g., memory managers) that are themselves semi-portable! For example, the Cyclone run-time system assumes 32-bit integers and pointers, and support for 64-bit platforms is a top request from users.

5.3 Formalizing C

Recent work by Leroy et al. [16, 4] uses Coq to prove a C compiler correct. Their (large-step) operational semantics for C distinguishes left and right expressions much as we do. However, the language omits structs (avoiding many alignment and padding issues), and the metatheory proves correctness only for correct source programs (hence saying nothing about implementation-dependent code).

Norrish's formalization of C in HOL includes structs [24]. Like our work, a global namespace maps struct names to sequences of typed fields. However, he purposely omits padding and alignment from the model. Without a separable notion of implementation, he models implementation choices as non-determinism.

6 Conclusions

We have developed a sound, formal description of implementation-dependencies in low-level software. The key insight is a semantic definition of "implementation" that directs a low-level operational semantics *and* models a syntactic constraint that is produced via static analysis on a source program. Giving implementations a clear identity and identifying "sensibility constraints" clarifies a number of poorly understood issues. We believe we are the first to consider describing a *set of implementations* on which a low-level program can run safely.

Our next step is to complete a practical tool to determine implementation assumptions of real C code, which may require a more precise static analysis while still using our basic approach. Another area for future work is a stronger guarantee, such as establishing program equivalence on a set of implementations.

References

1. A. Ahmed and D. Walker. The logical approach to stack typing. In *ACM International Workshop on Types in Language Design and Implementation*, 2003.
2. *The ARMLinux Book Online*. 2005. http://www.aleph1.co.uk/armlinux/book.
3. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. on Programming Languages and Systems*, 15(4), Sept. 1993.
4. S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *14th International Symposium on Formal Methods*, Aug. 2006.

5. S. Chandra and T. Reps. Physical type checking for C. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, 1999.
6. J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back-end optimization. In *ACM PLDI*, 2003.
7. J. Condit, M. Harren, S. McPeak, G. Necula, and W. Weimer. CCured in the real world. In *ACM PLDI*, 2003.
8. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *European Symposium on Programming*, 2005.
9. D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM PLDI*, 2006.
10. N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *ACM PLDI*, 2003.
11. D. Grossman. Quantified types in imperative languages. *ACM Trans. on Programming Languages and Systems*, 28(3), 2006.
12. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM PLDI*, 2002.
13. N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. *Journal of Automated Reasoning*, 31(3–4), 2003.
14. *ISO/IEC 9899:1999, International Standard—Programming Languages—C.* International Standards Organization, 1999.
15. T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.
16. X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *ACM POPL*, 2006.
17. R. Love. *Linux Kernel Development, 2nd Edition.* Novell Press, 2005. Page 328.
18. B. Martin, A. Rettinger, and J. Singh. Multiplatform porting to 64 bits. *Dr. Dobb's Journal*, Dec. 2005. http://www.ddj.com/184406427.
19. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3), 1999.
20. G. Necula. Proof-carrying code. In *ACM POPL*, 1997.
21. G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM POPL*, 2002.
22. G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. on Programming Languages and Systems*, 27(3), 2005.
23. M. Nita, D. Grossman, and C. Chambers. A theory of implementation-dependent low-level software. Technical Report 2006-10-01, Univ. of Wash. Dept. of Computer Science & Engineering, Oct. 2006. Available at http://www.cs.washington.edu/homes/marius/papers/tid/.
24. M. Norrish. *C formalised in HOL.* PhD thesis, University of Cambridge, 1998.
25. L. Petersen, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout. In *ACM POPL*, 2003.
26. N. Ramsey, S. P. Jones, and C. Lindig. The C-- language specification version 2.0, Feb. 2005. http://www.cminusminus.org/extern/man2.pdf.
27. M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *7th European Software Engineering Conference and 7th ACM Symposium on the Foundations of Software Engineering*, 1999.