# A Theory of Platform-Dependent Low-Level Software

Marius Nita Dan Grossman Craig Chambers

Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350
{marius, djg, chambers}@cs.washington.edu

## Abstract

The C language definition leaves the sizes and layouts of types partially unspecified. When a C program makes assumptions about type layout, its semantics is defined only on platforms (C compilers and the underlying hardware) on which those assumptions hold. Previous work on formalizing C-like languages has ignored this issue, either by assuming that programs do not make such assumptions or by assuming that all valid programs target only one platform. In the latter case, the platform's choices are hard-wired in the language semantics.

In this paper, we present a practically-motivated model for a C-like language in which the memory layouts of types are left largely unspecified. The dynamic semantics is parameterized by a platform's layout policy and makes manifest the consequence of platform-dependent (i.e., unspecified) steps. A type-and-effect system produces a *layout constraint*: a logic formula encoding layout conditions under which the program is memory-safe. We prove that if a program type-checks, it is memory-safe on all platforms satisfying its constraint.

Based on our theory, we have implemented a tool that discovers unportable layout assumptions in C programs. Our approach should generalize to other kinds of platform-dependent assumptions.

***Categories and Subject Descriptors*** F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.4 [*Software Engineering*]: Software/Program Verification

***General Terms*** Languages, Verification, Theory

## 1. Introduction

In recent years, research has demonstrated many ways to improve the quality of low-level software (typically written in C) by using programming-language and program-analysis technology. Such work has detected safety violations (array-bounds errors, dangling-pointer dereferences, uninitialized memory, etc.), enforced temporal protocols, and provided new languages and compilers that support reliable systems programming. The results are an important and practical success for programming-language theory. However, there remains a crucial and complementary set of complications that this paper begins to address:



*POPL'08* January 7–12, 2008, San Francisco, California, USA.


*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 laid out in memory (including padding), the size of values, and alignment restrictions on memory accesses. To our knowledge, existing work on safe low-level code (see Section 6) either (1) checks or simply assumes full layout portability (e.g., that the program is unaffected by structure padding) or (2) checks the program assuming a particular C platform, making no guarantee for other platforms.

Requiring that code makes no platform-dependent assumptions — e.g., by enforcing poorly understood and informally specified (C Standard 1999) restrictions on C programs — is too strict because low-level code often has inherently non-portable parts. An impractical solution is to rewrite large legacy applications in fully portable languages or to use libraries that abstract all platform dependencies. Such approaches ignore legacy issues, can be a poor match for low-level code, and assume that language or library implementations are available for an ever-increasing number of platforms.

Conversely, allowing implicit platform-dependent assumptions can lead to pernicious defects that lie dormant until one uses a platform violating the assumptions. Whereas defects like dangling-pointer dereferences are largely independent of the language implementation, so testing or verification on the "old platform" can find many of them, defects like assuming two `struct` types have similar data layouts are not. The results can be severe. Conceptually simple tasks like porting an application from a 32-bit machine 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 relies on undocumented behavior. Section 2.2 discusses some specific real-world examples.

In practice, C programmers identify and isolate platform dependencies manually. They may retest on each platform or use ad hoc tool support, such as compiler flags and lint-like technology, relying on informal knowledge of how target platforms lay out data. In contrast, the work presented in this paper informs the development of a principled, fully automatic tool that discovers particular unportable assumptions in C code.

More generally, we develop a semantics for low-level software that has an explicit and separable notion of a platform. Without such a notion, formal models or language-based tools for C face the same dilemma as the C code they are designed to help: either they apply only to fully portable code or they assume C implementation details that do not hold on all platforms. With our approach, one can "plug in" a platform description into a generic framework for the operational semantics. We advocate our general approach for making any work on C semantics (or semantics for any language with some unspecified behavior) robust to platform dependencies.

Moreover, we use logic formulas to describe a program's platform-dependent assumptions. Such descriptions give a formal definition to the idea of *semi-portability* — a piece of software or an analysis may be correct given some assumptions, i.e., portable to those platforms on which the assumptions hold. Software and analyses could use these formulas as documentation for their assumptions. In our work, we extract them automatically but conservatively from C code.

### 1.1 Overview of Our Approach

The key to our formal model is isolating the notion of *platform*, which we can think of as an oracle that answers queries about how types are laid out. In our model, a platform has two roles: (1) as a parameter to the operational semantics, and (2) as something we can describe with a *layout constraint*. The key insight is this: Given a program $P$, we can extract a layout constraint $S$ from $P$ and show that $P$ is memory-safe on all platforms satisfying $S$.

To see how platforms work as parameters to the operational semantics, suppose we have a pointer dereference `*e`. The number of bytes accessed depends on the size of the type of `e`, and this size is determined by the platform. Therefore, our dynamic semantics has the form $\Pi \vdash P \to P'$ where $\Pi$ is a platform and $P$ is a program state. That way, the dereference rule can use $\Pi$ to guide the memory access and become stuck if $\Pi$ deems the access misaligned.

As for layout constraints, they are formulas in a first-order theory in which platforms are models. For example, the constraint "$\text{access}(4, 8) \wedge \text{size}(\text{long}) = 8$" is modeled by any platform in which values of type long occupy 8 bytes and 8-byte loads of 4-byte aligned data are allowed.[1] Per convention, we write $\Pi \models S$ when platform $\Pi$ models constraint $S$.

Now given a program $P$ we can try to find a constraint $S$ such that if $\Pi \models S$, then the abstract machine does not get stuck when running $P$ given $\Pi$. For our operational semantics, that means it will not treat an integer as a pointer, read beyond the end of a `struct` value, perform an improperly aligned memory access, etc. Finding an $S$ that describes exactly the set of platforms on which the program does not get stuck is trivially undecidable, so a sound approximation is warranted. In our theory, we take a very conservative approach: A type system for source programs produces $S$ using no flow-sensitivity or alias information. Our tool uses points-to information, but the general setup remains the same.

The key metatheoretic result is showing that the $S$ our system produces is indeed sound, i.e., its only models are platforms on which the program does not get stuck. For the proof, we define a second type system for program states. This second type system, which exists only to show safety, is parameterized by a $\Pi$ like the dynamic semantics. Our type-safety argument then has two parts:

1. The second type system and operational semantics enjoy the conventional preservation and progress properties.
2. If the first type system produces $S$ given $P$, then $P$ type-checks in the second type system for any $\Pi$ such that $\Pi \models S$.

### 1.2 Contributions and Caveats

To our knowledge, this work is the first to consider describing a *set of platforms* on which a low-level program can run safely. At a more detailed level, our development clarifies several points:

- We can define a sound type system for a language with partially unspecified type layouts. The soundness theorem is proved once and for all instead of once for each platform.
- Layout-portability questions are reduced to pointer-cast questions, namely, "when can a pointer to a $\tau_1$ be treated as a pointer to a $\tau_2$?" This question, clearly akin to subtyping, depends on the platform.
- Platform constraints can be described in a first-order theory and extracted statically from the program.
- There should be a notion of "sensible" platform, meaning platforms on which every cast-free program cannot get stuck.

For tractability, the formal model considers only a small expression language inspired by C. It has many relevant features, including structs, heap allocation, and taking the address of fields, but we omit some relevant features (e.g., bit-fields), and many irrelevant ones (e.g., functions and `goto`). We also make some simplifying assumptions in our definition of platform. In particular, we assume all pointers have the same size and that alignment restrictions depend on the size, but not the type, of data. We see no fundamental problems extending our model in these directions.

The formal model directly informs the implementation of an automatic tool we wrote to detect layout-portability problems in C programs. In addition to producing a constraint describing the platforms on which the program is portable, the tool checks the constraint against a set of platforms of interest and outputs informative warnings when the constraint is violated. No access to the platforms of interest is needed.

### 1.3 Outline

Section 2 presents examples of platform-dependent code and the constraints describing their layout assumptions. Section 3 presents our core formal model, including the definition of platforms, our first-order theory, the dynamic and static semantics of our language, and our soundness theorem. Section 4 describes important extensions to the base model. Section 5 describes our tool. The last two sections discuss related work and conclude.

## 2. Examples

Section 2.1 presents several tiny examples of C code to explain issues of layout portability and relevant platform constraints. Section 2.2 complements this "tutorial" with actual platforms, systems, and coping strategies related to these concepts.

### 2.1 Small Code Fragments

***Example 1: Accessing Memory***

```
(*e).f
```

A memory access such as `(*e).f` reads or writes $s$ bytes at some alignment $a$. If `e` has type `struct T*` and the `f` field has type $\tau$, then $s$ is the *size* of $\tau$ and $a$ is the greatest common divisor of the *alignment* of `struct T` and the *offset* of `f`.

Platforms 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 alignments, a compiler might put pad bytes before `f` fields or break 8-byte accesses into two 4-byte accesses. In the latter case, the compiler "supports" 8-byte accesses on 4-byte alignments.

In this paper, we assume platforms include an *access* function of type $\text{Int} \to \text{Int} \to \text{Bool}$, as well as *size* and *alignment* functions that map types to integers. Our example `(*e).f` therefore induces the constraint $\text{access}(a, s)$ where $a$ and $s$ are defined above. However, this constraint assumes `e` actually evaluates to a pointer with alignment $a$ and a $\tau$ at the right offset. The constraints for cast expressions must ensure this.

***Example 2: Prefix***

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

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

A cast from `struct S1*` to `struct D1*` is safe if `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. Our formal model captures this requirement precisely.

