Alias Types *

Frederick Smith David Walker Greg Morrisett

October 28, 1999

Abstract

Linear type systems allow destructive operations such as object deallocation and imperative updates of functional data structures. These operations and others, such as the ability to reuse memory at different types, are essential in low-level typed languages. However, traditional linear type systems are too restrictive for use in low-level code where it is necessary to exploit pointer aliasing. We present a new typed language that allows functions to specify the shape of the store that they expect and to track the flow of pointers through a computation. Our type system is expressive enough to represent pointer aliasing and yet safely permit destructive operations.

1 Introduction

Linear type systems [24, 23] give programmers explicit control over memory resources. The critical invariant of a linear type system is that every linear value is used exactly once. After its single use, a linear value is dead and the system can immediately reclaim its space or reuse it to store another value. Although this single-use invariant enables compile-time garbage collection and imperative updates to functional data structures, it also limits the use of linear values. For example, x is used twice in the following expression: $\texttt{let } x = \langle 1, 2\rangle \texttt{ in let } y = fst(x) \texttt{ in let } z = snd(x) \texttt{ in } y + z$. Therefore, x cannot be given a linear type, and consequently cannot be deallocated early.

Several authors have extended pure linear type systems to allow greater flexibility. Wadler [24], for example, introduced a new let-form $\texttt{let!}\,(x)\ y = e_1 \texttt{ in } e_2$ that permits the variable x to be used as a non-linear value in e_1 (*i.e.* it can be used many times, albeit in a restricted fashion) and then later used as a linear value in e_2. Similarly, Kobayashi [9] replaced linear values with *pseudo-linear* values that can be used locally a number of times before being deallocated. Concurrent Clean is a production-quality lazy function language that uses a related notion called *uniqueness types* [3] to enable static garbage collection.

Because these solutions have focused on high-level user programming languages, they have emphasized simple typing rules that programmers can understand and/or typing rules that admit effective

*This material is based on work supported in part by the AFOSR grant F49620-97-1-0013 and the National Science Foundation under Grant No. EIA 97-03470. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

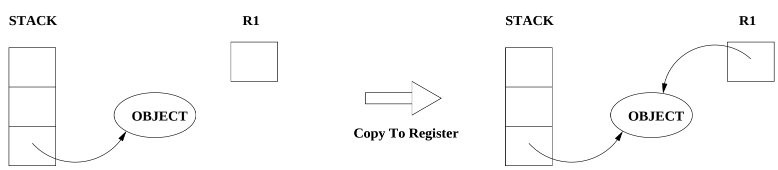
type inference techniques. These issues are not as much of a concern for low-level typed languages designed as compiler intermediate languages [20, 18] or as secure mobile code platforms, such as the Java Virtual Machine [10], Proof-Carrying Code [13] or Typed Assembly Language (TAL) [12]. These languages are designed for machine, not human, consumption. However, the implementation of strongly typed low-level languages requires a variety of new type-theoretic mechanisms. In particular, our experience with TAL has revealed at least two new challenges:

1. Low-level languages require even more destructive operations than their high-level counterparts. In high-level languages, every location is stamped with a single type for the lifetime of the program. Failing to maintain this invariant has resulted in unsound type systems or misfeatures (witness the interaction between parametric polymorphism and references in ML [21, 26], and covariant arrays in Java [14]). In low-level languages that aim to expose the resource constraints of the underlying machine, this invariant is untenable. For instance, because machines contain a limited number of registers, each register cannot be stamped with a single type. Also, when two stack-allocated objects have disjoint lifetimes, compilers naturally reuse the stack space, even when the two objects have different types. Finally, in a low-level language exposing initialization, even the simplest objects change type. For example, a pair x of type $\langle int, int\rangle$ may be created through the following sequence of instructions:

$$\begin{array}{ll}
\texttt{malloc}\, x, 2\ ; & (*\ x \text{ has type } \langle \texttt{junk}, \texttt{junk}\rangle\ *) \\
x[1]{:=}1\ ; & (*\ x \text{ has type } \langle int, \texttt{junk}\rangle\ *) \\
x[2]{:=}2\ ; & (*\ x \text{ has type } \langle int, int\rangle\ *) \\
\vdots &
\end{array}$$

Type systems for low-level languages should support values whose types change.

2. Efficient and natural low-level code often copies values. In fact, the job of the register allocator is to copy values between the stack and registers intelligently. However, when pointer values are copied, the objects they point to are shared:



This sharing can be eliminated by invalidating one of the copies (presumably the pointer on the stack). However, should the pointer be live, it might not be possible to keep it in a register for the rest of its lifetime. For example, imagine that x is the current stack frame and assume the top of the stack contains a pointer: $\texttt{let}\ r_1 = x[1]\ \texttt{in}\ \cdots$ spill register $r_1\ \cdots \texttt{let}\ r_1 = x[1]$. In an untyped calculus, the spilling operation need not copy the pointer back onto the stack because the register allocator can easily remember that the pointer is already stored on the stack. Unfortunately, if the type system forces us to invalidate one of our copies, we will have to perform this extra copy.

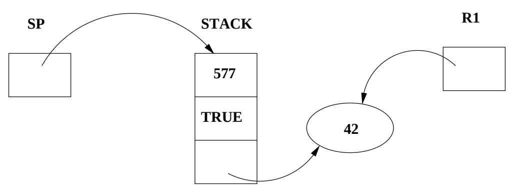
Pointer aliasing and data sharing also occur naturally in other data structures introduced by a compiler. For example, compilers often use a top-of-stack pointer and a frame pointer, both of which point to the same data structure. Compiling a language like Pascal using displays [1] generalizes this problem to having an arbitrary (but statically known) number of pointers into the same data structure. In each of these examples, a flexible type system will allow aliasing but ensure that no inconsistencies arise.

Type systems for low-level languages should represent sharing.

1.1 Overview

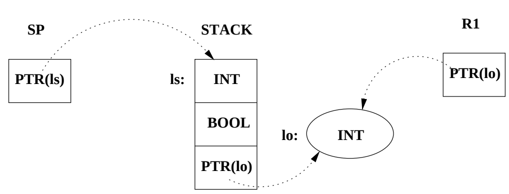
We have devised a new type system that is capable of tracking sharing in data structures and that admits operations for memory reuse at different types, object initialization, and deallocation. This paper formalizes the type system and provides a theoretical foundation for safely integrating operations that depend upon pointer aliasing with type systems that include polymorphism and higher-order functions.

The main new feature of our language is a collection of *aliasing constraints*. Aliasing constraints describe the shape of the store and every function uses them to specify the store that it expects. If the current store does not conform to the constraints specified, then the type system ensures that the function cannot be called. To illustrate how our constraints abstract a concrete store, we will consider the following example:



Here, *sp* is a pointer to a stack frame, which has been allocated on the heap (as might be done in the SML/NJ compiler [2], for instance). This frame contains a pointer to a second object, which is also pointed to by register r_1.

In our program model, every heap-allocated object occupies a particular memory location. For example, the stack frame might occupy location ℓ_s and the second object might occupy location ℓ_o. In order to track the flow of pointers to these locations accurately, we reflect locations into the type system: A pointer to a location ℓ is given the singleton type $ptr(\ell)$. Each singleton type contains exactly one value (the pointer in question). This property allows the type system to reason about pointers in a very fine-grained way. In fact, it allows us to represent the graph structure of our example store precisely:



We represent this picture in our formal syntax by declaring the program variable *sp* to have type $ptr(\ell_s)$ and r_1 to have type $ptr(\ell_o)$. The store itself is described by the constraints $\{\ell_s \mapsto \langle int, bool, ptr(\ell_o)\rangle\} \oplus \{\ell_o \mapsto \langle int\rangle\}$, where the type $\langle \tau_1, \ldots, \tau_n\rangle$ denotes a memory block containing values with types τ_1 through τ_n.

