# A Theory of Implementation-Dependent Low-Level Software

Marius Nita Dan Grossman Craig Chambers
{marius,djg,chambers}@cs.washington.edu
Department of Computer Science and Engineering
University of Washington

UW-CSE Technical Report 2006-10-01

**Abstract**

This technical report is an extended version of the recent paper *A Theory of Implementation-Dependent Low-Level Software* by Nita, Grossman, and Chambers. The report contains additional discussion of the work and full proofs of the theorems described in the paper. The differences between the report and the paper are more preciesly outlined in Appendix B. If the reader is interested only in the safety proof, please skip to Appendix A.

We present a theory for describing implementation-dependent assumptions that a program in a C-like language 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. More specifically, the constraint produced by the analysis is a formula in first-order logic and implementations are models for the logic. By defining an abstract machine in such a way that it takes an implementation as a parameter, we can prove the analysis is sound. We use this foundation to explain some common but non-portable 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 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:

*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 (including padding), the size of values, and alignment restrictions on memory accesses. To our knowledge, existing work on safe low-level code (see Section 5) either (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 a particular language implementation (making no guarantee for a different compiler or architecture).

Requiring full portability for all code (e.g., by enforcing poorly understood and informally specified [22] 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 perfect libraries that abstract all implementation dependencies. Such high-level 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 computing 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. Whereas 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), 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 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". Section 2.2 discusses some specific real-world examples.

Common practice confronts this "somewhat but not completely portable" dilemma by manually isolating and minimizing implementation-dependent assumptions. For example, the Linux source code has an `arch` subdirectory; avoiding assumptions in the rest of the kernel is left to programmer discipline. 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, run-time system code for accessing object headers may 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" (and the source-code locations that influence the description). Defining the implementation-description language has forced us to give precise meaning to poorly understood platform-specific issues. The result is a foundation not only for a porting tool, but for any analysis seeking to account for implementation dependencies. While our focus is C, where incorrect assumptions can violate memory safety, many safe languages also have implementation-dependent behavior for which our approach of isolating the implementation-definition should apply.

## 1.1 Key Questions

Many compilers and low-level language tools have had to wrestle with defining semi-portability and implementation dependencies, but the issues have not been isolated and considered rigorously. In hindsight, the key questions are:

- How can we define the notion of an *implementation* to expose issues relevant to semi-portable software without exposing a full translation to assembly language?
- Can we analyze a C program statically to determine (conservatively) what implementation-dependent assumptions must hold for it to be memory-safe?
- What soundness guarantee can we provide? Why does memory-safety (unfortunately) not imply that a program behaves equivalently on different implementations?
- How can we prove a soundness theorem for a source-code analysis given that the code runs on an implementation-dependent low-level machine?

To answer these questions, we have built a novel model for a small-but-relevant C-level expression language.

## 1.2 Approach

The key to our formal model is isolating the notion of an *implementation*. An implementation has two roles: (1) as a parameter to the operational semantics, and (2) as something we can describe with a portability constraint. The actual definition of an implementation includes things like how to determine the offset of a `struct` field and what alignment restrictions a memory access must obey. The details are fully described in this work, but they are less important than the general insight: by parameterizing the operational semantics

by an implementation, we can take a program $P$ and state that a property (such as safety) holds for $P$ on implementations satisfying a constraint $S$. That is, $P$ is semi-portable as described by $S$.

To see how implementations 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 implementation. Therefore, our operational semantics has 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 guide the memory access (and become stuck if *impl* deems the access misaligned).

As for portability constraints, they are formulas in a theory of first-order logic that we interpret by having implementations serve as models. For example, the constraint "$\text{access}(4, 8) \wedge \text{size}(\text{long}) = 8$" 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 $\mathit{impl} \models S$, then the abstract machine does not get stuck when running $P$ given *impl*. 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 "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 will use a more sophisticated analysis, but the set-up (given $P$, produce an $S$) will remain the same.

The key metatheoretic result is showing that the $S$ our system produces is indeed sound. To do so, we define a second type system for program states. This second type system, which exists only to prove safety, is parameterized by an *impl* just 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 (not including orthogonal issues like array-bounds violations and uninitialized memory, which we could prevent at the cost of losing our focus on semi-portability).

2. If the first type system produces $S$ given $P$, then $P$ type-checks in the second type system for any *impl* such that $\mathit{impl} \models S$.

Together, these lemmas ensure the program cannot get stuck when run on any model of $S$.

## 1.3 Contributions and Caveats

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

- Most semi-portability questions can be 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, often depends on the implementation.

- A theory in first-order logic can describe implementation constraints.

- There should be a notion of "sensible" implementation, meaning implementations on which every program without casts cannot get stuck. The constraint language lets us write a formula for which the models are the "sensible" implementations.

- A program's safety might depend on a pointer not being treated as an array or a pointer not being written through.

For tractability, the model we present considers only a small expression language inspired by C. It has many relevant features, including structs (and assignment of struct values), 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

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

and `goto`). We see no fundamental problems extending our model in such directions. We also present a definition of implementation that is slightly simpler than one may encounter in practice. In particular, we assume all pointers have the same size and that alignment restrictions do not depend on the destination of a memory access (e.g., we do not distinguish floating-point operations). Again, extending our definition of implementation is straightforward.

### 1.4 Outline

Section 2 presents examples of semi-portable code and the constraints describing their assumptions. 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 describes how to extend the model in several directions, most importantly to support arrays. 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 semi-portability and relevant implementation constraints. Section 2.2 complements this "tutorial" with actual platforms, systems, and coping strategies related to these concepts.

### 2.1 Small Code Fragments

**Example 0: Accessing Memory**

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

A memory access such as `(*e).f` reads or writes $n$ bytes at some alignment $m$. If `e` has type `struct T*` and the `f` field has type $\tau$, then $n$ is the *size* of $\tau$ 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 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 implementation (which comprises compiler and machine) "supports" 8-byte accesses on 4-byte alignments. In this paper, we assume implementations include an *access* function of type Int $\to$ Int $\to$ Bool, as well as *size* and *alignment* functions that map types to integers.[2]

Our example `(*e).f` therefore induces the constraint access$(n, m)$ where $m$ and $n$ are defined above. However, this constraint assumes `e` actually evaluates to a pointer with alignment $m$ and a $\tau$ at the right offset. The constraints for cast expressions must ensure this.

**Example 1: Prefix**

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

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.

[2] We actually generalize the notion of alignment to include an offset, e.g., [8,0] would describe an 8-byte aligned address and [8,2] would describe a 2-byte offset from an 8-byte aligned address.

**Example 2: 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 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 will generate constraints preventing such a representation mismatch if the program has this cast.

**Example 3: 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, 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.

**Example 4: Arrays and Prefixes**

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

Example 1 (and 2–3) implicitly assumed the destination pointer was not used as an array (i.e., it was used as a pointer to one and not more than one `struct D1`). 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 4.1 therefore extends our model to make the necessary distinction between pointers to single-objects and pointers to arrays.

**Example 5: Deep Subtyping**

```
struct S4 {
  struct {int* f1; int* f2;} *f3;
};
struct D4 {
  const struct {int* g1;} *g2;
};
```

A cast from `struct S4*` to `struct D4*` is safe only because `const` qualifies the type of `g2`. As expected, read-only access permits more casts (i.e., fewer implementation constraints) for the same reasons covariant subtyping is sound on read-only fields. Section 4.2 adds this orthogonal feature to our model.

**Example 6: Skipping Pad Bytes**

```
struct S6 {    struct D6 {
  int* f1;       int* g1;
  int* f2;       double g2;
  double f3;   };
};
```

A cast from `struct S6*` to `struct D6*` might appear safe if pointers are 4 bytes, `struct S6` has no padding, and `struct D6` has 4 bytes of padding before `g2`. However, an assignment such as `*p=*q` where `p` and `q` have type `struct D6*` may overwrite the pad bytes (which thanks to casting actually need to hold a pointer). Section 4.3 extends our model to let implementations indicate on a per-pad basis how they implement assignment to `struct` values.

**Example 7: Safe-But-Inequivalent Implementations**

```
struct S7 {    struct D7 {
  long f1;       short g1;
};               short g2;
               };
```

Assuming there is no padding, that `long` is twice the size of `short`, and that there are no misaligned accesses, a cast from `struct S7*` to `struct D7*` is safe. No misaligned memory access or treating an integer as pointer can result. However, endianness can cause different implementations to behave differently. We leave such notions of equivalence to future work, focusing here only on safety, which we believe will still prove incredibly useful in writing semi-portable code and debugging ported applications. In particular, by preventing reading beyond the end of a struct, we detect many no-padding assumptions.

Interestingly, once one focuses on safety, certain constraints are actually unnecessary even though correct C implementations must support them. For example, type-compatible fields of a `struct` could have the same offsets (like a `union`). So omitting constraints like "fields have disjoint offsets" does not ruin our safety result.

## 2.2 Practical Scenarios

The scope of the portability problem is not precisely known because defects lie dormant until one changes hardware or compiler (or at least compiler flags). Therefore, like date bugs (such as the famous Y2K problem of last decade), offending code can be difficult to locate and fix.[3]

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 [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.[4]

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" [25].

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 [21]. 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 [26].

[3] Furthermore, semantics-based Y2K solutions [13] serve as inspiration for us, though portability bugs fortunately do not share a worldwide deadline.

[4] Emphasis in original.

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

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

A third example comes from the common practice of writing garbage collectors for high-level languages in C. If objects are allocated in-line by generated assembly code but the garbage collector accesses them via a C struct, there are unchecked and often undocumented assumptions that the code generator and C code make about each other. In principle, our constraint language could enable documenting the relevant assumptions, and a tool could automatically test a C implementation against the documentation.

# 3 Core Language

This section develops a formal model that can explain examples 0–3 from Section 2. We define an appropriate core language, a definition of *implementation*, a dynamic semantics, a first-order theory that constrains implementations, a type-and-effect system to produce constraints, and a type-soundness result (acquired via a second, lower-level type system). Figure 8 on page 15 summarizes the judgments defined for the model.

## 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 ($\overline{t}$) 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 ($(\tau*)\&\tau*\rightarrow ef$), 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 lets us support "suffix casts" as in Example 3 of Section 2.

The remaining expression forms are for memory allocation or control flow. $\mathsf{new}\ \tau$ 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, a while-loop evaluates to a number if it terminates. Finally, $\tau\ x;\ e$ creates a local variable $x$ of type $\tau$ bound in $e$. Because memory management is not our concern, the dynamic semantics uses heap allocation even for local variables.

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

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

Figure 2: Syntax extensions for run-time behavior

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

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

Figure 3: Implementations and low-level types

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 $\mathsf{uninit}$, or pointers $\ell{+}i$. A pointer is a label $\ell$ and an offset $i$ because we roughly model the heap as a mapping from labels to values (which recall are 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 "just right" for modeling semi-portable C.

Actually, heaps also map labels to alignments $[a, o]$ which means the address $\ell$ is $o$ mod $a$. (Typically $o$ is zero.) As the next section shows, an implementation specifies an alignment for each type; allocating an object of type $\tau$ produces a fresh heap location with this alignment. Note that if $\ell$ has alignment $[a, o]$, then $\ell{+}i$ has alignment $\gcd(a, a + i + o)$ where gcd computes the greatest-common-divisor.

## 3.2 Implementations

An implementation (*impl*) has several components, summarized in Figure 3. Together these components suffice to guide an abstract machine parameterized by an implementation.

- 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 $\tau$ (assuming type definitions $\overline{t}$).
- An *alignment* function (*impl.align*) returns the alignment $\alpha$ used to allocate space for a $\tau$.
- 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, but an implementation comprises all components. It is clear 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 these restrictions precisely.

Low-level types (the target of *impl.xtype*) are $\overline{\sigma}$, a sequence of $\sigma$. For example, if long is four bytes, the translation is byte byte byte byte, i.e., byte$^4$. 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 $\alpha$ (i.e., $\alpha$ 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 for expressions, parameterized by an implementation and a sequence of type definitions. Figure 4 holds the full definition for $\mathit{impl}; \overline{t} \vdash H; e \to H'; e'$. 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 ($\xrightarrow{\text{l}}$ and $\xrightarrow{\text{r}}$) for each sort of context [16]. 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 *impl*, 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. We do not restrict the length of the byte-sequences.[5] 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 implementation 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 is much like D-New (its has identical hypotheses) because we model local variables by heap allocating them. The resulting expression is the substitution of $*(\tau*)(\ell$+0) for $x$.

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 *impl.access* to model alignment constraints on the memory access.

D-Assign has the exact 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 $H(\ell)$.

D-Faddr takes a pointer value and increases its offset by the offset of the field $f$, which is defined by *impl*. 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 *impl* to project a subsequence of a value. We do not use the access function here because we are not accessing the heap.[6]

Finally, D-Short and D-Long use the implementation directly to translate literals to byte-sequences.

[5] We also disallow pointer values in the sequence, but allowing them would cause no problems.

[6] 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 implementation.

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

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

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

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
$$
\frac{\begin{array}{ll} \ell \notin \mathrm{Dom}(H) & impl.align(\overline{t}, \tau) = \alpha \\ impl.xtype(\overline{t}, \tau) = \overline{\sigma} & \mathrm{size}(impl, \overline{\sigma}) = i \end{array}}{impl; \overline{t} \vdash H; \mathsf{new}\ \tau \xrightarrow{\mathsf{r}} (H, \ell \mapsto \mathsf{uninit}^i, \alpha); \ell{+}0}
$$

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

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

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

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

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

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

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

$$
\mathrm{size}(impl, \sigma) = \begin{cases} 1 & \text{if } \sigma \in \{\mathrm{byte}, \mathrm{skip}\} \\ i & \text{if } \sigma = \mathrm{pad}[i] \\ impl.ptrsize & \text{if } \sigma \in \{\mathrm{ptr}_\alpha(N), \mathrm{ptr}_\alpha(\overline{\sigma})\} \end{cases}
$$

$$
\mathrm{size}(impl, w) = \begin{cases} 1 & \text{if } w \in \{b, \mathsf{uninit}\} \\ impl.ptrsize & \text{if } w = \ell{+}i \end{cases}
$$

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



Figure 4: Dynamic Semantics

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

### 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 (including fields $f$, types $\tau$, low-level types $\overline{\sigma}$, alignments $\alpha$, etc.), (2) arithmetic, and (3) function symbols relevant to implementation-dependencies. Figure 5 defines these function symbols and their interpretation. These interpretations induce the full definition of $impl \models S$ as usual (e.g., $impl \models S_1 \wedge S_2$ if and only if $impl \models S_1$ and $impl \models S_2$).

Consider two example constraints, which are just 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 $impl.align(\bar{t}, \tau)$ 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 that every "sensible" implementation would satisfy. We are not interested in other implementations, but stating these requirements syntactically is much simpler than revisiting our definition of implementations. Second, there are constraints that describe 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 for type safety are straightforward to enumerate and eminently justifiable:

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

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

Figure 5: Function Symbols for the First-Order Theory

$$\textsc{ptr}\ \frac{\vdash \alpha_1 \leq \alpha_2}{D \vdash \text{ptr}_{\alpha_1}(\overline{\sigma}_1\overline{\sigma}_2) \leq \text{ptr}_{\alpha_2}(\overline{\sigma}_1)} \qquad \textsc{unroll}\ \frac{\mathit{impl}.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}}{\mathit{impl}; \overline{t} \vdash \text{ptr}_\alpha(N) \leq \text{ptr}_\alpha(\overline{\sigma})} \qquad \textsc{roll}\ \frac{\mathit{impl}.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}}{\mathit{impl}; \overline{t} \vdash \text{ptr}_\alpha(\overline{\sigma}) \leq \text{ptr}_\alpha(N)}$$

$$\textsc{pad}\ \frac{\text{size}(\mathit{impl}, \sigma) = i}{\mathit{impl}; \overline{t} \vdash \sigma \leq \text{pad}[i]} \qquad \textsc{add}\ \frac{}{\mathit{impl}; \overline{t} \vdash \text{pad}[i]\text{pad}[j] \leq \text{pad}[i+j]} \qquad \textsc{seq}\ \frac{D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2 \qquad D \vdash \overline{\sigma}_3 \leq \overline{\sigma}_4}{D \vdash \overline{\sigma}_1\overline{\sigma}_3 \leq \overline{\sigma}_2\overline{\sigma}_4} \qquad \textsc{refl}\ \frac{}{D \vdash \overline{\sigma} \leq \overline{\sigma}}$$

$$\textsc{trans}\ \frac{D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2 \qquad D \vdash \overline{\sigma}_2 \leq \overline{\sigma}_3}{D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_3} \qquad \textsc{align-base}\ \frac{a_1 = a_2 \times i}{\vdash [a_1, o] \leq [a_2, o]} \qquad \textsc{align-offset}\ \frac{o_1 \equiv o_2 \mod a}{\vdash [a, o_1] \leq [a, o_2]} \qquad \textsc{align-trans}\ \frac{\vdash \alpha_1 \leq \alpha_2 \qquad \vdash \alpha_2 \leq \alpha_3}{\vdash \alpha_1 \leq \alpha_3}$$

Figure 6: Physical Subtyping (and Subtyping on Alignments)

These constraints are necessary for portable code in the sense that without them certain 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 three examples:

- long is at least as big as short:
  $\forall \overline{t}.\ \text{xtype}(\overline{t}, \text{long}) \geq \text{xtype}(\overline{t}, \text{short})$
- The first field always has offset 0:
  $\forall f, \overline{t}, \tau.\ (N\{\tau f \ldots\} \in \overline{t}) \Rightarrow \text{offset}(\overline{t}, f) = 0$
- Fields are in order and do not overlap:
  $\forall \tau_1, f_1, \tau_2, f_2.\ N\{\ldots \tau_1\ f_1 \ldots \tau_2\ f_2 \ldots\} \in \overline{t}\} \Rightarrow$
  $(\text{offset}(\overline{t}, f_1) + \text{size}(\text{xtype}(\overline{t}, \tau_1)) < \text{offset}(\overline{t}, f_2))$

## 3.5 Physical Subtyping

As in prior work [6, 35, 31], we use a subtyping relation on low-level types to formalize that data described by $\overline{\sigma}$ can also be treated as a $\overline{\sigma}'$. This notion has been called *physical subtyping* because it relies on actual memory layouts. The rules for our judgment $\mathit{impl}; \overline{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ appear in Figure 6 (recall we abbreviate $\mathit{impl}; \overline{t}$ as $D$ when their form is unimportant).

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 1 in Section 2. We also

allow assuming a less restrictive alignment (see the rules for $\vdash \alpha_1 \leq \alpha_2$), which also can only restrict how a pointer can be used.

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. We allow such use of $N$ in order to support recursive types.

Rules PAD and ADD let us forget about the form of data (not under a pointer) without forgetting its size.[7]

Rule SEQ lifts subtyping to sequences. As usual, subtyping is reflexive and transitive.

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`.) The static semantics enforces this restriction by disallowing casts as left-expressions.

## 3.6 Static Semantics and Constraint Generation

The preceding definitions of constraints and subtyping provide what we need to define a static semantics for source programs (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 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. The constraints directly describe the implicit assumptions made in Examples 1, 2, and 3 in Section 2. The S-FADDR constraint is much more complicated because it is not required in general that every subsequence of fields have an alignment appropriate for treating it as a type.

By using separate but mutually recursive typing rules for left-expressions and right-expressions we ensure left-expressions conform to a restricted grammar (as in C), namely expressions of the form $x$, $*(\tau*)(e_1)$, and $e_2.f$ where $e_2$ is itself a left-expression. Because subsumption is explicit (via casts) and casts are not left-expressions, there is no subsumption for left-expressions. Note the "cast" in dereference expressions is not really a cast (S-DEREF requires a type equality); the explicit type indicates a run-time size used by D-DEREF.

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 to ensure that the casted value actually has the result of the cast. Techniques used in existing safe-C approaches [31, 23] include implicit run-time type information, explicit discriminated unions, (bounded) parametric polymorphism, etc.

## 3.7 Metatheory and Low-Level Static Semantics

**Safety:**

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

However, this claim is not quite true because our type system does not prevent trying to use uninitialized data. Therefore, we must relax our safety guarantee to admit that $e'$ might also be "legally stuck," which

[7] For technical reasons we cannot allow $\mathit{impl}; \bar{t} \vdash \text{pad}[i+j] \leq \text{pad}[i]\text{pad}[j]$; it could allow a program to access "part of a pointer". On one level this is sound (it could not do anything with the result because of type pad[$i$]), but our abstract machine would get stuck when trying to extract part of a pointer.

S-Short
$$\frac{}{\overline{t}; \Gamma \vdash_{\mathsf{r}} s : \text{short}; \text{true}}$$

S-Long
$$\frac{}{\overline{t}; \Gamma \vdash_{\mathsf{r}} l : \text{long}; \text{true}}$$

S-New
$$\frac{}{\overline{t}; \Gamma \vdash_{\mathsf{r}} \mathsf{new}\ \tau : \tau*; \text{true}}$$

S-Var
$$\frac{\Gamma(x) = \tau}{\overline{t}; \Gamma \vdash_{\mathsf{r}} x : \tau; \text{true}}$$

S-Assn
$$\frac{\overline{t}; \Gamma \vdash_{\mathsf{l}} e_1 : \tau; S_1 \qquad \overline{t}; \Gamma \vdash_{\mathsf{r}} e_2 : \tau; S_2}{\overline{t}; \Gamma \vdash_{\mathsf{r}} e_1 = e_2 : \tau; S_1 \wedge S_2}$$

S-Fetch
$$\frac{\overline{t}; \Gamma \vdash_{\mathsf{r}} e : N; S \qquad N\{\ldots \tau\ f \ldots\} \in \overline{t}}{\overline{t}; \Gamma \vdash_{\mathsf{r}} e.f : \tau; S}$$

S-Seq
$$\frac{\overline{t}; \Gamma \vdash_{\mathsf{r}} e_1 : \tau'; S_1 \qquad \overline{t}; \Gamma \vdash_{\mathsf{r}} e_2 : \tau; S_2}{\overline{t}; \Gamma \vdash_{\mathsf{r}} e_1; e_2 : \tau; S_1 \wedge S_2}$$

S-Deref
$$\frac{\overline{t}; \Gamma \vdash_{\mathsf{r}} e : \tau*; S}{\overline{t}; \Gamma \vdash_{\mathsf{r}} *(\tau*)(e) : \tau; S}$$

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

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

S-If
$$\frac{\overline{t}; \Gamma \vdash_{\mathsf{r}} e_1 : \text{long}; S_1 \qquad \overline{t}; \Gamma \vdash_{\mathsf{r}} e_2 : \tau; S_2 \qquad \overline{t}; \Gamma \vdash_{\mathsf{r}} e_3 : \tau; S_3}{\overline{t}; \Gamma \vdash_{\mathsf{r}} \mathsf{if}\ e_1\ e_2\ e_3 : \tau; S_1 \wedge S_2 \wedge S_3}$$

S-While
$$\frac{\overline{t}; \Gamma \vdash_{\mathsf{r}} e_1 : \text{long}; S_1 \qquad \overline{t}; \Gamma \vdash_{\mathsf{r}} e_2 : \tau; S_2}{\overline{t}; \Gamma \vdash_{\mathsf{r}} \mathsf{while}\ e_1\ e_2 : \text{short}; S_1 \wedge S_2}$$

S-Decl
$$\frac{\overline{t}; \Gamma, x : \tau_1 \vdash_{\mathsf{r}} e : \tau_2; S}{\overline{t}; \Gamma \vdash_{\mathsf{r}} \tau_1\ x;\ e : \tau_2; S}$$

S-VarL
$$\frac{\Gamma(x) = \tau}{\overline{t}; \Gamma \vdash_{\mathsf{l}} x : \tau; \text{true}}$$

S-DerefL
$$\frac{\overline{t}; \Gamma \vdash_{\mathsf{r}} e : \tau*; S}{\overline{t}; \Gamma \vdash_{\mathsf{l}} *(\tau*)(e) : \tau; S}$$

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 $\Gamma ::= \cdot \mid \Gamma, x{:}\tau$)

| | |
|---|---|
| $impl; \overline{t} \vdash H; e \to H'; e'$ | small step in dynamic semantics |
| $impl; \overline{t} \vdash H; e \xrightarrow{\mathsf{r}} H'; e'$ | primitive right-step reduction in dynamic semantics |
| $impl; \overline{t} \vdash H; e \xrightarrow{\mathsf{l}} H'; e'$ | primitive left-step reduction in dynamic semantics |
| $impl \models S$ | implementation *impl* models formula $S$ |
| $\vdash \alpha_1 \leq \alpha_2$ | subtyping on alignments |
| $impl; \overline{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ | low-level implementation-dependent subtyping |
| $\overline{t}; \Gamma \vdash_{\mathsf{r}} e : \tau; S$ | static semantics for right-expressions |
| $\overline{t}; \Gamma \vdash_{\mathsf{l}} e : \tau; S$ | static semantics for left-expressions |
| $impl; \overline{t}; \Psi; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}$ | low-level typing for right-expressions |
| $impl; \overline{t}; \Psi; \Gamma \vdash_{\mathsf{l}} e : \overline{\sigma}, \alpha\alpha$ | low-level typing for left-expressions |

Figure 8: Summary of Judgments

we define as expressions of the form $\mathsf{R}[stuck]_{\mathsf{r}}$ or $\mathsf{R}[stuck]_{\mathsf{l}}$ where:

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

The proof, available in the appendix, employs a "low-level, run-time type system" that captures the relevant invariants that evaluation preserves. The main judgment of this type system has the form $impl; \overline{t}; \Psi; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}$ where $\Psi$ gives a type to the heap.[8] This system has implicit subsumption, which is necessary for a step via D-Cast to preserve typing:

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

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

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

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

See the appendix for the complete system, which includes rules for run-time forms (such as $\overline{w}$) and heaps.