***Example 3: Flattening and Alignment***

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

A cast from `struct S2*` to `struct D2*` has similar constraints as in Example 2. However, in this case, some platforms may put pad bytes before `f4` because of alignment restrictions. Our system will generate constraints preventing such a representation mismatch if the program has this cast.

***Example 4: Suffix***

```
struct S3 {   struct D3 {
 int* f1;      int* g1;
 int* f2;      double g2;
 double f3;   };
};
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, a platform with 4-byte pointers, 8-byte doubles, and 8-byte alignment of doubles cannot support this cast because `struct D3` has more padding. Platforms without padding can allow this cast, even though `&x->f2` has type `int**` in C.

***Example 5: Arrays***

```
struct S4 { int f1; short f2; };
struct D4 { int g1; };

struct S4 * x = ...;
struct D4   y = ((struct D4 *)x)[7];
```

Previous examples implicitly assumed the destination pointer was not used as an array (i.e., it was used as a pointer to exactly one struct). On many platforms that add trailing padding after field `f2`, a cast from `struct S4*` to `struct D4*` is legal. However, the cast above is broken due to intermittent pad bytes in the layout pointed to by `x`. This issue is orthogonal to array-bounds violations; we must reject the cast even if `x` points to more than 7 elements. Section 4.1 extends our model to distinguish pointers to single-objects from pointers to arrays.

### 2.2 Practical Scenarios

The scope of the platform dependency problem is not precisely known because defects can lie dormant until one changes hardware or compiler. Therefore, like date bugs (such as the famous Y2K problem), offending code can be difficult to locate and fix.

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

> 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.[2]

They also note that defects sometimes lead to alignment traps, but sometimes lead to silent data corruption. Kernel developers are basically told to, "be careful" (Love 2005).

Ports to 64-bit platforms provide another evidence source. Some vendors do little more than suggest using lint-like technology, such as gcc's `-Wpadded` flag for reporting when a struct type has padding (IBM 2004). However, others find that aggressive warning levels produce so much information for legacy code that they recommend using multiple independent compilers and looking only at lines for which they all produce warnings (Martin et al. 2005).

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

**Figure 1.** Source-Language Syntax: A program has the form $\bar{t}; e$

## 3. Core Language

This section develops a formal model that can explain examples 1–4 from Section 2. We define an appropriate core language, a definition of *platform*, a dynamic semantics, a first-order theory that constrains platforms, a type-and-effect system to produce constraints, and the type-soundness result, acquired via a lower-level, platform-dependent type system. Figure 8 on page 8 summarizes the model's judgments.

### 3.1 Idealized Syntax

For source programs, we consider a small subset of C with some convenient syntactic changes, as defined in Figure 1. Most significantly, we omit functions and make all terms expressions. A program is a sequence of struct definitions ($\bar{t}$) and an expression ($e$) to be evaluated. (We consistently write $\overline{x}$ for a sequence of elements from syntactic category $x$ and $\cdot$ for the empty sequence. We also write $x^i$ for a length $i$ sequence.) Type definitions have global scope, allowing mutually recursive types.

Types $\tau$ include short and long (for two sizes of data), pointers ($\tau*$), and struct types ($N$ rather than the more verbose `struct` $N$). As in C, all pointers (levels of indirection) 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 $l$; we leave their exact form unspecified), 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. (Our particular choice happens to correspond to C's syntax.) Dereference in C *is* type-directed (if $e$ has type $\tau*$, then $*e$ reads `sizeof(`$\tau$`)` bytes); our type decoration makes this explicit. The cast in address-of-field expressions helps us support "suffix casts" as in Example 4 of Section 2.

The remaining expression forms are for memory allocation or control flow. `new` $\tau$ heap-allocates uninitialized space to hold a $\tau$; it is less verbose than `malloc(sizeof(`$\tau$`))`. $(e; e)$ is sequence. $(\mathsf{if}\ e\ e\ e)$ is a conditional, branching on whether the first subexpression is 0. To avoid distinguishing statements from expressions,

[2] Emphasis in original.

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

**Figure 2.** Syntax extensions for run-time behavior

a while-loop evaluates to a number if it terminates. Finally, $(\tau\ x;\ e)$ creates a local variable $x$ of type $\tau$ bound in $e$.

As defined in Section 3.3, program evaluation depends on a platform $\Pi$ and modifies a heap $H$. Because $\overline{t}$ does not change during evaluation, we write $\Pi; \overline{t} \vdash H; e \to H'; e'$ for one evaluation step. Rather than define a *translation* (i.e., a compiler) from $e$ to a lower-level platform-dependent language, we *extend* $e$ with new lower-level forms. This equivalent approach of consulting the platform lazily (at run-time) simplifies the metatheory while fully exposing the intricacies of platform dependencies.

Figure 2 defines the syntactic extensions for run-time expressions and heaps. A value $v$ is a sequence of atomic values $w$, which can be initialized bytes $b$, uninitialized bytes $\mathsf{uninit}$, or pointers $\ell{+}i$. A pointer is a label $\ell$ and an offset $i$ because the heap maps labels to value sequences, so a pointer into the middle of a value has a non-zero offset. This heap model is higher level than assembly language but low enough for middle pointers, suffix casts, etc. In other words, it is ideal for modeling layout dependencies.

Heaps also map labels to alignments. We model alignments as pairs $[a, o]$ where $o$ is an offset from alignment $a$. Typically $o$ is 0, e.g., $[4, 0]$ describes 4-byte aligned pointers. Supporting offsets adds some precision, e.g., if we add 2 bytes to a $[4, 0]$ pointer we get $[4, 2]$ and adding 2 more bytes gives $[4, 4]$, which is isomorphic to $[4, 0]$. Without offsets, adding 2 bytes to a 4-byte aligned pointer would produce a 2-byte aligned pointer. Section 3.5 describes a subalignment relation precisely.

### 3.2 Platforms

Before we show the semantics of our language, we need to introduces a precise notion of platform, as platforms play central roles in both the dynamic and static semantics. A platform ($\Pi$) is a record of functions with the following components, summarized in Figure 3:

- A translation of types into a lower-level representation ($\overline{\sigma}$), described below. We write $\Pi.\mathit{xtype}(\overline{t}, \tau)$ for the $\overline{\sigma}$ corresponding to the translation of a type $\tau$ assuming type definitions $\overline{t}$.
- An *alignment* function ($\Pi.\mathit{align}$) returns the alignment $\alpha$ used to allocate space for a $\tau$.
- An *offset* function ($\Pi.\mathit{offset}$) 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 $\alpha$ and a size $i$ and returns true if accessing $i$ bytes at an alignment $\alpha$ is not an error.
- The size of pointers ($\Pi.\mathit{ptrsize}$) is a constant $i$.[3]
- An *xliteral* function translates integer literals into byte sequences. The layout of values in memory is platform-dependent.

The *access* function is typically associated with hardware and the other components with compilers, but a platform comprises all components. It is clear a "sensible" platform cannot define its components in isolation (e.g., the type translation must mind the access function); our constraint language will let us define these restrictions precisely.

[3] This is a slight simplification since a C implementation could use different sizes for different pointers.

$$
\begin{array}{l}
\sigma \ ::= \ \mathrm{byte} \mid \mathrm{pad}[i] \mid \mathrm{ptr}_\alpha(\overline{\sigma}) \mid \mathrm{ptr}_\alpha(N) \\
\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma} \\
\Pi.\mathit{align}(\overline{t}, \tau) = \alpha \\
\Pi.\mathit{offset}(\overline{t}, f) = i \\
\Pi.\mathit{access}(\alpha, i) = \{\mathrm{true}, \mathrm{false}\} \\
\Pi.\mathit{ptrsize} = i \\
\Pi.\mathit{xliteral}(s) = \overline{b} \\
\Pi.\mathit{xliteral}(l) = \overline{b}
\end{array}
$$

**Figure 3.** Platforms and low-level types

Low-level types (the target of $\Pi.\mathit{xtype}$) are $\overline{\sigma}$, a sequence of $\sigma$. For example, if long is four bytes, the translation is $\mathrm{byte}^4$. The type $\mathrm{pad}[i]$ represents $i$ bytes of padding. The type $\mathrm{ptr}_\alpha(\overline{\sigma})$ describes pointers to data described by $\overline{\sigma}$ at alignment $\alpha$. As a technical point, we disallow the type $N$ for low-level types except for the form $\mathrm{ptr}_\alpha(N)$. This restriction simplifies type equalities without restricting platforms or disallowing recursive types.

### 3.3 Dynamic Semantics

The dynamic semantics is a small-step rewrite system for expressions, parameterized by a platform and a sequence of type declarations. Figure 4 holds the full definition for $\Pi; \overline{t} \vdash H; e \to H'; e'$. It is defined via evaluation contexts for conciseness. As in C, the left side of assignments (called left-expressions) are evaluated differently from other expressions (called right-expressions). Therefore, we have two sorts of contexts ($\mathsf{L}$ and $\mathsf{R}$) defined by mutual induction and a different sort of primitive reduction ($\xrightarrow{\mathsf{l}}$ and $\xrightarrow{\mathsf{r}}$) for each sort of context (Grossman 2003). In particular, $\mathsf{R}[e]_\mathsf{r}$ is a right-context $\mathsf{R}$ containing a right-hole filled by $e$ and $\mathsf{R}[e]_\mathsf{l}$ is a right-context $\mathsf{R}$ containing a left-hole filled by $e$. Each context contains exactly one right-hole or exactly one left-hole, but not both.

Most primitive reductions depend on $\Pi$, but let us first dispense with those that do not. D-CAST shows that casts have no run-time effect. D-SEQ is typical. D-IFF and D-IFT are typical except we treat 0 as false (as in C) and other byte-sequences as true. D-WHILE is a typical small-step unrolling; we make the arbitrary choice that a terminating loop produces some $s$ literal nondeterministically.

D-NEW extends the heap with a new label holding uninitialized data. The platform 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 $\ell{+}0$ is a pointer to the beginning of the space. The type system does not prevent getting stuck due to uninitialized data; this issue is orthogonal. D-LET has the same hypotheses as D-NEW. Because memory management is not our concern, we use heap allocation even for local variables. We substitute $*(\tau*)(\ell{+}0)$ for $x$ in the resulting expression.

D-DEREF reads data from the heap and the resulting expression is the data. In particular, it extracts a sequence "from the middle" of $H(\ell)$. This sequence is from offset $j$ (where the expression before the step is $*(\tau*)(\ell + j)$) to $j + k$ (where $k$ is the size of the translation of $\tau$). If it is not possible to "carve up" $H(\ell)$ in this way, then the rule does not apply and the machine is stuck. As expected, we also use $\Pi.\mathit{access}$ to model alignment constraints on the memory access.

D-ASSIGN has exactly the same hypotheses as 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 $H(\ell)$.

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

$$D \quad ::= \quad \Pi; \overline{t}$$

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

D-CAST
$$\frac{}{D \vdash H; (\tau*)\overline{w} \xrightarrow{\mathsf{r}} H; \overline{w}}$$

D-SEQ
$$\frac{}{D \vdash H; (v; e) \xrightarrow{\mathsf{r}} H; e}$$

D-WHILE
$$\frac{}{D \vdash H; \mathsf{while}\ e_1\ e_2 \xrightarrow{\mathsf{r}} H; \mathsf{if}\ e_1\ (e_2; \mathsf{while}\ e_1\ e_2)\ s}$$

D-IFF
$$\frac{}{D \vdash H; \mathsf{if}\ 0^i\ e_1\ e_2 \xrightarrow{\mathsf{r}} H; e_2}$$

D-IFT
$$\frac{b_1 \ldots b_i \neq 0^i}{D \vdash H; \mathsf{if}\ (b_1 \ldots b_i)\ e_1\ e_2 \xrightarrow{\mathsf{r}} H; e_1}$$

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

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

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

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

D-FADDR
$$\frac{\Pi.\mathit{offset}(\overline{t}, f) = j'}{\Pi; \overline{t} \vdash H; (\tau*)\&(\ell{+}j){\to}f \xrightarrow{\mathsf{r}} H; \ell{+}(j + j')}$$

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

D-FETCH
$$\frac{N\{\ldots \tau\ f \ldots\} \in \overline{t} \qquad \Pi.\mathit{offset}(\overline{t}, f) = \mathrm{size}(\Pi, \overline{w}_1) \\ \Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma} \qquad \mathrm{size}(\Pi, \overline{\sigma}) = \mathrm{size}(\Pi, \overline{w}_2)}{\Pi; \overline{t} \vdash H; \overline{w}_1\overline{w}_2\overline{w}_3.f \xrightarrow{\mathsf{r}} H; \overline{w}_2}$$

D-SHORT
$$\frac{\Pi.\mathit{xliteral}(s) = \overline{b}}{\Pi; \overline{t} \vdash H; s \xrightarrow{\mathsf{r}} H; \overline{b}}$$

D-LONG
$$\frac{\Pi.\mathit{xliteral}(l) = \overline{b}}{\Pi; \overline{t} \vdash H; l \xrightarrow{\mathsf{r}} H; \overline{b}}$$

$$\mathrm{size}(\Pi, \sigma) = \begin{cases} 1 & \text{if } \sigma = \mathrm{byte} \\ i & \text{if } \sigma = \mathrm{pad}[i] \\ \Pi.\mathit{ptrsize} & \text{if } \sigma \in \{\mathrm{ptr}_\alpha(N), \mathrm{ptr}_\alpha(\overline{\sigma})\} \end{cases} \qquad \mathrm{size}(\Pi, w) = \begin{cases} 1 & \text{if } w \in \{b, \mathsf{uninit}\} \\ \Pi.\mathit{ptrsize} & \text{if } w = \ell{+}i \end{cases}$$

$$\mathrm{size}(\Pi, \sigma_1 \ldots \sigma_n) = \textstyle\sum_{i=1}^{n} \mathrm{size}(\Pi, \sigma_i) \qquad \mathrm{size}(\Pi, w_1 \ldots w_n) = \textstyle\sum_{i=1}^{n} \mathrm{size}(\Pi, w_i)$$