Constraints of the form $\{\ell \mapsto \tau\}$ are a reasonable starting point for an abstraction of the store. However, they are actually *too precise* to be useful for general-purpose programs. Consider, for example, the simple function *deref*, which retrieves an integer from a reference cell. There are two immediate problems if we demand that code call *deref* when the store has a shape described by $\{\ell \mapsto \langle int \rangle\}$. First, *deref* can only be used to derefence the location ℓ, and not, for example, ℓ' or ℓ''. This problem is easily solved by adding *location polymorphism*. The exact name of a location is usually unimportant; we need only establish a dependence between pointer type and constraint. Hence we could specify that *deref* requires a store $\{\rho \mapsto \langle int \rangle\}$ where ρ is a location variable instead of some specific location ℓ. Second, the constraint $\{\ell \mapsto \langle int \rangle\}$ specifies a store with exactly one location ℓ although we may want to dereference a single integer reference amongst a sea of other heap-allocated objects. Since *deref* does not use or modify any of these other references, we should be able to abstract away the size and shape of the rest of the store. We accomplish this task using *store polymorphism*. An appropriate constraint for the function *deref* is $\epsilon \oplus \{\rho \mapsto \langle int \rangle\}$ where ϵ is a constraint variable that may instantiated with any other constraint..

The third main feature of our constraint language is the capability to distinguish between linear constraints $\{\rho \mapsto \tau\}$ and non-linear constraints $\{\rho \mapsto \tau\}^{\omega}$. Linear constraints come with the additional guarantee that the location on the left-hand side of the constraint (ρ) is not aliased by any other location (ρ'). This invariant is maintained despite the presence of location polymorphism and store polymorphism. Intuitively, because ρ is unaliased, we can safely deallocate its memory or change the types of the values stored there. The key property that makes our system more expressive than traditional linear systems is that although the aliasing constraints may be linear, the pointer values that flow through a computation are not. Hence, there is no *direct* restriction on the copying and reuse of pointers.

The following example illustrates how the type system uses aliasing constraints and singleton types to track the evolution of the store across a series of instructions that allocate, initialize, and then deallocate storage. In this example, the instruction `malloc` x, ρ, n allocates storage which is bound to the variable ρ (this instruction binds both ρ and x). We trust that `malloc` has been implemented so that it always returns a fresh location ℓ which is used to instantiate ρ. The `free` instruction deallocates storage. Deallocated storage has type *junk* and the type system prevents any future use of that space.

	Instructions	Constraints (Initially the constraints ϵ)	
1.	`malloc` $sp, \rho_1, 2$;	$\epsilon \oplus \{\rho_1 \mapsto \langle junk, junk \rangle\}$	$sp : ptr(\rho_1)$
2.	$sp[1]{:=}1$;	$\epsilon \oplus \{\rho_1 \mapsto \langle int, junk \rangle\}$	
3.	`malloc` $r_1, \rho_2, 1$;	$\epsilon \oplus \{\rho_1 \mapsto \langle int, junk \rangle, \rho_2 \mapsto \langle junk \rangle\}$	$r_1 : ptr(\rho_2)$
4.	$sp[2]{:=}r_1$;	$\epsilon \oplus \{\rho_1 \mapsto \langle int, ptr(\rho_2) \rangle, \rho_2 \mapsto \langle junk \rangle\}$	
5.	$r_1[1]{:=}2$;	$\epsilon \oplus \{\rho_1 \mapsto \langle int, ptr(\rho_2) \rangle, \rho_2 \mapsto \langle int \rangle\}$	
6.	`free` r_1;	$\epsilon \oplus \{\rho_1 \mapsto \langle int, ptr(\rho_2) \rangle, \rho_2 \mapsto junk\}$	
7.	`free` sp;	$\epsilon \oplus \{\rho_1 \mapsto junk, \rho_2 \mapsto junk\}$	

Again, we can intuitively think of sp as the stack pointer and r_1 as a register that holds an alias of an object on the stack. Notice that on line 5, the initialization of r_1 updates the type of the memory at location ρ_2. This has the effect of simultaneously updating the type of r_1 and of $sp[1]$. Both of these paths are similarly affected when r_1 is freed in the next instruction. Despite the presence of the dangling pointer at $sp[1]$, the type system will not allow that pointer to be derefenced.

By using singleton types to accurately track pointers, and linear constraints to model the shape of the store, our type system can represent aliasing and simultaneously ensure safety in the presence of destructive operations

1.2 Summary

We have extended the Typed Assembly Language (TAL) implementation with the features described in this paper.[1] It was quite straightforward to augment the existing F^ω-based type system because many of the basic mechanisms, including polymorphism and singleton types, were already present in the type constructor language. Popcorn, an optimizing compiler for a safe C-like language, generates code for the new TAL type system and uses the alias tracking features of our type system.

Rather than formalizing the type system in the context of TAL, we present our ideas in terms of a more familiar lambda calculus. Section 2 describes the core language including the linear aliasing constraints. In Section 3, we extend this language with non-linear constraints. Even though non-linear constraints do not admit destructive operations, the aliasing information they contain is still useful. In particular, when code performs a dynamic type test, it is possible to refine the types of several aliases simultaneously. We explore this application in Section 3 as well. In Section 4 we show how to compile a simple imperative language with displays into the language of locations. The key feature of our translation is that the stack is explicitly allocated and deallocated. Finally, in Section 5 and Section 6, we discuss future and related work.

2 The Language of Locations

This section describes our new type-safe "language of locations." The syntax for the language appears in Figure 1.

2.1 Values, Instructions, and Programs

A program is a pair of a store (S) and a list of instructions (ι). The store maps locations (ℓ) to values (v). Normally, the values held in the store are memory blocks ($\langle \tau_1, \ldots, \tau_n \rangle$), but after the memory at a location has been deallocated, that location will point to the unusable value `junk`. Other values include integer constants (i), variables (x or f), and, of course, pointers ($\texttt{ptr}(\ell)$).

The main instructions manipulate memory blocks. As discussed in the introduction, the instruction `malloc` x, ρ, n allocates a memory block of size n at a new location ℓ, binds the variable x to the pointer $\texttt{ptr}(\ell)$ and binds the location variable ρ to the concrete location ℓ. After allocation, the components of the new memory block are uninitialized (filled with `junk`). There are two memory access instructions. The instruction $x{=}v[i]$ substitutes the ith component of the memory block pointed to by v for x. The variable x is considered bound by this expression and its scope extends through the rest of the instruction sequence. The instruction $v[i]{:=}v'$ stores v' in the ith component of the block pointed to by v. The final memory management primitive, `free` v, deallocates the

[1] See `http://www.cs.cornell.edu/talc` for the latest software release.

$$\rho \in \text{LabelVar} \qquad \epsilon \in \text{ConstraintVar} \qquad \ell \in \text{Locations} \qquad x, f \in \text{ValueVar}$$

$$\begin{array}{llcl}
\textit{locations} & \eta & ::= & \ell \mid \rho \\
\textit{constraints} & C & ::= & \emptyset \mid \epsilon \mid \{\eta \mapsto \tau\} \mid C_1 \oplus C_2 \\
\textit{types} & \tau & ::= & \mathit{int} \mid \mathit{junk} \mid \mathit{ptr}(\eta) \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \forall[\Delta; C].(\tau_1, \ldots, \tau_n) \rightarrow 0 \\
\textit{value ctxts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\textit{constructor ctxts} & \Delta & ::= & \cdot \mid \Delta, \rho \mid \Delta, \epsilon \\
\textit{program ctxts} & \Phi & ::= & \Delta; C; \Gamma \\
\textit{values} & v & ::= & x \mid i \mid \texttt{junk} \mid \texttt{ptr}(\ell) \mid \langle v_1, \ldots, v_n \rangle \mid \texttt{fix}\, f[\Phi].\iota \mid \\
& & & v[\eta] \mid v[C] \\
\textit{instructions} & \iota & ::= & \texttt{malloc}\, x, \rho, n; \iota \mid x{=}v[i]; \iota \mid v[i]{:=}v'; \iota \mid \texttt{free}\; v; \iota \mid \\
& & & v(v_1, \ldots, v_n) \mid \texttt{halt} \\
\textit{stores} & S & ::= & \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \\
\textit{programs} & P & ::= & (S, \iota)
\end{array}$$

Figure 1: Language of Locations: Syntax

storage pointed to by v. If v is the pointer $\texttt{ptr}(\ell)$ then deallocation is modeled by updating the store (S) so that the location ℓ maps to `junk`.