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

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 $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}$.

The proof, by induction on the derivation of $\overline{t}; \cdot \vdash_{\mathsf{r}} e : \tau; S$, uses the definition of $\models$ 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 implementations model $\mathrm{access}(\mathrm{align}(\overline{t}, \tau), \mathrm{size}(\mathrm{xtype}(\overline{t}, \tau)))$.

Given the lemma above, showing preservation and progress [36] (modulo legally stuck states) for the dynamic semantics relative to the low-level type system suffices to establish 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

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

Syntax:

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

Implementations:

$$impl.val(\overline{b}) = i$$

Dynamic semantics:

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

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

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:

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

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

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

Figure 9: Extensions for Arrays

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).

To be precise, a program $\overline{t}; e$ is cast-free if:

- The only occurrences of $(\tau{*})e'$ in $e$ are in expressions of the form $*((\tau{*})e')$ and $(\tau{*})\&\tau{*}{\rightarrow}e'f$.
- For every expression of the form $(\tau{*})\&\tau{*}{\rightarrow}e'f$ 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 an implementation-dependent suffix cast).

# 4 Extensions

This section sketches how the core model we have developed is flexible enough to be extended with other relevant features of C and its implementations. We focus primarily on arrays because they are ubiquitous and require restricting our subtyping definition. Other extensions we consider merely permit additional subtyping.

## 4.1 Arrays

As Example 4 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 [31, 23], can be approximated via static analysis over C code, and is necessary to identify what implementation 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 $\overline{b}$, an implementation must interpret $\overline{b}$ as an integer; we use *impl.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.[9]

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 then be dereferenced or assigned through. (Adding a form that produces a $\tau*^{\omega}$ type instead of a $\tau*$ would work fine, but we omit it here.) 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.[10] By performing the bounds-check on $\&((\tau*^{\omega})(e_1))[e_2]$, we ensure an ensuing dereference can never fail.

With this economical addition of arrays to the abstract machine, we can design constraints and subtyping such that the only failures are bounds-checks (not unaligned memory accesses or treating bytes as pointers). 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 $e_2$? (This issue does not arise with $(\tau*)\&\tau*\rightarrow ef$ precisely because the offset of $f$ is known statically.) The solution taken by every sensible C implementation 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. Interestingly, this sensibility constraint is not needed without 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 $\text{ptr}^{\omega}_{\alpha}(\overline{\sigma})$, 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 (for some $i$). This is more lenient than strict invariance. For example, it supports the semi-portable practice 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. (See Appendix A).

The Arr rule does *not* support subtyping such as:
$D \vdash \text{ptr}^{\omega}_{\alpha}(\text{byte byte byte}) \leq \text{ptr}^{\omega}_{\alpha}(\text{byte byte})$. 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 must have known size), which is why CCured [31] allows casts like this.

An alternate approach could allow such subtyping for pointers to arrays of unknown size, but change the run-time behavior of casts to check that the array length is an appropriate multiple. In any case, our model provides an excellent starting point for considering subtle variations of subtyping and how it relies on implementation dependencies, which is exactly the goal of our work.

## 4.2 Read-Only Pointers

We do not allow subtyping under pointer types because a pointer can be dereferenced on the left side of an assignment. In C, `const` $\tau*$ describes pointers that cannot be used to write to the pointed-to data (though

[9] In C, $e$ is unsigned, but large allocations due to conversion from negative numbers are a well-known defect source.
[10] This check disallows pointing just past the end of the array, unlike C.

the lack of qualifier polymorphism [15] causes `const` to be used rarely and often removed via unsafe casts).

Adding this new flavor of pointer type to our model (for both high-level and low-level types) is straightforward:

- There are no changes to the dynamic semantics.
- S-DerefL must require a non-`const` pointer; S-Deref can allow a `const` or non-`const` pointer; and S-Faddr must produce a type with the same qualifier as its subexpression.
- For subtyping rules Ptr, Unroll, and Roll, we can add versions where the two types are both `const`.
- Finally, we add two new subtyping rules to show that `const` permits deep subtyping and allows less access than non-`const`:

$$\frac{D \vdash \overline{\sigma} \leq \overline{\sigma}'}{D \vdash \texttt{const}\ \mathrm{ptr}_\alpha(\overline{\sigma}) \leq \texttt{const}\ \mathrm{ptr}_\alpha(\overline{\sigma}')} \qquad \frac{}{D \vdash \mathrm{ptr}_\alpha(\overline{\sigma}) \leq \texttt{const}\ \mathrm{ptr}_\alpha(\overline{\sigma})}$$

This addition is synergistic with more expressive recursive subtyping, discussed below.

## 4.3 Byte-Skipping

Our dynamic semantics assumes that an assignment copies all bytes of the right-hand value into the corresponding heap location. Actually, C implementations may choose to *skip* pad bytes. In practice, skipping reduces the amount of memory written but can increase the number of store instructions.

Example 6 showed a contrived example where skipping could allow more subtyping. Conversely, although we did not add equality on structures to our expression language, skipping can lead to equality failing because some bytes remain uninitialized (in practice, holding unpredictable bits) despite the struct value being initialized. Though these issues are probably rare, our model is flexible enough to handle them and shed light on the meaning of skipping. To summarize the changes:

- Add a form skip[$i$] to $\sigma$ and let $D \vdash \mathrm{pad}[i] \leq \mathrm{skip}[i]$.
- Change D-Assn to "merge" the new value with the old one by skipping over any "skip bytes" as indicated by $\mathit{impl.xtype}(\overline{t}, \tau)$. That is, we use the type translation to indicate where skipping does and does not occur.
- The new skip type allows additional subtyping under pointers:

$$\frac{D \vdash \overline{\sigma}_2 \leq \mathrm{skip}[i]}{D \vdash \mathrm{ptr}_\alpha(\overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3) \leq \mathrm{ptr}_\alpha(\overline{\sigma}_1 \mathrm{skip}[i]\overline{\sigma}_3)}$$

## 4.4 Recursive Subtyping

Our definition of named struct types ($\overline{t}$) allows recursive types, but for simplicity our definition of physical subtyping is overly restrictive. For example, given two isomorphic structs defining linked-lists of integers, $N_1\{\text{long } f_1;\ N_1 *\ f_2\}$ and $N_2\{\text{long } g_1;\ N_2 *\ g_2\}$, we might have $\mathit{impl.xtype}(\overline{t}, N_1) = \mathrm{byte}^4\mathrm{ptr}_\alpha(N_1)$ and $\mathit{impl.xtype}(\overline{t}, N_2) = \mathrm{byte}^4\mathrm{ptr}_\alpha(N_2)$. We would be unable to show $\mathit{impl}; \overline{t} \vdash \mathrm{byte}^4\mathrm{ptr}_\alpha(N_1) \leq \mathrm{byte}^4\mathrm{ptr}_\alpha(N_2)$.

Sound solutions to this sort of limitation are well-known [3]. For this example, one could maintain a context of valid subtyping assumptions and to derive $\mathrm{ptr}_\alpha(N_1) \leq \mathrm{ptr}_\alpha(N_2)$ one can show the translation of $N_1$ is a subtype of the translation of $N_2$ while assuming $\mathrm{ptr}_\alpha(N_1) \leq \mathrm{ptr}_\alpha(N_2)$.

The issues regarding subtyping recursive types are the usual ones. We have opted for simplicity of presentation over expressiveness. In an actual tool, one should prefer expressiveness.

# 5 Previous Work

To our knowledge, previous work considering implementation-dependent data-layout assumptions or low-level type-safety has taken one of a few approaches:

- Consider only one or two implementations (e.g., verify that one compiler to one architecture produces type-safe code) 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.
- Restrict C, e.g., prohibit dynamic memory allocation.

The first approach is less helpful for writing semi-portable code because we must reverify the code for each implementation. The second approach is good only for (parts of) applications that should be written in a higher-level language. The third and fourth approaches essentially relegate some issues to work such as ours, much as we relegate some issues like array-bounds errors to other work [9, 12].

## 5.1 Assuming an Implementation

Most closely related is the "physical type-checking" work of Chandra et al. [6, 35], 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. Hence checking code against a new implementation would require reverification and changing their tool. They present no metatheory validating their approach.

CCured [31], a memory-safe C implementation, 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 implementations. The formal model establishing CCured's soundness shares similarities with our work (particularly the subtyping between arrays and single-objects), but it lacks a distinct notion of *implementation*, alignment constraints, local variables with `struct` types, memory allocation, `const`, recursive types, etc.

Work on typed assembly language and proof-carrying code [29, 28, 10, 7, 20] 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 [33, 1] 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 semi-portable code because verification of type-safety is repeated for each implementation. In practice, defining a new implementation is an enormous amount of work.

For external data such as network packets [27] or *ad hoc* data streams [14], one can produce parsers directly from explicit bit-by-bit type declarations. While this is a robust solution for external data, it is less appropriate for describing implementation decisions for program data. For semi-portable code, we want to specify some data-representation constraints, but not fix every bit.

## 5.2 Safe C

Memory-safe dialects or implementations of C, such as Cyclone [23, 18, 17], CCured [31, 30, 8], and SAFECode [11], do not solve the semi-portability problem. Rather, they may reject (at compile-time) or terminate

(at run-time) programs that attempt implementation-dependent operations, or they may support only certain implementations (e.g., certain C compilers as back-ends). These approaches are fine for fully portable code or code that is correct assuming particular compilers.

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

### 5.3 Formalizing C

Recent work by Leroy et al. [24, 5] 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, 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 implementation-dependent code).

Norrish's formalization of C [32] uses HOL and includes structs. Like in our work, he uses a global namespace mapping struct names to sequences of typed fields. However, he purposely omits padding and alignment from his formalism. He has no separable notion of an implementation; instead he models implementation choices as non-determinism.

### 5.4 Low-Level Code without C

Our work has been somewhat C-centric, whereas other projects have started with languages at higher levels of abstraction and added bit-level views for low-level programming. (See [4, 19] 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. That is, our model provides a foundation for semi-portable types, which are useful for building low-level but retargetable systems in any language.

C-- [34] 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. It has explicit padding on data (a compiler just inserts bits where desired) and explicit alignment on all memory accesses.[11] 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.

## 6 Conclusions

This work has developed a 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 we can produce via static analysis on a source program. We have proven soundness for a small core language and a simple static analysis, and extended the approach to account for arrays and other language features. Giving implementations a clear identity in our framework clarifies a number of poorly understood issues.

The technique of implementation-as-parameter can apply broadly since high-level languages also have implementation-defined behavior. As examples, SML programs may depend on the size of `int`, Scheme programs may depend on evaluation order, or Java programs may depend on fair thread-scheduling. A program property relying on such assumptions would be stronger than memory-safety (which is always assured); the point is describing implementations via constraints would let us identify a program's assumptions.

Our next step is to complete a practical tool to determine implementation assumptions of real C code. We expect the simple static analysis formalized here will prove too imprecise. However, the problem decomposition in this work still applies; we just replace the "high-level type system" with a more accurate

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

producer of an implementation constraint. In fact, for a bug-finding porting tool, we may wish to sacrifice soundness to reduce false positives.

Another area for future work is a stronger guarantee, namely that a program has similar (ideally equivalent) behavior on a set of implementations. Tackling this problem requires arithmetic reasoning in the presence of overflow and different endianesses.

# 7 Acknowledgments

Matthew Fluet, Greg Morrisett, and David Walker provided excellent feedback on the presentation.

# References

[1] A. Ahmed and D. Walker. The logical approach to stack typing. In *ACM International Workshop on Types in Language Design and Implementation*, pages 74–85, New Orleans, LA, Jan. 2003.

[2] *The ARMLinux Book Online,* Chapter 10. May 2005. http://www.aleph1.co.uk/armlinux/book.

[3] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, Sept. 1993.

[4] D. F. Bacon. Kava: a Java dialect with a uniform object model for lightweight classes. *Concurrency and Computation: Practice and Experience*, 15(3–5):185–206, 2003.

[5] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *14th International Symposium on Formal Methods*, Aug. 2006.

[6] S. Chandra and T. Reps. Physical type checking for C. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 66–75, Toulouse, France, Sept. 1999.

[7] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back-end optimization. In *ACM Conference on Programming Language Design and Implementation*, pages 208–219, San Diego, CA, June 2003.

[8] J. Condit, M. Harren, S. McPeak, G. Necula, and W. Weimer. CCured in the real world. In *ACM Conference on Programming Language Design and Implementation*, pages 232–244, June 2003.

[9] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer-Verlag, Apr. 2005.

[10] K. Crary. Toward a foundational typed assembly language. In *30th ACM Symposium on Principles of Programming Languages*, pages 198–212, New Orleans, LA, Jan. 2003.

[11] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006.

[12] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *ACM Conference on Programming Language Design and Implementation*, pages 155–167, San Diego, CA, June 2003.

[13] P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sørensen, and M. Tofte. AnnoDomini: from type theory to year 2000 conversion tool. In *26th ACM Symposium on Principles of Programming Languages*, pages 1–14, San Antonio, TX, Jan. 1999.

[14] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *33rd ACM Symposium on Principles of Programming Languages*, pages 2–15, Charleston, SC, Jan. 2006.

[15] J. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, GA, May 1999.

[16] D. Grossman. Type-safe multithreading in Cyclone. In *ACM International Workshop on Types in Language Design and Implementation*, pages 13–25, New Orleans, LA, Jan. 2003.

[17] D. Grossman. Quantified types in imperative languages. *ACM Transactions on Programming Languages and Systems*, 28(3):429–475, May 2006.

[18] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002.

[19] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in Haskell. In *10th ACM International Conference on Functional Programming*, pages 116–128, Tallinn, Estonia, Sept. 2005.

[20] 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):191–229, Sept. 2003.

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

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

[23] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[24] X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *33rd ACM Symposium on Principles of Programming Languages*, pages 42–54, Jan. 2006.

[25] R. Love. *Linux Kernel Development, Second Edition.* Novell Press, 2005. Page 328.

[26] B. Martin, A. Rettinger, and J. Singh. Multiplatform porting to 64 bits. *Dr. Dobb's Journal*, Dec. 2005. http://www.ddj.com/184406427.

[27] P. J. McCann and S. Chandra. Packet types: abstract specification of network protocol messages. In *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 321–333, 2000.

[28] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[29] G. Necula. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, Jan. 1997.

[30] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages*, pages 128–139, Portland, OR, Jan. 2002.

[31] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, May 2005.

[32] M. Norrish. *C formalised in HOL.* PhD thesis, University of Cambridge, 1998.

[33] L. Petersen, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout. In *30th ACM Symposium on Principles of Programming Languages*, pages 172–184, New Orleans, LA, Jan. 2003.

[34] N. Ramsey, S. P. Jones, and C. Lindig. The C-- language specification version 2.0, Feb. 2005. http://www.cminusminus.org/extern/man2.pdf.

[35] 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*, pages 180–198, Toulouse, France, Sept. 1999.

[36] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

# A Safety Proof

We use the notation $\overline{x}$ to denote a sequence of objects drawn from the syntactic class $x$, the notation $x^i$ to denote a sequence of length $i$, and let $\cdot$ denote the empty sequence. Figure 10 gives the syntax for a C-like source language. Figure 12 gives an extended, "low-level" version of this language, where the types are replaced by types at the byte-sequence level, runtime values are added, and syntax for heaps, heap typings, and evaluation contexts is also shown. Figure 13 gives the dynamic semantics for the low-level language and Figure 14 defines a set of value and type size functions used throughout the low-level semantics. Figures 15 and 16 give the static semantics for the low-level language. Figure 17 gives the static semantics for the high-level language.

## A.1 Implementations

Our dynamic and low-level static semantics are parameterized by *implementations*. An implementation is an oracle that guides the semantics in various ways. Each implementation is a record of six functions, as follows:

- xtype : $\overline{t} \times \tau \to \overline{\sigma}$: Translates a high-level type into a low-level type.
- align : $\overline{t} \times \tau \to \alpha$: Gives the alignment of type $\tau$.
- offset : $\overline{t} \times f \to \mathbb{N}$: Gives the offset of field $f$. Without loss of generality, we assume that all fields in $\overline{t}$ are uniquely named.
- access : $\alpha \times \mathbb{N} \to \{\text{true}, \text{false}\}$: $\mathit{impl}.\text{access}(\alpha, k)$ determines whether it is okay to access a memory chunk of size $k$ at alignment $\alpha$ on implementation *impl*.
- xlit : $\{s, l\} \to \overline{w}$: Translates a literal expression into a sequence of bytes.
- ptr_size : $\mathbb{N}$: Gives the size of pointers. We limit our focus to implementations on which all pointers have the same size.

## A.2 Constraint Language

The high-level semantics (Figure 17) is a type and effect system that, in addition to a type, outputs a constraint $S$. The language used to express $S$ is a multi-sort first-order logic in which each of the $\overline{t}, \tau, \overline{\sigma}, f, \alpha, \mathbb{N}$ syntactic classes is assigned a corresponding sort. Moreover, the logic is enriched with function symbols corresponding to each implementation component plus a couple of others. Implementations are models in this theory; we write $\mathit{impl} \models S$ when *impl* satisfies formula $S$. The $(\models)$ judgment is entirely ordinary, defined inductively over the structure of $S$. For example, $\mathit{impl} \models S_1 \wedge S_2$ exactly when $\mathit{impl} \models S_1$ and

| | | | |
|---|---|---|---|
| (types) | $\tau$ | ::= | short \| long \| $\tau*$ \| $N$ |
| (declarations) | $t$ | ::= | $N\{\overline{\tau\ f}\}$ |
| (expressions) | $e$ | ::= | $s \mid l \mid x \mid e = e \mid e.f \mid *(\tau*)(e) \mid \mathsf{new}\ \tau \mid (\tau*)e \mid (\tau*)\&e{\rightarrow}f$ |
| | | \| | $e; e \mid \mathsf{if}\ e\ e\ e \mid \mathsf{while}\ e\ e \mid \tau\ x;\ e$ |
| (contexts) | $\Gamma$ | ::= | $\overline{x{:}\tau}$ |

Figure 10: Syntax for the Source Language

| symbol | interpretation under *impl* |
|---|---|
| xtype$(\bar{t}, \tau)$ | *impl*.xtype$(\bar{t}, \tau)$ |
| align$(\bar{t}, \tau)$ | *impl*.align$(\bar{t}, \tau)$ |
| offset$(\bar{t}, f)$ | *impl*.offset$(\bar{t}, f)$ |
| access$(\alpha, i)$ | *impl*.access$(\alpha, i)$ |
| xlit$(s)$ | *impl*.xlit$(s)$ |
| xlit$(l)$ | *impl*.xlit$(l)$ |
| ptr_size | *impl*.ptr_size |
| size$(\overline{\sigma})$ | size$(impl, \overline{\sigma})$ |
| size$(\overline{w})$ | size$(impl, \overline{w})$ |
| subtype$(\bar{t}, \overline{\sigma}_1, \overline{\sigma}_2)$ | $impl; \bar{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ is derivable |
| subalign$(\alpha_1, \alpha_2)$ | $\vdash \alpha_1 \leq \alpha_2$ is derivable |

Figure 11: Constraint Language Function Symbols

$impl \models S_2$; and $impl \models \forall x : s.S$ exactly when for all objects $t$ of sort $s$, $S[t/x]$ is true, where $S[t/x]$ denotes the capture-avoiding substitution of $t$ for $x$ into $S$. We generally omit explicit sorts on quantified variables; it is clear what is meant from the context.

Figure 11 summarizes the function symbols and their interpretation under *impl*. As a final example, involving function symbols, suppose we have a constraint $S = (\text{align}(\bar{t}, N) = [a, o] \wedge \text{align}(\bar{t}, \tau) = [a, o + \text{offset}(\bar{t}, f)])$ and want to know whether an implementation *impl* satisfies $S$ (alternatively, *impl* models $S$). *impl* satisfies $S$ iff $impl \models \text{align}(\bar{t}, N) = [a, o]$ and $impl \models \text{align}(\bar{t}, \tau) = [a, o + \text{offset}(\bar{t}, f)]$. To satisfy the former conjunct, $impl.\text{align}(\bar{t}, N) = [a, o]$ must be true. To satisfy the latter, $impl.\text{align}(\bar{t}, \tau) = [a, o + impl.\text{offset}(\bar{t}, f)]$ must be true.

## A.3 Metatheory

**Definition 1 (Sensible Implementations)**
*An implementation impl is said to be sensible if*

1. $\forall \tau.$ $impl.\text{access}(impl.\text{align}(\bar{t}, \tau), \text{size}(impl, impl.\text{xtype}(\bar{t}, \tau)))$.
2. $\exists i.$ $impl.\text{xtype}(\bar{t}, \text{short}) = \text{byte}^i$ *and* $\forall s \exists \bar{b}.$ $impl.\text{xlit}(s) = \bar{b}$ *and* $\text{size}(impl, \bar{b}) = i$.
   $\exists i.$ $impl.\text{xtype}(\bar{t}, \text{long}) = \text{byte}^i$ *and* $\forall s \exists \bar{b}.$ $impl.\text{xlit}(l) = \bar{b}$ *and* $\text{size}(impl, \bar{b}) = i$.
3. $\forall \bar{t}.$ *if* $N\{\ldots \tau\ f \ldots\} \in \bar{t}$ *and* $impl.\text{xtype}(\bar{t}, \tau) = \overline{\sigma}_2$ *then* $\exists \overline{\sigma}_1, \overline{\sigma}_3, a, o, \alpha$ *such that* $impl.\text{xtype}(\bar{t}, N) = \overline{\sigma}_1 \overline{\sigma}_2 \overline{\sigma}_3$ *where* $impl.\text{offset}(\bar{t}, f) = \text{size}(impl, \overline{\sigma}_1)$, $impl.\text{align}(\bar{t}, N) = [a, o]$, $impl.\text{align}(\bar{t}, \tau) = \alpha$, *and* $\vdash [a, o + impl.\text{offset}(\bar{t}, f)] \leq \alpha$.

   The reason we have the "subalignment" requirement here, rather than equality, is to admit more implementations without sacrificing soundness; otherwise we would be unnecessarily restrictive. For example, suppose $impl.\text{align}(\bar{t}, N) = [8, 0]$, $impl.\text{offset}(\bar{t}, f) = 8$, and $impl.\text{align}(\bar{t}, \tau) = [4, 0]$. Then, clause (3) would require $\tau$ to be 8-byte aligned (alignment $[8, 0 + impl.\text{offset}(\bar{t}, f)] \equiv [8, 8] \equiv [8, 0]$), which is unreasonable. It is okay for $\tau$ to be 4-byte, 2-byte, or 1-byte aligned.

$$
\begin{array}{rrcl}
\text{(types)} & \sigma & ::= & \text{byte} \mid \text{pad}[i] \mid \text{ptr}_\alpha(\overline{\sigma}) \mid \text{ptr}_\alpha(N) \\
\text{(alignments)} & \alpha & ::= & [a, o] \\
\text{(atomic values)} & w & ::= & b \mid \ell{+}i \mid \mathsf{uninit} \\
\text{(bytes)} & b & ::= & 0 \mid 1 \mid \ldots \mid 255 \\
\text{(expressions)} & e & ::= & \ldots \mid \overline{w} \\
\text{(values)} & v & ::= & \overline{w} \\
\text{(heaps)} & H & ::= & \overline{\ell \mapsto v, \alpha} \\
\text{(heap typings)} & \Psi & ::= & \overline{\ell{:}\overline{\sigma}, \alpha} \\
 & D & ::= & \mathit{impl}; \overline{t} \\
 & C & ::= & \Psi; \Gamma \\
 & i, j, k, a, o & \in & \mathbb{N} \\
 & & & \\
\text{(left contexts)} & \mathsf{L} & ::= & [\cdot]_{\mathsf{L}} \mid \mathsf{L}.f \mid *(\tau*)(\mathsf{R}) \\
\text{(right contexts)} & \mathsf{R} & ::= & [\cdot]_{\mathsf{R}} \mid \mathsf{L} = e \mid *(\tau*)(\ell{+}i) = \mathsf{R} \mid \mathsf{R}.f \mid *(\tau*)(\mathsf{R}) \mid (\tau*)\mathsf{R} \\
 & & \mid & \mathsf{R}; e \mid (\tau*)\&\mathsf{R}{\to}f \mid \mathsf{if}\ \mathsf{R}\ e\ e
\end{array}
$$

Figure 12: Syntax for the Full Language (Extends Source Language) and Eval Contexts

4. $\forall \tau \exists \alpha, \overline{\sigma}.$ *impl*.xtype$(\overline{t}, \tau*) = \text{ptr}_\alpha(\overline{\sigma})$ *and* *impl*.xtype$(\overline{t}, \tau) = \overline{\sigma}$ *and* *impl*.align$(\overline{t}, \tau) = \alpha$.
5. *If* *impl*.access$(\alpha, i)$ *and* $\vdash \alpha' \leq \alpha$ *then* *impl*.access$(\alpha', i)$.

