A Theory of Platform-Dependent Low-Level Software (Extended Version)

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

Abstract

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

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

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

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

Requiring that code makes no platform-dependent assumptions (e.g., by enforcing poorly understood and informally specified [23] 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 platform 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, allowing implicit platform-dependent assumptions can lead to pernicious defects that lie dormant until one uses a platform violating the assumptions. Whereas defects like dangling-pointer dereferences are largely independent of the language implementation, so testing or verification on the "old platform" can find many of them, defects like assuming two `struct` types have similar data layouts are not. The results can be severe. Conceptually simple tasks like porting an application from a 32-bit machine to a 64-bit machine become expensive and error-prone. Software tested on widely available platforms can break when run on novel hardware such as embedded systems. Widely used compilers cannot change data-representation strategies without breaking legacy code that implicitly relies on undocumented behavior. Section 2.2 discusses some specific real-world examples.

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

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

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

1.1 Overview of Our Approach

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

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

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

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

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

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

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

1.2 Contributions and Caveats

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

- We can define a sound type system for a language with partially unspecified type layouts. The soundness theorem is proved once and for all instead of once for each platform.
- Layout-portability questions are reduced to pointer-cast questions, namely, "when can a pointer to a τ_1 be treated as a pointer to a τ_2?" This question, clearly akin to subtyping, depends on the platform.
- Platform constraints can be described in a first-order theory and extracted statically from the program.
- There should be a notion of "sensible" platform, meaning platforms on which every cast-free program cannot get stuck.

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

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

1.3 Outline

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

2 Examples

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

2.1 Small Code Fragments

Example 1: Accessing Memory

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

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

Platforms choose sizes, alignments, and offsets such that cast-free programs do not fail. For example, if a machine prohibits 8-byte accesses on 4-byte alignments, a compiler might put pad bytes before `f` fields or break 8-byte accesses into two 4-byte accesses. In the latter case, the compiler "supports" 8-byte accesses on 4-byte alignments.

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

Example 2: 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 platform to meet these constraints (and for `g1` and `f1` to have offset 0 and `g2` to have the same offset as `f2`). Our formal model captures this notion precisely.

Example 3: Flattening and Alignment

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

A cast from `struct S2*` to `struct D2*` has similar constraints as in Example 2, but this time the C standard provides no guarantee. In fact, some platforms put pad bytes before `f4` because of alignment restrictions. Our system will generate constraints preventing such a representation mismatch if the program has this cast.

Example 4: Suffix

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

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

Example 5: Arrays

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

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

Previous examples implicitly assumed the destination pointer was not used as an array (i.e., it was used as a pointer to exactly one struct). On many platforms that add trailing padding after field `f2`, a cast from `struct S4*` to `struct D4*` is legal. However, the cast above is broken due to intermittent pad bytes in the layout pointed to by `x`. This issue is orthogonal to array-bounds violations; we must reject the cast even if `x` points to more than 7 elements. Section A.4 extends our model to distinguish pointers to single-objects from pointers to arrays.

Example 6: 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 4.3 adds this orthogonal feature to our model.

Example 7: 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 4.4 extends our model to let implementations indicate on a per-pad basis how they implement assignment to `struct` values.

Example 8: 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.

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

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

2.2 Practical Scenarios

The scope of the platform dependency problem is not precisely known because defects can lie dormant until one changes hardware or compiler. Therefore, like date bugs (such as the famous Y2K problem of last decade), offending code can be difficult to locate and fix.[2]

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 [5]:

> 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.[3]

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" [26].

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 [22]. 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 [27].

3 Core Language

This section develops a formal model that can explain examples 1–4 from Section 2. We define an appropriate core language, a definition of *platform*, a dynamic semantics, a first-order theory that constrains platforms, a type-and-effect system to produce constraints, and the type-soundness result, acquired via a lower-level, platform-dependent type system. Figure 8 on page 15 summarizes the model's judgments.

3.1 Idealized Syntax

For source programs, we consider a small subset of C with some convenient syntactic changes, as defined in Figure 1. Most significantly, we omit functions and make all terms expressions. A program is a sequence of struct definitions ($\overline{t}$) 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 τ 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 ($(\tau*)e$).

[2] Furthermore, semantics-based Y2K solutions [15] serve as inspiration for us, though portability bugs fortunately do not share a worldwide deadline.

[3] Emphasis in original.

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

Figure 2: Syntax extensions for run-time behavior

For pointer dereference ($*(\tau*)(e)$) and pointing to a field ($((\tau*)\&e \to f)$), it is a technical convenience to require a type annotation in the syntax. (Our particular choice happens to correspond to C's syntax.) Dereference in C *is* type-directed (if e has type $\tau*$, then $*e$ reads `sizeof(`τ`)` bytes); our type decoration makes this explicit. The cast in address-of-field expressions helps us support "suffix casts" as in Example 4 of Section 2.

The remaining expression forms are for memory allocation or control flow. $\mathsf{new}\ \tau$ heap-allocates uninitialized space to hold a τ; it is less verbose than `malloc(sizeof(`τ`))`. $(e; e)$ 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 τ bound in e.

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

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

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

3.2 Platforms

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

- A translation of types into a lower-level representation ($\overline{\sigma}$), described below. We write $\Pi.\mathit{xtype}(\overline{t}, \tau)$ for the $\overline{\sigma}$ corresponding to the translation of a type τ assuming type definitions $\overline{t}$.
- An *alignment* function ($\Pi.\mathit{align}$) returns the alignment α used to allocate space for a τ.
- An *offset* function ($\Pi.\mathit{offset}$) takes a field f and returns the number of bytes from the beginning of the nearest enclosing struct to the field f.
- An *access* function takes an alignment α and a size i and returns true if accessing i bytes at an alignment α is not an error.

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

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

Figure 3: Platforms and low-level types

- The size of pointers ($\Pi.\mathit{ptrsize}$) is a constant i.[4]
- An *xliteral* function translates integer literals into byte sequences. The layout of values in memory is platform-dependent.

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

Low-level types (the target of $\Pi.\mathit{xtype}$) are $\overline{\sigma}$, a sequence of σ. For example, if long is four bytes, the translation is byte[4]. The type pad$[i]$ represents i bytes of padding. The type $\text{ptr}_\alpha(\overline{\sigma})$ describes pointers to data described by $\overline{\sigma}$ at alignment α. 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 platforms or disallowing recursive types.

3.3 Dynamic Semantics

The dynamic semantics is a small-step rewrite system for expressions, parameterized by a platform and a sequence of type declarations. Figure 4 holds the full definition for $\Pi; \overline{t} \vdash H; e \to H'; e'$. It is defined via evaluation contexts for conciseness. As in C, the left side of assignments (called left-expressions) are evaluated differently from other expressions (called right-expressions). Therefore, we have two sorts of contexts (L and R) defined by mutual induction and a different sort of primitive reduction ($\xrightarrow{\text{l}}$ and $\xrightarrow{\text{r}}$) for each sort of context [17]. In particular, $\mathsf{R}[e]_\text{r}$ is a right-context R containing a right-hole filled by e and $\mathsf{R}[e]_\text{l}$ is a right-context R containing a left-hole filled by e. Each context contains exactly one right-hole or exactly one left-hole, but not both.

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

D-New extends the heap with a new label holding uninitialized data. The platform determines the alignment and size of the new space, with the latter computed by applying the auxiliary size function to the translation of the allocated type. The resulting value ℓ+0 is a pointer to the beginning of the space. The type system does not prevent getting stuck due to uninitialized data; this issue is orthogonal. D-Let has the same hypotheses as D-New. Because memory management is not our concern, we use heap allocation even for local variables. We substitute $*(\tau*)(\ell\text{+}0)$ for x in the resulting expression.

D-Deref reads data from the heap and the resulting expression is the data. In particular, it extracts a sequence "from the middle" of $H(\ell)$. This sequence is from offset j (where the expression before the step is

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 4: Dynamic Semantics

$*(\tau*)(\ell + j))$ to $j + k$ (where k is the size of the translation of τ). If it is not possible to "carve up" $H(\ell)$ in this way, then the rule does not apply and the machine is stuck. As expected, we also use $\Pi.\mathit{access}$ to model alignment constraints on the memory access.

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

D-Faddr takes a pointer value and increases its offset by the offset of the field f, which is defined by Π. 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 Π to project a subsequence of a value. We do not use the access function here because we are not accessing the heap.[5]

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

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

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

3.4 Constraint Language

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

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

Consider two example formulas:

- $\forall \tau, \bar{t}.\ \mathrm{access}(\mathrm{align}(\bar{t}, \tau), \mathrm{size}(\mathrm{xtype}(\bar{t}, \tau)))$
- 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:
  $\mathrm{subtype}(\bar{t}_0, \mathrm{xtype}(\bar{t}_0, N_1*), \mathrm{xtype}(\bar{t}_0, N_2*))$.