The following example shows the evaluation of a program to allocate, initialize, and finally deallocate a pair of integers beginning with an empty store:

Store	Instructions	
$\{\,\}$	$\texttt{malloc}\, x, \rho, n$	`(* allocate new location` ℓ`,   *)`
		`(* substitute ptr(`ℓ`),`ℓ` for `x,ρ` *)`
$\{\ell \mapsto \langle \texttt{junk}, \texttt{junk} \rangle\}$	$\texttt{ptr}(\ell)[1]{:=}3$	`(* initialize field 1   *)`
$\{\ell \mapsto \langle 3, \texttt{junk} \rangle\}$	$\texttt{ptr}(\ell)[2]{:=}5$	`(* initialize field 2   *)`
$\{\ell \mapsto \langle 3, 5 \rangle\}$	$\texttt{free}\,\texttt{ptr}(\ell)$	`(* free storage   *)`
$\{\ell \mapsto \texttt{junk}\}$		

A sequence of instructions (ι) ends in either a `halt` instruction, which stops computation immediately, or a function application ($v(v_1, \ldots, v_n)$). In order to simplify the language and its typing constructs, our functions never return. However, a higher-level language that contains call and return statements can be compiled into our language of locations by performing a *continuation-passing style* (CPS) transformation [15, 16]. It is possible to define a direct-style language, but doing so is less elegant. Such a choice would force us to adopt an awkward syntax that allows functions to return portions of the store. In a CPS style, all control-flow transfers are handled symmetrically by calling a continuation.

Functions are defined using the form $\texttt{fix}\, f[\Delta; C; \Gamma].\iota$. These functions are recursive (f may appear in ι). The context $(\Delta; C; \Gamma)$ specifies a pre-condition that must be satisfied before the function can be invoked. The type context Δ binds the set of type variables that can occur free in the term; C is a collection of aliasing constraints that statically approximates a portion of the store; and Γ assigns types to free variables in ι.

To call a polymorphic function, code must first instantiate the type variables in Δ using the value

$$
\begin{array}{lll}
(S, \mathtt{malloc}\, x, \rho, n; \iota) & \longmapsto & (S\{\ell \mapsto \langle \mathtt{junk}_1, \ldots, \mathtt{junk}_n \rangle\}, \iota[\ell/\rho][\mathtt{ptr}(\ell)/x]) \\
\quad \text{where } \ell \notin S & & \\
(S\{\ell \mapsto v\}, \mathtt{free}\, \mathtt{ptr}(\ell); \iota) & \longmapsto & (S\{\ell \mapsto \mathtt{junk}\}, \iota) \\
\quad \text{if } v = \langle v_1, \ldots, v_n \rangle & & \\
(S\{\ell \mapsto v\}, \mathtt{ptr}(\ell)[i] {:=} v'; \iota) & \longmapsto & (S\{\ell \mapsto \langle v_1, \ldots, v_{i-1}, v', v_{i+1}, \ldots, v_n \rangle\}, \iota) \\
\quad \text{if } v = \langle v_1, \ldots, v_n \rangle \text{ and } 1 \leq i \leq n & & \\
(S\{\ell \mapsto v\}, x {=} \mathtt{ptr}(\ell)[i]; \iota) & \longmapsto & (S\{\ell \mapsto v\}, \iota[v_i/x]) \\
\quad \text{if } v = \langle v_1, \ldots, v_n \rangle \text{ and } 1 \leq i \leq n & & \\
(S, v(v_1, \ldots, v_n)) & \longmapsto & (S, \iota[c_1, \ldots, c_m/\beta_1, \ldots, \beta_m][v', v_1, \ldots, v_n/f, x_1, \ldots, x_n]) \\
\quad \text{if } v = v'[c_1, \ldots, c_m] & & \\
\quad \text{and } v' = \mathtt{fix}\, f[\Delta; C; x_1{:}\tau_1, \ldots, x_n{:}\tau_n].\iota & & \\
\quad \text{and } Dom(\Delta) = \beta_1, \ldots, \beta_m \qquad \text{(where } \beta \text{ ranges over } \rho \text{ and } \epsilon) & &
\end{array}
$$

Figure 2: Language of Locations: Operational Semantics

form: $v[\eta]$ or $v[C]$. These forms are treated as values because type application has no computational effect (types and constraints are only used for compile-time checking; they can be erased before executing a program). Figure 2 formally defines the operational semantics. Here and elsewhere, the notation $X[c_1, \ldots, c_n/x_1, \ldots, x_n]$ denotes capture-avoiding substitution of $c_1, \ldots, c_n$ for variables $x_1, \ldots, x_n$ in X.

2.2 Type Constructors

There are three kinds of type constructors: locations[2] (η), types (τ), and aliasing constraints (C).

The simplest types are the base types, which we have chosen to be integers (*int*). A pointer to a location η is given the singleton type $ptr(\eta)$. The only value in the type $ptr(\eta)$ is the pointer $\mathtt{ptr}(\eta)$, so if v_1 and v_2 both have type $ptr(\eta)$, then they must be aliases. Memory blocks have types ($\langle \tau_1, \ldots, \tau_n \rangle$) that describe their contents.

A collection of constraints, C, establishes the connection between pointers of type $ptr(\eta)$ and the contents of the memory blocks they point to. The main form of constraint, written $\{\eta \mapsto \tau\}$, models a store with a single location η containing a value of type τ. Collections of constraints are constructed from more primitive constraints using the join operator ($\oplus$). The empty constraint is denoted by $\emptyset$. We often abbreviate $\{\eta \mapsto \tau\} \oplus \{\eta' \mapsto \tau'\}$ with $\{\eta \mapsto \tau, \eta' \mapsto \tau'\}$.

2.3 Static Semantics

Store Typing The central invariant maintained by the type system is that the current constraints C are a faithful description of the current store S. We write this *store-typing invariant* as the judgment $\vdash S : C$. Intuitively, whenever a location ℓ contains a value v of type τ, the constraints

[2] We use the meta-variable ℓ to denote concrete locations, ρ to denote location *variables*, and η to denote either.

should specify that location ℓ maps to τ (or an equivalent type τ'). Formally:

$$\frac{\cdot;\cdot \vdash v_1 : \tau_1 \qquad \cdots \qquad \cdot;\cdot \vdash v_n : \tau_n}{\vdash \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} : \{\ell_1 \mapsto \tau_1, \ldots, \ell_n \mapsto \tau_n\}}$$

where for $1 \leq i \leq n$, the locations ℓ_i are all distinct. And,

$$\frac{\vdash S : C' \qquad \cdot \vdash C' = C}{\vdash S : C}$$

Instruction Typing Instructions are type checked in a context $\Delta; C; \Gamma$. The judgment $\Delta; C; \Gamma \vdash \iota$ states that the instruction sequence is well-formed. A related judgment, $\Delta; \Gamma \vdash v : \tau$, ensures that the value v is well-formed and has type τ.