**Definition 2 (Substitution)**

$$
\begin{array}{rcll}
s\{e/x\} & = & s & \\
l\{e/x\} & = & l & \\
x\{e/x\} & = & e & \\
y\{e/x\} & = & y & \\
\overline{w}\{e/x\} & = & \overline{w} & \\
(e_1 = e_2)\{e/x\} & = & e_1\{e/x\} = e_2\{e/x\} & \\
(e_1.f)\{e/x\} & = & (e_1\{e/x\}).f & \\
(*(\tau*)(e_1))\{e/x\} & = & *(\tau*)((e_1\{e/x\})) & \\
(\mathsf{new}\ \tau)\{e/x\} & = & \mathsf{new}\ \tau & \\
((\tau*)e_1)\{e/x\} & = & (\tau*)(e_1\{e/x\}) & \\
((\tau*)\&e_1{\to}f)\{e/x\} & = & (\tau*)\&(e_1\{e/x\}){\to}f & \\
(e_1; e_2)\{e/x\} & = & e_1\{e/x\}; e_2\{e/x\} & \\
(\mathsf{if}\ e_1\ e_2\ e_3)\{e/x\} & = & \mathsf{if}\ (e_1\{e/x\})\ (e_2\{e/x\})\ (e_3\{e/x\}) & \\
(\mathsf{while}\ e_1\ e_2)\{e/x\} & = & \mathsf{while}\ (e_1\{e/x\})\ (e_2\{e/x\}) & \\
(\tau\ y;\ e_1)\{e/x\} & = & \tau\ z;\ (e_1\{z/y\})\{e/x\} & (z \text{ fresh})
\end{array}
$$

**Definition 3 (Legal Stuck State)** *Let*

$$
\begin{array}{rcl}
\mathit{rstuck} & ::= & \mathsf{if}\ \overline{w}_1\ \mathsf{uninit}\ \overline{w}_2\ e\ e \mid *(\tau*)(\mathsf{uninit}^i) \mid (\tau*)\&\mathsf{uninit}^i{\to}f \\
\mathit{lstuck} & ::= & *(\tau*)(\mathsf{uninit}^i)
\end{array}
$$

*A state* $H; e$ *is* <u>*legally stuck*</u> *if one of the following is true:*

- $e = \mathsf{R}[\mathit{lstuck}]_{\mathsf{l}}$
- $e = \mathsf{R}[\mathit{rstuck}]_{\mathsf{r}}$
- $e = \mathsf{L}[\mathit{lstuck}]_{\mathsf{l}}$
- $e = \mathsf{L}[\mathit{rstuck}]_{\mathsf{r}}$

$$
\begin{array}{lllll}
\text{(D-Cast)} & D \vdash H; (\tau*)\overline{w} & \xrightarrow{\mathsf{r}} & H; \overline{w} & \\
\text{(D-Short)} & D \vdash H; s & \xrightarrow{\mathsf{r}} & H; \overline{b} & \\
 & & & \quad \text{if} \quad impl.\mathrm{xlit}(s) = \overline{b} & \\
\text{(D-Long)} & D \vdash H; l & \xrightarrow{\mathsf{r}} & H; \overline{b} & \\
 & & & \quad \text{if} \quad impl.\mathrm{xlit}(l) = \overline{b} & \\
\text{(D-Seq)} & D \vdash H; (v; e) & \xrightarrow{\mathsf{r}} & H; e & \\
\text{(D-IfF)} & D \vdash H; \mathsf{if}\ 0^i\ e_1\ e_2 & \xrightarrow{\mathsf{r}} & H; e_2 & \\
\text{(D-IfT)} & D \vdash H; \mathsf{if}\ b_1 \ldots b_i\ e_1\ e_2 & \xrightarrow{\mathsf{r}} & H; e_1 & \\
 & & & \quad \text{if} \quad b_1 \ldots b_i \neq 0^i & \\
\text{(D-While)} & 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 & \\
\text{(D-New)} & impl; \overline{t} \vdash H; \mathsf{new}\ \tau & \xrightarrow{\mathsf{r}} & H, \ell \mapsto \mathsf{uninit}^i, \alpha; \ell\mathsf{+}0 & \\
 & & & \quad \text{if} \quad \ell \notin \mathrm{Dom}(H) & \\
 & & & \qquad impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma} & \\
 & & & \qquad \mathrm{size}(impl, \overline{\sigma}) = i & \\
 & & & \qquad impl.\mathrm{align}(\overline{t}, \tau) = \alpha & \\
\text{(D-Decl)} & impl; \overline{t} \vdash H; \tau\ x;\ e & \xrightarrow{\mathsf{r}} & H, \ell \mapsto \mathsf{uninit}^i, \alpha; e\{*(\tau*)(\ell\mathsf{+}0)/x\} & \\
 & & & \quad \text{if} \quad \ldots \text{same as above} \ldots & \\
\text{(D-Fetch)} & impl; \overline{t} \vdash H; \overline{w}_1\overline{w}_2\overline{w}_3.f & \xrightarrow{\mathsf{r}} & H; \overline{w}_2 & \\
 & & & \quad \text{if} \quad impl.\mathrm{offset}(\overline{t}, f) = \mathrm{size}(impl, \overline{w}_1) & \\
 & & & \qquad N\{\ldots \tau\ f \ldots\} \in \overline{t} & \\
 & & & \qquad impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma} & \\
 & & & \qquad \mathrm{size}(impl, \overline{\sigma}) = \mathrm{size}(impl, \overline{w}_2) & \\
\text{(D-FAddr)} & impl; \overline{t} \vdash H; (\tau*)\&\ell\mathsf{+}j \to f & \xrightarrow{\mathsf{r}} & H; \ell\mathsf{+}(j + impl.\mathrm{offset}(\overline{t}, f)) & \\
\text{(D-Deref)} & impl; \overline{t} \vdash H; *(\tau*)(\ell\mathsf{+}j) & \xrightarrow{\mathsf{r}} & H; \overline{w}_2 & \\
 & & & \quad \text{if} \quad H(\ell) = \overline{w}_1\overline{w}_2\overline{w}_3, [a, o] & \\
 & & & \qquad \mathrm{size}(impl, \overline{w}_1) = j & \\
 & & & \qquad impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma} & \\
 & & & \qquad \mathrm{size}(impl, \overline{w}_2) = \mathrm{size}(impl, \overline{\sigma}) = k & \\
 & & & \qquad impl.\mathrm{access}([a, o + j], k) & \\
\text{(D-Assn)} & impl; \overline{t} \vdash H; *(\tau*)(\ell\mathsf{+}j) = \overline{w} & \xrightarrow{\mathsf{r}} & H, \ell \mapsto \overline{w}_1\overline{w}\,\overline{w}_3, [a, o]; \overline{w} & \\
 & & & \quad \text{if} \quad \ldots \text{same as above} \ldots & \\
 & & & \qquad \mathrm{size}(impl, \overline{w}) = \mathrm{size}(impl, \overline{w}_2) & \\
\text{(D-FetchL)} & impl; \overline{t} \vdash H; (*(\tau_1*)(\ell\mathsf{+}j)).f & \xrightarrow{\mathsf{l}} & H; *(\tau_2*)(\ell\mathsf{+}(j + j')) & \\
 & & & \quad \text{if} \quad impl.\mathrm{offset}(f) = j' & \\
 & & & \qquad N\{\ldots \tau_2\ f \ldots\} \in \overline{t} &
\end{array}
$$

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

Figure 13: Dynamic Semantics

$$
\begin{array}{lcl}
\text{size} & : & (\mathit{impl} \times \sigma) \to \mathbb{N} \\
\text{size}(\mathit{impl}, \sigma) & = & \begin{cases} 1 & \text{if } \sigma = \text{byte} \\ i & \text{if } \sigma = \text{pad}[i] \\ \mathit{impl}.\text{ptr\_size} & \text{if } \sigma \in \{\text{ptr}_\alpha(N), \text{ptr}_\alpha(\overline{\sigma})\} \end{cases} \\
 & & \\
\text{size} & : & (\mathit{impl} \times \overline{\sigma}) \to \mathbb{N} \\
\text{size}(\mathit{impl}, \sigma_1 \ldots \sigma_n) & = & \sum_{i=1}^{n} \text{size}(\mathit{impl}, \sigma_i) \\
 & & \\
\text{size} & : & (\mathit{impl} \times w) \to \mathbb{N} \\
\text{size}(\mathit{impl}, w) & = & \begin{cases} 1 & \text{if } w \in \{b, \mathsf{uninit}\} \\ \mathit{impl}.\text{ptr\_size} & \text{if } w = \ell\text{+}i \end{cases} \\
 & & \\
\text{size} & : & (\mathit{impl} \times \overline{w}) \to \mathbb{N} \\
\text{size}(\mathit{impl}, w_1 \ldots w_n) & = & \sum_{i=1}^{n} \text{size}(\mathit{impl}, w_i)
\end{array}
$$

Figure 14: Size Functions

**Definition 4 (Illegal Stuck State)**
*A state $H; e$ is illegally stuck on an implementation impl and declarations $\overline{t}$ if $H; e$ is* not *legally stuck, $e$ is not a value, and there exist no $H'$ and $e'$ such that $\mathit{impl}; \overline{t} \vdash H; e \to H'; e'$.*

**Definition 5 (Heap Typing Extension)**
*A heap typing $\Psi$ extends a heap typing $\Psi'$ iff there exists $\Psi''$ such that $\Psi = \Psi'\Psi''$.*

### A.3.1 Type Soundness

**Lemma 6 (Weakening)**

1. *(a) If $D; \Psi; \Gamma \vdash_{\text{r}} e : \overline{\sigma}$ then $D; \Psi\Psi'; \Gamma\Gamma' \vdash_{\text{r}} e : \overline{\sigma}$.*
   *(b) If $D; \Psi; \Gamma \vdash_{\text{l}} e : \overline{\sigma}, \alpha$ then $D; \Psi\Psi'; \Gamma\Gamma' \vdash_{\text{l}} e : \overline{\sigma}, \alpha$.*
2. *If $D; \Psi \vdash H : \Psi'$ then $D; \Psi\Psi'' \vdash H : \Psi'$.*

**Proof:**

1. By simultaneous induction on the assumed typing derivations, using the facts that if $\ell \in \text{Dom}(\Psi)$ then $(\Psi\Psi')(\ell) = \Psi(\ell)$, and if $x \in \text{Dom}(\Gamma)$ then $(\Gamma\Gamma')(x) = \Gamma(x)$.
2. By straightforward induction on the assumed heap typing derivation, using the result of part (1).

**Lemma 7 (Heap Canonical Forms)**
*If $D; \Psi \vdash H : \Psi$ and $\Psi(\ell) = \overline{\sigma}, \alpha$, then $\exists \overline{w}$ such that $H(\ell) = \overline{w}, \alpha$ and $D; \Psi; \cdot \vdash_{\text{r}} \overline{w} : \overline{\sigma}$.*

**Proof:** By straightforward induction on the assumed heap typing derivation.

**Lemma 8 (Uninit Type)**
*If $\text{size}(\mathit{impl}, \overline{\sigma}) = i$ then $\mathit{impl}; \overline{t}; \Psi; \cdot \vdash_{\text{r}} \mathsf{uninit}^i : \overline{\sigma}$.*

**Proof:** By induction on the length of $\overline{\sigma}$. In the base case, $\overline{\sigma} = \cdot$ and the claim follows immediately. In the inductive case, $\overline{\sigma} = \sigma\overline{\sigma}'$ where by induction we have $\text{size}(\mathit{impl}, \overline{\sigma}') = j$ and

**1** $\mathit{impl}; \overline{t}; \Psi; \cdot \vdash_{\text{r}} \mathsf{uninit}^j : \overline{\sigma}'$

L-Short
$$\frac{impl.\text{xtype}(\overline{t}, \text{short}) = \text{byte}^i}{impl; \overline{t}; C \vdash_{\text{r}} s : \text{byte}^i}$$

L-Long
$$\frac{impl.\text{xtype}(\overline{t}, \text{long}) = \text{byte}^i}{impl; \overline{t}; C \vdash_{\text{r}} l : \text{byte}^i}$$

L-VarR
$$\frac{\Gamma(x) = \tau \qquad impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}}{impl; \overline{t}; \Psi; \Gamma \vdash_{\text{r}} x : \overline{\sigma}}$$

L-VarL
$$\frac{\Gamma(x) = \tau \qquad impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma} \qquad impl.\text{align}(\overline{t}, \tau) = \alpha}{impl; \overline{t}; \Psi; \Gamma \vdash_{\text{l}} x : \overline{\sigma}, \alpha}$$

L-Assn
$$\frac{D; C \vdash_{\text{l}} e_1 : \overline{\sigma}, \alpha \qquad D; C \vdash_{\text{r}} e_2 : \overline{\sigma}}{D; C \vdash_{\text{r}} e_1 = e_2 : \overline{\sigma}}$$

L-FetchR
$$\frac{impl; \overline{t}; C \vdash_{\text{r}} e : \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3 \qquad impl.\text{offset}(f) = \text{size}(impl, \overline{\sigma}_1) \qquad N\{\ldots\ \tau\ f \ldots\} \in \overline{t} \qquad impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}_2}{impl; \overline{t}; C \vdash_{\text{r}} e.f : \overline{\sigma}_2}$$

L-FetchL
$$\frac{\begin{array}{c} impl; \overline{t}; C \vdash_{\text{l}} e : \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3, [a, o] \qquad impl.\text{offset}(\overline{t}, f) = \text{size}(impl, \overline{\sigma}_1) = o' \\ N\{\ldots\ \tau\ f \ldots\} \in \overline{t} \qquad impl.\text{align}(\overline{t}, N) = \alpha \qquad \vdash [a, o] \leq \alpha \qquad impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}_2 \end{array}}{impl; \overline{t}; C \vdash_{\text{l}} e.f : \overline{\sigma}_2, [a, o + o']}$$

L-Deref{R,L}
$$\frac{impl; \overline{t}; C \vdash_{\text{r}} e : \text{ptr}_{\alpha}(\overline{\sigma}_1\overline{\sigma}_2) \qquad impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}_1 \qquad impl.\text{access}(\alpha, \text{size}(impl, \overline{\sigma}_1))}{\begin{array}{c} impl; \overline{t}; C \vdash_{\text{r}} *(\tau*)(e) : \overline{\sigma}_1 \\ impl; \overline{t}; C \vdash_{\text{l}} *(\tau*)(e) : \overline{\sigma}_1, \alpha \end{array}}$$

L-New
$$\frac{impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma} \qquad impl.\text{align}(\overline{t}, \tau) = \alpha}{impl; \overline{t}; C \vdash_{\text{r}} \mathsf{new}\ \tau : \text{ptr}_{\alpha}(\overline{\sigma})}$$

L-Cast
$$\frac{impl; \overline{t}; C \vdash_{\text{r}} e : \text{ptr}_{\alpha}(\overline{\sigma})}{impl; \overline{t}; C \vdash_{\text{r}} (\tau*)e : \text{ptr}_{\alpha}(\overline{\sigma})}$$

L-FAddr
$$\frac{impl; \overline{t}; C \vdash_{\text{r}} e : \text{ptr}_{[a,o_1]}(\overline{\sigma}_1\overline{\sigma}_2) \qquad impl.\text{offset}(f) = \text{size}(impl, \overline{\sigma}_1)}{impl; \overline{t}; C \vdash_{\text{r}} (\tau*)\&e \rightarrow f : \text{ptr}_{[a,o_2]}(\overline{\sigma}_2)}$$

L-Seq
$$\frac{D; C \vdash_{\text{r}} e_1 : \overline{\sigma}' \qquad D; C \vdash_{\text{r}} e_2 : \overline{\sigma}}{D; C \vdash_{\text{r}} e_1; e_2 : \overline{\sigma}}$$

L-Decl
$$\frac{D; \Psi; \Gamma, x : \tau \vdash_{\text{r}} e : \overline{\sigma}}{D; \Psi; \Gamma \vdash_{\text{r}} \tau\ x;\ e : \overline{\sigma}}$$

L-If
$$\frac{impl; \overline{t}; C \vdash_{\text{r}} e_1 : \text{byte}^i \qquad impl; \overline{t}; C \vdash_{\text{r}} e_2 : \overline{\sigma} \qquad impl; \overline{t}; C \vdash_{\text{r}} e_3 : \overline{\sigma}}{impl; \overline{t}; C \vdash_{\text{r}} \mathsf{if}\ e_1\ e_2\ e_3 : \overline{\sigma}}$$

L-While
$$\frac{impl; \overline{t}; C \vdash_{\text{r}} e_1 : \text{byte}^j \qquad impl; \overline{t}; C \vdash_{\text{r}} e_2 : \overline{\sigma}_2 \qquad impl.\text{xtype}(\overline{t}, \text{short}) = \text{byte}^i}{impl; \overline{t}; C \vdash_{\text{r}} \mathsf{while}\ e_1\ e_2 : \text{byte}^i}$$

L-Value
$$\frac{D; \Psi \vdash_{\text{w}} \overline{w}_1 : \sigma_1 \qquad D; \Psi; \Gamma \vdash_{\text{r}} \overline{w}_2 : \overline{\sigma}_2}{D; \Psi; \Gamma \vdash_{\text{r}} \overline{w}_1\overline{w}_2 : \sigma_1\overline{\sigma}_2}$$

L-ValueEmpty
$$\frac{}{D; \Psi \vdash_{\text{r}} \cdot : \cdot}$$

L-Sub
$$\frac{D; C \vdash_{\text{r}} e : \overline{\sigma}_1 \qquad D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2}{D; C \vdash_{\text{r}} e : \overline{\sigma}_2}$$

Figure 15: Static Semantics for the Low-Level Language

Heap Typing

$$\textsc{Ht-Ax}\ \frac{}{D; \Psi \vdash \cdot : \cdot} \qquad \textsc{Ht-Inf}\ \frac{D; \Psi \vdash H : \Psi' \qquad D; \Psi; \cdot \models_{\mathrm{r}} v : \overline{\sigma}}{D; \Psi \vdash H, \ell \mapsto v, \alpha : \Psi', \ell : \overline{\sigma}, \alpha}$$

Alignment Subtyping

$$\textsc{Alst-Off}\ \frac{o_1 \equiv o_2 \mod a}{\vdash [a, o_1] \leq [a, o_2]} \qquad \textsc{Alst-Base}\ \frac{a_1 = a_2 \times k}{\vdash [a_1, o] \leq [a_2, o]} \qquad \textsc{Alst-Trans}\ \frac{\vdash \alpha_1 \leq \alpha_2 \qquad \vdash \alpha_2 \leq \alpha_3}{\vdash \alpha_1 \leq \alpha_3}$$

Physical Subtyping

$$\textsc{St-Unroll}\ \frac{impl.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}}{impl; \overline{t} \vdash \mathrm{ptr}_\alpha(N) \leq \mathrm{ptr}_\alpha(\overline{\sigma})} \qquad \textsc{St-Roll}\ \frac{impl.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}}{impl; \overline{t} \vdash \mathrm{ptr}_\alpha(\overline{\sigma}) \leq \mathrm{ptr}_\alpha(N)} \qquad \textsc{St-Ptr}\ \frac{\vdash \alpha_1 \leq \alpha_2}{D \vdash \mathrm{ptr}_{\alpha_1}(\overline{\sigma}_1\overline{\sigma}_2) \leq \mathrm{ptr}_{\alpha_2}(\overline{\sigma}_1)}$$

$$\textsc{St-Pad}\ \frac{\mathrm{size}(impl, \sigma) = i}{impl; \overline{t} \vdash \sigma \leq \mathrm{pad}[i]} \qquad \textsc{St-PadAdd}\ \frac{}{impl; \overline{t} \vdash \mathrm{pad}[i]\mathrm{pad}[j] \leq \mathrm{pad}[i+j]} \qquad \textsc{St-Refl}\ \frac{}{D \vdash \overline{\sigma} \leq \overline{\sigma}}$$

$$\textsc{St-Seq}\ \frac{D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2 \qquad D \vdash \overline{\sigma}_3 \leq \overline{\sigma}_4}{D \vdash \overline{\sigma}_1\overline{\sigma}_3 \leq \overline{\sigma}_2\overline{\sigma}_4} \qquad \textsc{St-Trans}\ \frac{D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2 \qquad D \vdash \overline{\sigma}_2 \leq \overline{\sigma}_3}{D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_3}$$

Value Typing

$$\textsc{Lw-Byte}\ \frac{}{D; \Psi \models_{\mathrm{w}} b : \mathrm{byte}} \qquad \textsc{Lw-Lbl}\ \frac{\Psi(\ell) = \overline{\sigma}_1\overline{\sigma}_2, [a, o] \qquad i = \mathrm{size}(impl, \overline{\sigma}_1)}{impl; \overline{t}; \Psi \models_{\mathrm{w}} \ell{+}i : \mathrm{ptr}_{[a, o+i]}(\overline{\sigma}_2)} \qquad \textsc{Lw-Uninit1}\ \frac{}{D; \Psi \models_{\mathrm{w}} \mathsf{uninit} : \mathrm{byte}}$$

$$\textsc{Lw-Uninit2}\ \frac{impl.\mathrm{ptr\_size} = i}{impl; \overline{t}; \Psi \models_{\mathrm{w}} \mathsf{uninit}^i : \mathrm{ptr}_\alpha(\overline{\sigma})}$$

Figure 16: Auxiliary Relations

If $\mathrm{size}(impl, \sigma) = k$ then we can derive

**2** $impl; \overline{t}; \Psi; \cdot \models_{\mathrm{w}} \mathsf{uninit}^k : \sigma$

If $\sigma$ is one of (byte,$\mathrm{ptr}_\alpha(\overline{\sigma})$), this is immediate from Lw-Uninit1 and Lw-Uninit2. For $\mathrm{pad}[i]$ we can use Lw-Uninit1, St-Pad, and St-PadAdd $k$ times. Plugging (1,2) into L-Value yields the desired conclusion. Finally, if $\sigma$ is $\mathrm{ptr}_\alpha(N)$, we can use Lw-Uninit2 to derive $impl; \overline{t}; \Psi; \cdot \models_{\mathrm{r}} \mathsf{uninit}^i : \mathrm{ptr}_\alpha(\overline{\sigma})$ where $impl.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}$, then use St-Roll and L-Sub to assign type $\mathrm{ptr}_\alpha(N)$ to $\mathsf{uninit}^i$.

**Lemma 9 (Constant-Size Subtyping)**
*If* $D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ *then* $\mathrm{size}(impl, \overline{\sigma}_1) = \mathrm{size}(impl, \overline{\sigma}_2)$.