**Figure 4.** Dynamic Semantics

D-FADDR takes a pointer value and increases its offset by the offset of the field $f$, which is defined by $\Pi$. D-FETCHL, the one primitive reduction in left contexts, is similar, but we also have to 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 $*(\tau*)(\ell{+}j)$.

D-FETCH uses the offset and size information from $\Pi$ to project a subsequence of a value. We do not use the access function here because we are not accessing the heap.[4] Finally, D-SHORT and D-LONG use the platform to translate literals to byte-sequences.

Several of the rules require computing the size of a value $\overline{w}$ or a type $\overline{\sigma}$. Figure 4 includes these platform-dependent functions.

There are many ways to get stuck in the dynamic semantics, especially in the presence of arbitrary, unchecked pointer casts. To characterize memory-safe programs and the platforms on which they will not become stuck, we need a way to write down the platform-dependent layout assumptions made by a program and then a way to extract the assumptions from the program. The next two sections address these issues.

### 3.4 Constraint Language

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

The syntax for formulas $S$ is a first-order theory with (1) arithmetic, (2) sorts for aspects of our language (including fields $f$, types $\tau$, low-level types $\overline{\sigma}$, etc.), and (3) function symbols relevant to platform-dependencies. Figure 5 defines the function symbols and their interpretations. The interpretations induce the full definition of $\Pi \models S$ (e.g., $\Pi \models S_1 \wedge S_2$ iff $\Pi \models S_1$ and $\Pi \models S_2$).

[4] On actual machines, large values do not fit in registers so alignment remains a concern. We could model this by treating field access as an address-of-field computation followed by a dereference. However, the computation that produced the $v$ in $v.f$ must have done a properly aligned memory access, so if $v$ has the right type, then the more complicated treatment of field-access also would not have failed for any sensible platform.

| syntax | interpretation under $\Pi$ | defined in |
|---|---|---|
| $\text{xtype}(\bar{t}, \tau)$ | $\Pi.\mathit{xtype}(\bar{t}, \tau)$ | Figure 3 |
| $\text{align}(\bar{t}, \tau)$ | $\Pi.\mathit{align}(\bar{t}, \tau)$ | |
| $\text{offset}(\bar{t}, f)$ | $\Pi.\mathit{offset}(\bar{t}, f)$ | |
| $\text{access}(\alpha, i)$ | $\Pi.\mathit{access}(\alpha, i)$ | |
| $\text{xliteral}(s)$ | $\Pi.\mathit{xliteral}(s)$ | |
| $\text{xliteral}(l)$ | $\Pi.\mathit{xliteral}(l)$ | |
| $\text{size}(\overline{\sigma})$ | $\text{size}(\Pi, \overline{\sigma})$ | Figure 4 |
| $\text{size}(\overline{w})$ | $\text{size}(\Pi, \overline{w})$ | |
| $\text{subtype}(\bar{t}, \overline{\sigma}_1, \overline{\sigma}_2)$ | $\Pi; \bar{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ | Figure 6 |
| $\text{subalign}(\alpha_1, \alpha_2)$ | $\vdash \alpha_1 \leq \alpha_2$ | |

**Figure 5.** Function Symbols for the First-Order Theory

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 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 $\Pi.\mathit{align}(\bar{t}, \tau)$ for the space allocated for $x$. The second formula requires a low-level subtyping relationship between two pointer types. This is the constraint our static semantics generates for a cast like in Example 2 from Section 2.

These examples demonstrate the two flavors of formulas that arise in practice. First, there are constraints that every "sensible" platform would satisfy. We are not interested in other platforms, but stating these requirements as a constraint is much simpler than revisiting our definition of platforms. Second, there are constraints that we do not expect every platform to satisfy. Our static semantics produces a formula for these extra assumptions that a particular program makes.

The sensibility clauses we assume for type safety are straightforward to enumerate. We collect them in a constraint, called $S_{sensible}$, defined as the conjunction of the following formulas:

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:
   $\forall s, l, \bar{t}.\ \text{size}(\text{xliteral}(s)) = \text{size}(\text{xtype}(\bar{t}, \text{short}))$
   $\quad \wedge\ \text{size}(\text{xliteral}(l)) = \text{size}(\text{xtype}(\bar{t}, \text{long}))$
3. Greater alignment does not restrict access:
   $\forall \alpha_1, \alpha_2, i.\ (\text{access}(\alpha_1, i) \wedge \text{subalign}(\alpha_1, \alpha_2)) \Rightarrow \text{access}(\alpha_2, i)$
4. Translation of $\tau*$ respects the alignment and translation of $\tau$:
   $\forall \tau, \bar{t}.\text{subtype}(\bar{t},\ \text{ptr}_{\text{align}(\bar{t},\tau)}(\text{xtype}(\bar{t}, \tau)),\ \ \text{xtype}(\bar{t}, \tau*))$
5. Struct translation respects the offset and alignment of each field:
   $\forall \bar{t}, \tau, f, \overline{\sigma}, N.$
   $(N\{\ldots \tau\ f \ldots\} \in \bar{t} \wedge (\text{xtype}(\bar{t}, \tau) = \overline{\sigma}) \Rightarrow$
   $(\exists \overline{\sigma}_1, \overline{\sigma}_2, \alpha, o, o', i.$
   $\quad \text{xtype}(\bar{t}, N) = \overline{\sigma}_1 \overline{\sigma}\, \overline{\sigma}_2\ \wedge\ \text{size}(\overline{\sigma}_1) = \text{offset}(\bar{t}, f) = o'$
   $\wedge\ \text{align}(\bar{t}, N) = [a, o] \wedge \text{subalign}([a, o+o'], \text{align}(\bar{t}, \tau))))$

This gives rise to a precise notion of sensible platform:

DEFINITION 1. *A platform* $\Pi$ *is* sensible *if* $\Pi \models S_{\text{sensible}}$.

The sensibility constraints are necessary for portable code in the sense that without them, cast-free programs could get stuck. The C standard also allows other assumptions that we can write in our logic but that our safety theorem need not assume. Here are just two examples:

- The first field always has offset 0:
  $\forall f, \bar{t}, \tau, N.\ (N\{\tau\ f \ldots\} \in \bar{t}) \Rightarrow \text{offset}(\bar{t}, f) = 0$
- Fields are in order and do not overlap:
  $\forall \tau_1, f_1, \tau_2, f_2, N.\ N\{\ldots \tau_1\ f_1 \ldots \tau_2\ f_2 \ldots\} \in \bar{t}\} \Rightarrow$
  $(\text{offset}(\bar{t}, f_1) + \text{size}(\text{xtype}(\bar{t}, \tau_1)) \leq \text{offset}(\bar{t}, f_2))$

### 3.5 Static Semantics and Constraint Generation

With constraints in hand, we can now define a type-and-effect system where the main judgment $\bar{t}; \Gamma \vdash_{\mathsf{r}} e : \tau; S$ gives a constraint $S$ suitable for $e$. Because it is the pointer casts in $e$ that give rise to the constraints, this type system needs an expressive notion of pointer subtyping. Therefore, we consider subtyping (Figure 6) before describing the typing judgments for expressions (Figure 7).

***Subtyping:*** We use a subtyping relation on low-level types to formalize when data described by $\overline{\sigma}$ can also be described by $\overline{\sigma}'$, and hence when pointer casts are safe. This notion has been called *physical subtyping* because it relies on actual memory layouts. Because we take a byte-for-byte view of memory, our notion of physical subtyping is richer than those defined in prior work (Chandra and Reps 1999; Siff et al. 1999; Necula et al. 2005), which are at the level of ground C types instead of bytes. For example, our definition allows casting a `struct S{short x;short y}*` to a `struct D{long a;}*` on many platforms, whereas prior definitions forbid it. The rules for our judgment $\Pi; \bar{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ appear in Figure 6.

As expected in a language with mutation, pointer types have invariant subtyping (rule PTR). However, we do allow forgetting fields under a pointer type as this corresponds to restricting access to a prefix of the data previously accessible. This encodes the core concept behind casts like Example 2 in Section 2. We also allow assuming a less restrictive alignment. For example, a 4-byte aligned value can be safely treated as if it were 2- or 1-byte aligned.

Although we allow sequence-shortening under pointer types, it is *not* correct to allow shortening as a subtyping rule because a supertype should have the same size as a subtype (we can prove our rules have this property by induction on a subtyping derivation). This fact 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).

Rules UNROLL and ROLL witness the equivalence between a struct name and its definition. Recall we restrict a type $N$ to occur under pointers.

Rules PAD and ADD let us forget about the form of data (not under a pointer) without forgetting its size. Note that we disallow $\Pi; \bar{t} \vdash \text{pad}[i + j] \leq \text{pad}[i]\text{pad}[j]$ to prohibit accessing "part of a pointer", which would cause the abstract machine to get stuck. Rule SEQ lifts subtyping to sequences.

As usual, subsumption is sound for right-expressions but unsound for left-expressions. The static semantics enforces this restriction by disallowing casts as left-expressions.

***Static Semantics:*** The static semantics is shown in Figure 7. The judgments $\bar{t}; \Gamma \vdash_{\mathsf{r}} e : \tau; S$ and $\bar{t}; \Gamma \vdash_{\mathsf{l}} e : \tau; S$ (for right- and left-expressions respectively) produce types as usual, but also layout constraints $S$. This constraint is a conjunction of the layout assumptions the program is making. An alternative approach could parameterize the typing judgment by a platform instead of outputting a constraint, to essentially perform platform-dependent type-checking. This approach can be recovered from ours: we can check the constraint against a platform.

The interesting rules are S-CAST and S-FADDR because the constraint $S_{sensible}$ in Section 3.4 suffices to ensure other expression

$$\text{PTR}\ \frac{\vdash \alpha_1 \le \alpha_2}{D \vdash \mathrm{ptr}_{\alpha_1}(\overline{\sigma}_1\overline{\sigma}_2) \le \mathrm{ptr}_{\alpha_2}(\overline{\sigma}_1)} \qquad \text{UNROLL}\ \frac{\Pi.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}}{\Pi; \overline{t} \vdash \mathrm{ptr}_\alpha(N) \le \mathrm{ptr}_\alpha(\overline{\sigma})} \qquad \text{ROLL}\ \frac{\Pi.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}}{\Pi; \overline{t} \vdash \mathrm{ptr}_\alpha(\overline{\sigma}) \le \mathrm{ptr}_\alpha(N)} \qquad \text{PAD}\ \frac{\mathrm{size}(\Pi, \sigma) = i}{\Pi; \overline{t} \vdash \sigma \le \mathrm{pad}[i]}$$

$$\text{ADD}\ \frac{}{\Pi; \overline{t} \vdash \mathrm{pad}[i]\mathrm{pad}[j] \le \mathrm{pad}[i+j]} \qquad \text{SEQ}\ \frac{D \vdash \overline{\sigma}_1 \le \overline{\sigma}_2 \qquad D \vdash \overline{\sigma}_3 \le \overline{\sigma}_4}{D \vdash \overline{\sigma}_1\overline{\sigma}_3 \le \overline{\sigma}_2\overline{\sigma}_4} \qquad \text{REFL}\ \frac{}{D \vdash \overline{\sigma} \le \overline{\sigma}} \qquad \text{TRANS}\ \frac{D \vdash \overline{\sigma}_1 \le \overline{\sigma}_2 \qquad D \vdash \overline{\sigma}_2 \le \overline{\sigma}_3}{D \vdash \overline{\sigma}_1 \le \overline{\sigma}_3}$$

$$\text{ALIGN-BASE}\ \frac{a_1 = a_2 \times i}{\vdash [a_1, o] \le [a_2, o]} \qquad \text{ALIGN-OFFSET}\ \frac{o_1 \equiv o_2 \mod a}{\vdash [a, o_1] \le [a, o_2]} \qquad \text{ALIGN-TRANS}\ \frac{\vdash \alpha_1 \le \alpha_2 \qquad \vdash \alpha_2 \le \alpha_3}{\vdash \alpha_1 \le \alpha_3}$$

**Figure 6.** Physical Subtyping and Alignment Subtyping

$$\text{S-SHORT}\ \frac{}{\overline{t}; \Gamma \vdash_\mathsf{r} s : \mathrm{short}; \mathrm{true}} \qquad \text{S-LONG}\ \frac{}{\overline{t}; \Gamma \vdash_\mathsf{r} l : \mathrm{long}; \mathrm{true}} \qquad \text{S-NEW}\ \frac{}{\overline{t}; \Gamma \vdash_\mathsf{r} \mathsf{new}\ \tau : \tau*; \mathrm{true}} \qquad \text{S-VAR}\ \frac{\Gamma(x) = \tau}{\overline{t}; \Gamma \vdash_\mathsf{r} x : \tau; \mathrm{true}}$$

$$\text{S-ASSN}\ \frac{\overline{t}; \Gamma \vdash_\mathsf{r} e_1 : \tau; S_1 \qquad \overline{t}; \Gamma \vdash_\mathsf{r} e_2 : \tau; S_2}{\overline{t}; \Gamma \vdash_\mathsf{r} e_1 = e_2 : \tau; S_1 \wedge S_2} \qquad \text{S-FETCH}\ \frac{\overline{t}; \Gamma \vdash_\mathsf{r} e : N; S \qquad N\{\ldots \tau\ f \ldots\} \in \overline{t}}{\overline{t}; \Gamma \vdash_\mathsf{r} e.f : \tau; S} \qquad \text{S-SEQ}\ \frac{\overline{t}; \Gamma \vdash_\mathsf{r} e_1 : \tau'; S_1 \qquad \overline{t}; \Gamma \vdash_\mathsf{r} e_2 : \tau; S_2}{\overline{t}; \Gamma \vdash_\mathsf{r} e_1; e_2 : \tau; S_1 \wedge S_2}$$

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

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

$$\text{S-IF}\ \frac{\overline{t}; \Gamma \vdash_\mathsf{r} e_1 : \mathrm{long}; S_1 \qquad \overline{t}; \Gamma \vdash_\mathsf{r} e_2 : \tau; S_2 \qquad \overline{t}; \Gamma \vdash_\mathsf{r} e_3 : \tau; S_3}{\overline{t}; \Gamma \vdash_\mathsf{r} \mathsf{if}\ e_1\ e_2\ e_3 : \tau; S_1 \wedge S_2 \wedge S_3} \qquad \text{S-WHILE}\ \frac{\overline{t}; \Gamma \vdash_\mathsf{r} e_1 : \mathrm{long}; S_1 \qquad \overline{t}; \Gamma \vdash_\mathsf{r} e_2 : \tau; S_2}{\overline{t}; \Gamma \vdash_\mathsf{r} \mathsf{while}\ e_1\ e_2 : \mathrm{short}; S_1 \wedge S_2} \qquad \text{S-DECL}\ \frac{\overline{t}; \Gamma, x : \tau_1 \vdash_\mathsf{r} e : \tau_2; S}{\overline{t}; \Gamma \vdash_\mathsf{r} \tau_1\ x;\ e : \tau_2; S}$$

$$\text{S-VARL}\ \frac{\Gamma(x) = \tau}{\overline{t}; \Gamma \vdash_\mathsf{l} x : \tau; \mathrm{true}} \qquad \text{S-DEREFL}\ \frac{\overline{t}; \Gamma \vdash_\mathsf{r} e : \tau*; S}{\overline{t}; \Gamma \vdash_\mathsf{l} *(\tau*)(e) : \tau; S} \qquad \text{S-FETCHL}\ \frac{N\{\ldots \tau\ f \ldots\} \in \overline{t} \qquad \overline{t}; \Gamma \vdash_\mathsf{l} e : N; S}{\overline{t}; \Gamma \vdash_\mathsf{l} e.f : \tau; S}$$

**Figure 7.** Static Semantics (letting $\Gamma ::= \cdot \mid \Gamma, x{:}\tau$)

forms (such as dereferences and assignments) cannot fail due to a platform dependency. The constraints directly describe the implicit assumptions made in Examples 2, 3, and 4 in Section 2. The key insight here is that we can allow a pointer cast to assign an arbitrary type to an expression, but the cast will only be deemed legal on platforms that model the associated layout constraint. Recall $\mathrm{subtype}(\overline{t}, \overline{\sigma}_1, \overline{\sigma}_2)$ is the logical formula corresponding to $\Pi; \overline{t} \vdash \overline{\sigma}_1 \le \overline{\sigma}_2$. The S-FADDR constraint is much more complicated because it must state that there is some sequence of fields starting at the offset of field $f$ that can be viewed as a $\tau$.