Our presentation of the typing rules for instructions focuses on how each rule maintains the store-typing invariant. With this invariant in mind, consider the rule for projection:

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \mathit{ptr}(\eta) \\ \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} \qquad \Delta; C; \Gamma, x{:}\tau_i \vdash \iota\end{array}}{\Delta; C; \Gamma \vdash x{=}v[i]; \iota} \begin{pmatrix} x \notin \Gamma \\ 1 \leq i \leq n \end{pmatrix}$$

The first pre-condition ensures that v is a pointer. The second uses C to determine the contents of the location pointed to by v. More precisely, it requires that C equal a store description $C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}$. The store is unchanged by the operation so the final pre-condition requires that the rest of the instructions be well-formed under the same collection of constraints C.

Next, examine the rule for the assignment operation:

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \mathit{ptr}(\eta) \qquad \Delta; \Gamma \vdash v' : \tau' \\ \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} \qquad \Delta; C' \oplus \{\eta \mapsto \tau_{\text{after}}\}; \Gamma \vdash \iota\end{array}}{\Delta; C; \Gamma \vdash v[i]{:=}v'; \iota} \; (1 \leq i \leq n)$$

$$\text{where } \tau_{\text{after}} \text{ is } \langle \tau_1, \ldots, \tau_{i-1}, \tau', \tau_{i+1}, \ldots, \tau_n \rangle$$

Once again, the value v must be a pointer to some location η. The type of the contents of η are given in C and must be a block with type $\langle \tau_1, \ldots, \tau_n \rangle$. This time the store has changed, and the remaining instructions are checked under the appropriately modified constraint $C' \oplus \{\eta \mapsto \tau_{\text{after}}\}$.

How can the type system ensure that the new constraints $C' \oplus \{\eta \mapsto \tau_{\text{after}}\}$ correctly describe the store? If v' has type τ' and the contents of the location η have type $\langle \tau_1, \ldots, \tau_n \rangle$, then $\{\eta \mapsto \tau_{\text{after}}\}$ describes the contents of the location η after the update accurately. However, we must avoid a situation in which C' continues to contain an outdated type for the contents of the location η. This task may appear trivial: Search C' for all occurrences of a constraint $\{\eta \mapsto \tau\}$ and update all of the mappings appropriately. Unfortunately, in the presence of location polymorphism, this approach will fail. Suppose a value is stored in location ρ_1 and the current constraints are $\{\rho_1 \mapsto \tau, \rho_2 \mapsto \tau\}$. We cannot determine whether or not ρ_1 and ρ_2 are aliases and therefore whether the final constraint set should be $\{\rho_1 \mapsto \tau', \rho_2 \mapsto \tau'\}$ or $\{\rho_1 \mapsto \tau', \rho_2 \mapsto \tau\}$.

Our solution uses a technique from the literature on linear type systems. Linear type systems prevent duplication of assumptions by disallowing uses of the contraction rule. We use an analogous restriction in the definition of constraint equality: The join operator $\oplus$ is associative, and commutative, but *not* idempotent. By ensuring that linear constraints cannot be duplicated, we can prove that ρ_1 and ρ_2 from the example above cannot be aliases. The other equality rules are unsurprising. The empty constraint collection is the identity for $\oplus$ and equality on types τ is syntactic up to alpha-conversion of bound variables and modulo equality on constraints. Therefore:

$$\{\rho_1 \mapsto \langle \mathit{int} \rangle\} \oplus \{\rho_2 \mapsto \langle \mathit{bool} \rangle\} = \{\rho_2 \mapsto \langle \mathit{bool} \rangle\} \oplus \{\rho_1 \mapsto \langle \mathit{int} \rangle\}$$

but,

$$\{\rho_1 \mapsto \langle \mathit{int} \rangle\} \oplus \{\rho_2 \mapsto \langle \mathit{bool} \rangle\} \neq \{\rho_1 \mapsto \langle \mathit{int} \rangle\} \oplus \{\rho_1 \mapsto \langle \mathit{int} \rangle\} \oplus \{\rho_2 \mapsto \langle \mathit{bool} \rangle\}$$

Given these equality rules, we can prove that after an update of the store with a value with a new type, the store typing invariant is preserved:

Lemma 1 ((Store Update)) *If* $\vdash S\{\ell \mapsto v\} : C \oplus \{\ell \mapsto \tau\}$ *and* $\cdot;\cdot \vdash v' : \tau'$ *then* $\vdash S\{\ell \mapsto v'\} : C \oplus \{\ell \mapsto \tau'\}$.

where $S\{\ell \mapsto v\}$ denotes the store S extended with the mapping $\ell \mapsto v$ (provided ℓ does not already appear on the left-hand side of any elements in S).

Function Typing The rule for function application $v(v_1, \ldots, v_n)$ is the rule one would expect. In general, v will be a value of the form $v'[c_1] \cdots [c_n]$ where v' is a function polymorphic in locations and constraints and the type constructors c_1 through c_n instantiate its polymorphic variables. After substituting c_1 through c_n for the polymorphic variables, the current constraints must equal the constraints expected by the function v. This check guarantees that the no-duplication property is preserved across function calls. To see why, consider the polymorphic function *foo* where the type context Δ is $(\rho_1, \rho_2, \epsilon)$ and the constraints C are $\epsilon \oplus \{\rho_1 \mapsto \langle \mathit{int} \rangle, \rho_2 \mapsto \langle \mathit{int} \rangle\}$:

$\texttt{fix}\, \mathit{foo}[\Delta; C; x{:}\mathit{ptr}(\rho_1), y{:}\mathit{ptr}(\rho_2), \mathit{cont}{:}\forall[\cdot;\epsilon].(\mathit{int}) \to 0].$
    $\texttt{free}\; x;$ `(* constraints` $= \epsilon \oplus \{\rho_2 \mapsto \langle \mathit{int} \rangle\}$ `*)`
    $z{=}y[0];$ `(* ok because` $y : \mathit{ptr}(\rho_2)$ `and` $\{\rho_2 \mapsto \langle \mathit{int} \rangle\}$ `*)`
    $\texttt{free}\; y;$ `(* constraints` $= \epsilon$ `*)`
    $\mathit{cont}(z)$ `(* return/continue *)`

This function deallocates its two arguments, x and y, before calling its continuation with the contents of y. It is easy to check that this function type-checks, but should it? If *foo* is called in a state where ρ_1 and ρ_2 are aliases, a run-time error will result when the second instruction is executed because the location pointed to by y will already have been deallocated. Fortunately, our type system guarantees that *foo* can never be called from such a state.

Suppose that the store currently contains a single integer reference: $\{\ell \mapsto \langle 3 \rangle\}$. This store can be described by the constraints $\{\ell \mapsto \langle \mathit{int} \rangle\}$. If the programmer attempts to instantiate both ρ_1 and ρ_2 with the same label ℓ, the function call $\mathit{foo}[\ell, \ell, \emptyset](\texttt{ptr}(\ell))$ will fail to type check because the constraints $\{\ell \mapsto \langle \mathit{int} \rangle\}$ do not equal the pre-condition $\emptyset \oplus \{\ell \mapsto \langle \mathit{int} \rangle, \ell \mapsto \langle \mathit{int} \rangle\}$.

Figure 3 contains the typing rules for values and instructions.