**Proof:** By induction on the assumed subtyping derivation, by cases on the last rule used.

- St-Roll, St-Unroll, St-Ptr: Two pointers have the same size.
- St-Pad: $\mathrm{size}(impl, \sigma) = \mathrm{size}(impl, \mathrm{pad}[i]) = i$.

$$\text{H-Short}\quad \frac{}{\overline{t};\Gamma \vdash_{\mathrm{r}} i : \mathrm{short};\top} \qquad \text{H-Long}\quad \frac{}{\overline{t};\Gamma \vdash_{\mathrm{r}} d : \mathrm{long};\top} \qquad \text{H-New}\quad \frac{}{\overline{t};\Gamma \vdash_{\mathrm{r}} \mathsf{new}\ \tau : \tau*;\top}$$

$$\text{H-Var}\{\mathrm{R,L}\}\quad \frac{\Gamma(x) = \tau}{\overline{t};\Gamma \vdash_{\mathrm{r}} x : \tau;\top \quad \overline{t};\Gamma \vdash_{\mathrm{l}} x : \tau;\top}$$

$$\text{H-Assn}\quad \frac{\overline{t};\Gamma \vdash_{\mathrm{l}} e_1 : \tau; S_1 \qquad \overline{t};\Gamma \vdash_{\mathrm{r}} e_2 : \tau; S_2}{\overline{t};\Gamma \vdash_{\mathrm{r}} e_1 = e_2 : \tau; S_1 \wedge S_2}$$

$$\text{H-FetchR}\quad \frac{N\{\ldots \tau\ f \ldots\} \in \overline{t} \qquad \overline{t};\Gamma \vdash_{\mathrm{r}} e : N; S}{\overline{t};\Gamma \vdash_{\mathrm{r}} e.f : \tau; S}$$

$$\text{H-FetchL}\quad \frac{N\{\ldots \tau\ f \ldots\} \in \overline{t} \qquad \overline{t};\Gamma \vdash_{\mathrm{l}} e : N; S}{\overline{t};\Gamma \vdash_{\mathrm{l}} e.f : \tau; S}$$

$$\text{H-Seq}\quad \frac{\overline{t};\Gamma \vdash_{\mathrm{r}} e_1 : \tau'; S_1 \qquad \overline{t};\Gamma \vdash_{\mathrm{r}} e_2 : \tau; S_2}{\overline{t};\Gamma \vdash_{\mathrm{r}} e_1; e_2 : \tau; S_1 \wedge S_2}$$

$$\text{H-Deref}\{\mathrm{R,L}\}\quad \frac{\overline{t};\Gamma \vdash_{\mathrm{r}} e : \tau*; S}{\overline{t};\Gamma \vdash_{\mathrm{r}} *(\tau*)(e) : \tau; S \quad \overline{t};\Gamma \vdash_{\mathrm{l}} *(\tau*)(e) : \tau; S}$$

$$\text{H-Cast}\quad \frac{\overline{t};\Gamma \vdash_{\mathrm{r}} e : \tau*; S}{\overline{t};\Gamma \vdash_{\mathrm{r}} (\tau'*)e : \tau'*; S \wedge \mathrm{subtype}(\mathrm{xtype}(\overline{t}, \tau*), \mathrm{xtype}(\overline{t}, \tau'*))}$$

$$\text{H-Faddr}\quad \frac{N\{\ldots \tau'\ f \ldots\} \in \overline{t} \qquad \overline{t};\Gamma \vdash_{\mathrm{r}} e : N*; S}{\begin{array}{l}\overline{t};\Gamma \vdash_{\mathrm{r}} (\tau*)(\&e \to f) : \tau*; S \wedge \exists \overline{\sigma}_1, \overline{\sigma}_2, a, o \quad . \quad \mathrm{xtype}(\overline{t}, N*) = \mathrm{ptr}_{[a,o]}(\overline{\sigma}_1 \overline{\sigma}_2) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge \quad \mathrm{subtype}(\overline{t}, \mathrm{ptr}_{[a,o+\mathrm{offset}(\overline{t},f)]}(\overline{\sigma}_2), \mathrm{xtype}(\overline{t}, \tau*)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge \quad \mathrm{offset}(\overline{t}, f) = \mathrm{size}(\overline{\sigma}_1)\end{array}}$$

$$\text{H-If}\quad \frac{\overline{t};\Gamma \vdash_{\mathrm{r}} e_1 : \mathrm{long}; S_1 \qquad \overline{t};\Gamma \vdash_{\mathrm{r}} e_2 : \tau; S_2 \qquad \overline{t};\Gamma \vdash_{\mathrm{r}} e_3 : \tau; S_3}{\overline{t};\Gamma \vdash_{\mathrm{r}} \mathsf{if}\ e_1\ e_2\ e_3 : \tau; S_1 \wedge S_2 \wedge S_3}$$

$$\text{H-While}\quad \frac{\overline{t};\Gamma \vdash_{\mathrm{r}} e_1 : \mathrm{long}; S_1 \qquad \overline{t};\Gamma \vdash_{\mathrm{r}} e_2 : \tau; S_2}{\overline{t};\Gamma \vdash_{\mathrm{r}} \mathsf{while}\ e_1\ e_2 : \mathrm{short}; S_1 \wedge S_2}$$

$$\text{H-Decl}\quad \frac{\overline{t};\Gamma, x : \tau_1 \vdash_{\mathrm{r}} e : \tau_2; S}{\overline{t};\Gamma \vdash_{\mathrm{r}} \tau_1\ x;\ e : \tau_2; S}$$

Figure 17: Static semantics for high level language

- St-PadAdd: Immediate.
- St-Refl: Immediate.
- St-Seq: By induction, $\mathrm{size}(impl, \overline{\sigma}_1) = \mathrm{size}(impl, \overline{\sigma}_2)$ and $\mathrm{size}(impl, \overline{\sigma}_3) = \mathrm{size}(impl, \overline{\sigma}_4)$. It follows that $\mathrm{size}(impl, \sigma_1) + \mathrm{size}(impl, \overline{\sigma}_3) = \mathrm{size}(impl, \sigma_2) + \mathrm{size}(impl, \overline{\sigma}_4)$.
- St-Trans: By induction twice, $\mathrm{size}(impl, \overline{\sigma}_1) = \mathrm{size}(impl, \overline{\sigma}_2) = \mathrm{size}(impl, \overline{\sigma}_3)$.

**Lemma 10 (Value-Type Size)**

1. *If* $impl; \overline{t}; \Psi \vDash_{\mathrm{w}} \overline{w} : \overline{\sigma}$ *then* $\mathrm{size}(impl, \overline{w}) = \mathrm{size}(impl, \overline{\sigma})$.
2. *If* $impl; \overline{t}; \Psi; \Gamma \vdash_{\mathrm{r}} \overline{w} : \overline{\sigma}$ *then* $\mathrm{size}(impl, \overline{w}) = \mathrm{size}(impl, \overline{\sigma})$.

**Proof:**

1. By inspection of the assumed typing derivation:

- LW-BYTE: $\text{size}(impl, b) = \text{size}(impl, \text{byte}) = 1$.
- LW-LBL: $\text{size}(impl, \ell{+}i) = \text{size}(impl, \text{ptr}_\alpha(\overline{\sigma}_2)) = impl.\text{ptr\_size}$.
- LW-UNINIT1: $\text{size}(impl, \mathsf{uninit}) = \text{size}(impl, \text{byte}) = 1$.
- LW-UNINIT2: $\text{size}(impl, \mathsf{uninit}^i) = \text{size}(impl, \text{ptr}_\alpha(\overline{\sigma})) = impl.\text{ptr\_size} = i$.

2. By induction on the assumed typing derivation, by cases on the last rule used (all but 3 is impossible):

   - L-VALUEEMPTY: trivial; both sizes are 0.
   - L-VALUE: we have $impl; \overline{t}; \Psi; \Gamma \vdash_{\text{r}} \overline{w}_1\overline{w}_2 : \sigma_1\overline{\sigma}_2$ where inversion gives $impl; \overline{t}; \Psi \vdash_{\text{w}} \overline{w}_1 : \sigma_1$ and $impl; \overline{t}; \Psi; \Gamma \vdash_{\text{r}} \overline{w}_2 : \overline{\sigma}_2$. Part (1) of the theorem gives $\text{size}(impl, \overline{w}_1) = \text{size}(impl, \sigma_1)$. The induction hypothesis gives $\text{size}(impl, \overline{w}_2) = \text{size}(impl, \overline{\sigma}_2)$. It follows that $\text{size}(impl, \overline{w}_1\overline{w}_2) = \text{size}(impl, \overline{\sigma}_1\overline{\sigma}_2)$.
   - L-SUB: Follows from induction and the Constant-Size Subtyping Lemma.

**Lemma 11 (Sequence Typing)**

1. *If* $D; C \vdash_{\text{r}} \overline{w}_1 : \overline{\sigma}_1$ *and* $D; C \vdash_{\text{r}} \overline{w}_2 : \overline{\sigma}_2$, *then* $D; C \vdash_{\text{r}} \overline{w}_1\overline{w}_2 : \overline{\sigma}_1\overline{\sigma}_2$.
2. *If* $D; C \vdash_{\text{r}} \overline{w}_1 : \overline{\sigma}_1$, ..., $D; C \vdash_{\text{r}} \overline{w}_n : \overline{\sigma}_n$, *then* $D; C \vdash_{\text{r}} \overline{w}_1 \ldots \overline{w}_n : \overline{\sigma}_1 \ldots \overline{\sigma}_n$.

**Proof:**

1. By induction on the derivation of $D; C \vdash_{\text{r}} \overline{w}_1 : \overline{\sigma}_1$, by cases on the last rule used (all but 3 are impossible):

   - L-VALUEEMPTY: trivial; the second assumption is what we need.
   - L-VALUE: Then $\overline{w}_1 = \overline{w}_a\overline{w}_b$ and induction (applied to $\overline{w}_b$ and $\overline{w}_2$) and L-VALUE suffice.
   - L-SUB: By inversion $D; C \vdash_{\text{r}} \overline{w}_1 : \overline{\sigma}_3$ and $D \vdash \overline{\sigma}_3 \leq \overline{\sigma}_1$. So by induction $D; C \vdash_{\text{r}} \overline{w}_1\overline{w}_2 : \overline{\sigma}_3\overline{\sigma}_2$. So with $D \vdash \overline{\sigma}_3 \leq \overline{\sigma}_1$, ST-SEQ, ST-REFL, and L-SUB we can derive $D; C \vdash_{\text{r}} \overline{w}_1\overline{w}_2 : \overline{\sigma}_1\overline{\sigma}_2$.

2. By induction on $n$. Cases $n = 0$ and $n = 1$ are trivial. For $n > 1$, use induction and part(1).

**Lemma 12 (Subtyping Partition)**
*If* $D \vdash \overline{\sigma}_1 \leq \sigma_{21} \ldots \sigma_{2n}$, *then* $\exists \overline{\sigma}_{11}, \ldots, \overline{\sigma}_{1n}$ *such that* $\overline{\sigma}_1 = \overline{\sigma}_{11} \ldots \overline{\sigma}_{1n}$ *and for all* $1 \leq i \leq n$, $D \vdash \overline{\sigma}_{1i} \leq \sigma_{2i}$.

**Proof:** By induction on the assumed subtyping derivation, by cases on the last rule used:

- ST-ROLL, ST-UNROLL, ST-PTR, ST-PAD, ST-PADADD: Immediate because $n = 1$ so $\overline{\sigma}_1 = \overline{\sigma}_{11}$.
- ST-REFL: Trivial, let $\overline{\sigma}_{1i} = \sigma_{2i}$.
- ST-SEQ: Follows from two invocations of the induction hypothesis and the *union* of the partitions they prove exist.
- ST-TRANS: Follows from two invocations of the induction hypothesis and *composing* the partitions they prove exist.

**Lemma 13 (Subtyping Split)**
*If* $D \vdash \overline{\sigma}_3 \leq \overline{\sigma}_1\overline{\sigma}_2$, *then there exists* $\overline{\sigma}'_1$ *and* $\overline{\sigma}'_2$ *such that* $\overline{\sigma}_3 = \overline{\sigma}'_1\overline{\sigma}'_2$, $D \vdash \overline{\sigma}'_1 \leq \overline{\sigma}_1$, *and* $D \vdash \overline{\sigma}'_2 \leq \overline{\sigma}_2$.

**Proof:** Let $\overline{\sigma}_1 = \sigma_{11} \ldots \sigma_{1n}$ and $\overline{\sigma}_2 = \sigma_{21} \ldots \sigma_{2m}$. Then by the Subtyping Partition Lemma, there exist $\overline{\sigma}_{11}, \ldots, \overline{\sigma}_{1n}, \overline{\sigma}_{21}, \ldots \overline{\sigma}_{2m}$ such that $\overline{\sigma}_3 = \overline{\sigma}_{11} \ldots \overline{\sigma}_{1n}\overline{\sigma}_{21} \ldots \overline{\sigma}_{2m}$, $D \vdash \overline{\sigma}_{11} \leq \sigma_{11}$, $\ldots D \vdash \overline{\sigma}_{1n} \leq \sigma_{1n}$, $D \vdash \overline{\sigma}_{21} \leq \sigma_{21}$, $\ldots D \vdash \overline{\sigma}_{2m} \leq \sigma_{2m}$. So letting $\overline{\sigma}'_1 = \overline{\sigma}_{11} \ldots \overline{\sigma}_{1n}$ (respectively $\overline{\sigma}'_2 = \overline{\sigma}_{21} \ldots \overline{\sigma}_{2m}$), we can use $n$ (respectively $m$) uses of ST-SEQ and ST-TRANS to derive what we need.

**Lemma 14 (Typing Partition)**
*If* $D; C \vdash_{\mathrm{r}} \overline{w} : \sigma_1 \dots \sigma_n$, *then* $\exists \overline{w}_1, \dots, \overline{w}_n$ *such that* $\overline{w} = \overline{w}_1 \dots \overline{w}_n$ *and for all* $1 \le i \le n$, $D; C \vdash_{\mathrm{r}} \overline{w}_i : \sigma_i$.

**Proof:** By induction on the assumed typing derivation, by cases on the last rule used (all but 3 are impossible):

1. L-ValueEmpty: trivial because $n = 0$.

2. L-Value: Then $\overline{w} = \overline{w}_1 \overline{w}_2$, $D; \Psi \vDash_{\mathrm{w}} \overline{w}_1 : \sigma_1$ follows from inversion, and the other results follow from inversion and induction.

3. L-Sub: Then by inversion there exists $\sigma'_1 \dots \sigma'_m$ such that $D; C \vdash_{\mathrm{r}} \overline{w} : \sigma'_1 \dots \sigma'_m$ and $D \vdash \sigma'_1 \dots \sigma'_m \le \sigma_1 \dots \sigma_n$. So by induction there exists $\overline{w}'_1, \dots, \overline{w}'_m$ such that $\overline{w} = \overline{w}'_1 \dots \overline{w}'_m$ and $1 \le i \le m$, $D; C \vdash_{\mathrm{r}} \overline{w}'_i : \sigma'_i$. And by the Subtyping Partition Lemma, there exist $\overline{\sigma}'_1, \dots, \overline{\sigma}'_n$ such that $\sigma'_1 \dots \sigma'_m = \overline{\sigma}'_1 \dots \overline{\sigma}'_n$ and for all $1 \le i \le n$, $D \vdash \overline{\sigma}'_i \le \sigma_i$. Therefore, we can use the $\sigma'_i \dots \sigma'_j$ that is $\overline{\sigma}'_k$ and the corresponding $\overline{w}'_i \dots \overline{w}'_j$ and use the Sequence Typing Lemma to conclude $D; C \vdash_{\mathrm{r}} \overline{w}'_i \dots \overline{w}'_j : \overline{\sigma}'_k$ and then L-Sub to conclude $D; C \vdash_{\mathrm{r}} \overline{w}'_i \dots \overline{w}'_j : \sigma_k$. So letting $\overline{w}'_i \dots \overline{w}'_j = \overline{w}_k$ suffices.

**Lemma 15 (Value Split)**

1. *If* $D; C \vdash_{\mathrm{r}} \overline{w} : \overline{\sigma}_1 \overline{\sigma}_2$, *then* $\exists \overline{w}_1, \overline{w}_2$ *such that* $\overline{w} = \overline{w}_1 \overline{w}_2$, $D; C \vdash_{\mathrm{r}} \overline{w}_1 : \overline{\sigma}_1$, *and* $D; C \vdash_{\mathrm{r}} \overline{w}_2 : \overline{\sigma}_2$.

2. *If* $D; C \vdash_{\mathrm{r}} \overline{w} : \overline{\sigma}_1 \overline{\sigma}_2 \overline{\sigma}_3$, *then* $\exists \overline{w}_1, \overline{w}_2, \overline{w}_3$ *such that* $\overline{w} = \overline{w}_1 \overline{w}_2 \overline{w}_3$, $D; C \vdash_{\mathrm{r}} \overline{w}_1 : \overline{\sigma}_1$, $D; C \vdash_{\mathrm{r}} \overline{w}_2 : \overline{\sigma}_2$, *and* $D; C \vdash_{\mathrm{r}} \overline{w}_3 : \overline{\sigma}_3$.

**Proof:**

1. Let $\overline{\sigma}_1 = \sigma_{11} \dots \sigma_{1n}$ and $\overline{\sigma}_2 = \sigma_{21} \dots \sigma_{2m}$. Then by the Typing Partition Lemma there exists $\overline{w}_{11}, \dots, \overline{w}_{1n}, \overline{w}_{21}, \dots, \overline{w}_{2m}$ such that $\overline{w} = \overline{w}_{11} \dots \overline{w}_{1n} \overline{w}_{21} \dots \overline{w}_{2m}$ and $D; C \vdash_{\mathrm{r}} \overline{w}_{11} : \sigma_{11}$, …, $D; C \vdash_{\mathrm{r}} \overline{w}_{1n} : \sigma_{1n}$, $D; C \vdash_{\mathrm{r}} \overline{w}_{21} : \sigma_{21}$, … $D; C \vdash_{\mathrm{r}} \overline{w}_{2m} : \sigma_{2m}$. So letting $\overline{w}_1 = \overline{w}_{11} \dots \overline{w}_{1n}$ and $\overline{w}_2 = \overline{w}_{21} \dots \overline{w}_{2m}$, two uses of the Sequence Typing Lemma provide what we need.

2. Letting $\overline{\sigma}'_2 = \overline{\sigma}_2 \overline{\sigma}_3$, part 1 ensures there exist $\overline{w}_1$ and $\overline{w}'_2$ such that $\overline{w} = \overline{w}_1 \overline{w}'_2$, $D; C \vdash_{\mathrm{r}} \overline{w}_1 : \overline{\sigma}_1$, and $D; C \vdash_{\mathrm{r}} \overline{w}'_2 : \overline{\sigma}'_2$. Applying part 1 again to the last conclusion provides $D; C \vdash_{\mathrm{r}} \overline{w}_2 : \overline{\sigma}_2$ and $D; C \vdash_{\mathrm{r}} \overline{w}_3 : \overline{\sigma}_3$. (Note we can extend this to $n$ by a trivial induction should we need it).

**Non-Lemma: (Type-Split)** If $D; C \vdash_{\mathrm{r}} \overline{w}_1 \overline{w}_2 : \overline{\sigma}$, then $\exists \overline{\sigma}_1, \overline{\sigma}_2$ such that $\overline{\sigma} = \overline{\sigma}_1 \overline{\sigma}_2$, $D; C \vdash_{\mathrm{r}} \overline{w}_1 : \overline{\sigma}_1$, and $D; C \vdash_{\mathrm{r}} \overline{w}_2 : \overline{\sigma}_2$.
**Non-Proof:** This is FALSE. Let $\overline{w}_1 = \mathsf{uninit}^2$, $\overline{w}_2 = \mathsf{uninit}^2$ and $\overline{\sigma} = \mathrm{ptr}_\alpha(\cdot)$ when the size of pointers is 4.

**Lemma 16 (Value-Type Split)** *If*

$$impl; \overline{t}; C \vdash_{\mathrm{r}} \overline{w}_1 \overline{w}_2 : \overline{\sigma}_1 \overline{\sigma}_2$$
$$\mathrm{size}(impl, \overline{w}_1) = \mathrm{size}(impl, \overline{\sigma}_1)$$

*then*

$$impl; \overline{t}; C \vdash_{\mathrm{r}} \overline{w}_1 : \overline{\sigma}_1$$
$$impl; \overline{t}; C \vdash_{\mathrm{r}} \overline{w}_2 : \overline{\sigma}_2$$

**Proof:** If $\overline{w}_1 = \cdot$, then $\overline{\sigma}_1 = \cdot$ and the results follow from L-ValueEmpty and the assumption. So assume $\overline{w}_1 \neq \cdot$ and proceed by induction on the assumed typing derivation, by cases on the last rule applied (all but 2 are impossible):

- L-Value: By inversion and the assumptions, $\overline{w}_1 = \overline{w}_{11} \overline{w}_{12}$, $\overline{\sigma}_1 = \sigma_{11} \overline{\sigma}_{12}$,

  **1** $impl; \overline{t}; \Psi \vDash_{\mathrm{w}} \overline{w}_{11} : \sigma_{11}$

**2** $impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_{12}\overline{w}_2 : \overline{\sigma}_{12}\overline{\sigma}_2$

From the assumption, we know $\mathrm{size}(impl, \overline{w}_{11}\overline{w}_{12}) = \mathrm{size}(impl, \sigma_{11}\overline{\sigma}_{12})$. By the Value-Type Size Lemma, we have $\mathrm{size}(impl, \overline{w}_{11}) = \mathrm{size}(impl, \sigma_{11})$. It follows that

**3** $\mathrm{size}(impl, \overline{w}_{12}) = \mathrm{size}(impl, \overline{\sigma}_{12})$

From (2,3), the induction hypothesis yields

**4** $impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_{12} : \overline{\sigma}_{12}$

**5** $impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_2 : \overline{\sigma}_2$

Plugging (1,4) into L-Value gives $impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_{11}\overline{w}_{12} : \sigma_{11}\overline{\sigma}_{12}$, which, together with (5), gives the desired conclusion.

- L-Sub: By inversion there exists a $\overline{\sigma}_3$ such that $D; C \vdash_{\mathrm{r}} \overline{w}_1\overline{w}_2 : \overline{\sigma}_3$ and $D \vdash \overline{\sigma}_3 \leq \overline{\sigma}_1\overline{\sigma}_2$. So by the Subtyping Split Lemma, there exists $\overline{\sigma}'_1$ and $\overline{\sigma}'_2$ such that $\overline{\sigma}_3 = \overline{\sigma}'_1\overline{\sigma}'_2$, $D \vdash \overline{\sigma}'_1 \leq \overline{\sigma}_1$, and $D \vdash \overline{\sigma}'_2 \leq \overline{\sigma}_2$. By the Constant-Size Subtyping Lemma $\mathrm{size}(impl, \overline{\sigma}'_1) = \mathrm{size}(impl, \overline{\sigma}_1)$, so the assumption $\mathrm{size}(impl, \overline{w}_1) = \mathrm{size}(impl, \overline{\sigma}_1)$ ensures $\mathrm{size}(impl, \overline{w}'_1) = \mathrm{size}(impl, \overline{\sigma}'_1)$. So by induction $impl; \bar{t}; C \vdash_{\mathrm{r}} \overline{w}_1 : \overline{\sigma}'_1$ and $impl; \bar{t}; C \vdash_{\mathrm{r}} \overline{w}_2 : \overline{\sigma}'_2$. So with one use of L-Sub each, we can derive $impl; \bar{t}; C \vdash_{\mathrm{r}} \overline{w}_1 : \overline{\sigma}_1$ and $impl; \bar{t}; C \vdash_{\mathrm{r}} \overline{w}_2 : \overline{\sigma}_2$.

**Lemma 17 (Subsequence Replacement)**

1. *If* $D; C \vdash_{\mathrm{r}} \overline{w}_1\overline{w}_2 : \overline{\sigma}_1\overline{\sigma}_2$ *and* $D; C \vdash_{\mathrm{r}} \overline{w}_2 : \overline{\sigma}_2$ *and* $D; C \vdash_{\mathrm{r}} \overline{w}_3 : \overline{\sigma}_2$ *then* $D; C \vdash_{\mathrm{r}} \overline{w}_1\overline{w}_3 : \overline{\sigma}_1\overline{\sigma}_2$.
2. *If* $D; C \vdash_{\mathrm{r}} \overline{w}_1\overline{w}_2 : \overline{\sigma}_1\overline{\sigma}_2$ *and* $D; C \vdash_{\mathrm{r}} \overline{w}_1 : \overline{\sigma}_1$ *and* $D; C \vdash_{\mathrm{r}} \overline{w}_3 : \overline{\sigma}_1$ *then* $D; C \vdash_{\mathrm{r}} \overline{w}_3\overline{w}_2 : \overline{\sigma}_1\overline{\sigma}_2$.

**Proof:** Follows from the Value Split and Sequence Typing Lemmas.

**Lemma 18 (Alignment Subtyping Reflexivity)**
*For all* $[a, o]$, *we can derive* $\vdash [a, o] \leq [a, o]$.

**Proof:** Can be derived with either Alst-Off ($o \equiv o \mod a$) or Alst-Base ($a = a \times 1$).

**Lemma 19 (Alignment Subtyping Form)**
*If* $\vdash [a, o + i] \leq \alpha$ *then* $\exists a', o'$ *such that* $\alpha = [a', o' + i]$.

**Proof:** By induction on the assumed derivation.

**Lemma 20 (Alignment Addition)**
$\vdash [a, o] \leq [a', o']$ *if and only if* $\vdash [a, o + k] \leq [a', o' + k]$.

**Proof:** Both directions proceed by induction on the assumed derivation, using the fact that $o \equiv o' \mod a$ iff $o + k \equiv o' + k \mod a$.

**Lemma 21 (Subtyping Type Form)**

- *If* $impl; \bar{t} \vdash \overline{\sigma}' \leq \mathrm{ptr}_\alpha(\overline{\sigma})$ *then* $\exists \overline{\sigma}'', \alpha'$ *such that* $\overline{\sigma}' = \mathrm{ptr}_{\alpha'}(\overline{\sigma}\,\overline{\sigma}'')$ *or* $\overline{\sigma}' = \mathrm{ptr}_{\alpha'}(N)$ *where* $impl.\mathrm{xtype}(\bar{t}, N) = \overline{\sigma}\,\overline{\sigma}''$, *and* $\vdash \alpha' \leq \alpha$.
- *If* $impl; \bar{t} \vdash \overline{\sigma}' \leq \mathrm{ptr}_\alpha(N)$ *and* $impl.\mathrm{xtype}(\bar{t}, N) = \overline{\sigma}$ *then* $\exists \overline{\sigma}'', \alpha'$ *such that* $\overline{\sigma}' = \mathrm{ptr}_{\alpha'}(\overline{\sigma}\,\overline{\sigma}'')$ *or* $\overline{\sigma}' = \mathrm{ptr}_{\alpha'}(N')$ *where* $impl.\mathrm{xtype}(\bar{t}, N') = \overline{\sigma}\,\overline{\sigma}''$, *and* $\vdash \alpha' \leq \alpha$.