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

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

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

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

Figure 5: Function Symbols for the First-Order Theory

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

1. Size and alignment allows access of all types:
   $\forall \tau, \overline{t}.\ \text{access}(\text{align}(\overline{t}, \tau), \text{size}(\text{xtype}(\overline{t}, \tau)))$

2. Translation of literals respects the translation of their types:
   $\forall s, l, \overline{t}.\ \text{size}(\text{xliteral}(s)) = \text{size}(\text{xtype}(\overline{t}, \text{short}))$
   $\quad \wedge\ \text{size}(\text{xliteral}(l)) = \text{size}(\text{xtype}(\overline{t}, \text{long}))$

3. Greater alignment does not restrict access:
   $\forall \alpha_1, \alpha_2, i.\ (\text{access}(\alpha_1, i)\ \wedge\ \text{subalign}(\alpha_1, \alpha_2)) \Rightarrow \text{access}(\alpha_2, i)$

4. Translation of $\tau*$ respects the alignment and translation of τ:
   $\forall \tau, \overline{t}.\text{subtype}(\overline{t},\ \text{ptr}_{\text{align}(\overline{t},\tau)}(\text{xtype}(\overline{t}, \tau)),\ \ \text{xtype}(\overline{t}, \tau*))$

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

This gives rise to a precise notion of sensible platform:

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

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

- The first field always has offset 0:
  $\forall f, \overline{t}, \tau, N.\ (N\{\tau\ f \dots\} \in \overline{t}) \Rightarrow \text{offset}(\overline{t}, f) = 0$

- Fields are in order and do not overlap:
  $\forall \tau_1, f_1, \tau_2, f_2, N.\ N\{\dots \tau_1\ f_1 \dots \tau_2\ f_2 \dots\} \in \overline{t}\} \Rightarrow$
  $(\text{offset}(\overline{t}, f_1) + \text{size}(\text{xtype}(\overline{t}, \tau_1)) \leq \text{offset}(\overline{t}, f_2))$

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

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

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

Figure 6: Physical Subtyping and Alignment Subtyping

3.5 Static Semantics and Constraint Generation

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

Subtyping: We use a subtyping relation on low-level types to formalize when data described by $\overline{\sigma}$ can also be described by $\overline{\sigma}'$, and hence when pointer casts are safe. This notion has been called *physical subtyping* because it relies on actual memory layouts. Because we take a byte-for-byte view of memory, our notion of physical subtyping is richer than those defined in prior work [9, 36, 31], which are at the level of ground C types instead of bytes. For example, our definition allows casting a `struct S{short x;short y}*` to a `struct D{long a;}*` on many platforms, whereas prior definitions forbid it. The rules for our judgment $\Pi; \overline{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ appear in Figure 6.

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

Although we allow sequence-shortening under pointer types, it is *not* correct to allow shortening as a subtyping rule because a supertype should have the same size as a subtype (we can prove our rules have this property by induction on a subtyping derivation). This fact may seem odd to readers not used to subtyping in a language with explicit pointers. It is why C correctly disallows casts between struct types (as opposed to pointers to structs).

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

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

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

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

$$\text{S-Short}\ \frac{}{\overline{t};\Gamma \vdash_{\mathsf{r}} s : \text{short};\text{true}} \qquad \text{S-Long}\ \frac{}{\overline{t};\Gamma \vdash_{\mathsf{r}} l : \text{long};\text{true}} \qquad \text{S-New}\ \frac{}{\overline{t};\Gamma \vdash_{\mathsf{r}} \mathsf{new}\ \tau : \tau*;\text{true}} \qquad \text{S-Var}\ \frac{\Gamma(x) = \tau}{\overline{t};\Gamma \vdash_{\mathsf{r}} x : \tau;\text{true}}$$

$$\text{S-Assn}\ \frac{\overline{t};\Gamma \vdash_{\mathsf{l}} e_1 : \tau; S_1 \qquad \overline{t};\Gamma \vdash_{\mathsf{r}} e_2 : \tau; S_2}{\overline{t};\Gamma \vdash_{\mathsf{r}} e_1 = e_2 : \tau; S_1 \wedge S_2} \qquad \text{S-Fetch}\ \frac{\overline{t};\Gamma \vdash_{\mathsf{r}} e : N; S \qquad N\{\ldots\tau\ f\ldots\} \in \overline{t}}{\overline{t};\Gamma \vdash_{\mathsf{r}} e.f : \tau; S}$$

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

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

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

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

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

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

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

platform-dependent type-checking. This approach can be recovered from ours: we can check the constraint against a platform.

The interesting rules are S-Cast and S-Faddr because the constraint $S_{sensible}$ in Section 3.4 suffices to ensure other expression forms (such as dereferences and assignments) cannot fail due to a platform dependency. The constraints directly describe the implicit assumptions made in Examples 2, 3, and 4 in Section 2. The key insight here is that we can allow a pointer cast to assign an arbitrary type to an expression, but the cast will only be deemed legal on platforms that model the associated layout constraint. Recall $\text{subtype}(\overline{t}, \overline{\sigma}_1, \overline{\sigma}_2)$ is the logical formula corresponding to $\Pi; \overline{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$. The S-Faddr constraint is much more complicated because it must state that there is some sequence of fields starting at the offset of field f that can be viewed as a τ.

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 [31, 24] to ensure that the casted value actually has the type dictated by the cast.

3.6 Metatheory and Low-Level Static Semantics

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

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

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

Our memory-safety proof employs a low-level type system that captures the relevant invariants that evaluation preserves. The main judgment of this type system has the form $\Pi; \overline{t}; \Psi; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}$ where Ψ gives a type to the heap.[6] This system has implicit subsumption, which is necessary for a step via D-Cast to preserve typing:

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

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

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

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

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

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

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

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

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

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

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

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

This is a corollary to Theorems 2 and 3.

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

Dynamic Semantics:

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

High−Level Static Semantics:

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

Low−Level Static Semantics:

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

Figure 8: Summary of Judgments

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

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

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

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

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

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

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

4 Extensions

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

4.1 Arrays

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

Rather than conservatively assume all pointers may point to arrays, the types distinguish pointers to one object ($\tau*$ as already defined) from pointers to arrays ($\tau*^{\omega}$; the ω just distinguishes it from $\tau*$). This

Syntax: $\tau ::= \ldots \mid \tau*^{\omega}$ Platforms: $\Pi.val(\overline{b}) = i$

$e ::= \ldots \mid \mathsf{new}\ \tau[e] \mid \&((\tau*^{\omega})(e))[e]$

$\mathsf{R} ::= \ldots \mid \mathsf{new}\ \tau[\mathsf{R}] \mid \&((\tau*^{\omega})(\mathsf{R}))[e] \mid \&((\tau*^{\omega})(\ell{+}i))[\mathsf{R}]$

$\sigma ::= \ldots \mid \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma}) \mid \mathrm{ptr}^{\omega}_{\alpha}(N)$

Dynamic semantics:

D-NewArr

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

D-ArrElt

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

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:

Arr

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

S-NewArr

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

S-ArrElt

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

Figure 9: Additions for Arrays

dichotomy is common in safe C-like languages [31, 24], can be approximated via static analysis over C code, and is necessary to identify what platform assumptions are due only to arrays. The low-level types (σ) make the same distinction.

We add two right-expression forms. First, $\mathsf{new}\ \tau[e]$ creates a pointer to a heap-allocated array of length e. (Because e will evaluate to a byte-sequence $\overline{b}$, a platform must interpret $\overline{b}$ as an integer; we use $\Pi.val$ 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 $\mathsf{new}\ \tau[e]$ from being stuck if e has uninitialized bytes or e is negative.[7]

Second, $\&((\tau*^{\omega})(e_1))[e_2]$ is more easily read as $\&e_1[e_2]$; the size of τ guides the dynamic semantics like it does with pointer dereferences. This form produces a pointer to one array element, which can be dereferenced or assigned through. The dynamic rule D-ArrElt produces the pointer $\ell{+}(i + j \times k)$ where the array begins at $\ell{+}i$, elements have size j, and e_2 evaluates to k. However, the two hypotheses on the right perform a *run-time bounds check*; our type system does not prevent this check from failing and therefore the machine being stuck.[8] The bounds-check on $\&((\tau*^{\omega})(e_1))[e_2]$ ensures an ensuing dereference can never fail.

With this economical addition of arrays, we can design constraints and subtyping such that the only failures are bounds-checks. A key issue is alignment: Given the alignment of e_1, how can we know the alignment of $\&((\tau*^{\omega})(e_1))[e_2]$ without statically constraining the value of e_2? The solution taken by every sensible C platform is to ensure the size of τ 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.

Finally but most importantly, we consider subtyping for pointer-to-array types. Analogues of Unroll and Roll are sound for types of the form $\mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})$, but Ptr must be replaced with a more restrictive rule. Therefore, Arr requires the element type of the subtype to be the element type of the supertype repeated i

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

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

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

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

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

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

Figure 10: Additions for Platform Selection

times. This is more lenient than strict invariance. For example, it supports the platform-dependent idiom of treating an array of: `struct { short i1; short i2; short i3; short i4;};` as an array of `short`. We have proven safety given this subtyping rule (Appendix A).

The Arr rule does *not* support subtyping such as:
$D \vdash \mathrm{ptr}^{\omega}_{\alpha}(\text{byte byte byte}) \le \mathrm{ptr}^{\omega}_{\alpha}(\text{byte byte})$. 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.

4.2 Platform Selection

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

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

If the size of `int` equals the size of `long`, the buffer can be directly used at type `long*`. Otherwise, the function `convert` allocates a new `long*` buffer, copies the elements from `ib` into it, and assigns it into `lb`. The test offers a common-case short path, avoiding a copy on many platforms while remaining portable.

Figure 10 shows the additions to our model to support this idiom. The pcase form has n branches, each guarded by a constraint. Given a platform Π, pcase steps to a branch whose guard is true under Π (shown in D-Pcase). The output constraint in the high-level typing rule (S-Pcase) encodes two important properties. The first conjunct requires the constraint for each branch's body to hold only if the constraint guarding the branch holds. In particular, an implication holds vacuously on platforms not modeling the guard. The second conjunct demands that at least one of the guards is true. The low-level typing rule (L-Pcase) similarly demands that at least one guard is true under Π and its corresponding expression type-checks under Π. Using pcase, the previous example can be written as:

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

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

4.3 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 the lack of qualifier polymorphism [16] 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.

4.4 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 7 showed a contrived situation 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 $\mathrm{skip}[i]$ to σ 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 $\Pi.\mathit{xtype}(\overline{t}, \tau)$. 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:

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

4.5 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 $\Pi.\mathit{xtype}(\overline{t}, N_1) = \mathrm{byte}^4\mathrm{ptr}_\alpha(N_1)$ and $\Pi.\mathit{xtype}(\overline{t}, N_2) = \mathrm{byte}^4\mathrm{ptr}_\alpha(N_2)$. We would be unable to show $\Pi; \overline{t} \vdash \mathrm{byte}^4\mathrm{ptr}_\alpha(N_1) \leq \mathrm{byte}^4\mathrm{ptr}_\alpha(N_2)$.

Sound solutions to this sort of limitation are well-known [6]. 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 $\mathrm{ptr}_\alpha(N_1) \leq \mathrm{ptr}_\alpha(N_2)$.