$\boxed{\Delta; \Gamma \vdash v : \tau}$

$$\frac{}{\Delta; \Gamma \vdash i : \mathit{int}} \qquad \frac{}{\Delta; \Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Delta; \Gamma \vdash \mathtt{junk} : \mathit{junk}}$$

$$\frac{\Delta \vdash \eta}{\Delta; \Gamma \vdash \mathtt{ptr}(\eta) : \mathit{ptr}(\eta)} \qquad \frac{\Delta; \Gamma \vdash v_1 : \tau_1 \qquad \cdots \qquad \Delta; \Gamma \vdash v_n : \tau_n}{\Delta; \Gamma \vdash \langle v_1, \ldots, v_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle}$$

$$\frac{\begin{array}{c}\Delta \vdash \forall[\Delta'; C].(\tau_1, \ldots, \tau_n) \rightarrow 0 \\ \Delta, \Delta'; C; \Gamma, f{:}\forall[\Delta'; C].(\tau_1, \ldots, \tau_n) \rightarrow 0, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash \iota\end{array}}{\Delta; \Gamma \vdash \mathtt{fix}\, f[\Delta'; C; x_1{:}\tau_1, \ldots, x_n{:}\tau_n].\iota : \forall[\Delta'; C].(\tau_1, \ldots, \tau_n) \rightarrow 0} \ (f, x_1, \ldots, x_n \notin \Gamma)$$

$$\frac{\Delta \vdash \eta \qquad \Delta; \Gamma \vdash v : \forall[\rho, \Delta'; C].(\tau_1, \ldots, \tau_n) \rightarrow 0}{\Delta; \Gamma \vdash v[\eta] : \forall[\Delta'; C].(\tau_1, \ldots, \tau_n) \rightarrow 0[\eta/\rho]}$$

$$\frac{\Delta \vdash C \qquad \Delta; \Gamma \vdash v : \forall[\epsilon, \Delta; C'].(\tau_1, \ldots, \tau_n) \rightarrow 0}{\Delta; \Gamma \vdash v[C] : \forall[\Delta; C'].(\tau_1, \ldots, \tau_n) \rightarrow 0[C/\epsilon]} \qquad \frac{\Delta; \Gamma \vdash v : \tau' \qquad \Delta \vdash \tau' = \tau}{\Delta; \Gamma \vdash v : \tau}$$

$\boxed{\Phi \vdash \iota}$

$$\frac{\Delta, \rho; C \oplus \{\rho \mapsto \langle \mathit{junk}_1, \ldots, \mathit{junk}_n \rangle\}; \Gamma, x{:}\mathit{ptr}(\rho) \vdash \iota}{\Delta; C; \Gamma \vdash \mathtt{malloc}\, x, \rho, n; \iota} \ (x \notin \Gamma, \rho \notin \Delta)$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \mathit{ptr}(\eta) \\ \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} \qquad \Delta; C' \oplus \{\eta \mapsto \mathit{junk}\}; \Gamma \vdash \iota\end{array}}{\Delta; C; \Gamma \vdash \mathtt{free}\, v; \iota}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \mathit{ptr}(\eta) \qquad \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} \\ \Delta; \Gamma \vdash v' : \tau' \quad \Delta; C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_{i-1}, \tau', \tau_{i+1}, \ldots, \tau_n \rangle\}; \Gamma \vdash \iota\end{array}}{\Delta; C; \Gamma \vdash v[i]{:=}v'; \iota} \ (1 \leq i \leq n)$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \mathit{ptr}(\eta') \\ \Delta \vdash C = C' \oplus \{\eta' \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} \qquad \Delta; C; \Gamma, x{:}\tau_i \vdash \iota\end{array}}{\Delta; C; \Gamma \vdash x{=}v[i]; \iota} \ \left(\begin{array}{c} x \notin \Gamma \\ 1 \leq i \leq n \end{array}\right)$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \forall[\cdot; C'].(\tau_1, \ldots, \tau_n) \rightarrow 0 \qquad \Delta \vdash C = C' \\ \Delta; \Gamma \vdash v_1 : \tau_1 \qquad \cdots \qquad \Delta; \Gamma \vdash v_n : \tau_n\end{array}}{\Delta; C; \Gamma \vdash v(v_1, \ldots, v_n)} \qquad \frac{}{\Delta; C; \Gamma \vdash \mathtt{halt}}$$

Figure 3: Language of Locations: Value and Instruction Typing

2.4 Soundness

Our typing rules enforce the property that well-typed programs cannot enter *stuck states*. A state (S, ι) is stuck when no reductions of the operational semantics apply and $\iota \neq \texttt{halt}$. The following theorem captures this idea formally:

Theorem 2 (Soundness) *If* $\vdash S : C$ *and* $\cdot; C; \cdot \vdash \iota$ *and* $(S, \iota) \longmapsto \ldots \longmapsto (S', \iota')$ *then* (S', ι') *is not a stuck state.*

We prove soundness syntactically in the style of Wright and Felleisen [27]. The full proof appears in Appendix A.

3 Non-linear Constraints

Most linear type systems contain a class of non-linear values that can be used in a completely unrestricted fashion. Our system is similar in that it admits non-linear constraints, written $\{\eta \mapsto \tau\}^\omega$. They are characterized by the axiom:

$$\Delta \vdash \{\eta \mapsto \tau\}^\omega = \{\eta \mapsto \tau\}^\omega \oplus \{\eta \mapsto \tau\}^\omega$$

Unlike the constraints of the previous section, non-linear constraints may be duplicated. Therefore, it is not sound to deallocate memory described by non-linear constraints or to use it at different types. Because there are strictly fewer operations on non-linear constraints than linear constraints, there is a natural subtyping relation between the two: $\{\eta \mapsto \tau\} \leq \{\eta \mapsto \tau\}^\omega$. We extend the subtyping relationship on single constraints to collections of constraints with rules for reflexivity, transitivity, and congruence. For example, assume *add* has type $\forall[\rho_1, \rho_2, \epsilon; \{\rho_1 \mapsto \langle int \rangle\}^\omega \oplus \{\rho_2 \mapsto \langle int \rangle\}^\omega \oplus \epsilon].(ptr(\rho_1), ptr(\rho_2)) \rightarrow \mathtt{0}$ and consider this code:

<u>Instructions</u>	<u>Constraints (Initially $\emptyset$)</u>
$\texttt{malloc}\ x, \rho, 1;$	$C_1 = \{\rho \mapsto \langle junk \rangle\},\ x : ptr(\rho)$
$x[0]{:=}3;$	$C_2 = \{\rho \mapsto \langle int \rangle\}$
$add[\rho, \rho, \emptyset](x, x)$	$C_2 \leq \{\rho \mapsto \langle int \rangle\}^\omega = \{\rho \mapsto \langle int \rangle\}^\omega \oplus \{\rho \mapsto \langle int \rangle\}^\omega \oplus \emptyset$

Typing rules for non-linear constraints are presented in Figure 4.

3.1 Non-linear Constraints and Dynamic Type Tests

Although data structures described by non-linear constraints cannot be deallocated or used to store objects of varying types, we can still take advantage of the sharing implied by singleton pointer types. More specifically, code can use weak constraints to perform a dynamic type test on a particular object and simultaneously refine the types of many aliases of that object.

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash v : \mathit{ptr}(\eta) \\ \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^\omega \qquad \Delta; C; \Gamma, x{:}\tau_i \vdash \iota\end{array}}{\Delta; C; \Gamma \vdash x{=}v[i]; \iota} \begin{pmatrix} x \notin \Gamma \\ 1 \leq i \leq n \end{pmatrix}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash v : \mathit{ptr}(\eta) \qquad \Delta;\Gamma \vdash v' : \tau' \\ \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^\omega \qquad \Delta \vdash \tau' = \tau_i \qquad \Delta; C; \Gamma \vdash \iota\end{array}}{\Delta; C; \Gamma \vdash v[i]{:=}v'; \iota} \ (1 \leq i \leq n)$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash v : \forall[\cdot; C'].(\tau_1, \ldots, \tau_n) \to 0 \qquad \Delta \vdash C \leq C' \\ \Delta;\Gamma \vdash v_1 : \tau_1 \qquad \cdots \qquad \Delta;\Gamma \vdash v_n : \tau_n\end{array}}{\Delta; C; \Gamma \vdash v(v_1, \ldots, v_n)} \qquad \frac{\vdash S : C' \qquad \vdash C' \leq C}{\vdash S : C}$$

Figure 4: Language of Locations: Non-linear Constraints

To demonstrate this application, we extend the language discussed in the previous section with a simple form of option type $?\langle \tau_1, \ldots, \tau_n \rangle$ (see Figure 5). Options may be `null` or a memory block $\langle \tau_1, \ldots, \tau_n \rangle$. The `mknull` operation associates the name ρ with `null` and the `tosum` v, τ instruction injects the value v (a location containing null or a memory block) into a location for the option type $?\langle \tau_1, \ldots, \tau_n \rangle$. In the typing rules for `tosum` and `ifnull`, the annotation ϕ may either be ω, which indicates a non-linear constraint or $\cdot$, the empty annotation, which indicates a linear constraint.