**Proof:** By simultaneous induction on the assumed subtyping derivations, by cases on the last rule used. The cases St-Pad and St-PadAdd cannot occur.

- St-Roll: $\overline{\sigma}' = \text{ptr}_\alpha(N)$ and inversion gives $\mathit{impl}.\text{xtype}(\bar{t}, N) = \overline{\sigma}$. We satisfy the property with $\overline{\sigma}'' = \cdot$ and $\alpha' = \alpha$.
- St-Unroll: $\overline{\sigma}' = \text{ptr}_\alpha(\overline{\sigma}\,\overline{\sigma}'')$ where inversion gives $\mathit{impl}.\text{xtype}(\bar{t}, N) = \overline{\sigma}$. We satisfy the property with $\overline{\sigma}'' = \cdot$ and $\alpha' = \alpha$.
- St-Ptr: Follows from inspection and inversion.
- St-Refl: Immediate, with $\alpha' = \alpha$, $N' = N$, and $\overline{\sigma}'' = \cdot$.
- St-Seq: We have $\mathit{impl}; \bar{t} \vdash \overline{\sigma}_1\overline{\sigma}_3 \leq \overline{\sigma}_2\overline{\sigma}_4$. It must be the case that either $\overline{\sigma}_1 = \overline{\sigma}_2 = \cdot$ or $\overline{\sigma}_3 = \overline{\sigma}_4 = \cdot$, because pointer types cannot be broken into subsequences. The property follows by induction.
- St-Trans: We have $\mathit{impl}; \bar{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_3$ where inverting gives $\mathit{impl}; \bar{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ and $\mathit{impl}; \bar{t} \vdash \overline{\sigma}_2 \leq \overline{\sigma}_3$. There are two possibilities for $\overline{\sigma}_3$:
  1. $\overline{\sigma}_3 = \text{ptr}_\alpha(N)$ where $\mathit{impl}.\text{xtype}(\bar{t}, N) = \overline{\sigma}'$. By induction on the second subderivation, we get either
     - $\overline{\sigma}_2 = \text{ptr}_{\alpha'}(\overline{\sigma}'\overline{\sigma}'')$, where $\vdash \alpha' \leq \alpha$. By induction on the first subderivation, we have either $\overline{\sigma}_1 = \text{ptr}_{\alpha''}(\overline{\sigma}'\overline{\sigma}''\overline{\sigma}''')$ or $\overline{\sigma}_1 = \text{ptr}_{\alpha''}(N')$ where $\mathit{impl}.\text{xtype}(\bar{t}, N') = \overline{\sigma}'\overline{\sigma}''\overline{\sigma}'''$ and $\vdash \alpha'' \leq \alpha'$, and by Alst-Trans, $\vdash \alpha'' \leq \alpha$, which is what we want.
     - $\overline{\sigma}_2 = \text{ptr}_{\alpha'}(N')$ where $\mathit{impl}.\text{xtype}(\bar{t}, N') = \overline{\sigma}'\overline{\sigma}''$ and $\vdash \alpha' \leq \alpha$. By induction on the first subderivation, we have either $\overline{\sigma}_1 = \text{ptr}_{\alpha''}(\overline{\sigma}'\overline{\sigma}''\overline{\sigma}''')$ or $\overline{\sigma}_1 = \text{ptr}_{\alpha''}(N)$ where $\mathit{impl}.\text{xtype}(\bar{t}, N) = \overline{\sigma}'\overline{\sigma}''\overline{\sigma}'''$ and $\vdash \alpha'' \leq \alpha'$, and by Alst-Trans, $\vdash \alpha'' \leq \alpha$, which is what we want.
  2. $\overline{\sigma}_3 = \text{ptr}_\alpha(\overline{\sigma}')$. Proceeds exactly as above.

**Lemma 22 (Canonical Forms)**

1. *If* $D; \Psi; \cdot \vdash_{\text{r}} \overline{w} : \cdot$ *then* $\overline{w} = \cdot$.
2. *If* $D; \Psi; \cdot \vdash_{\text{r}} \overline{w} : \text{byte}$ *then* $\overline{w} = b$ *or* $\overline{w} = \mathsf{uninit}$.
3. *If* $D; \Psi; \cdot \vdash_{\text{r}} \overline{w} : \text{byte}^k$ *then* $\overline{w} = w_1 \ldots w_k$ *where* $w_i = b$ *or* $w_i = \mathsf{uninit}$ *for* $i \in \{1, \ldots, k\}$.
4. - *If* $\mathit{impl}; \bar{t}; \Psi; \cdot \vdash_{\text{r}} \overline{w} : \text{ptr}_\alpha(\overline{\sigma}_1)$ *then* $\exists \overline{\sigma}_0, \overline{\sigma}_2$ *s.t. either*
     - $\overline{w} = \mathsf{uninit}^i$ *where* $\mathit{impl}.\text{ptr\_size} = i$.
     - $\overline{w} = \ell{+}i$ *and* $\alpha = [a, o + i]$ *and* $\Psi(\ell) = \overline{\sigma}_0\overline{\sigma}_1\overline{\sigma}_2, \alpha'$ *where* $\vdash \alpha' \leq [a, o]$ *and* $\text{size}(\mathit{impl}, \overline{\sigma}_0) = i$.
   - *If* $\mathit{impl}; \bar{t}; \Psi; \cdot \vdash_{\text{r}} \overline{w} : \text{ptr}_\alpha(N)$ *where* $\mathit{impl}.\text{xtype}(\bar{t}, N) = \overline{\sigma}_1$, *then* $\exists \overline{\sigma}_0, \overline{\sigma}_2$ *s.t. either*
     - $\overline{w} = \mathsf{uninit}^i$ *where* $\mathit{impl}.\text{ptr\_size} = i$.
     - $\overline{w} = \ell{+}i$ *and* $\alpha = [a, o + i]$ *and* $\Psi(\ell) = \overline{\sigma}_0\overline{\sigma}_1\overline{\sigma}_2, \alpha'$ *where* $\vdash \alpha' \leq [a, o]$ *and* $\text{size}(\mathit{impl}, \overline{\sigma}_0) = i$.

**Proof:**

1. By induction on the typing derivation. Last rule applied is either L-ValueEmpty or L-Sub. The former follows immediately. If the latter, the only subtype of $\cdot$ is $\cdot$, so the property follows form induction.
2. By induction on the derivation. Last rule applied is either L-Value or L-Sub. In the former case, by inversion we have $D; \Psi; \cdot \models_{\text{w}} \overline{w} : \text{byte}$. Only the Lw-Byte or Lw-Uninit1 can apply, meaning either $\overline{w} = b$ or $\overline{w} = \mathsf{uninit}$. In the latter case, byte has no subtypes other than itself, so the property follows from induction.
3. By induction on the type derivation, by cases on the last rule used. If the last rule is L-Value, we have $\overline{w} = w\overline{w}'$ and inversion gives $D; \Psi \models_{\text{w}} w : \text{byte}$ and $D; \Psi; \cdot \vdash_{\text{r}} \overline{w}' : \text{byte}^{k-1}$. By induction, the property holds for the latter derivation. By part (2) of the theorem, the property holds for the former derivation. If the last rule applied is L-Sub, the property holds by induction, since $\text{byte}^i$ has no subtypes other than itself.

4. By simultaneous induction on the assumed typing derivations, by cases on the last rule applied. Reflexivity of alignment subtyping (Lemma 18) is used implicitly. There are only two possibilities:

   - L-Value: There are two sub-cases:
     - Assume $D; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}_{\alpha}(\overline{\sigma}_1)$ Inversion gives $D; \Psi; \cdot \vdash_{\mathrm{w}} \overline{w} : \mathrm{ptr}_{\alpha}(\overline{\sigma}_1)$. Quick inspection reveals that only the rules Lw-Lbl or Lw-Uninit2 can lead to this conclusion. In the latter case, clearly $\overline{w} = \mathsf{uninit}^i$. In the former case, inversion yields the desired result, with $\overline{\sigma}_2 = \cdot$ and $\alpha = [a, o+i]$ and $\alpha' = [a, o]$.
     - Assume $D; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}_{[a,o+i]}(N)$, where $\mathit{impl}.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}_1$. Vacuous; cannot be derived via L-Value.
   - L-Sub: There are two sub-cases:
     - Assume $D; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}_{\alpha}(\overline{\sigma}_1)$. Inversion gives $D; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \overline{\sigma}$ and $D \vdash \overline{\sigma} \leq \mathrm{ptr}_{\alpha}(\overline{\sigma}_1)$. By the Subtyping Type Form Lemma, $\overline{\sigma}$ is one of
       - $\mathrm{ptr}_{\alpha'}(N)$ where $\mathit{impl}.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}_1\overline{\sigma}_k$ for some $\overline{\sigma}_k$, and $\vdash \alpha' \leq \alpha$. Applying the IH to $D; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}_{\alpha'}(N)$ we get that either $\overline{w} = \mathsf{uninit}^i$ or $\overline{w} = \ell{+}i$ and $\alpha' = [a, o+i]$ and $\Psi(\ell) = \overline{\sigma}_0\overline{\sigma}_1\overline{\sigma}_k\overline{\sigma}_2, \alpha''$ where $\vdash \alpha'' \leq [a, o]$ and $\mathrm{size}(\mathit{impl}, \overline{\sigma}_0) = i$.
         It remains to be shown that $\alpha$ is of the form $[a', o'+i]$ and $\vdash \alpha'' \leq [a', o']$. We know $\vdash [a, o+i] \leq \alpha$. By the Alignment Subtyping Form Lemma, $\alpha = [a', o'+i]$ for some $a'$ and $o'$. By the Alignment Addition Lemma, $\vdash [a, o] \leq [a', o']$. We also know $\vdash \alpha'' \leq [a, o]$, so by Alst-Trans, $\vdash \alpha'' \leq [a', o']$, as desired.
       - $\mathrm{ptr}_{\alpha}(\overline{\sigma}_1\overline{\sigma}_k)$ for some $\overline{\sigma}_k$. Proceeds like the above case.
     - Assume $D; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}_{\alpha}(N)$, where $\mathit{impl}.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}_1$. Similar to the above case.

**Lemma 23 (Context Reordering)**
*If* $D; \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash_{\mathrm{r}} e : \overline{\sigma}$ *then* $D; \Gamma, x_2 : \tau_2, x_1 : \tau_1 \vdash_{\mathrm{r}} e : \overline{\sigma}$.

**Proof:** Straightforward induction.

**Lemma 24 (Substitution Preserves Types)**
*Suppose* $\mathit{impl}.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}_2$, $\mathit{impl}.\mathrm{align}(\overline{t}, \tau) = \alpha$, $\mathit{impl}; \overline{t}; \Psi; \Gamma \vdash_{\mathrm{r}} e_2 : \overline{\sigma}_2$, *and* $\mathit{impl}; \overline{t}; \Psi; \Gamma \vdash_{\mathrm{l}} e_2 : \overline{\sigma}_2, \alpha$.

1. *If* $\mathit{impl}; \overline{t}; \Psi; \Gamma, x : \tau \vdash_{\mathrm{r}} e_1 : \overline{\sigma}_1$, *then* $\mathit{impl}; \overline{t}; \Psi; \Gamma \vdash_{\mathrm{r}} e_1\{e_2/x\} : \overline{\sigma}_1$.
2. *If* $\mathit{impl}; \overline{t}; \Psi; \Gamma, x : \tau \vdash_{\mathrm{l}} e_1 : \overline{\sigma}_1, \alpha$, *then* $\mathit{impl}; \overline{t}; \Psi; \Gamma \vdash_{\mathrm{l}} e_1\{e_2/x\} : \overline{\sigma}_1, \alpha$.

**Proof:** By simultaneous induction the assumed typing derivations, by cases on the last rule used.

- L-Short, L-Long, L-New, L-ValueEmpty, L-Value: trivial because substitution and $x$ are irrelevant.
- L-VarR: We assume $\mathit{impl}; \overline{t}; \Psi; \Gamma, x : \tau \vdash_{\mathrm{r}} y : \overline{\sigma}_1$ and it follows that $\mathit{impl}.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}_1$. If $x = y$ then $\overline{\sigma}_1 = \overline{\sigma}_2$ and $y\{e_2/x\} = e_2$, and so $\mathit{impl}; \overline{t}; \Psi; \Gamma \vdash_{\mathrm{r}} e_2 : \overline{\sigma}_1$ by assumption, as desired. If $x \neq y$, then $y\{e_2/x\} = y$ and we can drop $x$ from the context and still derive $\mathit{impl}; \overline{t}; \Psi; \Gamma \vdash_{\mathrm{r}} y : \overline{\sigma}_1$.
- L-VarL: We assume $\mathit{impl}; \overline{t}; \Psi; \Gamma, x : \tau \vdash_{\mathrm{l}} y : \overline{\sigma}_1, \alpha$ and it follows that $\mathit{impl}.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}_1$ and $\mathit{impl}.\mathrm{align}(\overline{t}, \tau) = \alpha$. If $x = y$ then $\overline{\sigma}_1 = \overline{\sigma}_2$ and $y\{e_2/x\} = e_2$, and so $\mathit{impl}; \overline{t}; \Psi; \Gamma \vdash_{\mathrm{l}} e_2 : \overline{\sigma}_1, \alpha$ by assumption, as desired. If $x \neq y$, then $y\{e_2/x\} = y$ and we can drop $x$ from the context and still derive $\mathit{impl}; \overline{t}; \Psi; \Gamma \vdash_{\mathrm{l}} y : \overline{\sigma}_1, \alpha$.
- L-Assn: We assume $\mathit{impl}; \overline{t}; \Psi; \Gamma, x : \tau \vdash_{\mathrm{r}} e = e' : \overline{\sigma}_1$. By inversion and induction (once for left-typing and once for right-typing),

  **1** $\mathit{impl}; \overline{t}; \Gamma \vdash_{\mathrm{l}} e\{e_2/x\} : \overline{\sigma}_1, \alpha$

  **2** $\mathit{impl}; \overline{t}; \Gamma \vdash_{\mathrm{r}} e'\{e_2/x\} : \overline{\sigma}_1$

The desired property follows from an application of L-Assn to these facts and from the definition of substitution.

- L-FetchL: Follows from inversion, induction, and an application of L-FetchL.
- L-FetchR: Follows from inversion, induction, and an application of L-FetchR.
- L-Deref{L,R}: Inversion, induction, and an application of L-Deref{L,R}.
- L-Cast: Inversion, induction, and an application of L-Cast.
- L-FAddr: Inversion, induction, and an application of L-FAddr.
- L-Seq: Inversion, induction applied to each subexpression, and plugging the results into L-Seq.
- L-If: Similar to L-Seq.
- L-While: Similar to L-Seq.
- L-Decl: We assume $impl; \bar{t}; \Gamma, x : \tau \vdash_{\mathrm{r}} \tau'\ y;\ e : \overline{\sigma}_1$. By inversion, $impl; \bar{t}; \Gamma, x : \tau, y : \tau' \vdash_{\mathrm{r}} e : \overline{\sigma}_1$. By the Context Reordering Lemma, $impl; \bar{t}; \Gamma, y : \tau', x : \tau \vdash_{\mathrm{r}} e : \overline{\sigma}_1$. By Weakening, $impl; \bar{t}; \Gamma, y : \tau' \vdash_{\mathrm{r}} e_2 : \overline{\sigma}_2$. By induction, $impl; \bar{t}; \Gamma, y : \tau' \vdash_{\mathrm{r}} e\{e_2/x\} : \overline{\sigma}_1$. By D-Decl, $impl; \bar{t}; \Gamma \vdash_{\mathrm{r}} \tau'\ y;\ e\{e_2/x\} : \overline{\sigma}_1$.
- L-Sub: Inversion, induction, and an application of L-Sub.

**Lemma 25 (Subject Reduction)**
*Suppose impl is a sensible implementation. If*

$impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} e : \overline{\sigma}$ *(or* $impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{l}} e : \overline{\sigma}, \alpha$*)*
$impl; \bar{t} \vdash H; e \xrightarrow{\mathrm{r}} H'; e'$ *(or* $impl; \bar{t} \vdash H; e \xrightarrow{\mathrm{l}} H'; e'$*)*
$impl; \bar{t}; \Psi \vdash H : \Psi$

*then there exists* $\Psi'$*, extending* $\Psi$*, such that*

$impl; \bar{t}; \Psi'; \cdot \vdash_{\mathrm{r}} e' : \overline{\sigma}$ *(or* $impl; \bar{t}; \Psi'; \cdot \vdash_{\mathrm{l}} e' : \overline{\sigma}, \alpha$*)*
$impl; \bar{t}; \Psi' \vdash H' : \Psi'$

**Proof:** Proof proceeds by induction on the assumed typing derivation. The only inductive case is L-Sub. We assume $\Psi = \Psi'$ and $H = H'$ unless otherwise stated.

- L-Var{L,R}, L-ValueEmpty, L-Value: Vacuous; variables and values cannot step.
- L-Cast: The step must have been derived by D-Cast and the result follows from inversion.
- L-Short: The step must have been derived by D-Short and we have $e' = \overline{b}$ where $impl.\mathrm{xlit}(s) = \overline{b}$. From L-Short, $\overline{\sigma} = \mathrm{byte}^i$. Because *impl* is sensible (clause 2), we know $\mathrm{size}(impl, \overline{b}) = i$. We can apply Lw-Byte and L-Value $i$ times to derive $impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} e' : \overline{\sigma}$.
- L-Long: Similar to L-Short, where the step was derived by D-Long.
- L-Seq: The step was derived using D-Seq and the result follows from inversion.
- L-If: The step was derived using D-IfT or D-IfF and the result follows from inversion.
- L-While: The step was derived using D-While and we have $impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} \mathsf{while}\ e_1\ e_2 : \mathrm{byte}^i$ and $e' = \mathsf{if}\ e_1\ (e_2; \mathsf{while}\ e_1\ e_2)\ s$. Inversion gives

  **1** $impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} e_1 : \mathrm{byte}^j$

  **2** $impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} e_2 : \overline{\sigma}_2$

**3** $impl.\text{xtype}(\overline{t}, \text{short}) = \text{byte}^i$

From the assumed typing and (2), L-Seq gives

**4** $impl; \overline{t}; \Psi; \cdot \vdash_{\text{r}} (e_2; \mathsf{while}\ e_1\ e_2) : \text{byte}^i$

From L-Short, we have

**5** $impl; \overline{t}; \Psi; \cdot \vdash_{\text{r}} s : \text{byte}^i$

Plugging (1,4,5) into L-If yields $impl; \overline{t}; \Psi; \cdot \vdash_{\text{r}} e' : \text{byte}^i$.

- L-New: The step was derived using D-New. We assume $impl; \overline{t}; \Psi; \cdot \vdash_{\text{r}} \mathsf{new}\ \tau : \text{ptr}_\alpha(\overline{\sigma})$ and have $e' = \ell\text{+}0$ and $H' = H, \ell \mapsto \mathsf{uninit}^i, \alpha$. Let $\Psi' = \Psi, \ell : \overline{\sigma}, \alpha$, so

  **1** $\Psi'(\ell) = \overline{\sigma}, \alpha$

  From the side conditions of the assumed reduction, we have

  **2** $impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}$

  **3** $\text{size}(impl, \overline{\sigma}) = i$

  **4** $impl.\text{align}(\overline{t}, \tau) = \alpha$

  Note that the $\alpha$ and $\overline{\sigma}$ here are the same as those mentioned in the assumption, from inversion on L-New. From (1,3), Lw-Lbl followed by L-Value give $impl; \overline{t}; \Psi'; \cdot \vdash_{\text{r}} e' : \text{ptr}_\alpha(\overline{\sigma})$, which satisfies the first part of the claim.

  To satisfy the second part of the claim, we first apply the Heap Weakening Lemma to the third assumption to get

  **5** $impl; \overline{t}; \Psi' \vdash H : \Psi$

  From (3) and the Uninit Type Lemma, we have

  **6** $impl; \overline{t}; \Psi'; \cdot \vdash_{\text{r}} \mathsf{uninit}^i : \overline{\sigma}$ (6)

  Plugging (5,6) into the Ht-Inf rule yields $impl; \overline{t}; \Psi' \vdash H' : \Psi'$.

- L-Decl: The step was derived using D-Decl. We assume $impl; \overline{t}; \Psi; \cdot \vdash_{\text{r}} \tau\ x;\ e : \overline{\sigma}_1$, and we have $e' = e\{*(\tau*)(\ell\text{+}0)/x\}$ and $H' = H, \ell \mapsto \mathsf{uninit}^i, \alpha$, where the side conditions of the assumed reduction give

  **1** $impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}$

  **2** $impl.\text{align}(\overline{t}, \tau) = \alpha$

  By inversion on the assumed typing, we have

  **3** $impl; \overline{t}; \Psi; \cdot, x : \tau \vdash_{\text{r}} e : \overline{\sigma}_1$

  Let $\Psi' = \Psi, \ell : \overline{\sigma}, \alpha$, so

  **4** $\Psi'(\ell) = \overline{\sigma}, \alpha$

  Because the implementation is sensible (clause 5), we know $impl.\text{access}(impl.\text{align}(\overline{t}, \tau), \text{size}(impl, impl.\text{xtype}(\overline{t}, \tau)))$. In other words, using (1,2),

  **5** $impl.\text{access}(\alpha, \text{size}(impl, \overline{\sigma}))$

With (1,4,5), we can perform the following forward derivation:

$$\dfrac{\dfrac{\dfrac{\Psi'(\ell) = \overline{\sigma}, \alpha}{impl; \overline{t}; \Psi'; \cdot \vdash_{\mathrm{w}} \ell\text{+}0 : \mathrm{ptr}_{\alpha}(\overline{\sigma})} \quad impl; \overline{t}; \Psi'; \cdot \vdash_{\mathrm{r}} \cdot : \cdot}{impl; \overline{t}; \Psi'; \cdot \vdash_{\mathrm{r}} \ell\text{+}0 : \mathrm{ptr}_{\alpha}(\overline{\sigma})} \quad impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma} \quad impl.\mathrm{access}(\alpha, \mathrm{size}(impl, \overline{\sigma}))}{impl; \overline{t}; \Psi'; \cdot \vdash_{\mathrm{r}} *(\tau*)(\ell\text{+}0) : \overline{\sigma}}$$

Notice that under the same assumptions, we can also derive $impl; \overline{t}; \Psi'; \cdot \vdash_{\mathrm{l}} *(\tau*)(\ell\text{+}0) : \overline{\sigma}, \alpha$. (L-DerefR and L-DerefL have identical hypotheses.) With this, we can apply Substitution Preserves Types Lemma to the above conclusion and (1,2,3) to get $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e' : \overline{\sigma}_1$. The demonstration of the second part of the claim is identical to the D-New case.

- L-FetchL: The step was derived using D-FetchL; we assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{l}} *(\tau_1*)(\ell\text{+}j).f : \overline{\sigma}_2, [a, o + j']$ and have $e' = *(\tau*)(\ell\text{+}(j + j'))$ where

  **1** $impl.\mathrm{offset}(\overline{t}, f) = j'$

  By inversion we obtain

  **2** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{l}} *(\tau_1*)(\ell\text{+}j) : \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3, [a, o]$
  **3** $impl.\mathrm{offset}(\overline{t}, f) = \mathrm{size}(impl, \overline{\sigma}_1)$
  **4** $N\{\ldots \tau\ f \ldots\} \in \overline{t}$
  **5** $impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}_2$
  **6** $impl.\mathrm{align}(\overline{t}, N) = [a_N, o_N]$
  **7** $\vdash [a, o] \leq [a_N, o_N]$

  Inverting (2) (note this is a left-typing so there is no subsumption) gives

  **8** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \ell\text{+}j : \mathrm{ptr}_{[a,o]}(\overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3\overline{\sigma}_4)$

  Canonical Forms gives

  **9** $\Psi(\ell) = \overline{\sigma}_0\overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3\overline{\sigma}_4\overline{\sigma}_5, [a', o']$
  **10** $\mathrm{size}(impl, \overline{\sigma}_0) = j$
  **11** $o = o'' + j$
  **12** $\vdash [a', o'] \leq [a, o'']$

  From (1,3), we have $j' = \mathrm{size}(impl, \overline{\sigma}_1)$. From this and (10), it follows that $j + j' = \mathrm{size}(impl, \overline{\sigma}_0\overline{\sigma}_1)$. Plugging this along with (9) into Lw-Lbl, and the result into L-Value, gives

  **13** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \ell\text{+}(j + j') : \mathrm{ptr}_{[a', o'+j+j']}(\overline{\sigma}_2\overline{\sigma}_3\overline{\sigma}_4\overline{\sigma}_5)$

  From the Alignment Addition Lemma on (12), we get $\vdash [a', o' + j + j'] \leq [a, o'' + j + j']$. Plugging this into St-Ptr, we get $impl; \overline{t} \vdash \mathrm{ptr}_{[a', o'+j+j']}(\overline{\sigma}_2\overline{\sigma}_3\overline{\sigma}_4\overline{\sigma}_5) \leq \mathrm{ptr}_{[a, o''+j+j']}(\overline{\sigma}_2\overline{\sigma}_3\overline{\sigma}_4\overline{\sigma}_5)$. Plugging this and (13) into L-Sub, we get

  **14** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \ell\text{+}(j + j') : \mathrm{ptr}_{[a, o''+j+j']}(\overline{\sigma}_2\overline{\sigma}_3\overline{\sigma}_4\overline{\sigma}_5)$.