While our formal exposition opts for simplicity over expressiveness, our tool favors the latter and supports proper recursive subtyping.

5 Implementation

To give our theory a practical outlet, we have implemented a bug-finding tool that is directly inspired by the formal model. In general, a tool based on the model should extract a layout constraint S from a program and present informative warnings based on S. We see several ways to achieve this. For example:

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

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

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

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

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

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

5.1 Tool Overview

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

The cast gatherer uses an interprocedural points-to analysis[9] to determine which memory layouts an expression may point to at run-time. A first pass analyzes all `malloc`[10] sites and records a map of associated program points and their allocation-time types in a *type table*. In addition to allocation sites, program points for local variables that participate in pointer casts are also entered in the type table. A second pass analyzes each pointer cast `(`τ`*)e`. For each entry `(`p`,`τ`'*)` in the type table, if `e` may-alias the allocation expression at program point p (hence `e` may point to memory with run-time type τ`'*`), a pointer cast from τ`'*` to τ`*` is recorded.

The list of pointer casts output by the cast gatherer is passed to the cast analyzer along with a set of *platform descriptions* chosen by the user: one for the host and one or more for each target. A platform

[9] We use the points-to analysis that ships with CIL [1].

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

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

Figure 11: Tool Output on Spread

description is a module, written by us in Caml, that implements a platform's layout policy by defining exactly the platform functions in our model (recall Figure 3). We have implemented many such platform descriptions, most of which represent real platforms, and some of which implement imagined platforms that are still within the C language specification.

For each pointer cast, the cast analyzer generates and checks the corresponding constraint. First, the constraint is checked against the host description. If it is true (hence the cast is legal on the host), it is checked against each target. If it is false against any of the targets, the relevant warnings (described below) are output. The constraint checker was implemented manually and is specialized to our current set of constraints. It queries the physical subtyping relation, for which we have implemented an algorithm, and the platform description functions.

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

5.2 Case Study

We ran our tool on Spread [3], a high-performance messaging service intended for use by distributed applications. We chose Spread because (a) it contained a reported layout portability bug [2], and (b) it is intended to be portable. Our tool issued two warnings, one of which was the known bug, and the other a new bug that to our knowledge has not been reported on the developer mailing lists. No false positives were reported.

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

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

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

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

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

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

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

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

6 Previous Work

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

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

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

6.1 Assuming a Platform

Most closely related is the "physical type-checking" work of Chandra *et al.* [9, 36], which motivated our work considerably. Their tool classifies C casts as "upcasts", "downcasts", or "neither", reporting a warning for the last possibility. Their notion of physical subtyping is at a higher level than ours (requiring that types, offsets, and field names match), and they neither parameterize their system by a platform nor produce descriptions of sets of platforms. Checking code against a new platform would require reverification and changing their tool. They present no metatheory validating their approach.

CCured [31], a memory-safe C platform, includes physical type-checking to reduce the number of casts that require run-time checks. That is, CCured permits casts that work in practice but are not allowed by the C standard. The allowed casts are safe under a padding strategy used by common C compilers for the x86 architecture, which covers some but certainly not all platforms. An interesting avenue for future work is to reimplement the CCured type analysis engine using our platform-as-parameter approach, thus making its memory-safety guarantee portable.

Work on typed assembly language and proof-carrying code [30, 29, 13, 10, 21] 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 [34, 4] has taken a lower-level view than our formalism by treating addresses as integers and exposing that pointer arithmetic can move between adjacent data objects. These approaches provide less help for writing platform-dependent code because verification of type-safety is repeated for each platform. In practice, defining a new platform is an enormous amount of work.

Various program analyses for C (e.g., [37, 28]) have represented structs and unions with explicit bytes and offsets by assuming one particular platform.

6.2 Safe C

Memory-safe dialects or implementations of C, such as CCured [31, 32, 12], Deputy [11], SAFECode [14], and Cyclone [24, 19, 18], do not solve the platform dependency problem. Rather, they may reject (at compile-time) or terminate (at run-time) programs that attempt platform-dependent operations, or they may support only certain platforms (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 these systems include run-time systems (automatic memory managers, type-tag checkers, etc.) that themselves make platform-dependent assumptions! For example, Cyclone assumes 32-bit integers and pointers, and making this code more portable is a top request from actual users.

6.3 Formalizing C

Recent work by Leroy *et al.* [25, 8] uses Coq to prove a C compiler correct. Their operational semantics for C distinguishes left and right expressions much as we do. However, their source language omits structs, avoiding many alignment and padding issues, and their metatheory proves correctness only for correct source programs, presumably saying nothing about platform-dependent code.

Norrish's HOL formalization of C [33] includes structs and uses a global namespace mapping struct names to sequences of typed fields, like our work. However, he purposely omits padding and alignment from his formalism. He has no separable notion of a platform; instead he models platform choices as nondeterminism.

6.4 Low-Level Code without C

Our work has been C-centric, whereas other projects have started with languages at higher levels of abstraction and added bit-level views for low-level programming. (See [7, 20] for just two recent examples.) We believe this complementary approach would benefit from our constraint-based view rather than choosing just between completely high-level types and completely low-level ones.

C-- [35] makes data representation and alignment explicit, but C-- is not appropriate for writing platform-dependent code. Rather, it is a low-level language designed as a target for compiling high-level languages. It has explicit padding on data (a compiler inserts bits where desired) and explicit alignment on all memory accesses.[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.

7 Conclusions and Future Work

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

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

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

Looking to the future, we have begun investigating the issue of checking that a C program has observationally equivalent behavior on a set of platforms. This notion of portability needs to take into account issues such as such as endianness, integer overflow, and perhaps floating-point roundoff behavior. We believe that our notion of platform selection described in Section 4.2 (`pcase`) is instrumental in such a framework. The idea is to translate a non-preprocessed C program into a version of C with a `pcase` primitive, where preprocessor directives and `if` statements encoding platform selection are translated to corresponding `pcase` statements. Then, we can extract from this program the necessary proof obligations under which all `pcase` branches are equivalent when executed on their respective platforms. Since `pcase` branches can be arbitrary code, checking their equivalence is undecidable. We envision discharging the proof obligations in an interactive proof environment.

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

References

[1] CIL: Infrastructure for C program analysis and transformation. http://hal.cs.berkeley.edu/cil/.

[2] Spread: Portability bug on Solaris 8. http://commedia.cnds.jhu.edu/ pipermail/spread-users/2002-November/001185.html.

[3] The Spread Toolkit. http://www.spread.org.

[4] A. Ahmed and D. Walker. The logical approach to stack typing. In *International Workshop on Types in Language Design and Implementation*, 2003.

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

[6] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4), 1993.

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

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

[9] S. Chandra and T. Reps. Physical type checking for C. In *Workshop on Program Analysis for Software Tools and Engineering*, 1999.

[10] 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*, 2003.

[11] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. Necula. Dependent types for low-level programming. In *European Symposium on Programming*, 2007.

[12] 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*, 2003.

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

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

[15] 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*, 1999.

[16] J. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, 1999.

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

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

[19] 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*, 2002.

[20] 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*, 2005.

[21] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. *Journal of Automated Reasoning*, 31(3–4), 2003.

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

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

[24] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

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

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

[27] B. Martin, A. Rettinger, and J. Singh. Multiplatform porting to 64 bits. *Dr. Dobb's Journal*, 2005.

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

[29] 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), 1999.

[30] G. Necula. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages*, 1997.

[31] G. 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), 2005.

[32] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages*, 2002.

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

[34] 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*, 2003.

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

[36] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *7th European Software Engineering Conference 7th ACM Symposium on the Foundations of Software Engineering*, 1999.

[37] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM Conference on Programming Language Design and Implementation*, 1995.

A Proofs

This appendix is organized as follows. Section A.2 (page 30) develops the proof of memory safety relative to the low-level type system. Section A.3 (page 48) develops the connection between the two type systems and proves the key layout portability theorems. Sections A.4 (page 51) and A.5 (page 57) show how the metatheory extends for the array and platform selection extensions, respectively.

A.1 Preliminaries

Figure 12 gives the syntax for our C-like source language. Figure 14 gives an extended, "low-level" version of this language, where types are replaced by physical layouts, and runtime values are added. The syntax for heaps, heap typings, and evaluation contexts is also shown. Figure 15 gives the dynamic semantics for the low-level language and Figure 16 defines a set of value and type size functions used throughout the low-level semantics. Figures 17 and 18 give the static semantics for the low-level language. Figure 19 gives the static semantics for the high-level language.

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 $\cdot$ to denote the empty sequence. For space-saving purposes, we occasionally write two inference rules $\dfrac{D}{x}$ and $\dfrac{D}{y}$ as $\dfrac{D}{\begin{array}{c}x\\y\end{array}}$ when they have identical hypotheses.

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

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

Constraint Language The high-level semantics (Figure 19) 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 platform component plus a couple of others. Platforms are models in this theory; we write $\Pi \models S$ when Π satisfies formula S. The $(\models)$ judgment is entirely ordinary, defined inductively over the structure of S. For example, $\Pi \models S_1 \wedge S_2$ exactly when $\Pi \models S_1$ and $\Pi \models S_2$; and $\Pi \models \forall x : s.S$ 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 13 summarizes the function symbols and their interpretation under Π. For an example involving function symbols, suppose we have a constraint $S = (\text{align}(\overline{t}, N) = [a, o] \wedge \text{align}(\overline{t}, \tau) = [a, o + \text{offset}(\overline{t}, f)])$ and want to know whether a platform Π models S ($\Pi \models S$). Π models S iff $\Pi \models \text{align}(\overline{t}, N) = [a, o]$ and $\Pi \models \text{align}(\overline{t}, \tau) = [a, o + \text{offset}(\overline{t}, f)]$. To satisfy the former conjunct, $\Pi.\mathit{align}(\overline{t}, N) = [a, o]$ must be true. To satisfy the latter, $\Pi.\mathit{align}(\overline{t}, \tau) = [a, o + \Pi.\mathit{offset}(\overline{t}, f)]$ must be true.