Absent from this formal type system is support for downcasts, which are obviously important in practice. To support safe downcasts, we would just need to invert the direction of the subtyping constraint generated by the cast and employ existing techniques (Necula et al. 2005; Jim et al. 2002) to ensure that the casted value actually has the type dictated by the cast.

### 3.6 Metatheory and Low-Level Static Semantics

***Safety:*** Ideally, our type-safety result would claim that running a well-typed program on a "sensible" platform that also models the program's constraint would never lead to a stuck state. That is, given $\overline{t}; \cdot \vdash_\mathsf{r} e : \tau; S$, $\Pi$ is sensible, $\Pi \models S$, and $\Pi; \overline{t} \vdash \cdot; e \to^* H; e'$ (where $\to^*$ is the reflexive, transitive closure of $\to$), either $e'$ is a value or there exists $H', e''$ such that $\Pi; \overline{t} \vdash H; e' \to^* H'; e''$.

However, it is possible that the abstract machine can get stuck by accessing uninitialized data. Because preventing uninitialized accesses is not our focus, we relax our safety guarantee to admit that $e'$ might also be *legally stuck*. An expression $e$ is legally stuck if $e$ is of the form $\mathsf{R}[ls]_\mathsf{r}$ or $\mathsf{R}[ls]_\mathsf{l}$, where

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

Our memory-safety proof employs a low-level type system that captures the relevant invariants that evaluation preserves. The main judgment of this type system has the form $\Pi; \overline{t}; \Psi; \Gamma \vdash_\mathsf{r} e : \overline{\sigma}$ where $\Psi$ gives a type to the heap.[5] This system has implicit subsumption, which is necessary for a step via D-CAST to preserve typing:

$$\frac{\Pi; \overline{t}; \Psi; \Gamma \vdash_\mathsf{r} e : \overline{\sigma}_1 \qquad \Pi; \overline{t} \vdash \overline{\sigma}_1 \le \overline{\sigma}_2}{\Pi; \overline{t}; \Psi; \Gamma \vdash_\mathsf{r} e : \overline{\sigma}_2}$$

[5] $\Psi ::= \cdot \mid \Psi, \ell \mapsto \overline{\sigma}, \alpha$

Dynamic Semantics:

| | |
|---|---|
| $\Pi; \overline{t} \vdash H; e \rightarrow H'; e'$ | small step |
| $\Pi; \overline{t} \vdash H; e \xrightarrow{\mathsf{r}} H'; e'$ | primitive right-step |
| $\Pi; \overline{t} \vdash H; e \xrightarrow{\mathsf{l}} H'; e'$ | primitive left-step |

High−Level Static Semantics:

| | |
|---|---|
| $\Pi \models S$ | platform $\Pi$ models formula $S$ |
| $\vdash \alpha_1 \leq \alpha_2$ | subtyping on alignments |
| $\Pi; \overline{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ | platform-dependent subtyping |
| $\overline{t}; \Gamma \vdash_{\mathsf{r}} e : \tau; S$ | typing for right-expressions |
| $\overline{t}; \Gamma \vdash_{\mathsf{l}} e : \tau; S$ | typing for left-expressions |

Low−Level Static Semantics:

| | |
|---|---|
| $\Pi; \overline{t}; \Psi; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}$ | typing for right-expressions |
| $\Pi; \overline{t}; \Psi; \Gamma \vdash_{\mathsf{l}} e : \overline{\sigma}, \alpha$ | typing for left-expressions |

**Figure 8.** Summary of Judgments

Like in the source-level type system, we also have a judgment for left-expressions ($\Pi; \overline{t}; \Psi; \Gamma \vdash_{\mathsf{l}} e : \overline{\sigma}, \alpha$). This judgment does not have a subsumption rule, but does produce an alignment $\alpha$ for the location to which $e$ will evaluate.

Many of the low-level typing rules have hypotheses that refer directly to the platform. For example, the rule for type-checking dereferences is:

$$\frac{\Pi; \overline{t}; \Psi; \Gamma \vdash_{\mathsf{r}} e : \mathrm{ptr}_{\alpha}(\overline{\sigma}_1\overline{\sigma}_2) \quad \Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}_1 \quad \Pi.\mathit{access}(\alpha, \mathrm{size}(\Pi, \overline{\sigma}_1))}{\Pi; \overline{t}; \Psi; \Gamma \vdash_{\mathsf{r}} *(\tau*)(e) : \overline{\sigma}_1}$$

See the technical report (Nita et al. 2007) for the complete system, which includes rules for run-time forms (such as $\overline{w}$) and heaps.

The dynamic semantics and low-level type system enjoy the usual type soundness property (modulo legally stuck states), proven with the aid of the usual progress and preservation lemmas.

THEOREM 2. *(Low-Level Type Soundness) If* $\Pi; \overline{t}; \cdot; \cdot \vdash_{\mathsf{r}} e : \overline{\sigma}$ *and* $\Pi; \overline{t} \vdash \cdot; e \rightarrow^* H'; e'$*, then* $H'; e'$ *is not stuck on* $\Pi$.

The connection between the static semantics and the low-level type system is concisely stated by this theorem:

THEOREM 3. *If* $\overline{t}; \Gamma \vdash_{\mathsf{r}} e : \tau; S$*,* $\Pi$ *is sensible,* $\Pi \models S$*, and* $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}$*, then* $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}$.

The proof, by induction on the derivation of $\overline{t}; \Gamma \vdash_{\mathsf{r}} e : \tau; S$, uses the definition of $\Pi \models S$ in many cases. For example, a source derivation ending in S-DEREF can produce a low-level derivation ending in the dereference rule above because sensible platforms model $\mathrm{access}(\mathrm{align}(\overline{t}, \tau), \mathrm{size}(\mathrm{xtype}(\overline{t}, \tau)))$. Indeed, the proof of Theorem 3 ensures our definition of $S_{sensible}$ is sufficient.

Last but not least, we state the key theorem that a program will not get stuck on any platform on which its layout assumptions hold:

THEOREM 4. *(Layout Portability) If* $\overline{t}; \Gamma \vdash_{\mathsf{r}} e : \tau; S$*,* $\Pi$ *is sensible,* $\Pi \models S$*, and* $\Pi; \overline{t} \vdash \cdot; e \rightarrow^* H'; e'$*, then* $H'; e'$ *is not stuck on* $\Pi$.

This is a corollary to Theorems 2 and 3.

***Cast-Free Portability:*** The constraint produced by our type system is fairly expressive, ruling out only platforms for which some cast in the program would make no sense. To make this intuition precise, we prove that for the right definition of "cast-free", a cast-free program does not get stuck on any sensible platform.

DEFINITION 5. *(Cast-Free) A program* $\overline{t}; e$ *is* cast-free *if:*

- *No expressions of the form* $(\tau*)e'$ *occur in* $e$.
- *For every expression of the form* $(\tau*)\&e' \rightarrow f$ *in* $e$*, the type* $\tau$ *is the type of* $f$*. That is,* $N\{\ldots \tau\ f \ldots\} \in \overline{t}$.

The second point allows taking the address of a field but requires the resulting type to be the type of the field (rather than allowing a platform-dependent suffix cast). The key theorem is as follows:

THEOREM 6. *If* $\overline{t}; e$ *is cast-free and* $\overline{t}; \cdot \vdash_{\mathsf{r}} e : \tau; S$*, then* $S_{sensible} \Rightarrow S$.

The intuition that only casts threaten layout portability is captured by the following theorem, a corollary to Theorems 4 and 6:

THEOREM 7. *(Cast-Free Layout Portability) If* $\overline{t}; e$ *is cast-free,* $\overline{t}; \cdot \vdash_{\mathsf{r}} e : \tau; S$*,* $\Pi$ *is sensible, and* $\Pi; \overline{t} \vdash \cdot; e \rightarrow^* H'; e'$*, then* $H'; e'$ *is not stuck on* $\Pi$.

## 4. Extensions

This section sketches how the core model we have developed is flexible enough to be extended with some other relevant features of C and its platforms. We focus first on arrays because they are ubiquitous and require restricting our subtyping definition.

### 4.1 Arrays

As Example 5 demonstrates, a subtyping rule for pointers that drops a suffix of pointed-to fields (rule PTR in Figure 6) is unsound if the pointer may be used as a pointer to an array. Therefore, extending our model with arrays is important and requires some otherwise unnecessary restrictions. Figure 9 defines this extension formally.

Rather than conservatively assume all pointers may point to arrays, the types distinguish pointers to one object ($\tau*$ as already defined) from pointers to arrays ($\tau*^{\omega}$; the $\omega$ just distinguishes it from $\tau*$). This dichotomy is common in safe C-like languages (Necula et al. 2005; Jim et al. 2002), can be approximated via static analysis, and is necessary to identify what platform assumptions are due only to arrays. The low-level types ($\sigma$) make the same distinction.

We add two right-expression forms. First, new $\tau[e]$ creates a pointer to a heap-allocated array of length $e$. (Because $e$ will evaluate to a byte-sequence $\overline{b}$, a platform must interpret $\overline{b}$ as an integer; we use $\Pi.\mathit{val}$ for this conversion.) The dynamic rule D-NEWARR is exactly like D-NEW except it creates enough space at $H(\ell)$ for the array. Our type system does not prevent new $\tau[e]$ from being stuck if $e$ has uninitialized bytes or $e$ is negative.[6]

Second, $\&((\tau*^{\omega})(e_1))[e_2]$ is more easily read as $\&e_1[e_2]$; the size of $\tau$ guides the dynamic semantics like it does with pointer dereferences. This form produces a pointer to one array element, which can be dereferenced or assigned through. The dynamic rule D-ARRELT produces the pointer $\ell{+}(i + j \times k)$ where the array begins at $\ell{+}i$, elements have size $j$, and $e_2$ evaluates to $k$. However, the two hypotheses on the right perform a *run-time bounds check*; our type system does not prevent this check from failing and therefore the machine being stuck.[7] The bounds-check on $\&((\tau*^{\omega})(e_1))[e_2]$ ensures an ensuing dereference can never fail.

With this economical addition of arrays, we can design constraints and subtyping such that the only failures are bounds-checks. A key issue is alignment: Given the alignment of $e_1$, how can we know the alignment of $\&((\tau*^{\omega})(e_1))[e_2]$ without statically constraining the value of $e_2$? The solution taken by every sensible C platform is to ensure the size of $\tau$ is a multiple of its alignment; see Figure 9 for the formal constraint. That way, $\&((\tau*^{\omega})(e_1))[e_2]$ is at least as aligned as $e_1$. Assuming this constraint, the typing rules for the new expression forms add nothing notable.

[6] In C, $e$ is unsigned, but large allocations due to conversion from negative numbers are a well-known cause of defects.

[7] This check disallows pointing just past the end of the array, unlike C.

Syntax: $\tau ::= \ldots \mid \tau*^{\omega}$
$e ::= \ldots \mid \mathsf{new}\ \tau[e] \mid \&((\tau*^{\omega})(e))[e]$
$\mathsf{R} ::= \ldots \mid \mathsf{new}\ \tau[\mathsf{R}] \mid \&((\tau*^{\omega})(\mathsf{R}))[e] \mid \&((\tau*^{\omega})(\ell\text{+}i))[\mathsf{R}]$
$\sigma ::= \ldots \mid \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma}) \mid \mathrm{ptr}^{\omega}_{\alpha}(N)$

Platforms: $\Pi.val(\overline{b}) = i$

Dynamic semantics:

D-NEWARR
$$\frac{\ell \notin \mathrm{Dom}(H) \quad \Pi.align(\overline{t},\tau) = \alpha \quad \Pi.xtype(\overline{t},\tau) = \overline{\sigma} \quad \mathrm{size}(\Pi,\overline{\sigma}) = i \quad \Pi.val(\overline{b}) = j \geq 0}{\Pi;\overline{t} \vdash H; \mathsf{new}\ \tau[\overline{b}] \xrightarrow{r} H, \ell \mapsto \mathsf{uninit}^{i\times j}, \alpha; \ell\text{+}0}$$

D-ARRELT
$$\frac{\Pi.xtype(\overline{t},\tau) = \overline{\sigma} \quad H(\ell) = \overline{w},\alpha \quad \mathrm{size}(\Pi,\overline{\sigma}) = j \quad 0 \leq (i + j \times k) < \mathrm{size}(\Pi,\overline{w}) \quad \Pi.val(\overline{b}) = k}{\Pi;\overline{t} \vdash H; \&((\tau*^{\omega})(\ell\text{+}i))[\overline{b}] \xrightarrow{r} H; \ell\text{+}(i + j \times k)}$$

Sensibility constraint: size is a multiple of alignment

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

Subtyping and static semantics:

ARR
$$\frac{\overline{\sigma}_1 = \overline{\sigma}_2^{\,i} \qquad \vdash \alpha_1 \leq \alpha_2}{\Pi;\overline{t} \vdash \mathrm{ptr}^{\omega}_{\alpha_1}(\overline{\sigma}_1) \leq \mathrm{ptr}^{\omega}_{\alpha_2}(\overline{\sigma}_2)}$$

S-NEWARR
$$\frac{\overline{t};\Gamma \vdash_{\mathsf{r}} e : \mathrm{long}; S}{\overline{t};\Gamma \vdash_{\mathsf{r}} \mathsf{new}\ \tau[e] : \tau*^{\omega}; S}$$

S-ARRELT
$$\frac{\overline{t};\Gamma \vdash_{\mathsf{r}} e_1 : \tau*^{\omega}; S_1 \qquad \overline{t};\Gamma \vdash_{\mathsf{r}} e_2 : \mathrm{long}; S_2}{\overline{t};\Gamma \vdash_{\mathsf{r}} \&((\tau*^{\omega})(e_1))[e_2] : \tau*; S_1 \wedge S_2}$$

**Figure 9.** Additions for Arrays

Finally but most importantly, we consider subtyping for pointer-to-array types. Analogues of UNROLL and ROLL are sound for types of the form $\mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})$, but PTR must be replaced with a more restrictive rule. Therefore, ARR requires the element type of the subtype to be the element type of the supertype repeated $i$ times. This is more lenient than strict invariance. For example, it supports the platform-dependent idiom of treating an array of: `struct { short i1; short i2; short i3; short i4;};` as an array of `short`. We have proven safety given this subtyping rule (Nita et al. 2007).

The ARR rule does *not* support subtyping such as: $D \vdash \mathrm{ptr}^{\omega}_{\alpha}(\mathrm{byte\ byte\ byte}) \leq \mathrm{ptr}^{\omega}_{\alpha}(\mathrm{byte\ byte})$. A cast requiring this subtyping makes sense if the pointed-to-array has an element count divisible by 6, else it is memory-safe but probably a bug since the target type will "forget" the last byte in the array. We have not extended our formal model with arrays of known size, but we see no problems doing so. Such arrays are common in C, particularly with multidimensional arrays (all but one dimension have known size), which is why CCured (Necula et al. 2005) allows such casts.

### 4.2 Platform Selection

In practice, programs written in low-level languages can selectively run code based on features of the underlying platform. For example, in the following snippet, `lb` has type `long*`, `ib` points to a buffer filled with integers, and the program needs to treat `ib` as if it were an array of `long`:

```
if (sizeof(int) == sizeof(long))
  lb = (long*)ib;
else
  lb = convert(ib);
```

If the size of `int` equals the size of `long`, the buffer can be directly used at type `long*`. Otherwise, the function `convert` allocates a new `long*` buffer, copies the elements from `ib` into it, and assigns it into `lb`. The test offers a common-case short path, avoiding a copy on many platforms while remaining portable. Figure 10 shows the additions to our model to support this idiom. The pcase form has $n$ branches, each guarded by a constraint. Given a platform $\Pi$, pcase steps to a branch whose guard is true under $\Pi$ (shown in D-PCASE). The output constraint in the high-level typing rule (S-PCASE) encodes two important properties. The first conjunct requires the constraint for each branch's body to hold only if the constraint guarding the branch holds. In particular, an implication

$$e ::= \ldots \mid \mathsf{pcase}\ S_1\ \texttt{=>}\ e_1 \ldots S_n\ \texttt{=>}\ e_n$$

D-PCASE
$$\frac{\Pi \models S_k}{\Pi;\overline{t} \vdash (\mathsf{pcase}\ S_1\ \texttt{=>}\ e_1 \ldots S_n\ \texttt{=>}\ e_n) \xrightarrow{r} e_k}$$

S-PCASE
$$\frac{\forall 1 \leq i \leq n\ .\ \overline{t};\Gamma \vdash_{\mathsf{r}} e_i : \tau; S_i'}{\overline{t};\Gamma \vdash_{\mathsf{r}} \mathsf{pcase}\ S_1\ \texttt{=>}\ e_1 \ldots S_n\ \texttt{=>}\ e_n : \tau; \bigwedge_{i=1}^{n}(S_i \Rightarrow S_i') \wedge \bigvee_{i=1}^{n} S_i}$$

L-PCASE
$$\frac{\forall 1 \leq i \leq n\ .\ \text{if}\ \ \Pi \models S_i\ \ \text{then}\ \ \Pi;\overline{t};\Psi;\Gamma \vdash_{\mathsf{r}} e_i : \overline{\sigma} \quad \exists 1 \leq i \leq n\ .\ \Pi \models S_i}{\Pi;\overline{t};\Psi;\Gamma \vdash_{\mathsf{r}} \mathsf{pcase}\ S_1\ \texttt{=>}\ e_1 \ldots S_n\ \texttt{=>}\ e_n : \overline{\sigma}}$$

**Figure 10.** Additions for Platform Selection

holds vacuously on platforms not modeling the guard. The second conjunct demands that at least one of the guards is true. The low-level typing rule (L-PCASE) similarly demands that at least one guard is true under $\Pi$ and its corresponding expression type-checks under $\Pi$. Using pcase, the previous example can be written as:

```
pcase
  size(xtype(int)) = size(xtype(long)) => lb=(long*)ib
  size(xtype(int)) ≠ size(xtype(long)) => lb=convert(ib)
```

In addition to the idiom exemplified above, pcase effectively explains *selective compilation*, where the preprocessor is used to compile code on a platform-dependent basis.

### 4.3 Other Extensions

Additional discussion of extensions, including support for read-only (`const`) pointers, for platforms that choose to skip pad bytes when copying values, and for proper recursive subtyping (Amadio and Cardelli 1993), can be found in the companion technical report (Nita et al. 2007). All these extensions permit more subtyping.

## 5. Implementation

To give our theory a practical outlet, we have implemented a bug-finding tool that is directly inspired by the formal model. In general,

a tool based on the model should extract a layout constraint $S$ from a program and present informative warnings based on $S$. We see several ways to achieve this. For example:

- Directly explain $S$ to the user, *simplifying* it into a legible form and connecting it to relevant locations in the code.
- Check $S$ against a set of platforms. That is, the user chooses a set of platforms of interest $\Pi_1, \ldots, \Pi_n$, and warnings are reported whenever $\Pi_i \not\models S$, for $1 \leq i \leq n$.

Our tool takes the latter approach; we leave the former to future work. Implementing the formal model poses two main challenges:

1. The type system relies on syntactic types to produce constraints, using no flow or alias information. While convenient for the exposition and metatheory, such simplicity is too imprecise.
2. Real C programs contain many downcasts that are safe at run-time. In a simple implementation of the model, these downcasts would result in spurious warnings.

To address (1), our tool replaces the simple type system with a points-to analysis. Using points-to information yields much more precise constraints and correctly handles "round-trip" casts through `void*`. For example, `D*t=(D*)(void*)e`, where `e` points to a value of type `S*`, is properly identified as a cast from `S*` to `D*`.

To address (2), we built our tool assuming a standard scenario: The program was developed and tested on a platform called the *host*, and a programmer is now interested in porting it to some *targets*. We assume that if a pointer cast is legal on the host, it should also be legal on a target. If a cast is illegal on both the host and a target, we do not report a warning. Our experience suggests that (a) if a cast is legal on the host and illegal on a target, it is highly likely to be a bug, and (b) if a cast is illegal on both the host and a target, then it is highly *unlikely* to be a bug. The warning is likely a result of imprecision in the static analysis. In essence, our host-target scenario is an effective false-positive filter.

While it remains future work to use the tool to investigate thoroughly the extent of layout-portability bugs in C software, preliminary experience suggests it is a valuable addition to the developer's toolset when porting or programming with portability in mind. The rest of this section describes the tool's architecture and a case study that discovered a previously unknown bug.

### 5.1 Tool Overview

The tool has two main components. The *cast gatherer* is a static analysis that takes a C program and outputs a list of pointer casts that may occur at run-time. The *cast analyzer* inputs this list of casts, generates and checks their corresponding constraints, and outputs a list of warnings with code locations.

The cast gatherer uses an interprocedural points-to analysis[8] to determine which memory layouts an expression may point to at run-time. A first pass analyzes all `malloc`[9] sites and records a map of associated program points and their allocation-time types in a *type table*. In addition to allocation sites, program points for local variables that participate in pointer casts are also entered in the type table. A second pass analyzes each pointer cast `(`$\tau$`*)e`. For each entry `(`$p$`,`$\tau$`'*)` in the type table, if `e` may-alias the allocation expression at program point $p$ (hence `e` may point to memory with run-time type $\tau$`'*`), a pointer cast from $\tau$`'*` to $\tau$`*` is recorded.

The list of pointer casts output by the cast gatherer is passed to the cast analyzer along with a set of *platform descriptions* chosen by the user: one for the host and one or more for each target. A platform description is a module, written by us in Caml, that implements a platform's layout policy by defining exactly the platform functions in our model (recall Figure 3). We have implemented many such platform descriptions, most of which represent real platforms, and some of which implement imagined platforms that are still within the C language specification.

For each pointer cast, the cast analyzer generates the appropriate constraint. It then checks the constraint against the host description. If the constraint is true (i.e., the cast is legal on the host), it is checked against each target. If it is false for any target, the relevant warnings (described below) are output. We implemented the constraint checker manually and specialized it to our constraint language. It queries the physical subtyping relation, for which we have implemented an algorithm, and the platform description functions.

While a whole-program analysis is necessary for soundness, the tool can be run on a subset of the program at the expense of coverage. Also, it is easy to plug in a different cast gatherer. For example, we have experimented with a dynamic analysis that produces exact results per run.