  It remains to be shown that $impl.\mathrm{access}([a, o'' + j + j'], \mathrm{size}(impl, \overline{\sigma}_2))$. This, plugged into L-DerefL together with (5,14) would yield $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{l}} *(\tau*)(\ell\text{+}(j + j')) : \overline{\sigma}_2, [a, o'' + j + j']$, as desired.

  To prove this, suppose $impl.\mathrm{align}(\overline{t}, \tau) = \alpha$. By clause (1) of sane implementations we have $impl.\mathrm{access}(\alpha, \overline{\sigma}_2)$. By clause (3) of sane implementations, we have $\vdash [a_N, o_N + j'] \leq \alpha$ (where $[a_N, o_N]$ is the alignment of $N$ as established in (7)). By clause (5) of sane implementations, we then have $impl.\mathrm{access}([a_N, o_N + j'], \mathrm{size}(impl, \overline{\sigma}_2))$. From (7) and the Alignment Addition Lemma, we get $\vdash [a, o + j'] \leq [a_N, o_N + j']$, which, with (11), can be rewritten as $\vdash [a, o'' + j + j'] \leq [a_N, o_N + j']$. Again, by clause (5) of sane implementations, we have $impl.\mathrm{access}([a, o'' + j + j'], \mathrm{size}(impl, \overline{\sigma}_2))$, which concludes the proof case.

- L-FetchR: The step was derived using D-Fetch, so we assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_1\overline{w}_2\overline{w}_3.f : \overline{\sigma}_2$ and $e' = \overline{w}_2$. The step side condition gives

  **1** $impl.\mathrm{offset}(\overline{t}, f) = \mathrm{size}(impl, \overline{w}_1)$

  **2** $\mathrm{size}(impl, \overline{\sigma}_2) = \mathrm{size}(impl, \overline{w}_2)$

  By inversion on the assumed typing we have

  **3** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_1\overline{w}_2\overline{w}_3 : \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3$

  **4** $impl.\mathrm{offset}(\overline{t}, f) = \mathrm{size}(impl, \overline{\sigma}_1)$

  Note that the metavariable $\overline{\sigma}_2$ is bound to the same type sequence in both the step side condition and inversion on the assumed typing, because $impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}_2$ is given by both.

  Applying the Value-Type Split Lemma to (1,3) gives

  **5** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_2\overline{w}_3 : \overline{\sigma}_2\overline{\sigma}_3$

  Applying the Value-Type Split Lemma to (2,5) yields $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_2 : \overline{\sigma}_2$.

- L-FAddr: The step was derived using D-FAddr; we assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} (\tau*)\&\ell\texttt{+}j \to f : \mathrm{ptr}_{[a,o_2]}(\overline{\sigma}_2)$ and $e' = \ell\texttt{+}(j + impl.\mathrm{offset}(\overline{t}, f))$. By inversion on the assumed typing, we get

  **1** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \ell\texttt{+}j : \mathrm{ptr}_{[a,o_1]}(\overline{\sigma}_1\overline{\sigma}_2)$

  **2** $impl.\mathrm{offset}(\overline{t}, f) = \mathrm{size}(impl, \overline{\sigma}_1)$

  **3** $o_2 = o_1 + impl.\mathrm{offset}(\overline{t}, f) = o_1 + \mathrm{size}(impl, \overline{\sigma}_1)$

  Canonical Forms gives

  **4** $\Psi(\ell) = \overline{\sigma}_0\overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3, [a', o'_0]$

  **5** $o_1 = o_0 + j$

  **6** $j = \mathrm{size}(impl, \overline{\sigma}_0)$

  **7** $\vdash [a', o'_0] \leq [a, o_0]$

  Adding $j$ to both sides of (2) and making use of (6), we obtain $j + impl.\mathrm{offset}(\overline{t}, f) = \mathrm{size}(impl, \overline{\sigma}_0\overline{\sigma}_1)$. We can plug this fact along with (4) into Lw-Lbl to get

  $$impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \ell\texttt{+}(j + impl.\mathrm{offset}(\overline{t}, f)) : \mathrm{ptr}_{[a', o'_0 + j + impl.\mathrm{offset}(\overline{t}, f)]}(\overline{\sigma}_2\overline{\sigma}_3)$$

  By the Alignment Addition Lemma on (7), $\vdash [a', o'_0 + j + impl.\mathrm{offset}(\overline{t}, f)] \leq [a, o_0 + j + impl.\mathrm{offset}(\overline{t}, f)]$. Plugging this into St-Aln1 and the result into L-Sub, we get

  $$impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \ell\texttt{+}(j + impl.\mathrm{offset}(\overline{t}, f)) : \mathrm{ptr}_{[a, o_0 + j + impl.\mathrm{offset}(\overline{t}, f)]}(\overline{\sigma}_2\overline{\sigma}_3)$$

  Notice however, by way of (3,5), that $o_2 = o_1 + impl.\mathrm{offset}(\overline{t}, f) = o_0 + j + impl.\mathrm{offset}(\overline{t}, f)$, so we get

  $$impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \ell\texttt{+}(j + impl.\mathrm{offset}(\overline{t}, f)) : \mathrm{ptr}_{[a,o_2]}(\overline{\sigma}_2\overline{\sigma}_3)$$

  A final application of L-Sub to this and $impl; \overline{t} \vdash \mathrm{ptr}_{[a,o_2]}(\overline{\sigma}_2\overline{\sigma}_3) \leq \mathrm{ptr}_{[a,o_2]}(\overline{\sigma}_2)$ (derivable via St-Ptr) yields the desired result.

- L-DerefL: Vacuous; no primitive step rule can apply.

- L-DerefR: The step was derived using D-Deref. We assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} *(\tau*)(\ell\texttt{+}j) : \overline{\sigma}_1$ and $e' = \overline{w}_2$. By inversion on L-DerefR, we obtain

**1** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \ell\texttt{+}j : \mathrm{ptr}_{\alpha}(\overline{\sigma}_1\overline{\sigma}_2)$

**2** $impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}_1$

The D-Deref side condition gives

**3** $H(\ell) = \overline{w}_1\overline{w}_2\overline{w}_3, [a', o']$

**4** $\mathrm{size}(impl, \overline{w}_1) = j$

**5** $impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}_1$

**6** $\mathrm{size}(impl, \overline{w}_2) = \mathrm{size}(impl, \overline{\sigma}_1) = k$

**7** $impl.\mathrm{access}([a', o' + j], k)$

From (1), Canonical Forms gives

**8** $\Psi(\ell) = \overline{\sigma}_0\overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3, [a', o']$

**9** $j = \mathrm{size}(impl, \overline{\sigma}_0)$

**10** $\alpha = [a, o + i]$

**11** $\vdash [a', o'] \leq [a, o]$

From the assumption $impl; \overline{t}; \Psi \vdash H : \Psi$ and (8), the Heap Canonical Forms Lemma gives $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w}_1\overline{w}_2\overline{w}_3 : \overline{\sigma}_0\overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3$. Since $\mathrm{size}(impl, \overline{w}_1) = \mathrm{size}(impl, \overline{\sigma}_0) = j$ (by (4,9)), the Value-Type Split Lemma gives $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w}_2\overline{w}_3 : \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3$. Since $\mathrm{size}(impl, \overline{w}_2) = \mathrm{size}(impl, \overline{\sigma}_1)$ (by (6)), the same lemma gives $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w}_2 : \overline{\sigma}_1$, as desired.

- L-Assn: The step must have been derived using D-Assn. We assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} *(\tau*)(\ell\texttt{+}j) = \overline{w} : \overline{\sigma}_1$ and $e' = \overline{w}$, where $H(\ell) = \overline{w}_1\overline{w}_2\overline{w}_3, [a, o]$. By inversion on L-Assn, we get $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w} : \overline{\sigma}_1$, which satisfies the first part of the claim.

  Identically to the L-FetchR case, we use the Canonical Forms, Heap Canonical Forms, and Value-Type Size Lemmas to establish that $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w}_2 : \overline{\sigma}_1$ and $\Psi(\ell) = \overline{\sigma}_0\overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3, [a, o]$. Using the Subsequence Replacement and Value-Type Split Lemmas, we get $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w}_1\overline{w}\overline{w}_2 : \overline{\sigma}_0\overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3$. Plug this and the assumption $impl; \overline{t}; \Psi \vdash H : \Psi$ into Ht-Inf to obtain $impl; \overline{t}; \Psi \vdash H' : \Psi'$, where $\Psi' = \Psi, \ell \mapsto \Psi(\ell)$. Using Weakening, we obtain $impl; \overline{t}; \Psi' \vdash H' : \Psi'$, as desired.

- L-Sub: Follows from inversion, induction, and an application of L-Sub.

**Lemma 26 (Replacement)**

- *If* $D; \Psi; \cdot \vdash_{\mathsf{r}} \mathsf{R}[e]_{\mathsf{r}} : \overline{\sigma}$, *then* $\exists \overline{\sigma}'$ *s.t.* $D; \Psi; \cdot \vdash_{\mathsf{r}} e : \overline{\sigma}'$ *and if* $D; \Psi; \cdot \vdash_{\mathsf{r}} e' : \overline{\sigma}'$ *then* $D; \Psi; \cdot \vdash_{\mathsf{r}} \mathsf{R}[e']_{\mathsf{r}} : \overline{\sigma}$.
- *If* $D; \Psi; \cdot \vdash_{\mathsf{l}} \mathsf{L}[e]_{\mathsf{r}} : \overline{\sigma}, \alpha$, *then* $\exists \overline{\sigma}'$ *s.t.* $D; \Psi; \cdot \vdash_{\mathsf{r}} e : \overline{\sigma}'$ *and if* $D; \Psi; \cdot \vdash_{\mathsf{r}} e' : \overline{\sigma}'$ *then* $D; \Psi; \cdot \vdash_{\mathsf{l}} \mathsf{L}[e']_{\mathsf{r}} : \overline{\sigma}, \alpha$.
- *If* $D; \Psi; \cdot \vdash_{\mathsf{r}} \mathsf{R}[e]_{\mathsf{l}} : \overline{\sigma}$, *then* $\exists \overline{\sigma}'$ *s.t.* $D; \Psi; \cdot \vdash_{\mathsf{l}} e : \overline{\sigma}', \alpha$ *and if* $D; \Psi; \cdot \vdash_{\mathsf{l}} e' : \overline{\sigma}', \alpha$ *then* $D; \Psi; \cdot \vdash_{\mathsf{r}} \mathsf{R}[e']_{\mathsf{l}} : \overline{\sigma}$.
- *If* $D; \Psi; \cdot \vdash_{\mathsf{l}} \mathsf{L}[e]_{\mathsf{l}} : \overline{\sigma}, \alpha$, *then* $\exists \overline{\sigma}'$ *s.t.* $D; \Psi; \cdot \vdash_{\mathsf{l}} e : \overline{\sigma}', \alpha$ *and if* $D; \Psi; \cdot \vdash_{\mathsf{l}} e' : \overline{\sigma}', \alpha$ *then* $D; \Psi; \cdot \vdash_{\mathsf{l}} \mathsf{L}[e']_{\mathsf{l}} : \overline{\sigma}, \alpha$.

**Proof:** Proof proceeds by simultaneous structural induction on $\mathsf{R}$ and $\mathsf{L}$, by cases on their top-level syntactic form. In the base cases we have $\mathsf{R} = [\cdot]_{\mathsf{R}}$ or $\mathsf{L} = [\cdot]_{\mathsf{L}}$ and the property follows immediately. All inductive cases follow directly from the inductive hypothesis and typing rules. For example, consider the case when $\mathsf{R} = (\mathsf{L} = e_2)$. We assume $D; \Psi; \cdot \vdash_{\mathsf{r}} \mathsf{L}[e]_{\mathsf{r}} = e_2 : \overline{\sigma}$ and proceed by cases on the last rule applied in order to reach this conclusion. All but two are impossible:

- L-Assn: By inversion we get

  **1** $D; \Psi; \cdot \vdash_{\mathsf{l}} \mathsf{L}[e]_{\mathsf{r}} : \overline{\sigma}, \alpha$

  **2** $D; \Psi; \cdot \vdash_{\mathsf{r}} e_2 : \overline{\sigma}$

By induction hypothesis on (1) we have that there exists a $\overline{\sigma}'$ s.t. $D; \Psi; \cdot \vdash_{\mathsf{r}} e : \overline{\sigma}'$, and also that $D; \Psi; \cdot \vdash_{\mathsf{l}} \mathsf{L}[e']_{\mathsf{r}} : \overline{\sigma}, \alpha$. Plugging this along with (2) into L-Assn, we get $D; \Psi; \cdot \vdash_{\mathsf{r}} \mathsf{L}[e']_{\mathsf{r}} = e_2 : \overline{\sigma}$, which is the same as $D; \Psi; \cdot \vdash_{\mathsf{r}} \mathsf{R}[e']_{\mathsf{r}} : \overline{\sigma}$.

- L-Sub: Follows from inversion, induction, and an application of L-Sub.

All other cases follow an identical pattern.

**Lemma 27 (Preservation)** *If*

$impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} e : \overline{\sigma}$ *(or* $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{l}} e : \overline{\sigma}, \alpha$*)*
$impl; \overline{t}; \Psi \vdash H : \Psi$
$impl; \overline{t} \vdash H; e \to H'; e'$

*then there exist* $\Psi'$*, extending* $\Psi$*, such that*

$impl; \overline{t}; \Psi'; \cdot \vdash_{\mathsf{r}} e' : \overline{\sigma}$ *(or* $impl; \overline{t}; \Psi'; \cdot \vdash_{\mathsf{l}} e' : \overline{\sigma}, \alpha$*)*
$impl; \overline{t}; \Psi' \vdash H' : \Psi'$

**Proof:** Since it is a right-expression, $e$ is of the form $\mathsf{R}[e_0]_{\mathsf{r}}$ or $\mathsf{R}[e_0]_{\mathsf{l}}$. Consider the former case. It must be the case that the step is of the form $impl; \overline{t} \vdash \mathsf{R}[e_0]_{\mathsf{r}} \to \mathsf{R}[e_0']_{\mathsf{r}}$ (the D-StepR rule) which can only happen under condition $impl; \overline{t} \vdash e_0 \xrightarrow{\mathsf{r}} e_0'$. Then:

- Replacement on the assumed typing derivation gives $\exists \overline{\sigma}'$ s.t. $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} e_0 : \overline{\sigma}'$.
- Subject Reduction gives $\exists \Psi'$ extending $\Psi$ such that $impl; \overline{t}; \Psi'; \cdot \vdash_{\mathsf{r}} e_0' : \overline{\sigma}'$.
- Weakening gives $impl; \overline{t}; \Psi'; \cdot \vdash_{\mathsf{r}} \mathsf{R}[e_0]_{\mathsf{r}} : \overline{\sigma}$.
- Replacement gives $impl; \overline{t}; \Psi'; \cdot \vdash_{\mathsf{r}} \mathsf{R}[e_0']_{\mathsf{r}} : \overline{\sigma}$.

The case when $e$ is of the form $\mathsf{R}[e_0]_{\mathsf{l}}$ proceeds similarly, using the D-StepL rule instead.

**Lemma 28 (Progress)**
*Suppose* $impl; \overline{t}; \Psi \vdash H : \Psi$ *and impl is a sensible implementation.*

- *If* $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} e : \overline{\sigma}$ *then one of the following is true:*
  - *e is of the form* $\overline{w}$ *– an r-value.*
  - *e is legally stuck on impl and* $\overline{t}$*.*
  - *e can step. I.e., there exist* $\mathsf{R}$*,* $e_1$*,* $e_2$*, and* $H'$*, such that*
    - $e = \mathsf{R}[e_1]_{\mathsf{r}}$ *and* $impl; \overline{t} \vdash H; e_1 \xrightarrow{\mathsf{r}} H'; e_2$*, or*
    - $e = \mathsf{R}[e_1]_{\mathsf{l}}$ *and* $impl; \overline{t} \vdash H; e_1 \xrightarrow{\mathsf{l}} H'; e_2$
- *If* $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{l}} e : \overline{\sigma}, \alpha$ *then one of the following is true:*
  - *e is of the form* $*(\tau*)(\ell+i)$ *– an l-value.*
  - *e is legally stuck on impl and* $\overline{t}$*.*
  - *e can step. I.e., there exist* $\mathsf{L}$*,* $e_1$*,* $e_2$*, and* $H'$*, such that*
    - $e = \mathsf{L}[e_1]_{\mathsf{r}}$ *and* $impl; \overline{t} \vdash H; e_1 \xrightarrow{\mathsf{r}} H'; e_2$*, or*
    - $e = \mathsf{L}[e_1]_{\mathsf{l}}$ *and* $impl; \overline{t} \vdash H; e_1 \xrightarrow{\mathsf{l}} H'; e_2$

**Proof:** Proof proceeds by simultaneous induction on the assumed typing derivations, by cases on the last rule used. In the cases where the induction hypothesis applies to a subexpression, there are three cases to consider: (a) the subexpression is a value, (b) the subexpression is legally stuck, or (c) the subexpression can take a step. In case (b), the subexpression is a context ($\mathsf{L}$ or $\mathsf{R}$) whose left- or right-hole is plugged by a legally stuck expression. In each case, we can show that the outer expression is also legally stuck. Take for example the outer expression $e = *(\tau*)(e')$. If the subexpression $e' = \mathsf{R}[\mathit{rstuck}]_\mathsf{r}$, let $\mathsf{R}' = *(\tau*)(\mathsf{R})$. Then, $e = \mathsf{R}'[\mathit{rstuck}]_\mathsf{r}$, which is legally stuck. The cases for left-contexts and left-holes are analogous. In case (c), the induction hypothesis says that the subexpression $e'$ is a context whose hole is plugged by an expression $e_1$ that can take a primitive step to an expression $e_2$. In each case, we can build a context that, when plugged with $e_1$, equals the outer expression $e$; it follows that $e$ can take a step. Take for example $e = *(\tau*)(e')$. If $e' = \mathsf{R}[e_1]_\mathsf{r}$ and $\mathit{impl}; \overline{t} \vdash H; e_1 \xrightarrow{\mathsf{r}} H'; e_2$, we take $\mathsf{R}' = *(\tau*)(\mathsf{R})$. Then $e = \mathsf{R}'[e_1]_\mathsf{r}$, which can take a step. The cases for left-contexts and left-holes are analogous. In the rest of the proof, whenever we apply the induction hypothesis, we consider only the case when the subexpression is a value. The other two cases follow the consistent pattern explained above. We assume $H = H'$ unless otherwise stated.

- L-Short: We assume $\mathit{impl}; \overline{t}; \Psi; \cdot \vdash_\mathsf{r} s : \text{byte}^i$. Take $\mathsf{R} = [\cdot]_\mathsf{R}$, $e_1 = s$, $e_2 = b^i$, and D-Short applies.
- L-Long: Analogous to L-Short, where D-Long applies.
- L-Var{L,R}: Holds vacuously as variables cannot type-check in an empty $\Gamma$.
- L-FetchR: We assume $\mathit{impl}; \overline{t}; \Psi; \cdot \vdash_\mathsf{r} e.f : \overline{\sigma}_2$ where inversion gives

  **1** $\mathit{impl}; \overline{t}; \Psi; \cdot \vdash_\mathsf{r} e : \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3$

  **2** $\mathit{impl}.\text{offset}(\overline{t}, f) = \text{size}(\mathit{impl}, \overline{\sigma}_1)$

  **3** $N\{\ldots \tau\ f \ldots\} \in \overline{t}$

  **4** $\mathit{impl}.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}_2$

  We apply the IH to (1) and consider the case when $e$ is a value, that is $e = \overline{w}$. The Value Split Lemma gives

  **5** $e = \overline{w}_1\overline{w}_2\overline{w}_3$

  such that $\mathit{impl}; \overline{t}; \Psi; \cdot \vdash_\mathsf{r} \overline{w}_j : \overline{\sigma}_j$ for $j \in \{1, 2, 3\}$. The Value-Type Size Lemma gives

  **6** $\text{size}(\mathit{impl}, \overline{\sigma}_1) = \text{size}(\mathit{impl}, \overline{w}_1)$

  **7** $\text{size}(\mathit{impl}, \overline{\sigma}_2) = \text{size}(\mathit{impl}, \overline{w}_2)$

  From (2,6) we have

  **8** $\mathit{impl}.\text{offset}(\overline{t}, f) = \text{size}(\mathit{impl}, \overline{w}_1)$

  (3,4,5,7) allow a step $\mathit{impl}; \overline{t} \vdash H; e \xrightarrow{\mathsf{r}} H'; \overline{w}_2$ via the D-Fetch rule. Thus, we satisfy the third requirement of the theorem with $\mathsf{R} = [\cdot]_\mathsf{R}.f$, $e_1 = e$ and $e_2 = \overline{w}_2$.
- L-FetchL: We assume $\mathit{impl}; \overline{t}; \Psi; \cdot \vdash_\mathsf{l} e.f : \overline{\sigma}_2, [a, o + o']$ where inversion gives

  **1** $\mathit{impl}; \overline{t}; \Psi; \cdot \vdash_\mathsf{l} e : \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3, [a, o]$

  **2** $N\{\ldots \tau\ f \ldots\} \in \overline{t}$

  We apply the IH to (1) and consider the case when $e$ is an l-value, i.e. $e = *(\tau_1*)(\ell\text{+}i)$. (2) is a sufficient condition for $e.f$ to take a primitive step via the D-FetchL rule. We satisfy the third requirement of the theorem with $\mathsf{R} = [\cdot]_\mathsf{L}.f$, $e_1 = e$ and $e_2 = *(\tau*)(\ell\text{+}(i + \mathit{impl}.\text{offset}(\overline{t}, f)))$.
- L-DerefR: We assume $\mathit{impl}; \overline{t}; \Psi; \cdot \vdash_\mathsf{r} *(\tau*)(e) : \overline{\sigma}$. Inverting, we get

**1** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e : \mathrm{ptr}_{\alpha}(\overline{\sigma}\overline{\sigma}_2)$

**2** $impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}$

**3** $impl.\mathrm{access}(\alpha, \mathrm{size}(impl, \overline{\sigma}))$

We apply the IH to (1) and consider the case when $e$ is a value.

Canonical Forms gives $e = \overline{w} = \mathsf{uninit}^i$ or $e = \overline{w} = \ell{+}i$. In the former case, we have an expression of form $*(\tau*)(\mathsf{uninit}^i)$, which is *legally stuck* so we are done. In the latter case, Canonical Forms also gives

**4** $\Psi(\ell) = \overline{\sigma}_0\overline{\sigma}\overline{\sigma}_2, [a', o']$

**5** $\alpha = [a, o + i]$

**6** $\mathrm{size}(impl, \overline{\sigma}_0) = i$

**7** $\vdash [a', o'] \leq [a, o]$

Heap Canonical Forms on (4) gives

**8** $H(\ell) = \overline{w}', [a', o']$

**9** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}' : \overline{\sigma}_0\overline{\sigma}\overline{\sigma}_2$

Value Split on (8) gives $\overline{w}' = \overline{w}_0\overline{w}_1\overline{w}_2$ where

**10** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_0 : \overline{\sigma}_0$

**11** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_1 : \overline{\sigma}$

Value-Type Size on (10,11) gives

**12** $\mathrm{size}(impl, \overline{w}_0) = \mathrm{size}(impl, \overline{\sigma}_0) = i$ (from (6))

**13** $\mathrm{size}(impl, \overline{w}_1) = \mathrm{size}(impl, \overline{\sigma})$

From (3,7), the Alignment Addition Lemma, and clause 5 of sensible implementations, we get

**14** $impl.\mathrm{access}([a', o' + i], \mathrm{size}(impl, \overline{\sigma}))$

Notice that (2,8,12,13,14) form sufficient conditions for $H; \overline{w}$ to take a right-step to $H; \overline{w}_1$ under rule D-Deref, so we satisfy the third requirement of the theorem with $\mathsf{R} = [\cdot]_{\mathsf{R}}$, $e_1 = *(\tau*)(\ell{+}i)$, and $e_2 = \overline{w}_1$.

- L-DerefL: We assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{l}} *(\tau*)(e) : \overline{\sigma}_1, \alpha$ and by inversion we get $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e : \mathrm{ptr}_{\alpha}(\overline{\sigma}_1\overline{\sigma}_2)$. We apply the IH to this and consider the case when $e$ is a value. Canonical Forms says that $e = \mathsf{uninit}^k$ or $e = \ell{+}i$. In the former case, $*(\tau*)(\mathsf{uninit}^k)$ is legally stuck and we are done. In the latter case, $*(\tau*)(\ell{+}i)$ is an l-value.