(types)	τ	::=	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)	Γ	::=	$\overline{x{:}\tau}$

Figure 12: Syntax for the Source Language

symbol	interpretation under Π
xtype$(\bar{t}, \tau)$	$\Pi.\mathit{xtype}(\bar{t}, \tau)$
align$(\bar{t}, \tau)$	$\Pi.\mathit{align}(\bar{t}, \tau)$
offset$(\bar{t}, f)$	$\Pi.\mathit{offset}(\bar{t}, f)$
access(α, i)	$\Pi.\mathit{access}(\alpha, i)$
xlit(s)	$\Pi.\mathit{xlit}(s)$
xlit(l)	$\Pi.\mathit{xlit}(l)$
ptrsize	$\Pi.\mathit{ptrsize}$
size$(\overline{\sigma})$	size$(\Pi, \overline{\sigma})$
size$(\overline{w})$	size$(\Pi, \overline{w})$
subtype$(\bar{t}, \overline{\sigma}_1, \overline{\sigma}_2)$	$\Pi; \bar{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$
subalign(α_1, α_2)	$\vdash \alpha_1 \leq \alpha_2$

Figure 13: Constraint Language Function Symbols

Definition 8 (Sensible Platforms)
A platform Π *is said to be* <u>*sensible*</u> *if*

1. $\forall\tau.\ \Pi.\mathit{access}(\Pi.\mathit{align}(\bar{t}, \tau), \text{size}(\Pi, \Pi.\mathit{xtype}(\bar{t}, \tau)))$.

2. $\exists i.\ \Pi.\mathit{xtype}(\bar{t}, \text{short}) = \text{byte}^i$ *and* $\forall s \exists \bar{b}.\ \Pi.\mathit{xlit}(s) = \bar{b}$ *and* $\text{size}(\Pi, \bar{b}) = i$.
   $\exists i.\ \Pi.\mathit{xtype}(\bar{t}, \text{long}) = \text{byte}^i$ *and* $\forall s \exists \bar{b}.\ \Pi.\mathit{xlit}(l) = \bar{b}$ *and* $\text{size}(\Pi, \bar{b}) = i$.

3. $\forall \bar{t}.$ *if* $N\{\ldots\tau\ f\ldots\} \in \bar{t}$ *and* $\Pi.\mathit{xtype}(\bar{t}, \tau) = \overline{\sigma}_2$ *then* $\exists \overline{\sigma}_1, \overline{\sigma}_3, a, o, \alpha$ *such that* $\Pi.\mathit{xtype}(\bar{t}, N) = \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3$ *where* $\Pi.\mathit{offset}(\bar{t}, f) = \text{size}(\Pi, \overline{\sigma}_1)$, $\Pi.\mathit{align}(\bar{t}, N) = [a, o]$, $\Pi.\mathit{align}(\bar{t}, \tau) = \alpha$, *and* $\vdash [a, o + \Pi.\mathit{offset}(\bar{t}, f)] \leq \alpha$.

   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 $\Pi.\mathit{align}(\bar{t}, N) = [8, 0]$, $\Pi.\mathit{offset}(\bar{t}, f) = 8$, and $\Pi.\mathit{align}(\bar{t}, \tau) = [4, 0]$. Then, clause (3) would require τ to be 8-byte aligned (alignment $[8, 0 + \Pi.\mathit{offset}(\bar{t}, f)] \equiv [8, 8] \equiv [8, 0]$), which is unreasonable. It is okay for τ to be 4-byte, 2-byte, or 1-byte aligned.

4. $\forall\tau\exists\alpha, \overline{\sigma}.\ \Pi.\mathit{xtype}(\bar{t}, \tau*) = \text{ptr}_\alpha(\overline{\sigma})$ *and* $\Pi.\mathit{xtype}(\bar{t}, \tau) = \overline{\sigma}$ *and* $\Pi.\mathit{align}(\bar{t}, \tau) = \alpha$.

5. *If* $\Pi.\mathit{access}(\alpha, i)$ *and* $\vdash \alpha' \leq \alpha$ *then* $\Pi.\mathit{access}(\alpha', i)$.

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

Figure 14: Syntax for the Full Language (Extends Source Language) and Eval Contexts

Definition 9 (Substitution)

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

Definition 10 (Legal Stuck State) *Let*

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

A state $H; e$ is legally stuck if one of the following is true:

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

Definition 11 (Illegal Stuck State)
A state $H; e$ is illegally stuck on a platform Π 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 $\Pi; \overline{t} \vdash H; e \rightarrow H'; e'$.*

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

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

Figure 15: Dynamic Semantics

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

Figure 16: Size Functions

Definition 12 (Heap Typing Extension)
A heap typing Ψ extends a heap typing Ψ' iff there exists Ψ'' such that $\Psi = \Psi'\Psi''$.

A.2 Low-Level Memory Safety

Lemma 13 (Weakening)

1. *(a) If $D; \Psi; \Gamma \vdash_r e : \overline{\sigma}$ then $D; \Psi\Psi'; \Gamma\Gamma' \vdash_r e : \overline{\sigma}$.*
   (b) If $D; \Psi; \Gamma \vdash_\mathsf{T} e : \overline{\sigma}, \alpha$ then $D; \Psi\Psi'; \Gamma\Gamma' \vdash_\mathsf{T} e : \overline{\sigma}, \alpha$.
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 14 (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 $D; \Psi; \cdot \vdash_r \overline{w} : \overline{\sigma}$.

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

Lemma 15 (Uninit Type)
If $\text{size}(\Pi, \overline{\sigma}) = i$ then $\Pi; \overline{t}; \Psi; \cdot \vdash_r \mathsf{uninit}^i : \overline{\sigma}$.

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 $\text{size}(\Pi, \overline{\sigma}') = j$ and

1 $\Pi; \overline{t}; \Psi; \cdot \vdash_\mathsf{r} \mathsf{uninit}^j : \overline{\sigma}'$

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

2 $\Pi; \overline{t}; \Psi; \cdot \vDash_\mathsf{w} \mathsf{uninit}^k : \sigma$

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

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

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

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

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

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

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

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

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

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

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

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

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 17: 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

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

$$\textsc{St-Pad}\ \frac{\mathrm{size}(\Pi, \sigma) = i}{\Pi;\overline{t} \vdash \sigma \leq \mathrm{pad}[i]} \qquad \textsc{St-PadAdd}\ \frac{}{\Pi;\overline{t} \vdash \mathrm{pad}[i]\mathrm{pad}[j] \leq \mathrm{pad}[i+j]} \qquad \textsc{St-Refl}\ \frac{}{D \vdash \overline{\sigma} \leq \overline{\sigma}} \qquad \textsc{St-Seq}\ \frac{D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2 \qquad D \vdash \overline{\sigma}_3 \leq \overline{\sigma}_4}{D \vdash \overline{\sigma}_1\overline{\sigma}_3 \leq \overline{\sigma}_2\overline{\sigma}_4}$$

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

Value Typing

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

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

Figure 18: Auxiliary Relations

If σ 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 σ is $\mathrm{ptr}_\alpha(N)$, we can use Lw-Uninit2 to derive $\Pi;\overline{t};\Psi;\cdot \models_{\mathrm{r}} \mathsf{uninit}^i : \mathrm{ptr}_\alpha(\overline{\sigma})$ where $\Pi.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}$, then use St-Roll and L-Sub to assign type $\mathrm{ptr}_\alpha(N)$ to uninit^i.

Lemma 16 (Constant-Size Subtyping)
If $D \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ *then* $\mathrm{size}(\Pi, \overline{\sigma}_1) = \mathrm{size}(\Pi, \overline{\sigma}_2)$.

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

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

H-Short
$$\overline{\bar{t};\Gamma \vdash_{\mathsf{r}} i : \text{short}; \top}$$

H-Long
$$\overline{\bar{t};\Gamma \vdash_{\mathsf{r}} d : \text{long}; \top}$$

H-New
$$\overline{\bar{t};\Gamma \vdash_{\mathsf{r}} \mathsf{new}\ \tau : \tau*; \top}$$

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

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

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

H-FetchL
$$\frac{N\{\ldots \tau\ f \ldots\} \in \bar{t} \qquad \bar{t};\Gamma \vdash_{\mathsf{l}} e : N; S}{\bar{t};\Gamma \vdash_{\mathsf{l}} e.f : \tau; S}$$

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

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

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

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

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

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

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

Figure 19: Static semantics for high level language

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

Lemma 17 (Value-Type Size)

1. *If* $\Pi; \bar{t}; \Psi \Vdash_{\mathbb{w}} \overline{w} : \overline{\sigma}$ *then* $\text{size}(\Pi, \overline{w}) = \text{size}(\Pi, \overline{\sigma})$.
2. *If* $\Pi; \bar{t}; \Psi; \Gamma \vdash_{\tau} \overline{w} : \overline{\sigma}$ *then* $\text{size}(\Pi, \overline{w}) = \text{size}(\Pi, \overline{\sigma})$.

Proof:

1. By inspection of the assumed typing derivation:
   - Lw-Byte: $\text{size}(\Pi, b) = \text{size}(\Pi, \text{byte}) = 1$.
   - Lw-Lbl: $\text{size}(\Pi, \ell{+}i) = \text{size}(\Pi, \text{ptr}_{\alpha}(\overline{\sigma}_2)) = \Pi.ptrsize$.

- LW-UNINIT1: $\text{size}(\Pi, \mathsf{uninit}) = \text{size}(\Pi, \text{byte}) = 1$.
- LW-UNINIT2: $\text{size}(\Pi, \mathsf{uninit}^i) = \text{size}(\Pi, \text{ptr}_\alpha(\overline{\sigma})) = \Pi.ptrsize = i$.