The `ifnull` v `then` ι_1 `else` ι_2 construct tests an option to determine whether it is `null` or not. Assuming v has type $\mathit{ptr}(\eta)$, we check the first branch (ι_1) with the constraint $\{\eta \mapsto \mathit{null}\}^\phi$ and the second branch with the constraint $\{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^\phi$ where $\langle \tau_1, \ldots, \tau_n \rangle$ is the appropriate non-null variant. As before, imagine sp is the stack pointer, which contains an integer option.

`(* constraints` $= \{\eta \mapsto \langle \mathit{ptr}(\eta') \rangle, \eta' \mapsto ?\langle \mathit{int} \rangle\}$, $sp{:}\mathit{ptr}(\eta)$	`*)`
$r_1{=}sp[1]$; `(*` $r_1{:}\mathit{ptr}(\eta')$	`*)`
`ifnull` r_1 `then halt` `(* null check`	`*)`
`else` $\cdots$ `(* constraints` $= \{\eta \mapsto \langle \mathit{ptr}(\eta') \rangle\} \oplus \{\eta' \mapsto \langle \mathit{int} \rangle\}^\omega$	`*)`

Notice that a single null test refines the type of multiple aliases; both r_1 and its alias on the stack $sp[1]$ can be used as integer references in the else clause. Future loads of r_1 or its alias will not have to perform a null-check.

We have proven these additional features of our language sound.

4 Compiling displays

One of Pascal's distinctive features is that it supports lexical scoping, nested functions, and yet treats functions as second class citizens (functions cannot be returned or stored in data structures).

Syntax:

$$\begin{array}{llll} \textit{types} & \tau & ::= & \ldots \mid ?\langle\tau_1,\ldots,\tau_n\rangle \mid \textit{null} \\ \textit{values} & v & ::= & \ldots \mid \texttt{null} \\ \textit{instructions} & \iota & ::= & \ldots \mid \texttt{mknull}\ x,\rho;\iota \mid \texttt{tosum}\ v, ?\langle\tau_1,\ldots,\tau_n\rangle \mid \\ & & & \texttt{ifnull}\ v\ \texttt{then}\ \iota_1\ \texttt{else}\ \iota_2 \end{array}$$

Operational semantics:

$$\begin{array}{lll} (S, \texttt{mknull}\ x,\rho;\iota) & \longmapsto & (S\{\ell \mapsto \texttt{null}\}, \iota[\ell/\rho][\texttt{ptr}(\ell)/x]) \\ \quad \text{where } \ell \notin S & & \\ (S, \texttt{tosum}\ v, ?\langle\tau_1,\ldots,\tau_n\rangle;\iota) & \longmapsto & (S,\iota) \\ (S\{\ell \mapsto \texttt{null}\}, & & \\ \quad \texttt{ifnull}\ \texttt{ptr}(\ell)\ \texttt{then}\ \iota_1\ \texttt{else}\ \iota_2) & \longmapsto & (S\{\ell \mapsto \texttt{null}\}, \iota_1) \\ (S\{\ell \mapsto \langle v_1,\ldots,v_n\rangle\}, & & \\ \quad \texttt{ifnull}\ \texttt{ptr}(\ell)\ \texttt{then}\ \iota_1\ \texttt{else}\ \iota_2) & \longmapsto & (S\{\ell \mapsto \langle v_1,\ldots,v_n\rangle\}, \iota_2) \end{array}$$

Static Semantics:

$$\frac{}{\Delta;\Gamma \vdash \texttt{null} : \textit{null}} \qquad \frac{\Delta,\rho; C \oplus \{\rho \mapsto \textit{null}\}; \Gamma, x{:}\textit{ptr}(\rho) \vdash \iota}{\Delta; C; \Gamma \vdash \texttt{mknull}\ x,\rho;\iota}\ (x \notin \Gamma, \rho \notin \Delta)$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash v : \textit{ptr}(\eta) \qquad \Delta \vdash C = C' \oplus \{\eta \mapsto \textit{null}\}^{\phi} \\ \Delta \vdash ?\langle\tau_1,\ldots,\tau_n\rangle \qquad \Delta; C' \oplus \{\eta \mapsto ?\langle\tau_1,\ldots,\tau_n\rangle\}^{\phi};\Gamma \vdash \iota\end{array}}{\Delta; C; \Gamma \vdash \texttt{tosum}\ v, ?\langle\tau_1,\ldots,\tau_n\rangle;\iota}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash v : \textit{ptr}(\eta) \\ \Delta \vdash C = C' \oplus \{\eta \mapsto \langle\tau_1,\ldots,\tau_n\rangle\}^{\phi} \qquad \Delta; C' \oplus \{\eta \mapsto ?\langle\tau_1,\ldots,\tau_n\rangle\}^{\phi};\Gamma \vdash \iota\end{array}}{\Delta; C; \Gamma \vdash \texttt{tosum}\ v, ?\langle\tau_1,\ldots,\tau_n\rangle;\iota}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash v : \textit{ptr}(\eta) \qquad \Delta \vdash C = C' \oplus \{\eta \mapsto ?\langle\tau_1,\ldots,\tau_n\rangle\}^{\phi} \\ \Delta; C' \oplus \{\eta \mapsto \textit{null}\}^{\phi};\Gamma \vdash \iota_1 \qquad \Delta; C' \oplus \{\eta \mapsto \langle\tau_1,\ldots,\tau_n\rangle\}^{\phi};\Gamma \vdash \iota_2\end{array}}{\Delta; C; \Gamma \vdash \texttt{ifnull}\ v\ \texttt{then}\ \iota_1\ \texttt{else}\ \iota_2}$$

Figure 5: Language of Locations: Extensions for option types

$$
\begin{array}{llcl}
 & x & \in & \text{ArgVar} \\
 & f & \in & \text{ProcedureVar} \\
 & & & \\
\textit{types} & \tau & ::= & \mathit{int} \mid (\mathit{int}, \ldots, \mathit{int}) \to \mathit{int} \\
\textit{declarations} & D & ::= & x = v \\
\textit{functions} & F & ::= & \mathtt{fun} f(x_1, \ldots, x_n) \\
 & & & \quad\quad \mathtt{let}\, F_1 \ldots F_m \; D_1 \ldots D_k \; \mathtt{in}\, \iota \\
\textit{values} & v & ::= & i \mid x \\
\textit{instructions} & \iota & ::= & x := v; \iota \mid x := f(v_1, \ldots, v_n); \iota \mid \mathtt{return}\; x
\end{array}
$$

Figure 6: A Simple Lexically-scoped Imperative Language (SIL)

This feature permits the use of a *display* to look up variables residing in outer scopes. Following the description in Aho, Sethi and Ullman [1], a display is a heap-allocated array of pointers to the dynamically closest enclosing activation record or stack frame. A function at lexical depth d accesses a variable at depth $d' \leq d$ from the stack frame pointed to by $display[d']$. The display is maintained by each function on entry and exit. On entry, a function at depth d saves the contents of $display[d]$ on its stack, and overwrites $display[d]$ with a pointer to its own stack frame. On exit, the function restores $display[d]$.