- L-New: The totality of $impl$.xtype, $impl$.ptr_size, and $impl$.align give sufficient conditions to step.

- L-Assn: We assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e_1 = e_2 : \overline{\sigma}$ where by inversion we have

  **1** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{l}} e_1 : \overline{\sigma}, \alpha$

  **2** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e_2 : \overline{\sigma}$

  By induction on (1,2), we have several cases to consider:

  - $e_1$ is legally stuck. In this case we can show that $e_1 = e_2$ is legally stuck, as explained in the proof prelude.
  - $e_1$ is an l-value and $e_2$ is legally stuck. Then we can show that $e_1 = e_2$ is legally stuck as explained in the prelude.

– $e_1$ is an l-value and $e_2$ can take a step. Then we can show that $e_1 = e_2$ can take a step, as explained in the proof prelude.

– $e_1$ is an l-value ($e_1 = *(\tau*)(\ell{+}i)$) and $e_2$ is a value ($e_2 = \overline{w}''$). Inversion on (1) (L-DerefL) gives

**3** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \ell{+}i : \mathrm{ptr}_{[a,o+i]}(\overline{\sigma\sigma}_2)$

**4** $impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}$

**5** $impl.\mathrm{access}([a, o+i], \mathrm{size}(impl, \overline{\sigma}))$

Canonical Forms on (3) yields

**6** $\Psi(\ell) = \overline{\sigma}_0\overline{\sigma\sigma}_2\overline{\sigma}_3, [a', o']$

**7** $\vdash [a', o'] \leq [a, o]$

**8** $\mathrm{size}(impl, \overline{\sigma}_0) = i$

Heap Canonical Forms on (6) gives

**9** $H(\ell) = \overline{w}', [a', o']$

**10** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}' : \overline{\sigma}_0\overline{\sigma\sigma}_2\overline{\sigma}_3$

The Value Split Lemma on (10) gives

**11** $\overline{w}' = \overline{w}_0\overline{w}_1\overline{w}_2\overline{w}_3$

**12** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_1 : \overline{\sigma}$

**13** $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_0 : \overline{\sigma}_0$

The Value-Type Size Lemma on (12,13) gives

**14** $\mathrm{size}(impl, \overline{w}_1) = \mathrm{size}(impl, \overline{\sigma})$

**15** $\mathrm{size}(impl, \overline{w}_0) = \mathrm{size}(impl, \overline{\sigma}_0) = i$ (by (8))

From (7) and the Alignment Addition Lemma, we get $\vdash [a', o'+i] \leq [a, o+i]$. From this and (5), clause 5 of implementation sensibility gives

**16** $impl.\mathrm{access}([a', o'+i], \mathrm{size}(impl, \overline{\sigma}))$

(9,11,14,15,16) form sufficient conditions for $e_1 = e_2$ to take a step via rule D-Assn. We satisfy the third part of the theorem with $\mathsf{R} = [\cdot]_{\mathsf{R}}$, $H' = H, \ell \mapsto \overline{w}_0\overline{w}''\overline{w}_2\overline{w}_3$, where $*(\tau*)(\ell{+}i) = \overline{w}''$ steps to $\overline{w}''$.

– $e_1$ can step. Then we can show that $e_1 = e_2$ can step as explained in the prelude.

- L-Cast: Supposing the subexpression is a value, D-Cast applies.
- L-FAddr: We assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} (\tau*)\&e \to f : \mathrm{ptr}_{[a,o_1+impl.\mathrm{offset}(\overline{t},f)]}(\overline{\sigma}_2)$ where by inversion we have $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e : \mathrm{ptr}_{a,o_1}(\overline{\sigma}_1\overline{\sigma}_2)$. We apply the IH to the latter and consider the case when $e$ is a value. Canonical Forms gives that either $e = \mathsf{uninit}^i$, in which case $(\tau*)\&\mathsf{uninit}^i \to f$ is legally stuck and we are done. Otherwise, $e = \ell{+}i$ and the expression can step via D-FAddr.
- L-Seq: We assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e_1; e_2 : \overline{\sigma}$ where inversion gives $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e_1 : \overline{\sigma}'$. Applying the IH to this, if $e_1$ is a value then $e_1; e_2$ can step by the D-Seq rule.
- L-Decl: The totality of $impl.$xtype, $impl.$ptr_size, and $impl.$align give sufficient conditions to step.
- L-If: We assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \mathsf{if}\ e_1\ e_2\ e_3 : \overline{\sigma}$, where inversion gives $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e_1 : \mathrm{byte}^i$. By induction, if $e_1$ is a value, Canonical Forms gives $e_1 = w_1 \ldots w_i$ where each $w_i$ can be either a $b$ or $\mathsf{uninit}$. If $\exists i$ s.t. $w_i = \mathsf{uninit}$, then the expression is legally stuck and we are done. Otherwise $e_1 = b_1 \ldots b_i$, which is a a sufficient condition for $(\mathsf{if}\ e_1\ e_2\ e_3)$ to step by D-IfT or D-IfF.
- L-While: D-While applies.
- L-Value, L-ValueEmpty: The expression in question is already a value.

- L-Sub: Falls out of the induction hypothesis.

**Theorem 29 (Type Soundness)** *Let* $(\rightarrow^*)$ *be the reflexive transitive closure of the transition relation* $(\rightarrow)$*. Then, if*

1. $\mathit{impl}; \overline{t}; \cdot; \cdot \vdash_{\mathsf{r}} e : \overline{\sigma}$
2. $\mathit{impl}; \overline{t} \vdash \cdot; e \rightarrow^* H'; e'$

*then* $H'; e'$ *is not illegally stuck on* $\mathit{impl}$ *and* $\overline{t}$*.*

**Proof:** Proof proceeds by induction on the length of the step sequence. In the case of 0 steps, Progress tells us that $\cdot; e$ is not illegally stuck. In the case of $n$ steps, by induction we have $\mathit{impl}; \overline{t} \vdash \cdot; e \rightarrow^{n-1} H''; e''$ and $H''; e''$ not illegally stuck. Preservation tells us that if $\mathit{impl}; \overline{t} \vdash H''; e'' \rightarrow H'; e'$, the type of $e''$ is preserved in the step to $e'$. It follows from Progress that $H'; e'$ is not illegally stuck.

### A.3.2 Implementation Dependencies

**Theorem 30 (Constraint Satisfaction)**

- *If* $\overline{t}; \Gamma \vdash_{\mathsf{r}} e : \tau; S$ *and* $\mathit{impl}$ *is sensible and* $\mathit{impl} \models S$ *and* $\mathit{impl}.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}$*, then* $\mathit{impl}; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}$*.*
- *If* $\overline{t}; \Gamma \vdash_{\mathsf{l}} e : \tau; S$ *and* $\mathit{impl}$ *is sensible and* $\mathit{impl} \models S$ *and* $\mathit{impl}.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}$ *and* $\mathit{impl}.\mathrm{align}(\overline{t}, \tau) = \alpha$*, then* $\mathit{impl}; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{l}} e : \overline{\sigma}, \alpha'$ *and* $\vdash \alpha' \leq \alpha$*.*

**Proof:** Proof proceeds by simultaneous induction on the assumed typing derivations, by cases on the last rule used. Reflexivity of alignment subtyping (Lemma 18) is used implicitly.

- H-Short: We assume $\overline{t}; \Gamma \vdash_{\mathsf{r}} s : \mathrm{short}; \top$. By clause 2 of the definition of a sensible implementation, $\mathit{impl}.\mathrm{xtype}(\overline{t}, \mathrm{short}) = \mathrm{byte}^i$. From this, $\mathit{impl}; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} s : \mathrm{byte}^i$ via L-Short.
- H-Long: Similar to H-Short.
- H-New: We assume $\overline{t}; \Gamma \vdash_{\mathsf{r}} \mathsf{new}\ \tau : \tau*; \top$. Because $\mathit{impl}$ is sensible (clause 4), we know $\mathit{impl}.\mathrm{xtype}(\overline{t}, \tau*) = \mathrm{ptr}_\alpha(\overline{\sigma})$ where $\mathit{impl}.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}$ and $\mathit{impl}.\mathrm{align}(\overline{t}, \tau) = \alpha$. We can plug the latter two facts into L-New to obtain $\mathit{impl}; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} \mathsf{new}\ \tau : \mathrm{ptr}_\alpha(\overline{\sigma})$.
- H-VarR: We assume $\overline{t}; \Gamma \vdash_{\mathsf{r}} x : \tau; \top$ and $\Gamma(x) = \tau$. Plug this along with $\mathit{impl}.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}$ into L-VarR to reach the desired conclusion.
- H-VarL: Assume $\overline{t}; \Gamma \vdash_{\mathsf{l}} x : \tau; \top$ and $\Gamma(x) = \tau$. Plug this along with $\mathit{impl}.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}$ and $\mathit{impl}.\mathrm{align}(\overline{t}, \tau) = \alpha$ into L-VarL to reach the desired conclusion.
- H-Assn: We assume $\overline{t}; \Gamma \vdash_{\mathsf{r}} e_1 = e_2 : \tau; S_1 \wedge S_2$, where inversion gives $\overline{t}; \Gamma \vdash_{\mathsf{l}} e_1 : \tau; S_1$ and $\overline{t}; \Gamma \vdash_{\mathsf{r}} e_2 : \tau; S_2$. Because $\mathit{impl} \models S_1 \wedge S_2$, we know by model definition that $\mathit{impl} \models S_1$ and $\mathit{impl} \models S_2$. Induction on the left subderivation gives $\mathit{impl}; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{l}} e_1 : \overline{\sigma}, \alpha$ where $\overline{\sigma} = \mathit{impl}.\mathrm{xtype}(\overline{t}, \tau)$. Induction on the right subderivation gives $\mathit{impl}; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e_2 : \overline{\sigma}$. Plugging these derivations into L-Assn yields the desired conclusion.
- H-FetchR: We assume $\overline{t}; \Gamma \vdash_{\mathsf{r}} e.f : \tau; S$ where inversion gives

  **1** $N\{\ldots\ \tau\ f\ \ldots\} \in \overline{t}$

  **2** $\overline{t}; \Gamma \vdash_{\mathsf{r}} e : N; S$

  By induction on (2) (with the assumption that $\mathit{impl} \models S$), we get

  **3** $\mathit{impl}; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}$

**4** $impl.\text{xtype}(\overline{t}, N) = \overline{\sigma}$

Because *impl* is sensible (clause 3), under assumption (1), we have

**5** $impl.\text{xtype}(\overline{t}, N) = \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3$

**6** $impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}_2$

**7** $\text{size}(impl, \overline{\sigma}_1) = impl.\text{offset}(\overline{t}, f)$

It follows from (3,5) that

**8** $impl; \overline{t}; \cdot; \Gamma \vdash_{\text{r}} e : \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3$

Plugging (1,6,7,8) into L-FetchR, we obtain $impl; \overline{t}; \cdot; \Gamma \vdash_{\text{r}} e.f : \overline{\sigma}_2$, as desired.

- H-FetchL: We assume $\overline{t}; \Gamma \vdash_{\text{l}} e.f : \tau; S$ with $impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}$ and $impl.\text{align}(\overline{t}, \tau) = [a, o]$. Inversion gives

  **1** $N\{\ldots\ \tau\ f\ \ldots\} \in \overline{t}$

  **2** $\overline{t}; \Gamma \vdash_{\text{l}} e : N; S$

  By induction on (2) (under the assumption that $impl \models S$), we get

  **3** $impl; \overline{t}; \cdot; \Gamma \vdash_{\text{l}} e : \overline{\sigma}', [a', o']$

  **4** $impl.\text{xtype}(\overline{t}, N) = \overline{\sigma}'$

  **5** $impl.\text{align}(\overline{t}, N) = [a'', o'']$

  **6** $\vdash [a', o'] \leq [a'', o'']$

  By clause 3 of implementation sensibility, we get

  **7** $\overline{\sigma}' = \overline{\sigma}_1\overline{\sigma}\,\overline{\sigma}_3$

  **8** $\text{size}(impl, \overline{\sigma}_1) = impl.\text{offset}(\overline{t}, f)$

  **9** $\vdash [a'', o'' + impl.\text{offset}(\overline{t}, f)] \leq [a, o]$

  Plugging (1,3,5,6,8) along with the assumption $impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}$ into L-FetchL yields

  **10** $impl; \overline{t}; \cdot; \Gamma \vdash_{\text{l}} e.f : \overline{\sigma}_2, [a', o' + impl.\text{offset}(\overline{t}, f)]$

  From (6) and the Alignment Addition Lemma we get $\vdash [a', o' + impl.\text{offset}(\overline{t}, f)] \leq [a'', o'' + impl.\text{offset}(\overline{t}, f)]$. Plugging this and (9) into Alst-Trans we get $\vdash [a', o' + impl.\text{offset}(\overline{t}, f)] \leq [a, o]$. This and (10) form the desired conclusion.

- H-Seq: Similar to H-Assn.

- H-DerefR: We assume $\overline{t}; \Gamma \vdash_{\text{r}} *(\tau*)(e) : \tau; S$ and inversion gives $\overline{t}; \Gamma \vdash_{\text{r}} e : \tau*; S$. By induction,

  **1** $impl; \overline{t}; \cdot; \Gamma \vdash_{\text{r}} e : \overline{\sigma}$

  **2** $\overline{\sigma} = impl.\text{xtype}(\overline{t}, \tau*)$

  Because *impl* is sensible (clause 4), we know

  **3** $\overline{\sigma} = \text{ptr}_{\alpha}(\overline{\sigma}')$

  **4** $impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}'$

  **5** $impl.\text{access}(\alpha, \text{size}(impl, \overline{\sigma}'))$

  From (1,3), we get

**6** $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e : \mathrm{ptr}_{\alpha}(\overline{\sigma}')$

Plugging (4,5,6) into L-DerefR, we get $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e : \overline{\sigma}'$, which, together with (4), forms our desired conclusion.

- H-DerefL: Similar to H-DerefR.
- H-Cast: We assume $\overline{t}; \Gamma \vdash_{\mathrm{r}} (\tau*)e : \tau*; S$, where $S = S' \wedge \mathrm{subtype}(\overline{t}, \mathrm{xtype}(\overline{t}, \tau'*), \mathrm{xtype}(\overline{t}, \tau*))$, where by inversion we obtain

  **1** $\overline{t}; \Gamma \vdash_{\mathrm{r}} e : \tau'*; S'$

  Because $impl \models S$, from the definition of $\models$ we get

  **2** $impl \models S'$

  Applying the induction hypothesis to (1,2), we get

  **3** $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e : \overline{\sigma}'$

  **4** $impl.\mathrm{xtype}(\overline{t}, \tau'*) = \overline{\sigma}'$

  Interpreting $S$ under $impl$, we get

  **5** $impl; \overline{t} \vdash \overline{\sigma}' \leq \overline{\sigma}$

  where $\overline{\sigma} = impl.\mathrm{xtype}(\overline{t}, \tau*)$. Plugging (3,5) into L-Sub, we get

  **6** $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e : \overline{\sigma}$

  From (6) and clause 4 of the definition of sensible implementations, we know $\overline{\sigma} = \mathrm{ptr}_{\alpha}(\overline{\sigma}'')$ where $impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}''$. Plugging (5) into L-Cast, we get $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} (\tau*)e : \mathrm{ptr}_{\alpha}(\overline{\sigma}'')$, as desired.
- H-Faddr: We assume $\overline{t}; \Gamma \vdash_{\mathrm{r}} (\tau*)\&e \rightarrow f : \tau*; S$, where

$$\begin{array}{lll} S = S' \wedge \exists \overline{\sigma}_1, \overline{\sigma}_2, a, o & . & \mathrm{xtype}(\overline{t}, N*) = \mathrm{ptr}_{[a,o]}(\overline{\sigma}_1 \overline{\sigma}_2) \\ & \wedge & \mathrm{subtype}(\overline{t}, \mathrm{ptr}_{[a,o+\mathrm{offset}(\overline{t},f)]}(\overline{\sigma}_2), \mathrm{xtype}(\overline{t}, \tau*)) \\ & \wedge & \mathrm{offset}(\overline{t}, f) = \mathrm{size}(\overline{\sigma}_1) \end{array}$$

  Interpreting this constraint under $impl$, and by clause 4 of the definition of sensible implementation, we obtain that there exist $\overline{\sigma}_1, \overline{\sigma}_2, a, o$, such that

  **1** $impl.\mathrm{align}(\overline{t}, N) = [a, o]$

  **2** $impl.\mathrm{xtype}(\overline{t}, N*) = \mathrm{ptr}_{[a,o]}(\overline{\sigma}_1 \overline{\sigma}_2)$

  **3** $impl.\mathrm{xtype}(\overline{t}, \tau*) = \mathrm{ptr}_{\alpha}(\overline{\sigma})$

  **4** $impl; \overline{t} \vdash \mathrm{ptr}_{[a,o+impl.\mathrm{offset}(\overline{t},f)]}(\overline{\sigma}_2) \leq \mathrm{ptr}_{\alpha}(\overline{\sigma})$

  **5** $impl.\mathrm{offset}(\overline{t}, f) = \mathrm{size}(impl, \overline{\sigma}_1)$

  By inversion,

  **6** $\overline{t}; \Gamma \vdash_{\mathrm{r}} e : N*; S'$

  By definition of the models relation,

  **7** $impl \models S'$

  By induction on (6,7), we obtain $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e : \overline{\sigma}'$ and $impl.\mathrm{xtype}(\overline{t}, N*) = \overline{\sigma}'$. From these two facts and (1,2), we obtain

**8** $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e : \mathrm{ptr}_{[a,o]}(\overline{\sigma}_1\overline{\sigma}_2)$

We can plug (5,8) into L-FAddr to derive $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} (\tau*)\&e{\rightarrow}f : \mathrm{ptr}_{[a,o+impl.\mathrm{offset}(\overline{t},f)]}(\overline{\sigma}_2)$. Plugging this and (4) into L-Sub, we obtain $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} (\tau*)\&e{\rightarrow}f : \mathrm{ptr}_{\alpha}(\overline{\sigma})$ (recall that $\mathrm{ptr}_{\alpha}(\overline{\sigma}) = \mathrm{xtype}(\overline{t}, \tau*)$, from (3)), as desired.

- H-If: Similar to H-Assn.
- H-While: Similar to H-Assn.
- H-Decl: We assume $\overline{t}; \Gamma \vdash_{\mathrm{r}} \tau_1\ x;\ e : \tau_2; S$, where inversion gives $\overline{t}; \Gamma, x : \tau_1 \vdash_{\mathrm{r}} e : \tau_2; S$. By induction, $impl; \overline{t}; \cdot; \Gamma, x : \tau_1 \vdash_{\mathrm{r}} e : \overline{\sigma}$ where $impl.\mathrm{xtype}(\overline{t}, \tau_2) = \overline{\sigma}$. Plugging these facts into L-Decl, we derive $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} \tau_1\ x;\ e : \overline{\sigma}$, as desired.

**Corollary 31 (Semi-Portability)**
*If $\overline{t}; \cdot \vdash_{\mathrm{r}} e : \tau; S$ and impl is sensible and $impl \models S$, then $\cdot; e$ is not illegally stuck on implementation impl and declarations $\overline{t}$.*

**Definition 32 (Cast-Free Expressions)**
*An expression e is cast-free if*

1. *$(\tau*)e'$ does not occur in e.*
2. *If $(\tau*)\&e'{\rightarrow}f$ occurs in e and $N\{\ldots \tau'\ f \ldots\} \in \overline{t}$ then $\tau = \tau'$.*

**Theorem 33 (Cast-Free Sufficiency)**

- *If $\overline{t}; \Gamma \vdash_{\mathrm{r}} e : \tau; S$, $impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}$, and e is cast-free, then for any sensible implementation impl, $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e : \overline{\sigma}$.*
- *If $\overline{t}; \Gamma \vdash_{\mathrm{r}} e : \tau; S$, $impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}$, $impl.\mathrm{align}(\overline{t}, \tau) = \alpha$, and e is cast-free, then for any sensible implementation impl, $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e : \overline{\sigma}, \alpha'$ and $\vdash \alpha' \leq \alpha$.*

**Proof:** The proof is similar to that of the Constraint Satisfaction theorem, by simultaneous induction on the assumed typing derivations. Notice that the only cases that yield constraints are those for H-Cast and H-Faddr. The rest of the cases are either trivial or follow from the definition of a sensible implementation; the corresponding proofs proceed identically as in Constraint Satisfaction. Moreover, the H-Cast case cannot occur, ruled out by the first cast-free requirement. The only remaining interesting case is that for H-Faddr. We assume $\overline{t}; \Gamma \vdash_{\mathrm{r}} (\tau*)\&e{\rightarrow}f : \tau*; S$ where inversion gives

**1** $\overline{t}; \Gamma \vdash_{\mathrm{r}} e : N*; S'$

**2** $N\{\ldots \tau\ f \ldots\} \in \overline{t}$

(Note that the types match as required by assumption (2).) By induction on (1), we obtain

**3** $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e : \mathrm{ptr}_{[a,o]}(\overline{\sigma})$

where $impl.\mathrm{xtype}(\overline{t}, N*) = \mathrm{ptr}_{[a,o]}(\overline{\sigma})$. By implementation sensibility clause 4, $impl.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}$. By sensibility clause 3, we have

**4** $\overline{\sigma} = \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3$

**5** $impl.\mathrm{offset}(\overline{t}, f) = \mathrm{size}(impl, \overline{\sigma}_1)$

**6** $impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}_2$

Plugging (3) into L-Sub with St-Ptr, we get

**7** $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e : \mathrm{ptr}_{[a,o]}(\overline{\sigma}_1\overline{\sigma}_2)$

Plugging (5,7) into L-FAddr yields $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} (\tau*)\&e{\rightarrow}f : \mathrm{ptr}_{[a,o+impl.\mathrm{offset}(\overline{t},f)]}(\overline{\sigma}_2)$ as desired.

It is important to note that this proof does not rely at all on constraints (we do not assume $impl \models S$) and that the sensibility of *impl* suffices.

### A.3.3 Array Extension

Figure 18 shows how the syntax and semantics can be augmented with C-like arrays. First, we augment implementations (see Section A.1) with a new component ($\text{val} : \overline{b} \to \mathbb{N}$) which interprets a sequence of bytes into a number in an implementation-prescribed manner. We now show how the definitions, lemmas, and theorems in Section A.3 can be extended for arrays.

**Extended Definition 34 (Sensible Implementations)**
*We add two array-related sensibility clauses:*

6. $\forall \tau, \overline{t}. \exists a, k, o.$ $impl.\text{align}(\overline{t}, \tau) = [a, o]$ *and* $impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}$ *and* $\text{size}(impl, \overline{\sigma}) = k \times a$.
7. $\forall \tau \exists \alpha, \overline{\sigma}.$ $impl.\text{xtype}(\overline{t}, \tau{*}^{\omega}) = \text{ptr}^{\omega}_{\alpha}(\overline{\sigma})$ *and* $impl.\text{xtype}(\overline{t}, \tau) = \overline{\sigma}$ *and* $impl.\text{align}(\overline{t}, \tau) = \alpha$.

**Extended Definition 35 (Legal Stuck State)**
*We extend the syntax for stuck expressions which can appear in right-holes.*

$$
\begin{array}{rcll}
\mathit{rstuck} & ::= & \ldots & \\
& | & \mathsf{new}\ \tau[\overline{w}_1\ \mathsf{uninit}\ \overline{w}_2] & \\
& | & \mathsf{new}\ \tau[\overline{b}] & \text{if } impl.\text{val}(\overline{b}) < 0 \\
& | & \&((\tau{*}^{\omega})(\mathsf{uninit}^{i}))[e] & \\
& | & \&((\tau{*}^{\omega})(\ell{+}i))[\overline{w}_1\ \mathsf{uninit}\ \overline{w}_2] & \\
& | & \&((\tau{*}^{\omega})(\ell{+}i))[\overline{b}] & \text{if the first 4 hypotheses of D-ArrElt hold but not the 5th}
\end{array}
$$

**Extended Lemma 36 (Uninit Type)**
*(No additions.)*

**Proof:** Consider the inductive case, when $\overline{\sigma} = \sigma\overline{\sigma}'$ and $\text{size}(impl, \overline{\sigma}') = j$ and by induction, $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \mathsf{uninit}^{j} : \overline{\sigma}'$. If $\text{size}(impl, \sigma) = k$ and $\sigma \in \{\text{ptr}^{\omega}_{\alpha}(\overline{\sigma}''), \text{ptr}^{\omega}_{\alpha}(N)\}$ then we can derive $impl; \overline{t}; \Psi \models_{\mathrm{w}} \mathsf{uninit}^{k} : \sigma$ by Lw-UninitArr. We can plug this and $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \mathsf{uninit}^{j} : \overline{\sigma}'$ into L-Value to achieve the desired result.

**Extended Lemma 37 (Constant-Size Subtyping)**
*(No additions.)*

**Proof:** In the array subtyping cases (St-RollArr, St-UnrollArr, St-Arr), two array pointers have the same size (given in Figure 18).