2. By induction on the assumed typing derivation, by cases on the last rule used (all but 3 are impossible):
   - L-VALUEEMPTY: trivial; both sizes are 0.
   - L-VALUE: we have $\Pi; \overline{t}; \Psi; \Gamma \vdash_{\text{r}} \overline{w}_1\overline{w}_2 : \sigma_1\overline{\sigma}_2$ where inversion gives $\Pi; \overline{t}; \Psi \vDash_{\text{w}} \overline{w}_1 : \sigma_1$ and $\Pi; \overline{t}; \Psi; \Gamma \vdash_{\text{r}} \overline{w}_2 : \overline{\sigma}_2$. Part (1) of the theorem gives $\text{size}(\Pi, \overline{w}_1) = \text{size}(\Pi, \sigma_1)$. The induction hypothesis gives $\text{size}(\Pi, \overline{w}_2) = \text{size}(\Pi, \overline{\sigma}_2)$. It follows that $\text{size}(\Pi, \overline{w}_1\overline{w}_2) = \text{size}(\Pi, \overline{\sigma}_1\overline{\sigma}_2)$.
   - L-SUB: Follows from induction and the Constant-Size Subtyping Lemma.

Lemma 18 (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 19 (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 20 (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 21 (Typing Partition)
If $D; C \vdash_{\text{r}} \overline{w} : \sigma_1 \ldots \sigma_n$, *then* $\exists \overline{w}_1, \ldots, \overline{w}_n$ *such that* $\overline{w} = \overline{w}_1 \ldots \overline{w}_n$ *and for all* $1 \leq i \leq n$, $D; C \vdash_{\text{r}} \overline{w}_i : \sigma_i$.

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 $\overline{w} = \overline{w}_1\overline{w}_2$, $D; \Psi \models_{\mathrm{w}} \overline{w}_1 : \sigma_1$ follows from inversion, and the other results follow from inversion and induction.

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

Lemma 22 (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 $\overline{\sigma}_1 = \sigma_{11} \ldots \sigma_{1n}$ and $\overline{\sigma}_2 = \sigma_{21} \ldots \sigma_{2m}$. Then by the Typing Partition Lemma there exists $\overline{w}_{11}, \ldots, \overline{w}_{1n}, \overline{w}_{21}, \ldots, \overline{w}_{2m}$ such that $\overline{w} = \overline{w}_{11} \ldots \overline{w}_{1n}\overline{w}_{21} \ldots \overline{w}_{2m}$ 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 $\overline{w}_1 = \overline{w}_{11} \ldots \overline{w}_{1n}$ and $\overline{w}_2 = \overline{w}_{21} \ldots \overline{w}_{2m}$, 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).

Lemma 23 (Value-Type Split) *If*

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

then

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

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 $\Pi; \overline{t}; \Psi \models_{\mathrm{w}} \overline{w}_{11} : \sigma_{11}$

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

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

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

  From (2,3), the induction hypothesis yields

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

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

Plugging (1,4) into L-Value gives $\Pi; \bar{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}_{11}\overline{w}_{12} : \sigma_{11}\overline{\sigma}_{12}$, 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 $\mathrm{size}(\Pi, \overline{\sigma}'_1) = \mathrm{size}(\Pi, \overline{\sigma}_1)$, so the assumption $\mathrm{size}(\Pi, \overline{w}_1) = \mathrm{size}(\Pi, \overline{\sigma}_1)$ ensures $\mathrm{size}(\Pi, \overline{w}'_1) = \mathrm{size}(\Pi, \overline{\sigma}'_1)$. So by induction $\Pi; \bar{t}; C \vdash_{\mathrm{r}} \overline{w}_1 : \overline{\sigma}'_1$ and $\Pi; \bar{t}; C \vdash_{\mathrm{r}} \overline{w}_2 : \overline{\sigma}'_2$. So with one use of L-Sub each, we can derive $\Pi; \bar{t}; C \vdash_{\mathrm{r}} \overline{w}_1 : \overline{\sigma}_1$ and $\Pi; \bar{t}; C \vdash_{\mathrm{r}} \overline{w}_2 : \overline{\sigma}_2$.

Lemma 24 (Subsequence Replacement)

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

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

Lemma 25 (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 26 (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 27 (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 28 (Subtyping Type Form)

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

Lemma 29 (Canonical Forms)

1. *If* $D; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \cdot$ *then* $\overline{w} = \cdot$.
2. *If* $D; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{byte}$ *then* $\overline{w} = b$ *or* $\overline{w} = \mathsf{uninit}$.
3. *If* $D; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{byte}^k$ *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* $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}_\alpha(\overline{\sigma}_1)$ *then* $\exists \overline{\sigma}_0, \overline{\sigma}_2$ *s.t. either*
     - $\overline{w} = \mathsf{uninit}^i$ *where* $\Pi.\mathit{ptrsize} = i$.
     - $\overline{w} = \ell\text{+}i$ *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* $\mathrm{size}(\Pi, \overline{\sigma}_0) = i$.
   - *If* $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}_\alpha(N)$ *where* $\Pi.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}_1$, *then* $\exists \overline{\sigma}_0, \overline{\sigma}_2$ *s.t. either*
     - $\overline{w} = \mathsf{uninit}^i$ *where* $\Pi.\mathit{ptrsize} = i$.
     - $\overline{w} = \ell\text{+}i$ *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* $\mathrm{size}(\Pi, \overline{\sigma}_0) = i$.

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 $D; \Psi; \cdot \vDash_{\mathrm{w}} \overline{w} : \mathrm{byte}$. 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 $D; \Psi \vDash_{\mathrm{w}} w : \mathrm{byte}$ and $D; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w}' : \mathrm{byte}^{k-1}$. 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 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 25) 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 $D; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w} : \mathrm{ptr}_{[a,o+i]}(N)$, where $\Pi.\mathit{xtype}(\bar{t}, N) = \overline{\sigma}_1$. Vacuous; cannot be derived via L-Value.

- L-Sub: There are two sub-cases:
  – Assume $D; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w} : \mathrm{ptr}_{\alpha}(\overline{\sigma}_1)$. Inversion gives $D; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w} : \overline{\sigma}$ 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 $\Pi.\mathit{xtype}(\bar{t}, N) = \overline{\sigma}_1 \overline{\sigma}_k$ for some $\overline{\sigma}_k$, and $\vdash \alpha' \leq \alpha$. Applying the IH to $D; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w} : \mathrm{ptr}_{\alpha'}(N)$ 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 $\mathrm{size}(\Pi, \overline{\sigma}_0) = i$.
      It remains to be shown that α 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 $D; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w} : \mathrm{ptr}_{\alpha}(N)$, where $\Pi.\mathit{xtype}(\bar{t}, N) = \overline{\sigma}_1$. Similar to the above case.

Lemma 30 (Context Reordering)
If $D; \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash_{\mathsf{r}} e : \overline{\sigma}$ *then* $D; \Gamma, x_2 : \tau_2, x_1 : \tau_1 \vdash_{\mathsf{r}} e : \overline{\sigma}$.

Proof: Straightforward induction.

Lemma 31 (Substitution Preserves Types)
Suppose $\Pi.\mathit{xtype}(\bar{t}, \tau) = \overline{\sigma}_2$, $\Pi.\mathit{align}(\bar{t}, \tau) = \alpha$, $\Pi; \bar{t}; \Psi; \Gamma \vdash_{\mathsf{r}} e_2 : \overline{\sigma}_2$, *and* $\Pi; \bar{t}; \Psi; \Gamma \vdash_{\mathsf{l}} e_2 : \overline{\sigma}_2, \alpha$.

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

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 $\Pi; \bar{t}; \Psi; \Gamma, x : \tau \vdash_{\mathsf{r}} y : \overline{\sigma}_1$ and it follows that $\Pi.\mathit{xtype}(\bar{t}, \tau) = \overline{\sigma}_1$. If $x = y$ then $\overline{\sigma}_1 = \overline{\sigma}_2$ and $y\{e_2/x\} = e_2$, and so $\Pi; \bar{t}; \Psi; \Gamma \vdash_{\mathsf{r}} e_2 : \overline{\sigma}_1$ 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 $\Pi; \bar{t}; \Psi; \Gamma \vdash_{\mathsf{r}} y : \overline{\sigma}_1$.
- L-VarL: We assume $\Pi; \bar{t}; \Psi; \Gamma, x : \tau \vdash_{\mathsf{l}} y : \overline{\sigma}_1, \alpha$ and it follows that $\Pi.\mathit{xtype}(\bar{t}, \tau) = \overline{\sigma}_1$ and $\Pi.\mathit{align}(\bar{t}, \tau) = \alpha$. If $x = y$ then $\overline{\sigma}_1 = \overline{\sigma}_2$ and $y\{e_2/x\} = e_2$, and so $\Pi; \bar{t}; \Psi; \Gamma \vdash_{\mathsf{l}} e_2 : \overline{\sigma}_1, \alpha$ 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 $\Pi; \bar{t}; \Psi; \Gamma \vdash_{\mathsf{l}} y : \overline{\sigma}_1, \alpha$.
- L-Assn: We assume $\Pi; \bar{t}; \Psi; \Gamma, x : \tau \vdash_{\mathsf{r}} e = e' : \overline{\sigma}_1$. By inversion and induction (once for left-typing and once for right-typing),

  1 $\Pi; \bar{t}; \Gamma \vdash_{\mathsf{l}} e\{e_2/x\} : \overline{\sigma}_1, \alpha$

  2 $\Pi; \bar{t}; \Gamma \vdash_{\mathsf{r}} e'\{e_2/x\} : \overline{\sigma}_1$

  The desired property follows from an application of L-Assn to these facts and from the definition of substitution.
- L-Fetch{L,R}, L-Deref{L,R}, L-Cast, L-FAddr, L-Seq, L-If, L-While: Inversion, induction applied to each subexpression, and plugging the results back into the rule.
- L-Decl: We assume $\Pi; \bar{t}; \Gamma, x : \tau \vdash_{\mathsf{r}} \tau'\ y;\ e : \overline{\sigma}_1$. By inversion, $\Pi; \bar{t}; \Gamma, x : \tau, y : \tau' \vdash_{\mathsf{r}} e : \overline{\sigma}_1$. By the Context Reordering Lemma, $\Pi; \bar{t}; \Gamma, y : \tau', x : \tau \vdash_{\mathsf{r}} e : \overline{\sigma}_1$. By Weakening, $\Pi; \bar{t}; \Gamma, y : \tau' \vdash_{\mathsf{r}} e_2 : \overline{\sigma}_2$. By induction, $\Pi; \bar{t}; \Gamma, y : \tau' \vdash_{\mathsf{r}} e\{e_2/x\} : \overline{\sigma}_1$. By D-Decl, $\Pi; \bar{t}; \Gamma \vdash_{\mathsf{r}} \tau'\ y;\ e\{e_2/x\} : \overline{\sigma}_1$.