It is possible to track the aliases created by displays precisely using alias types. To demonstrate this fact, we show how to compile a simple imperative language making use of displays into the language of locations in such a way that heap-allocated stack frames can be safely and explicitly deallocated. In order to highlight issues regarding displays, SIL has been kept extremely simple. Figure 6 shows its syntax. SIL only has arguments and results of integer type. Furthermore, declarations may not be mutually recursive. By design, SIL's only interesting feature is that it supports lexical scoping for nested functions. The typing rules and dynamic semantics for this language are easy to define and have therefore been omitted.

In the remainder of this section, we show how SIL terms are compiled into the language of locations. Our translation assumes that terms have been renamed so that all variable names are distinct, and that the whole program text is available. In order to simplify the presentation, we extend the language of locations with a new binding construct $x = v; \iota$ which has the operational effect of substituting the value v for x in ι, and has the obvious type-checking rule. Given these caveats, we believe our translation is type-preserving and semantics-preserving, although we do not have a formal proof.

Table 1 shows the naming conventions we use, and Figure 7 shows a snapshot of a translated program during execution. The display is bound to variable x_D, and the current stack frame to x_f. Throughout the translation we use M to denote the maximum lexical depth in the program. As shown in the figure, the display is an array of M elements. The translation is broken up into the type, value, declaration, function, and instruction sub-translations. Before continuing, we define some notation.

Definition 3 (Notation) *For any variable x, function variable f, and integer d,*

name	description
ρ_f	Location of the current stack frame
x_f	$ptr(\rho_f)$
ρ_{sp}	Location of the previous stack frame
ρ_{sv}	Location of the the saved display slot
$\rho_{f,i}$	ρ_f for parent at lexical depth i
$\rho_{sp,i}$	ρ_{sp} for parent at lexical depth i
$\rho_{sv,i}$	ρ_{sv} for parent at lexical depth i
ϵ_S	The rest of the store
ρ_D	Location of the display
x_D	$ptr(\rho_D)$
x_{cont}	Continuation

Table 1: Variable naming conventions

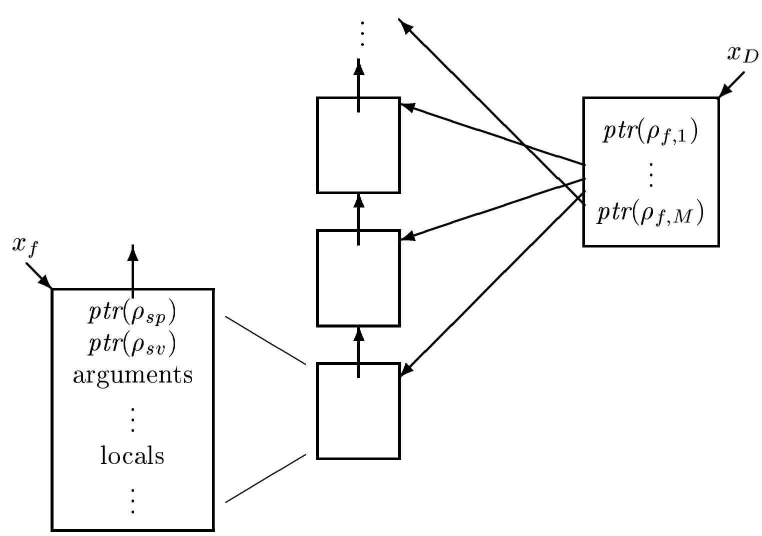


Figure 7: Snapshot of the stack and display during the execution of a translated program.

1. $(x)^d$ *is syntactic shorthand for* $x_1, \ldots, x_d$. *For variables with compound subscripts,* $(\rho_{sp})^d$ *corresponds to* $\rho_{sp,1}, \ldots, \rho_{sp,d}$.

2. $\texttt{depth}(x)$ *is the lexical depth at which* x *is defined in the program.*

3. $\texttt{offset}(x)$ *is the offset of* x *within its stack frame.*

4. $\texttt{locals}(f)$ *is the number of local variable declarations in the body of* f.

The most important part of the translation is the type translation because it encodes most of the dynamic invariants, and shows how the language of locations is able to track sharing and aliases.

Definition 4 (Type translation helper functions) *In the following* d, n, k, *and* M *denote integers.*

1. $\Delta(d, M) = \rho_f, \rho_{sp}, \rho_D, (\rho_{sp})^d, (\rho_{sv})^d, (\rho_f)^M, \epsilon_S$
   $\Delta(d, M)$ *is the translated typing context for a function at lexical depth* d. *The variables in* $(\rho_{sp})^d$ *and* $(\rho_{sv})^d$ *are needed to give types to the enclosing stack frames which mention these locations. The variables in* $(\rho_f)^M$ *are used to give a type to the display.*

2. $\tau_f(n, k) = \langle \mathit{ptr}(\rho_{sp}), \mathit{junk}, \mathrm{int}_1, \ldots, \mathrm{int}_n, \mathit{junk}_1, \ldots \mathit{junk}_k \rangle$
   $\tau_f(n, k)$ *is the type of the stack frame for an* n *argument function containing* k *local variables on entry.*

3. $C_{(\tau)^d} = \{\rho_{f,1} \mapsto \tau_1\} \oplus \ldots \oplus \{\rho_{f,d} \mapsto \tau_d\}$
   The types $\tau_1, \ldots, \tau_n$ *correspond to the types of the enclosing stack frames.* $C_{(\tau)^d}$ *represents the constraint which ties the types of the stack frames* (τ_i) *to their locations* ($\rho_{f,i}$). *These types are needed so that variables in outer scope, which reside in those stack frames, will be accessible.*

4. $D_M = \{\rho_D \mapsto \langle \mathit{ptr}(\rho_{f,1}), \ldots, \mathit{ptr}(\rho_{f,M}) \rangle\}$
   D_M *is the constraint giving the type of the display in a program with maximum lexical depth* M.

The type translation depends on three aspects of the program context: the globally maximum lexical scope M, the types of the stack frames of the lexically enclosing functions $(\tau)^d$, and the number of local variables contained in the function whose type we are translating k.

Definition 5 (Type translation)

$$
\begin{array}{rcl}
|\ \mathrm{int}\ |_{(\tau)^d;k} & = & \mathrm{int} \\
|\ (\mathrm{int}_1, \ldots, \mathrm{int}_n) \to \mathrm{int}\ |_{(\tau)^d;k} & = & \\
\multicolumn{3}{l}{\quad \forall[\Delta(d, M); \{\rho_f \mapsto \tau_f(n, k)\} \oplus C_{(\tau)^d} \oplus \epsilon_S \oplus D_M].(\mathit{ptr}(\rho_f), \mathit{ptr}(\rho_D), \tau_{cont}) \to 0}
\end{array}
$$

where $\tau_{cont} = \forall[\cdot; \{\rho_f \mapsto \mathit{junk}\} \oplus C_{(\tau)^d} \oplus \epsilon_S \oplus D_M].(\mathit{ptr}(\rho_D), \mathrm{int}) \to 0$

Because the translated types are so precise, they capture many of the invariants of the translation. The type of the current stack frame (τ_f) on function entry is

$$\langle \mathit{ptr}(\rho_{sp}), \mathit{junk}, \mathit{int}_1, \ldots, \mathit{int}_n, \mathit{junk}_1, \ldots \mathit{junk}_k \rangle$$

indicating that the caller will allocate the callee's stack frame (this is possible because all functions are statically known). Furthermore, the caller will put the arguments in positions 3 through $n+2$, and will install a pointer in the first position. The continuation demands that $\{\rho_f \mapsto \mathit{junk}\}$ forcing the callee to free its stack frame if it wishes to return. And, because the type of the display is unchanged in τ_{cont}, the continuation must be called with the same set of stack frames in the display – although the contents of the frames may have changed. It would be a typing error for the callee to forget to restore the display before returning!