**Extended Lemma 38 (Value-Type Size)**
*(No additions.)*

**Proof:**

- Lw-UninitArr: $\text{size}(impl, \mathsf{uninit}^{i}) = \text{size}(impl, \text{ptr}^{\omega}_{\alpha}(\overline{\sigma})) = impl.\text{ptr\_size} = i$.
- Lw-LblArr: $\text{size}(impl, \ell{+}0) = \text{size}(impl, \text{ptr}^{\omega}_{\alpha}(\overline{\sigma})) = impl.\text{ptr\_size}$.

**Extended Lemma 39 (Subtyping Partition)**
*(No additions.)*

**Proof:** The array subtyping cases (St-RollArr, St-UnrollArr, St-Arr) follow immediately because $n = 1$ so $\overline{\sigma}_1 = \overline{\sigma}_{11}$.

**Extended Lemma 40 (Subtyping Type Form)**

- *If* $impl; \overline{t} \vdash \overline{\sigma}' \leq \text{ptr}^{\omega}_{\alpha}(\overline{\sigma})$ *then* $\exists \alpha', i$ *such that* $\overline{\sigma}' = \text{ptr}^{\omega}_{\alpha'}(\overline{\sigma}^{i})$ *or* $\overline{\sigma}' = \text{ptr}^{\omega}_{\alpha'}(N)$ *where* $impl.\text{xtype}(\overline{t}, N) = \overline{\sigma}^{i}$, *and* $\vdash \alpha' \leq \alpha$.

- *If* $impl; \overline{t} \vdash \overline{\sigma}' \leq \mathrm{ptr}^{\omega}_{\alpha}(N)$ *and* $impl.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}$ *then* $\exists \alpha', i$ *such that* $\overline{\sigma}' = \mathrm{ptr}^{\omega}_{\alpha'}(\overline{\sigma}^i)$ *or* $\overline{\sigma}' = \mathrm{ptr}^{\omega}_{\alpha'}(N')$ *where* $impl.\mathrm{xtype}(\overline{t}, N') = \overline{\sigma}^i$, *and* $\vdash \alpha' \leq \alpha$.

**Proof:** By simultaneous induction on the assumed typing derivations, by cases on the last rule used. The cases St-Pad, St-PadAdd, St-Ptr, St-Roll, and St-Unroll cannot occur.

- St-Arr: Follows from inspection and inversion.
- St-UnrollArr: Follows from inspection and inversion, with $\alpha' = \alpha$.
- St-RollArr: Follows from inspection and inversion, with $\alpha' = \alpha$ and $i = 1$.
- St-Refl: Immediate, with $\alpha' = \alpha$, $N = N'$, and $i = 1$.
- St-Seq: We have $impl; \overline{t} \vdash \overline{\sigma}_1\overline{\sigma}_3 \leq \overline{\sigma}_2\overline{\sigma}_4$. It must be the case that either $\overline{\sigma}_1 = \overline{\sigma}_2 = \cdot$ or $\overline{\sigma}_3 = \overline{\sigma}_4 = \cdot$, since array types cannot be broken into subsequences. The property then follows form induction.
- St-Trans: We have $impl; \overline{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_3$ where inversion gives $impl; \overline{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ and $impl; \overline{t} \vdash \overline{\sigma}_2 \leq \overline{\sigma}_3$. There are two possibilities for $\overline{\sigma}_3$:
  1. $\overline{\sigma}_3 = \mathrm{ptr}^{\omega}_{\alpha}(N)$ where $impl.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}$. By induction on the second subderivation, we have either
     - $\overline{\sigma}_2 = \mathrm{ptr}^{\omega}_{\alpha'}(\overline{\sigma}^j)$ where $\vdash \alpha' \leq \alpha$. By induction on the first subderivation, we have either $\overline{\sigma}_1 = \mathrm{ptr}^{\omega}_{\alpha''}(\overline{\sigma}^{j\times k})$ or $\overline{\sigma}_1 = \mathrm{ptr}^{\omega}_{\alpha''}(N')$ where $impl.\mathrm{xtype}(\overline{t}, N') = \overline{\sigma}^j$ and $\vdash \alpha'' \leq \alpha'$. By Alst-Trans, $\vdash \alpha'' \leq \alpha$, as required.
     - $\overline{\sigma}_2 = \mathrm{ptr}^{\omega}_{\alpha'}(N')$ where $impl.\mathrm{xtype}(\overline{t}, N') = \overline{\sigma}$ and $\vdash \alpha' \leq \alpha$. By induction on the first subderivation, we have either $\overline{\sigma}_1 = \mathrm{ptr}^{\omega}_{\alpha''}(\overline{\sigma}^j)$ or $\overline{\sigma}_1 = \mathrm{ptr}^{\omega}_{\alpha''}(N')$ where $impl.\mathrm{xtype}(\overline{t}, N') = \overline{\sigma}^j$. By Alst-Trans, $\vdash \alpha'' \leq \alpha$, as required.
  2. $\overline{\sigma}_3 = \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma}')$. Similar to the above case.

**Extended Lemma 41 (Canonical Forms)**

- *If* $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})$ *then either*
  - $\overline{w} = \mathsf{uninit}^i$ *where* $impl.\mathrm{ptr\_size} = i$.
  - $\overline{w} = \ell{+}0$ *and* $\exists j, \overline{\sigma}', \alpha'$ *such that* $\Psi(\ell) = \overline{\sigma}^j, \alpha'$ *where* $\vdash \alpha' \leq \alpha$.
- *If* $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(N)$ *and* $impl.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}$, *then either*
  - $\overline{w} = \mathsf{uninit}^i$ *where* $impl.\mathrm{ptr\_size} = i$.
  - $\overline{w} = \ell{+}0$ *and* $\exists j, \overline{\sigma}', \alpha'$ *such that* $\Psi(\ell) = \overline{\sigma}^j, \alpha'$ *where* $\vdash \alpha' \leq \alpha$.

**Proof:** By simultaneous induction on the assumed derivations, by cases on the last rule applied. All but two are impossible. Reflexivity of the alignment subtype relation is used implicitly (Lemma 18).

- L-Value: There are two cases:
  - Assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma}_1)$. By inversion we get $impl; \overline{t}; \Psi \vDash_{\mathrm{w}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma}_1)$. Quick inspection reveals that this conclusion could have only been reached by Lw-UninitArr or Lw-LblArr. In the former case, $\overline{w} = \mathsf{uninit}^i$ where $i = impl.\mathrm{ptr\_size}$, and in the latter case, $\overline{w} = \ell{+}0$ where by inversion $\Psi(\ell) = \overline{\sigma}^j, \alpha'$ with $\alpha' = \alpha$.
  - Assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(N)$, where $impl.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}_1$. Vacuous; cannot be derived via L-Val.
- L-Sub: There are two cases:

– Assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})$. By inversion we get $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w} : \overline{\sigma}''$ and $impl; \overline{t} \vdash \overline{\sigma}'' \leq \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})$. By the Subtyping Type Form Lemma, $\overline{\sigma}''$ is one of
  * $\mathrm{ptr}^{\omega}_{\alpha'}(\overline{\sigma}^i)$ where $\vdash \alpha' \leq \alpha$. Applying the IH to $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha'}(\overline{\sigma}^i)$, we get that either $\overline{w} = \mathsf{uninit}^k$ or $\overline{w} = \ell\mathsf{+0}$ where $\Psi(\ell) = \overline{\sigma}^{i \times j}, \alpha''$ and $\vdash \alpha'' \leq \alpha'$. Plugging $\vdash \alpha'' \leq \alpha'$ and $\vdash \alpha' \leq \alpha$ into Alst-Trans we get $\vdash \alpha'' \leq \alpha$, as required.
  * $\mathrm{ptr}^{\omega}_{\alpha'}(N)$ where $impl.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}$ and $\vdash \alpha' \leq \alpha$. By induction and an application of Alst-Trans, like the above case.
– Assume $impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(N)$, where $impl.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}$. Similar to the above case.

**Extended Lemma 42 (Subject Reduction)**
*(No additions.)*

**Proof:**

- L-NewArr: $$\frac{impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma} \qquad impl.\mathrm{align}(\overline{t}, \tau) = \alpha \qquad impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{b} : \overline{\sigma}' \qquad impl.\mathrm{xtype}(\overline{t}, \mathrm{long}) = \overline{\sigma}'}{impl; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \mathsf{new}\ \tau[\overline{b}] : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})}$$

  The assumed step rule is

  $$impl; \overline{t} \vdash H; \mathsf{new}\ \tau[\overline{b}] \xrightarrow{\mathsf{r}} H, \ell \mapsto \mathsf{uninit}^{i \times j}, \alpha; \ell\mathsf{+0} \quad \text{if} \quad \begin{array}{l} \ell \notin \mathrm{Dom}(H) \\ impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma} \\ \mathrm{size}(impl, \overline{\sigma}) = i \\ impl.\mathrm{align}(\overline{t}, \tau) = \alpha \\ impl.\mathrm{val}(\overline{b}) = j \geq 0 \end{array}$$

  We have $H' = H, \ell \mapsto \mathsf{uninit}^{i \times j}, \alpha$. Let $\Psi' = \Psi, \ell \mapsto \overline{\sigma}^j, \alpha$, so $\Psi'(\ell) = \overline{\sigma}^j, \alpha$. Plugging this into Lw-LblArr, and the result into L-Value, yields $impl; \overline{t}; \Psi'; \cdot \vdash_{\mathsf{r}} \ell\mathsf{+0} : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})$, which satisfies the first part of the claim.

  To satisfy the second part of the claim, we first apply the Heap Weakening Lemma to the third assumption to get

  **1** $impl; \overline{t}; \Psi' \vdash H : \Psi$

  From the third step side condition and the Uninit Type Lemma, we have $impl; \overline{t}; \Psi'; \cdot \vdash_{\mathsf{r}} \mathsf{uninit}^i : \overline{\sigma}$. From the Sequence Typing Lemma applied $j$ times, we have $impl; \overline{t}; \Psi'; \cdot \vdash_{\mathsf{r}} \mathsf{uninit}^{i \times j} : \overline{\sigma}^j$. Plugging this and (1) into Ht-Inf we obtain $impl; \overline{t}; \Psi' \vdash H' : \Psi'$, as desired.

- L-ArrElt: $$\frac{\begin{array}{c} impl; \overline{t}; C \vdash_{\mathsf{r}} \ell\mathsf{+}i : \mathrm{ptr}^{\omega}_{[a,o]}(\overline{\sigma}) \\ impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma} \qquad \mathrm{size}(impl, \overline{\sigma}) = k \times a \qquad impl; \overline{t}; C \vdash_{\mathsf{r}} \overline{b} : \overline{\sigma}' \qquad impl.\mathrm{xtype}(\overline{t}, \mathrm{long}) = \overline{\sigma}' \end{array}}{impl; \overline{t}; C \vdash_{\mathsf{r}} \&((\tau*^{\omega})(\ell\mathsf{+}i))[\overline{b}] : \mathrm{ptr}_{[a,o]}(\overline{\sigma})}$$

  The assumed step rule is

  $$impl; ts \vdash H; \&((\tau*^{\omega})(\ell\mathsf{+}i))[\overline{b}] \xrightarrow{\mathsf{r}} H; \ell\mathsf{+}(i + j \times k) \quad \text{if} \quad \begin{array}{l} impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma} \\ \mathrm{size}(impl, \overline{\sigma}) = j \\ impl.\mathrm{val}(\overline{b}) = k \\ H(\ell) = \overline{w}, \alpha' \\ 0 \leq (i + j \times k) < \mathrm{size}(impl, \overline{w}) \end{array}$$

  Applying the Canonical Forms Lemma to the first typing hypothesis, we learn that $i = 0$ and

  **1** $\Psi(\ell) = \overline{\sigma}^h, [a', o']$ for some $h$

  **2** $\vdash [a', o'] \leq [a, o]$

Applying the Heap Canonical Forms Lemma to (1), we get

**3** $H(\ell) = \overline{w}, \alpha'$

**4** $impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \overline{\sigma}^h$

Applying the Value-Type Size Lemma to (4), we have

**5** $\mathrm{size}(impl, \overline{w}) = \mathrm{size}(impl, \overline{\sigma}^h)$

From this and the last side condition in the assumed reduction we get

**6** $j \times k < \mathrm{size}(impl, \overline{\sigma}^h)$

It must be the case then that we can write $\overline{\sigma}^h$ as $\overline{\sigma}^k\overline{\sigma}^g$ (since $\mathrm{size}(impl, \overline{\sigma}) = j$, according to the second side condition in the reduction) where $g > 0$. Using (1), we can write $\Psi(\ell) = \overline{\sigma}^k\overline{\sigma}^g, [a', o']$ and we naturally have $\mathrm{size}(impl, \overline{\sigma}^k) = k \times \mathrm{size}(impl, \overline{\sigma}) = k \times j$. Plugging these two facts into Lw-Lbl and the result into L-Value, we get

**7** $impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} \ell\texttt{+}(j \times k) : \mathrm{ptr}_{[a', o'+j\times k]}(\overline{\sigma}^g)\mathrm{g}$

Using the Alignment Addition Lemma, we can conclude

**8** $\vdash [a', o' + j \times k] \leq [a, o + j \times k]$

From the third hypothesis of the typing assumption, we know $j = \mathrm{size}(impl, \overline{\sigma}) = a \times k'$ for some $k'$. This means that $j = 0 \mod a$. With this, using Alst-Off, we can conclude $\vdash [a, o + j \times k] \leq [a, o]$. Plugging this and (8) into Alst-Trans, we get $\vdash [a', o' + j \times k] \leq [a, o]$. Plugging this into St-Ptr we get $impl; \bar{t} \vdash \mathrm{ptr}_{[a', o'+j\times k]}(\overline{\sigma}^g) \leq \mathrm{ptr}_{[a,o]}(\overline{\sigma})$. Plugging this along with (7) into L-Sub yields $impl; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} \ell\texttt{+}(j \times k) : \mathrm{ptr}_{[a,o]}(\overline{\sigma})$, as desired.

**Extended Lemma 43 (Progress)**
*(No additions.)*

**Proof:**

- L-NewArr: $\dfrac{impl.\mathrm{xtype}(\bar{t}, \tau) = \overline{\sigma} \qquad impl.\mathrm{align}(\bar{t}, \tau) = \alpha \qquad impl; \bar{t}; C \vdash_{\mathrm{r}} e : \overline{\sigma}' \qquad impl.\mathrm{xtype}(\bar{t}, \mathrm{long}) = \overline{\sigma}'}{impl; \bar{t}; C \vdash_{\mathrm{r}} \mathsf{new}\ \tau[e] : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})}$

  We apply the IH to $impl; \bar{t}; C \vdash_{\mathrm{r}} e : \overline{\sigma}'$ and consider the case when $e$ is a value. Clause 2 of implementation sensibility gives that $\overline{\sigma}' = \mathrm{byte}^i$. Canonical Forms gives that either $e = \overline{w}_1\ \mathsf{uninit}\ \overline{w}_2$ or $e = \overline{b}$. In the former case, $\mathsf{new}\ \tau[e]$ is legally stuck. In the latter case, if $impl.\mathrm{val}(\overline{b}) < 0$, then $\mathsf{new}\ \tau[e]$ is again legally stuck. Otherwise it can take a step via rule D-NewArr.

- L-ArrElt: $\dfrac{\begin{array}{c} impl; \bar{t}; C \vdash_{\mathrm{r}} e_1 : \mathrm{ptr}^{\omega}_{[a,o]}(\overline{\sigma}) \\ impl.\mathrm{xtype}(\bar{t}, \tau) = \overline{\sigma} \qquad \mathrm{size}(impl, \overline{\sigma}) = k \times a \qquad impl; \bar{t}; C \vdash_{\mathrm{r}} e_2 : \overline{\sigma}' \qquad impl.\mathrm{xtype}(\bar{t}, \mathrm{long}) = \overline{\sigma}' \end{array}}{impl; \bar{t}; C \vdash_{\mathrm{r}} \&((\tau{*}^{\omega})(e_1))[e_2] : \mathrm{ptr}_{[a,o]}(\overline{\sigma})}$

  We apply the IH to $impl; \bar{t}; C \vdash_{\mathrm{r}} e_1 : \mathrm{ptr}^{\omega}_{[a,o]}(\overline{\sigma})$ and $impl; \bar{t}; C \vdash_{\mathrm{r}} e_2 : \overline{\sigma}'$ and consider the case when both $e_1$ and $e_2$ are values. Canonical Forms on the former gives that either $e_1 = \mathsf{uninit}^{k_0}$ or $e_1 = \ell\texttt{+}0$ with $\Psi(\ell) = \overline{\sigma}^{k_1}, \alpha'$ where $\vdash \alpha' \leq [a, o]$. In the former case, $\&((\tau{*}^{\omega})(e_1))[e_2]$ is legally stuck. Suppose the latter case occurs.

  Clause 2 of implementation sensibility yields $\overline{\sigma}' = \mathrm{byte}^{k_2}$. Canonical Forms gives either $e_2 = \overline{w}_1\ \mathsf{uninit}\ \overline{w}_2$ or $e_2 = \overline{b}$. In the former case, $\&((\tau{*}^{\omega})(e_1))[e_2]$ is legally stuck. In the latter case, if $impl.\mathrm{val}(\overline{b}) < 0$, $\&((\tau{*}^{\omega})(e_1))[e_2]$ is again legally stuck. Suppose then that $impl.\mathrm{val}(\overline{b}) = k$ for $k \geq 0$.

  Heap Canonical Forms on $\Psi(\ell) = \overline{\sigma}^{k_1}, \alpha'$ gives $H(\ell) = \overline{w}, \alpha'$. Suppose that $i + \mathrm{size}(impl, \overline{\sigma}) \times k \geq \mathrm{size}(impl, \overline{w})$. Then $\&((\tau{*}^{\omega})(e_1))[e_2]$ is legally stuck. Otherwise, $H(\ell) = \overline{w}, \alpha'$ and $i{+}\mathrm{size}(impl, \overline{\sigma}){\times}k < \mathrm{size}(impl, \overline{w})$ give sufficient conditions to take a step via D-ArrElt.

**Extended Lemma 44 (Constraint Satisfaction)**
*(No additions.)*

**Proof:**

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

  By induction on $\overline{t}; \Gamma \vdash_{\mathrm{r}} e : \mathrm{long}; S$ we get $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e : \overline{\sigma}'$ where $impl.\mathrm{xtype}(\overline{t}, \mathrm{long}) = \overline{\sigma}'$. Plugging these two facts along with $impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}$ and $impl.\mathrm{align}(\overline{t}, \tau) = \alpha$ into L-NewArr yields $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} \mathsf{new}\ \tau[e] : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})$. From clause 7 of implementation sensibility, we get $impl.\mathrm{xtype}(\overline{t}, \tau{*}^{\omega}) = \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})$, as desired.

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

  We know $impl \models S_1 \wedge S_2$ so by definition of $(\models)$ we have $impl \models S_1$ and $impl \models S_2$. Using this, we can apply the induction hypothesis to the two subderivations, and use clause 7 of implementation sensibility to obtain

  **1** $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e_1 : \mathrm{ptr}^{\omega}_{[a,o]}(\overline{\sigma})$

  **2** $impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma}$

  **3** $impl.\mathrm{align}(\overline{t}, \tau) = [a, o]$

  **4** $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} e_2 : \overline{\sigma}'$

  **5** $impl.\mathrm{xtype}(\overline{t}, \mathrm{long}) = \overline{\sigma}'$

  From clause 6 of implementation sensibility we learn

  **6** $\mathrm{size}(impl, \overline{\sigma}) = k \times a$ for some $k$

  From clause 4 of implementation sensibility we have

  **7** $impl.\mathrm{xtype}(\overline{t}, \tau{*}) = \mathrm{ptr}_{[a,o]}(\overline{\sigma})$

  Plugging (1,2,4,5,6) into L-ArrElt yields $impl; \overline{t}; \cdot; \Gamma \vdash_{\mathrm{r}} \&((\tau{*}^{\omega})(e_1))[e_2] : \mathrm{ptr}^{\omega}_{[a,o]}(\overline{\sigma})$, which together with (7), gives the desired conclusion.

# B Differences Between Report and Paper

This technical report broadly extends the paper with the same title in the following ways:

- The full safety proof is given in Appendix A.
- Section 1.3, ommitted from the paper, contains additional discussion regarding the contributions of the work.
- Section 2 contains two additional motivating examples.
- The language given in the report contains one additional syntactic form for "while" loops.
- Section 3.6 contains additional discussion regarding the difference between left- and right-typings in the high-level semantics and the cast-like syntax on the dereference operator.
- Figures 6 (physical subtyping) and 7 (high-level semantics) are complete. The paper omits a subtyping rule for padding from the former and most rules from the latter.

- Section 3.7 contains additional discussion regarding the low-level type system.
- Section 4 contains a detailed discussion of the array extension, and other extensions which are ommitted from the paper: read-only pointers, byte skipping, and recursive subtyping.

New Syntax

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

New Dynamic Rules

$$\begin{array}{lll} \text{(D-NewArr)} & impl; \overline{t} \vdash H; \mathsf{new}\ \tau[\overline{b}] & \xrightarrow{\mathrm{r}} \quad H, \ell \mapsto \mathsf{uninit}^{i \times j}, \alpha; \ell{+}0 \\ & & \quad \text{if} \quad \ell \notin \mathrm{Dom}(H) \\ & & \qquad impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma} \\ & & \qquad \mathrm{size}(impl, \overline{\sigma}) = i \\ & & \qquad impl.\mathrm{align}(\overline{t}, \tau) = \alpha \\ & & \qquad impl.\mathrm{val}(\overline{b}) = j \geq 0 \\ \text{(D-ArrElt)} & impl; ts \vdash H; \&((\tau{*}^{\omega})(\ell{+}i))[\overline{b}] & \xrightarrow{\mathrm{r}} \quad H; \ell{+}(i + j \times k) \\ & & \quad \text{if} \quad impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma} \\ & & \qquad \mathrm{size}(impl, \overline{\sigma}) = j \\ & & \qquad impl.\mathrm{val}(\overline{b}) = k \\ & & \qquad H(\ell) = \overline{w}, \alpha \\ & & \qquad 0 \leq (i + j \times k) < \mathrm{size}(impl, \overline{w}) \end{array}$$

New High-Level Typing

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

New Subtyping

$$\text{St-Arr}\ \frac{\vdash \alpha_1 \leq \alpha_2}{impl; \overline{t} \vdash \mathrm{ptr}^{\omega}_{\alpha_1}(\overline{\sigma}^i) \leq \mathrm{ptr}^{\omega}_{\alpha_2}(\overline{\sigma})} \qquad \text{St-UnrollArr}\ \frac{impl.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}}{impl; \overline{t} \vdash \mathrm{ptr}^{\omega}_{\alpha}(N) \leq \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})} \qquad \text{St-RollArr}\ \frac{impl.\mathrm{xtype}(\overline{t}, N) = \overline{\sigma}}{impl; \overline{t} \vdash \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma}) \leq \mathrm{ptr}^{\omega}_{\alpha}(N)}$$

New Low-Level Typing

$$\text{L-NewArr}\ \frac{impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma} \qquad impl.\mathrm{align}(\overline{t}, \tau) = \alpha \qquad impl; \overline{t}; C \vdash_{\mathrm{r}} e : \overline{\sigma}' \qquad impl.\mathrm{xtype}(\overline{t}, \mathrm{long}) = \overline{\sigma}'}{impl; \overline{t}; C \vdash_{\mathrm{r}} \mathsf{new}\ \tau[e] : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})}$$

$$\text{L-ArrElt}\ \frac{\begin{array}{c} impl; \overline{t}; C \vdash_{\mathrm{r}} e_1 : \mathrm{ptr}^{\omega}_{[a,o]}(\overline{\sigma}) \\ impl.\mathrm{xtype}(\overline{t}, \tau) = \overline{\sigma} \qquad \mathrm{size}(impl, \overline{\sigma}) = k \times a \qquad impl; \overline{t}; C \vdash_{\mathrm{r}} e_2 : \overline{\sigma}' \qquad impl.\mathrm{xtype}(\overline{t}, \mathrm{long}) = \overline{\sigma}' \end{array}}{impl; \overline{t}; C \vdash_{\mathrm{r}} \&((\tau{*}^{\omega})(e_1))[e_2] : \mathrm{ptr}_{[a,o]}(\overline{\sigma})}$$

$$\text{Lw-UninitArr}\ \frac{impl.\mathrm{ptr\_size} = i}{impl; \overline{t}; \Psi \vdash_{\mathrm{w}} \mathsf{uninit}^{i} : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})} \qquad \text{Lw-LblArr}\ \frac{\Psi(\ell) = \overline{\sigma}^{i}, \alpha}{impl; \overline{t}; \Psi \vdash_{\mathrm{w}} \ell{+}0 : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})}$$

New Size Function Case

$$\mathrm{size}(impl, \sigma) = impl.\mathrm{ptr\_size}\ \text{if}\ \sigma \in \{\mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma}), \mathrm{ptr}^{\omega}_{\alpha}(N)\}$$

Figure 18: Array Syntax and Semantics