- L-Sub: Inversion, induction, and an application of L-Sub.

Lemma 32 (Subject Reduction)
Suppose Π *is a sensible platform. If*

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

then there exists Ψ'*, extending* Ψ*, such that*

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

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 $\Pi.\mathit{xlit}(s) = \overline{b}$. From L-Short, $\overline{\sigma} = \mathrm{byte}^i$. Because Π is sensible (clause 2), we know $\mathrm{size}(\Pi, \overline{b}) = i$. We can apply Lw-Byte and L-Value i times to derive $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e' : \overline{\sigma}$.
- 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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \mathsf{while}\ e_1\ e_2 : \mathrm{byte}^i$ and $e' = \mathsf{if}\ e_1\ (e_2; \mathsf{while}\ e_1\ e_2)\ s$. Inversion gives

  1 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e_1 : \mathrm{byte}^j$

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

  3 $\Pi.\mathit{xtype}(\overline{t}, \mathrm{short}) = \mathrm{byte}^i$

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

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

  From L-Short, we have

  5 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} s : \mathrm{byte}^i$

  Plugging (1,4,5) into L-If yields $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e' : \mathrm{byte}^i$.
- L-New: The step was derived using D-New. We assume $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \mathsf{new}\ \tau : \mathrm{ptr}_\alpha(\overline{\sigma})$ and have $e' = \ell\mathsf{+0}$ 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 $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}$

  3 $\mathrm{size}(\Pi, \overline{\sigma}) = i$

  4 $\Pi.\mathit{align}(\overline{t}, \tau) = \alpha$

Note that the α 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 $\Pi; \overline{t}; \Psi'; \cdot \vdash_{\mathrm{r}} e' : \mathrm{ptr}_{\alpha}(\overline{\sigma})$, 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 $\Pi; \overline{t}; \Psi' \vdash H : \Psi$

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

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

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

- L-Decl: The step was derived using D-Decl. We assume $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \tau\ x;\ e : \overline{\sigma}_1$, 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 $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}$

  2 $\Pi.\mathit{align}(\overline{t}, \tau) = \alpha$

  By inversion on the assumed typing, we have

  3 $\Pi; \overline{t}; \Psi; \cdot, x : \tau \vdash_{\mathrm{r}} e : \overline{\sigma}_1$

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

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

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

  5 $\Pi.\mathit{access}(\alpha, \mathrm{size}(\Pi, \overline{\sigma}))$

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

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

  Notice that under the same assumptions, we can also derive $\Pi; \overline{t}; \Psi'; \cdot \vdash_{\mathrm{l}} *(\tau*)(\ell\text{+}0) : \overline{\sigma}, \alpha$. (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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e' : \overline{\sigma}_1$. 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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{l}} *(\tau_1*)(\ell\text{+}j).f : \overline{\sigma}_2, [a, o + j']$ and have $e' = *(\tau*)(\ell\text{+}(j + j'))$ where

  1 $\Pi.\mathit{offset}(\overline{t}, f) = j'$

  By inversion we obtain

  2 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{l}} *(\tau_1*)(\ell\text{+}j) : \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3, [a, o]$

  3 $\Pi.\mathit{offset}(\overline{t}, f) = \mathrm{size}(\Pi, \overline{\sigma}_1)$

  4 $N\{\ldots \tau\ f \ldots\} \in \overline{t}$

5 $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}_2$

6 $\Pi.\mathit{align}(\overline{t}, N) = [a_N, o_N]$

7 $\vdash [a, o] \leq [a_N, o_N]$

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

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

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 $\mathrm{size}(\Pi, \overline{\sigma}_0) = j$

11 $o = o'' + j$

12 $\vdash [a', o'] \leq [a, o'']$

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

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

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 $\Pi; \overline{t} \vdash \mathrm{ptr}_{[a', o'+j+j']}(\overline{\sigma}_2\overline{\sigma}_3\overline{\sigma}_4\overline{\sigma}_5) \leq \mathrm{ptr}_{[a, o''+j+j']}(\overline{\sigma}_2\overline{\sigma}_3\overline{\sigma}_4\overline{\sigma}_5)$. Plugging this and (13) into L-Sub, we get

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

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

To prove this, suppose $\Pi.\mathit{align}(\overline{t}, \tau) = \alpha$. By clause (1) of platform sensibility we have $\Pi.\mathit{access}(\alpha, \overline{\sigma}_2)$. By clause (3), 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), we then have $\Pi.\mathit{access}([a_N, o_N + j'], \mathrm{size}(\Pi, \overline{\sigma}_2))$. 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 platform sensibility, we have $\Pi.\mathit{access}([a, o'' + j + j'], \mathrm{size}(\Pi, \overline{\sigma}_2))$, which concludes the proof case.

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

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

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

  By inversion on the assumed typing we have

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

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

  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 $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}_2$ is given by both.

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

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

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

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

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

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

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

  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 $j = \mathrm{size}(\Pi, \overline{\sigma}_0)$

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

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

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

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

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

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

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

  A final application of L-Sub to this and $\Pi; \overline{t} \vdash \mathrm{ptr}_{[a,o_2]}(\overline{\sigma}_2\overline{\sigma}_3) \leq \mathrm{ptr}_{[a,o_2]}(\overline{\sigma}_2)$ (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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} *(\tau*)(\ell{+}j) : \overline{\sigma}_1$ and $e' = \overline{w}_2$. By inversion on L-DerefR, we obtain

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

  2 $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}_1$

  The D-Deref side condition gives

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

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

  5 $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}_1$

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

  7 $\Pi.\mathit{access}([a', o' + j], k)$

  From (1), Canonical Forms gives

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

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

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

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

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

- L-Assn: The step must have been derived using D-Assn. We assume $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} *(\tau*)(\ell + j) = \overline{w} : \overline{\sigma}_1$ 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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w} : \overline{\sigma}_1$, 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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w}_2 : \overline{\sigma}_1$ 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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w}_1\overline{w}\overline{w}_2 : \overline{\sigma}_0\overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3$. Plug this and the assumption $\Pi; \overline{t}; \Psi \vdash H : \Psi$ into Ht-Inf to obtain $\Pi; \overline{t}; \Psi \vdash H' : \Psi'$, where $\Psi' = \Psi, \ell \mapsto \Psi(\ell)$. Using Weakening, we obtain $\Pi; \overline{t}; \Psi' \vdash H' : \Psi'$, as desired.

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

Lemma 33 (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 R and 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 34 (Preservation) *If*

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

then there exist Ψ'*, extending* Ψ*, such that*

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

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 $\Pi; \overline{t} \vdash \mathsf{R}[e_0]_{\mathsf{r}} \rightarrow \mathsf{R}[e_0']_{\mathsf{r}}$ (the D-StepR rule) which can only happen under condition $\Pi; \overline{t} \vdash e_0 \xrightarrow{\mathsf{r}} e_0'$. Then:

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

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

Lemma 35 (Progress)
Suppose $\Pi; \overline{t}; \Psi \vdash H : \Psi$ *and* Π *is a sensible platform.*

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

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 (L or 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 $\Pi; \overline{t} \vdash H; e_1 \xrightarrow{\mathsf{r}} H'; e_2$, 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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} s : \text{byte}^i$. 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 Γ.

- L-FetchR: We assume $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} e.f : \overline{\sigma}_2$ where inversion gives

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

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

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

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

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

  6 $\mathrm{size}(\Pi, \overline{\sigma}_1) = \mathrm{size}(\Pi, \overline{w}_1)$

  7 $\mathrm{size}(\Pi, \overline{\sigma}_2) = \mathrm{size}(\Pi, \overline{w}_2)$

  From (2,6) we have

  8 $\Pi.\mathit{offset}(\overline{t}, f) = \mathrm{size}(\Pi, \overline{w}_1)$

  (3,4,5,7) allow a step $\Pi; \overline{t} \vdash H; e \xrightarrow{r} H'; \overline{w}_2$ 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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{l}} e.f : \overline{\sigma}_2, [a, o + o']$ where inversion gives

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

  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. $e = *(\tau_1*)(\ell{+}i)$. (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 $e_2 = *(\tau*)(\ell{+}(i + \Pi.\mathit{offset}(\overline{t}, f)))$.

- L-DerefR: We assume $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} *(\tau*)(e) : \overline{\sigma}$. Inverting, we get

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

  2 $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}$

  3 $\Pi.\mathit{access}(\alpha, \mathrm{size}(\Pi, \overline{\sigma}))$

  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 $\mathrm{size}(\Pi, \overline{\sigma}_0) = i$

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

  Heap Canonical Forms on (4) gives

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

  9 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w}' : \overline{\sigma}_0\overline{\sigma}\overline{\sigma}_2$

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

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

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

Value-Type Size on (10,11) gives

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

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

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

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

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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{l}} *(\tau*)(e) : \overline{\sigma}_1, \alpha$ and by inversion we get $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} e : \mathrm{ptr}_{\alpha}(\overline{\sigma}_1\overline{\sigma}_2)$. 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 $\Pi.xtype$, $\Pi.ptrsize$, and $\Pi.align$ give sufficient conditions to step.