```
data_link.c:196: scat_element * ==> iovec *
    Host (Gcc/32-bit X86):
        Src:  ptr_4(ptr_1(b) bbbb)
        Dest: ptr_4(ptr_1(b) bbbb)
    Target (Gcc/LP-64):
        Src:  ptr_8(ptr_1(b) bbbb----)
        Dest: ptr_8(ptr_1(b) bbbbbbbb)
events.c:150: sp_time * ==> timeval *
    Host (Gcc/32-bit X86):
        Src:  ptr_4(bbbb bbbb)
        Dest: ptr_4(bbbb bbbb)
    Target (Gcc/LP-64):
        Src:  ptr_8(bbbbbbbb bbbbbbbb)
        Dest: ptr_8(bbbbbbbb bbbb----)
```

**Figure 11.** Tool Output on Spread

### 5.2 Case Study

We ran our tool on Spread[10], a high-performance messaging service intended for use by distributed applications. We chose Spread because (a) it contained a reported layout portability bug[11], and (b) it is intended to be portable. Our tool issued two warnings, one of which was the known bug, and the other a new bug that to our knowledge has not been reported on the developer mailing lists. No false positives were reported.

A relevant subset of the tool's output is given in Figure 11. The rest of the output was about these same casts at different locations in the code. The two warnings (the first of which is the known bug) are issued in the context of a conventional "Gcc on 32-bit X86" host description and a so-called "LP-64" target, on which integers are 32 bits and `long` and pointers are 64 bits. Each warning lists the bad pointer cast and its location, followed by the layouts of the source and destination type on each platform. The layouts are displayed in an ASCII version of the language for $\sigma$ in Figure 3, where '`b`' stands for $\mathrm{byte}$, '`-`' for $\mathrm{pad}[1]$, '`ptr_4(b)`' for $\mathrm{ptr}_{[4,0]}(\mathrm{byte})$, etc.

In the first case, we learn that the two types are laid out identically on the host, but on the target, the source type has four bytes of trailing padding where the destination type contains data bytes. The types involved in the cast are as follows:

```
struct scat_element { char *buf; int    len; }
struct iovec        { char *buf; size_t len; }
```

[8] We use the points-to analysis that ships with CIL (Necula et al. 2002a).

[9] A command-line flag allows specifying names of user-defined allocators.

[10] http://www.spread.org

[11] http://commedia.cnds.jhu.edu/pipermail/spread-users/2002-November/001185.html

The cast makes an assumption that `sizeof(int)` is equal to `sizeof(size_t)`. On LP-64 platforms, however, `int` is 4 bytes and `size_t` is 8 bytes, and the cast leads to a bigger-than-expected value in `iovec`'s `len` field and hence out-of-bounds `buf` accesses.

In the second warning, the target layout contains trailing padding where the host does not. The two types are:

```
struct sp_time { long sec; long usec; }
struct timeval { time_t tv_sec;
                 suseconds_t tv_usec; }
```

On both the host and target, `time_t` is a type alias for `long` and `suseconds_t` is a type alias for `int`. The cast leads to truncation of the `usec` field, only half of the bytes being visible through `tv_usec`. If the target machine is big-endian, most or all of the relevant data is lost, being mapped to the sequence of padding.

This bug is particularly interesting, because in addition to being impossible to detect on the host via testing, it is also very difficult to detect on the target. Since the `tv_usec` field is off by some number of nanoseconds, finding this bug by testing literally depends on the time of day. Our tool detects it statically and without access to the target platform.

## 6. Previous Work

To our knowledge, previous work considering platform-dependent layout assumptions or low-level type-safety has taken one of the following approaches:

- Consider only one platform or one complete "bit-by-bit" data description.
- Check that a C program is portable, giving a compile-time error or fail-stop run-time termination if it is not.
- Assume that a C program is portable, i.e., extend the C compiler's view that behavior for unportable programs is undefined.

The first approach is not helpful for writing code that works correctly on a set of platforms. The second approach suffices for applications that should be written in a higher-level language. The third approach relegates some issues to work such as ours, much as we relegate some issues like array-bounds errors to other work.

### 6.1 Assuming a Platform

Most closely related is the work on physical type-checking (Chandra and Reps 1999; Siff et al. 1999), which motivated our work considerably. Their tool classifies C casts as "upcasts", "downcasts", or "neither", reporting a warning for the last possibility. Their notion of physical subtyping is at a higher level than ours (requiring that types, offsets, and field names match), and they neither parameterize their system by a platform nor produce descriptions of sets of platforms. Checking code against a new platform would require reverification and changing their tool. They present no metatheory validating their approach.

CCured (Necula et al. 2005), a memory-safe C platform, includes physical type-checking to reduce the number of casts that require run-time checks. That is, CCured permits casts that work in practice but are not allowed by the C standard. The allowed casts are safe under a padding strategy used by common C compilers for the x86 architecture, which covers some but certainly not all platforms. An interesting avenue for future work is to reimplement the CCured type analysis engine using our platform-as-parameter approach, thus making its memory-safety guarantee portable.

Work on typed assembly language and proof-carrying code (Necula 1997; Morrisett et al. 1999; Crary 2003; Chen et al. 2003; Hamid et al. 2003) clearly needs a low-level view of memory. Such projects can establish that certifying compilers produce code that cannot get stuck due to uninitialized memory, unaligned memory access, segmentation faults, etc. In particular, work on allocation semantics (Petersen et al. 2003; Ahmed and Walker 2003) has taken a lower-level view than our formalism by treating addresses as integers and exposing that pointer arithmetic can move between adjacent data objects. These approaches provide less help for writing platform-dependent code because verification of type-safety is repeated for each platform. In practice, defining a new platform is an enormous amount of work.

Various program analyses for C, such as the work by Wilson and Lam (1995) and Miné (2006), have represented structs and unions with explicit bytes and offsets by assuming one particular platform.

### 6.2 Safe C

Memory-safe dialects or implementations of C, such as CCured (Necula et al. 2005, 2002b; Condit et al. 2003), Deputy (Condit et al. 2007), SAFECode (Dhurjati et al. 2006), and Cyclone (Jim et al. 2002; Grossman et al. 2002; Grossman 2006), do not solve the platform dependency problem. Rather, they may reject (at compile-time) or terminate (at run-time) programs that attempt platform-dependent operations, or they may support only certain platforms. These approaches are fine for fully portable code or code that is correct assuming particular compilers.

Furthermore, the implementations of these systems include run-time systems (automatic memory managers, type-tag checkers, etc.) that themselves make platform-dependent assumptions! For example, Cyclone assumes 32-bit integers and pointers, and making this code more portable is a top request from actual users.

### 6.3 Formalizing C

Some recent work (Leroy 2006; Blazy et al. 2006) uses Coq to prove a C compiler correct. Their operational semantics for C distinguishes left and right expressions much as we do. However, their source language omits structs, avoiding many alignment and padding issues, and their metatheory proves correctness only for correct source programs, presumably saying nothing about platform-dependent code.

The HOL formalization of C by Norrish (1998) includes structs and uses a global namespace mapping struct names to sequences of typed fields, like our work. However, he purposely omits padding and alignment from his formalism. He has no separable notion of a platform; instead he models platform choices as nondeterminism.

### 6.4 Low-Level Code without C

Our work has been C-centric, whereas other projects have started with languages at higher levels of abstraction and added bit-level views for low-level programming. See Bacon (2003) and Hallgren et al. (2005) for just two recent examples. We believe this complementary approach would benefit from our constraint-based view rather than choosing just between completely high-level types and completely low-level ones.

C-- (Ramsey et al. 2005) makes data representation and alignment explicit, but C-- is not appropriate for writing platform-dependent code. Rather, it is a low-level language designed as a target for compiling high-level languages. It has explicit padding on data (a compiler inserts bits where desired) and explicit alignment on all memory accesses.[12] Incorrect alignment is an unchecked run-time error. The purpose of C-- is to handle back-end code-generation issues for a compiler; it is still expected that the front-end compiler will generate different (but similar) code for each platform and provide a run-time system, probably written in C.

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

## 7. Conclusions and Future Work

We developed a formal description of platform dependencies in low-level code. The key insight is a semantic notion of "platform" that directs a low-level operational semantics *and* models a syntactic constraint that we can produce via static analysis on a source program. We have proven soundness for a small core language and a simple static analysis. Giving platforms a clear identity in our framework clarifies a number of poorly understood issues. The formal model directly informs the implementation of a useful tool that finds and reports layout portability problems in C programs.

The technique of platform-as-parameter can apply broadly since high-level languages also have platform-defined behavior. As examples, SML programs may depend on the size of `int`, Scheme programs may depend on evaluation order, and Java programs may depend on fair thread-scheduling.

In the future we hope to move beyond memory-safety by checking that a C program is observationally equivalent on a set of platforms. This level of portability must account for issues like endianness, integer overflow, and perhaps floating-point roundoff. We believe the notion of platform selection described in Section 4.2 (`pcase`) is essential in such an endeavor. The idea is to translate a C program into a version of C with a `pcase` primitive, where preprocessor directives and `if` statements encoding platform selection are translated to corresponding `pcase` statements. Then, we can extract from this program the necessary proof obligations under which all `pcase` branches are equivalent when executed under their respective assumptions. Since `pcase` branches can be arbitrary code, equivalence checking is undecidable. We envision discharging the proof obligations in an interactive proof environment.

A key feature of the work in this paper is that it detects a relevant subset of portability problems *fully automatically*. A portability checker requiring interactive theorem proving is not accessible to most C programmers.

## References


Amal Ahmed and David Walker. The logical approach to stack typing. In *International Workshop on Types in Language Design and Implementation*, 2003.

Aleph One Limited. *The ARMLinux Book Online,* Chapter 10. 2005. http://www.aleph1.co.uk/armlinux/book.

Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4), 1993.

David F. Bacon. Kava: a Java dialect with a uniform object model for lightweight classes. *Concurrency and Computation: Practice and Experience*, 15(3–5), 2003.

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *14th International Symposium on Formal Methods*, 2006.

C Standard 1999. *ISO/IEC 9899:1999, International Standard—Programming Languages—C*. International Standards Organization, 1999.

Satish Chandra and Tom Reps. Physical type checking for C. In *Workshop on Program Analysis for Software Tools and Engineering*, 1999.

Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *ACM Conference on Programming Language Design and Implementation*, 2003.

Jeremy Condit, Matthew Harren, Scott McPeak, George Necula, and Westley Weimer. CCured in the real world. In *ACM Conference on Programming Language Design and Implementation*, 2003.

Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. Dependent types for low-level programming. In *European Symposium on Programming*, 2007.

Karl Crary. Toward a foundational typed assembly language. In *30th ACM Symposium on Principles of Programming Languages*, 2003.

Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM Conference on Programming Language Design and Implementation*, 2006.

Dan Grossman. Type-safe multithreading in Cyclone. In *International Workshop on Types in Language Design and Implementation*, 2003.

Dan Grossman. Quantified types in imperative languages. *ACM Transactions on Programming Languages and Systems*, 28(3), 2006.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *ACM Conference on Programming Language Design and Implementation*, 2002.

Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *10th ACM International Conference on Functional Programming*, 2005.

Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. *Journal of Automated Reasoning*, 31(3–4), 2003.

IBM. Developing embedded software for the IBM PowerPC 970FX processor. Application Note 970, IBM, 2004. http://www.ibm.com/chips/techlib/.

Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

Xavier Leroy. Formal certification of a compiler back-end. In *33rd ACM Symposium on Principles of Programming Languages*, 2006.

Robert Love. *Linux Kernel Development, Second Edition*. Novell Press, 2005. Page 328.

Brad Martin, Anita Rettinger, and Jasmit Singh. Multiplatform porting to 64 bits. *Dr. Dobb's Journal*, 2005.

Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *Conference on Language, Compilers, and Tool Support for Embedded Systems*, 2006.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3), 1999.

George Necula. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages*, 1997.

George Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conference on Compiler Construction*, 2002a.

George Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages*, 2002b.

George Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3), 2005.

Marius Nita, Dan Grossman, and Craig Chambers. A theory of platform-dependent low-level software (extended version). 2007. Available at http://www.cs.washington.edu/homes/marius/papers/tpd/.

Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In *30th ACM Symposium on Principles of Programming Languages*, 2003.

Norman Ramsey, Simon Peyton Jones, and Christian Lindig. The C-- language specification version 2.0, 2005. http://www.cminusminus.org/extern/man2.pdf.

Micahel Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas Reps. Coping with type casts in C. In *7th European Software Engineering Conference 7th ACM Symposium on the Foundations of Software Engineering*, 1999.

Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM Conference on Programming Language Design and Implementation*, 1995.