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

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

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

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

    4 $\Pi.xtype(\overline{t}, \tau) = \overline{\sigma}$

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

    Canonical Forms on (3) yields

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

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

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

    Heap Canonical Forms on (6) gives

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

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

    The Value Split Lemma on (10) gives

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

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

13 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \overline{w}_0 : \overline{\sigma}_0$

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

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

15 $\mathrm{size}(\Pi, \overline{w}_0) = \mathrm{size}(\Pi, \overline{\sigma}_0) = i$ (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 platform sensibility gives

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

(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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} (\tau*)\&e \to f : \mathrm{ptr}_{[a, o_1 + \Pi.offset(\overline{t}, f)]}(\overline{\sigma}_2)$ where by inversion we have $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} e : \mathrm{ptr}_{a, o_1}(\overline{\sigma}_1\overline{\sigma}_2)$. 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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} e_1; e_2 : \overline{\sigma}$ where inversion gives $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} e_1 : \overline{\sigma}'$. 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 $\Pi.\mathit{xtype}$, $\Pi.\mathit{ptrsize}$, and $\Pi.\mathit{align}$ give sufficient conditions to step.
- L-If: We assume $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \mathsf{if}\ e_1\ e_2\ e_3 : \overline{\sigma}$, where inversion gives $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} e_1 : \mathrm{byte}^i$. 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 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 36 (Type Soundness) *Let $(\to^*)$ be the reflexive transitive closure of the transition relation $(\to)$. Then, if*

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

then $H'; e'$ is not illegally stuck on Π 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 $\Pi; \overline{t} \vdash \cdot; e \to^{n-1} H''; e''$ and $H''; e''$ not illegally stuck. Preservation tells us that if $\Pi; \overline{t} \vdash H''; e'' \to H'; e'$, the type of e'' is preserved in the step to e'. It follows from Progress that $H'; e'$ is not illegally stuck.

A.3 Platform Dependencies

Theorem 37 (Constraint Satisfaction)

- *If* $\overline{t}; \Gamma \vdash_{\mathsf{r}} e : \tau; S$ *and* Π *is sensible and* $\Pi \models S$ *and* $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}$*, then* $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}$.
- *If* $\overline{t}; \Gamma \vdash_{\mathsf{l}} e : \tau; S$ *and* Π *is sensible and* $\Pi \models S$ *and* $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}$ *and* $\Pi.\mathit{align}(\overline{t}, \tau) = \alpha$*, then* $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{l}} e : \overline{\sigma}, \alpha'$ *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 25) is used implicitly.

- H-Short: We assume $\overline{t}; \Gamma \vdash_{\mathsf{r}} s : \text{short}; \top$. By clause 2 of the definition of a sensible platform, $\Pi.\mathit{xtype}(\overline{t}, \text{short}) = \text{byte}^i$. From this, $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} s : \text{byte}^i$ 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 Π is sensible (clause 4), we know $\Pi.\mathit{xtype}(\overline{t}, \tau*) = \text{ptr}_\alpha(\overline{\sigma})$ where $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}$ and $\Pi.\mathit{align}(\overline{t}, \tau) = \alpha$. We can plug the latter two facts into L-New to obtain $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} \mathsf{new}\ \tau : \text{ptr}_\alpha(\overline{\sigma})$.
- H-VarR: We assume $\overline{t}; \Gamma \vdash_{\mathsf{r}} x : \tau; \top$ and $\Gamma(x) = \tau$. Plug this along with $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}$ 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 $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}$ and $\Pi.\mathit{align}(\overline{t}, \tau) = \alpha$ 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 $\Pi \models S_1 \wedge S_2$, we know by model definition that $\Pi \models S_1$ and $\Pi \models S_2$. Induction on the left subderivation gives $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{l}} e_1 : \overline{\sigma}, \alpha$ where $\overline{\sigma} = \Pi.\mathit{xtype}(\overline{t}, \tau)$. Induction on the right subderivation gives $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e_2 : \overline{\sigma}$. 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 $\Pi \models S$), we get

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

  4 $\Pi.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}$

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

  5 $\Pi.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3$

  6 $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}_2$

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

  It follows from (3,5) that

  8 $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}_1\overline{\sigma}_2\overline{\sigma}_3$

  Plugging (1,6,7,8) into L-FetchR, we obtain $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e.f : \overline{\sigma}_2$, as desired.
- H-FetchL: We assume $\overline{t}; \Gamma \vdash_{\mathsf{l}} e.f : \tau; S$ with $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}$ and $\Pi.\mathit{align}(\overline{t}, \tau) = [a, o]$. Inversion gives

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

2 $\overline{t}; \Gamma \vdash_{\intercal} e : N; S$

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

3 $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\intercal} e : \overline{\sigma}', [a', o']$

4 $\Pi.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}'$

5 $\Pi.\mathit{align}(\overline{t}, N) = [a'', o'']$

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

By clause 3 of platform sensibility, we get

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

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

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

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

10 $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\intercal} e.f : \overline{\sigma}_2, [a', o' + \Pi.\mathit{offset}(\overline{t}, f)]$

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

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

  1 $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}$

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

  Because Π is sensible (clause 4), we know

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

  4 $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}'$

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

  From (1,3), we get

  6 $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e : \text{ptr}_{\alpha}(\overline{\sigma}')$

  Plugging (4,5,6) into L-DerefR, we get $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}'$, which, together with (4), forms our desired conclusion.
- H-DerefL: Similar to H-DerefR.
- H-Cast: We assume $\overline{t}; \Gamma \vdash_{\mathsf{r}} (\tau*)e : \tau*; S$, where $S = S' \wedge \text{subtype}(\overline{t}, \text{xtype}(\overline{t}, \tau'*), \text{xtype}(\overline{t}, \tau*))$, where by inversion we obtain

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

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

  2 $\Pi \models S'$

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

  3 $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}'$

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

Interpreting S under Π, we get

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

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

6 $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}$

From (6) and clause 4 of the definition of sensible platforms, we know $\overline{\sigma} = \mathrm{ptr}_{\alpha}(\overline{\sigma}'')$ where $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}''$. Plugging (5) into L-Cast, we get $\Pi; \overline{t}; \cdot; \Gamma \vdash_{\mathsf{r}} (\tau*)e : \mathrm{ptr}_{\alpha}(\overline{\sigma}'')$, as desired.

- H-Faddr: We assume $\overline{t}; \Gamma \vdash_{\mathsf{r}} (\tau*)\&e \rightarrow f : \tau*; S$, where

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

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

1 $\Pi.\mathit{align}(\overline{t}, N) = [a, o]$

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

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

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

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

By inversion,

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

By definition of the models relation,

7 $\Pi \models S'$

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

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

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

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

Theorem 38 (Layout Portability) *If* $\overline{t}; \cdot \vdash_{\mathsf{r}} e : \tau; S$ *and* Π *is sensible,* $\Pi \models S$*, and* $\Pi; \overline{t} \vdash \cdot; e \rightarrow^* H'; e'$*, then* $H'; e'$ *is not illegally stuck on* Π *and declarations* $\overline{t}$*.*

Proof: Corollary to Theorems 36 (Low-Level Type Soundness) and 37 (Constraint Satisfaction).

Definition 39 (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* $N\{\ldots\tau'\ f\ldots\} \in \bar{t}$ *then* $\tau = \tau'$.

Theorem 40 (Cast-Free Typing)

- *If* $\bar{t};\Gamma \vdash_{\mathsf{r}} e : \tau; S$, $\Pi.\mathit{xtype}(\bar{t},\tau) = \overline{\sigma}$, *and* e *is cast-free, then for any sensible platform* Π, $\Pi;\bar{t};\cdot;\Gamma \vdash_{\mathsf{r}} e : \overline{\sigma}$.
- *If* $\bar{t};\Gamma \vdash_{\mathsf{T}} e : \tau; S$, $\Pi.\mathit{xtype}(\bar{t},\tau) = \overline{\sigma}$, $\Pi.\mathit{align}(\bar{t},\tau) = \alpha$, *and* e *is cast-free, then for any sensible platform* Π, $\Pi;\bar{t};\cdot;\Gamma \vdash_{\mathsf{T}} e : \overline{\sigma},\alpha'$ *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 platform; 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 $\bar{t};\Gamma \vdash_{\mathsf{r}} (\tau*)\&e\rightarrow f : \tau*; S$ where inversion gives

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

2 $N\{\ldots\tau\ f\ldots\} \in \bar{t}$

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

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

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

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

5 $\Pi.\mathit{offset}(\bar{t},f) = \mathrm{size}(\Pi,\overline{\sigma}_1)$

6 $\Pi.\mathit{xtype}(\bar{t},\tau) = \overline{\sigma}_2$

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

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

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

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

Theorem 41 (Cast-Free Sufficiency) *If* $\bar{t};\cdot \vdash_{\mathsf{r}} e : \tau; S$, e *is cast-free,* Π *is sensible, and* $\Pi;\bar{t} \vdash \cdot; e \rightarrow^* H'; e'$, *then* $H'; e'$ *is not illegally stuck on* Π *and declarations* $\bar{t}$.

Proof: Corollary to Theorems 36 (Low-Level Type Soundness) and 40 (Cast-Free Typing).

A.4 Array Extension

Figure 20 shows how the syntax and semantics can be augmented with C-like arrays. First, we augment platforms (see Section A.1) with a new component ($\mathrm{val} : \overline{b} \rightarrow \mathbb{N}$) which interprets a sequence of bytes into a number in an platform-prescribed manner. We now show how the definitions, lemmas, and theorems in the previous sections can be extended for arrays.

Extended Definition 42 (Sensible Platforms)
We add two array-related sensibility clauses:

6. $\forall \tau, \overline{t}. \exists a, k, o.\ \Pi.align(\overline{t}, \tau) = [a, o]$ *and* $\Pi.xtype(\overline{t}, \tau) = \overline{\sigma}$ *and* $\text{size}(\Pi, \overline{\sigma}) = k \times a$.

7. $\forall \tau \exists \alpha, \overline{\sigma}.\ \Pi.xtype(\overline{t}, \tau{*}^{\omega}) = \text{ptr}^{\omega}_{\alpha}(\overline{\sigma})$ *and* $\Pi.xtype(\overline{t}, \tau) = \overline{\sigma}$ *and* $\Pi.align(\overline{t}, \tau) = \alpha$.

Extended Definition 43 (Legal Stuck State)
We extend the syntax for stuck expressions which can appear in right-holes.

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

Extended Lemma 44 (Uninit Type)
(No additions.)

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

Extended Lemma 45 (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 20).

Extended Lemma 46 (Value-Type Size)
(No additions.)

Proof:

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

Extended Lemma 47 (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 48 (Subtyping Type Form)

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

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

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

New High-Level Typing

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

New Subtyping

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

New Low-Level Typing

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

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

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

New Size Function Case

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

Figure 20: Array Syntax and Semantics

- 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 $\Pi; \overline{t} \vdash \overline{\sigma}_1\overline{\sigma}_3 \leq \overline{\sigma}_2\overline{\sigma}_4$. 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 $\Pi; \overline{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_3$ where inversion gives $\Pi; \overline{t} \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ and $\Pi; \overline{t} \vdash \overline{\sigma}_2 \leq \overline{\sigma}_3$. There are two possibilities for $\overline{\sigma}_3$:
  1. $\overline{\sigma}_3 = \mathrm{ptr}^{\omega}_{\alpha}(N)$ where $\Pi.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}$. 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 $\Pi.\mathit{xtype}(\overline{t}, N') = \overline{\sigma}^j$ and $\vdash \alpha'' \leq \alpha'$. By Alst-Trans, $\vdash \alpha'' \leq \alpha$, as required.
     - $\overline{\sigma}_2 = \mathrm{ptr}^{\omega}_{\alpha'}(N')$ where $\Pi.\mathit{xtype}(\overline{t}, N') = \overline{\sigma}$ 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 $\Pi.\mathit{xtype}(\overline{t}, N') = \overline{\sigma}^j$. 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 49 (Canonical Forms)

- *If* $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})$ *then either*
  - $\overline{w} = \mathsf{uninit}^i$ *where* $\Pi.\mathit{ptrsize} = i$.
  - $\overline{w} = \ell\text{+}0$ *and* $\exists j, \overline{\sigma}', \alpha'$ *such that* $\Psi(\ell) = \overline{\sigma}'^j, \alpha'$ *where* $\vdash \alpha' \leq \alpha$.
- *If* $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(N)$ *and* $\Pi.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}$*, then either*
  - $\overline{w} = \mathsf{uninit}^i$ *where* $\Pi.\mathit{ptrsize} = i$.
  - $\overline{w} = \ell\text{+}0$ *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 25).

- L-Value: There are two cases:
  - Assume $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma}_1)$. By inversion we get $\Pi; \overline{t}; \Psi \vdash_{\mathrm{w}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma}_1)$. 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 $i = \Pi.\mathit{ptrsize}$, and in the latter case, $\overline{w} = \ell\text{+}0$ where by inversion $\Psi(\ell) = \overline{\sigma}^j, \alpha'$ with $\alpha' = \alpha$.
  - Assume $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(N)$, where $\Pi.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}_1$. Vacuous; cannot be derived via L-Val.
- L-Sub: There are two cases:
  - Assume $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})$. By inversion we get $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \overline{\sigma}''$ and $\Pi; \overline{t} \vdash \overline{\sigma}'' \leq \mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})$. 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 $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha'}(\overline{\sigma}^i)$, we get that either $\overline{w} = \mathsf{uninit}^k$ or $\overline{w} = \ell\text{+}0$ 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 $\Pi.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}$ and $\vdash \alpha' \leq \alpha$. By induction and an application of Alst-Trans, like the above case.
  - Assume $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathrm{r}} \overline{w} : \mathrm{ptr}^{\omega}_{\alpha}(N)$, where $\Pi.\mathit{xtype}(\overline{t}, N) = \overline{\sigma}$. Similar to the above case.

Extended Lemma 50 (Subject Reduction)
(No additions.)

Proof:

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

  The assumed step rule is

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

  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 $\Pi;\overline{t};\Psi';\cdot\vdash_{\mathrm{r}}\ell{+}0:\mathrm{ptr}^{\omega}_{\alpha}(\overline{\sigma})$, 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 $\Pi;\overline{t};\Psi'\vdash H:\Psi$

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

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

  The assumed step rule is

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

  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 $\Pi;\overline{t};\Psi;\cdot\vdash_{\mathrm{r}}\overline{w}:\overline{\sigma}^{h}$

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

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

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

6 $j \times k < \text{size}(\Pi, \overline{\sigma}^h)$

It must be the case then that we can write $\overline{\sigma}^h$ as $\overline{\sigma}^k\overline{\sigma}^g$ (since $\text{size}(\Pi, \overline{\sigma}) = j$, 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 $\text{size}(\Pi, \overline{\sigma}^k) = k \times \text{size}(\Pi, \overline{\sigma}) = k \times j$. Plugging these two facts into Lw-Lbl and the result into L-Value, we get

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

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 $j = \text{size}(\Pi, \overline{\sigma}) = a \times k'$ 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 $\Pi; \overline{t} \vdash \text{ptr}_{[a', o'+j\times k]}(\overline{\sigma}^g) \leq \text{ptr}_{[a,o]}(\overline{\sigma})$. Plugging this along with (7) into L-Sub yields $\Pi; \overline{t}; \Psi; \cdot \vdash_{\mathsf{r}} \ell\texttt{+}(j \times k) : \text{ptr}_{[a,o]}(\overline{\sigma})$, as desired.

Extended Lemma 51 (Progress)
(No additions.)

Proof:

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

  We apply the IH to $\Pi; \overline{t}; C \vdash_{\mathsf{r}} e : \overline{\sigma}'$ and consider the case when e is a value. Clause 2 of platform sensibility gives that $\overline{\sigma}' = \text{byte}^i$. 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 $\Pi.\text{val}(\overline{b}) < 0$, then $\mathsf{new}\ \tau[e]$ is again legally stuck. Otherwise it can take a step via rule D-NewArr.

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

  We apply the IH to $\Pi; \overline{t}; C \vdash_{\mathsf{r}} e_1 : \text{ptr}^{\omega}_{[a,o]}(\overline{\sigma})$ and $\Pi; \overline{t}; C \vdash_{\mathsf{r}} e_2 : \overline{\sigma}'$ 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 platform sensibility yields $\overline{\sigma}' = \text{byte}^{k_2}$. 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 $\Pi.\text{val}(\overline{b}) < 0$, $\&((\tau{*}^{\omega})(e_1))[e_2]$ is again legally stuck. Suppose then that $\Pi.\text{val}(\overline{b}) = k$ for $k \geq 0$.

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

Extended Lemma 52 (Constraint Satisfaction)
(No additions.)

Proof:

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

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

Syntax:

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

Dynamic semantics:

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

High-level typing:

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

Low-level typing:

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

Figure 21: Additions for Platform Selection

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

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

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

  2 $\Pi.\mathit{xtype}(\overline{t}, \tau) = \overline{\sigma}$

  3 $\Pi.\mathit{align}(\overline{t}, \tau) = [a, o]$

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

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

  From clause 6 of platform sensibility we learn

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

  From clause 4 of platform sensibility we have

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

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

A.5 Platform Selection Extension

The additions for platform selection are given in Figure 21. The metatheory extends as follows:

Extended Lemma 53 (Progress)
(No additions to the statement.)

Proof:

- L-Pcase: We assume $\Pi; \bar{t}; \Psi; \Gamma \vdash_{\mathsf{r}} \mathsf{pcase}\ S_1\ \texttt{=>}\ e_1 \ldots S_n\ \texttt{=>}\ e_n : \overline{\sigma}$. By inversion, we obtain $\Pi \models S_i$ for some i. Using this, we can step to e_i via D-Pcase.

Extended Lemma 54 (Subject Reduction)
(No additions to the statement.)

Proof:

- L-Pcase: We assume $\Pi; \bar{t}; \Psi; \Gamma \vdash_{\mathsf{r}} \mathsf{pcase}\ S_1\ \texttt{=>}\ e_1 \ldots S_n\ \texttt{=>}\ e_n : \overline{\sigma}$ and the step was derived via D-Pcase. By inversion, we get $\Pi; \bar{t}; \Psi; \Gamma \vdash_{\mathsf{r}} e_i : \overline{\sigma}$ for all i such that $\Pi \models S_i$. By inversion on D-Pcase, we get $\Pi \models S_k$ for a k such that the pcase expression stepped to e_k. Therefore, $\Pi; \bar{t}; \Psi; \Gamma \vdash_{\mathsf{r}} e_k : \overline{\sigma}$, as desired.

Extended Lemma 55 (Constraint Satisfaction)
(No additions to the statement.)

Proof:

- H-Pcase: We assume $\bar{t}; \Gamma \vdash_{\mathsf{r}} \mathsf{pcase}\ S_1\ \texttt{=>}\ e_1 \ldots S_n\ \texttt{=>}\ e_n : \tau; S$ and $\Pi \models S$ and we want to show $\Pi; \bar{t}; \cdot; \Gamma \vdash_{\mathsf{r}} \mathsf{pcase}\ S_1\ \texttt{=>}\ e_1 \ldots S_n\ \texttt{=>}\ e_n : \overline{\sigma}$ where $\Pi.\mathit{xtype}(\tau) = \overline{\sigma}$.

  We know $S = (\bigwedge_{i=1}^{n} S_i \Rightarrow S_i') \wedge \bigvee_{i=1}^{n} S_i$ and by inversion on the assumed typing we get $\bar{t}; \Gamma \vdash_{\mathsf{r}} e_i : \tau; S_i'$ for all $i \in 1..n$. By definition of $(\models)$, we have $\Pi \models \bigwedge_{i=1}^{n} S_i \Rightarrow S_i'$ and $\Pi \models \bigvee_{i=1}^{n} S_i$. The former means that $\forall 1 \leq i \leq n$, if $\Pi \models S_i$ then $\Pi \models S_i'$. The latter means that $\exists 1 \leq i \leq n$ s.t. $\Pi \models S_i$. Combining the results, we get $\exists 1 \leq i \leq n$ s.t. $\Pi \models S_i'$.

  We apply the induction hypothesis to all subderivations $\bar{t}; \Gamma \vdash_{\mathsf{r}} e_i : \tau; S_i'$ where $\Pi \models S_i'$ to obtain $\Pi; \bar{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e_i : \overline{\sigma}$. That is, $\forall 1 \leq i \leq n$, if $\Pi \models S_i'$ then $\Pi; \bar{t}; \cdot; \Gamma \vdash_{\mathsf{r}} e_i : \overline{\sigma}$.

  We plug the conclusions of the previous two paragraphs into L-Pcase to obtain the desired result.