The remaining translations are the obvious ones given the constraints placed on them by the type translation and by the intended semantics.

The value translation produces a sequence of instructions, and a value. As formally defined our instruction sequences must end in either `halt` or a function call. For the purpose of this translation, we allow sequences to end arbitrarily. We use the notation $\cdot$ to denote an empty sequence of instructions.

Definition 6 (Value Translation)

$$\begin{array}{rcl} \mid i \mid & = & (\cdot, i) \\ \mid x \mid & = & (x' {=} x_D[\mathtt{depth}(x)]; x'' {=} x'[\mathtt{offset}(x)],\ x'') \end{array}$$

where x' and x'' are fresh variables.

The declaration translation is similar to the value translation for variables. The difference is that we are initializing a variable in the current stack frame so we do not have to access the variable through the display.

Definition 7 (Declaration Translation)

$$\mid x = v \mid \quad = \quad \iota'; x_f[\mathtt{offset}(x)] {:=} v'$$

where $(\iota', v') = \mid v \mid$

As stated earlier, the function translation needs to bind the name of the nested function within the body of its parent. In keeping the language of locations simple, we did not include such a binding form. We introduce one now for convenience.

Definition 8 (Function Translation) *Let x' and x'' denote fresh variables in the following.*

$$\left|\begin{array}{l} \texttt{fun} f(x_1,\ldots,x_n) \\ \quad \texttt{let} \\ \quad\quad F_1 \ldots F_m \\ \quad\quad D_1 \ldots D_k \\ \quad \texttt{in} \\ \quad \iota \end{array}\right|_{(\tau)^d} = \begin{array}{l} f = \texttt{fix}\, f[\Delta(d,M); \\ \qquad \{\rho_f \mapsto \tau_f(n,k)\} \oplus C_{(\tau)^d} \oplus \epsilon_S \oplus D_M; \\ \qquad x_f{:}ptr(\rho_f), x_D{:}ptr(\rho_D), x_{cont}{:}\tau_{cont}]. \\ \quad \texttt{(* Save the display slot *)} \\ \quad x{=}x_D[d+1]; \\ \quad x_f[2]{:=}x; \\ \quad x_D[d+1]{:=}x_f; \\ \quad \texttt{(* Translate the nested functions *)} \\ \quad |\, F_1 \,|_{(\tau)^{d+1}}; \ldots; |\, F_m \,|_{(\tau)^{d+1}}; \\ \quad \texttt{(* Initialize the local variables *)} \\ \quad |\, D_1 \,|; \ldots; |\, D_k \,|; \\ \quad \texttt{(* Translate the body *)} \\ \quad |\, \iota \,|_{(\tau)^{d+1}} \end{array}$$

where $\tau_{d+1} = \langle ptr(\rho_{sp,d+1}), ptr(\rho_{sv,d+1}), \mathrm{int}_1, \ldots, \mathrm{int}_{n+k}\rangle$
and τ_{cont} *is the same as in the type translation.*

The instruction translation is straightforward except that the lexical depth d is one greater than the lexical depth of the function in which the instruction occurs (see the function translation). Therefore $\rho_{f,d}$ is not bound (see the definition of $\Delta(d,M)$). Instead it is represented by ρ_f. Similar problems occur with $\rho_{sv,d}$ and $\rho_{sp,d}$. These problems lead to the special casing of Δ' and ϵ'_S in the translation of function application depending on whether the call is to a function in an outer scope, or to an immediate child.

Definition 9 (Instruction Translation) *Let x', x'', ρ'_f, and x'_f denote fresh variables. Each instruction is translated as follows.*

1. *Let* $(\iota', v') = |\, v \,|_{(\tau)^d}$ *in the following.*

$$|\, x := v; \iota \,|_{(\tau)^d} = \begin{array}{l} \iota'; \\ x'{=}x_D[\texttt{depth}(x)]; \\ x'[\texttt{offset}(x)]{:=}v'; \\ |\, \iota \,|_{(\tau)^d} \end{array}$$

2. *In the following, let* $d' = \texttt{depth}(f)$, $k' = \texttt{locals}(f)$, *and* $(\iota'_i, v'_i) = \mid v_i \mid_{(\tau)^d}$ *for* $1 \le i \le m'$.

$$
\begin{array}{rl}
\mid x := f(v_1, \ldots, v_{m'}); \iota \mid_{(\tau)^d} = & \texttt{(* Allocate a new stack frame *)} \\
& \texttt{malloc}\, x'_f, \rho'_f, m' + k' + 2; \\
& \texttt{(* Initialize the parameters *)} \\
& \iota'_1; x'_f[3] := v'_1; \\
& \vdots \\
& \iota'_m; x'_f[m' + 2] := v'_{m'}; \\
& \texttt{(* Initialize the new frame's back pointer *)} \\
& x'_f[1] := x_f; \\
& \texttt{(* Call } f \texttt{ *)} \\
& f[\Delta'][\epsilon'_S](x'_f, x_D, v_{cont})
\end{array}
$$

where

$$
\begin{array}{rl}
v_{cont} = & \texttt{fix}\, f_{cont}[\cdot; \\
& \quad \{\rho'_f \mapsto \mathit{junk}\} \oplus \{\rho_f \mapsto \tau_d\} \oplus C_{(\tau)^{d-1}} \oplus \epsilon_S \oplus D_M; \\
& \quad x_D : \mathit{ptr}(\rho_D), x_{return} : \mathrm{int}]. \\
& x_f = x_D[d]; \\
& x'' = x_D[\texttt{depth}(x)]; \\
& x''[\texttt{offset}(x)] := x_{return}; \\
& \mid \iota \mid_{(\tau)^d}
\end{array}
$$

The definitions of Δ' *and* ϵ'_S *vary depending on whether* $d' = d$ *or* $d' < d$.

If $d' < d$ *then*

$$
\begin{array}{rcl}
\Delta' & = & \rho'_f, \rho_f, \rho_D, (\rho_{sp})^{d'}, (\rho_{sv})^{d'}, \rho_{f,1}, \ldots, \rho_{f,d-1}, \rho_f, \rho_{f,d+1}, \ldots, \rho_{f,M} \\
\epsilon'_S & = & \{\rho_{f,d'+1} \mapsto \tau_{d'+1}\} \oplus \ldots \oplus \{\rho_{f,d-1} \mapsto \tau_{d-1}\} \oplus \{\rho_f \mapsto \tau_d\} \oplus \epsilon_S
\end{array}
$$

If $d' = d$ *then*

$$
\begin{array}{rcl}
\Delta' & = & \rho'_f, \rho_f, \rho_D, (\rho_{sp})^{d'-1}, \rho_{sp}, (\rho_{sv})^{d'-1}, \rho_{f,d}, \rho_{f,1}, \ldots, \rho_{f,d-1}, \rho_f, \rho_{f,d+1}, \ldots, \rho_{f,M} \\
\epsilon'_S & = & \epsilon_S
\end{array}
$$

3. *Let* $(\iota', v') = \mid x \mid$ *in the following.*

$$
\begin{array}{rl}
\mid \texttt{return}\; x \mid_{(\tau)^d} = & \texttt{(* Load the return value *)} \\
& \iota' \\
& \texttt{(* Restore the saved display slot *)} \\
& x'' = x_f[2]; \\
& x_D[d] := x''; \\
& \texttt{free}\; x_f; \\
& \texttt{(* Call the continuation *)} \\
& x_{cont}(x_D, v')
\end{array}
$$

5 Related Work

Because our type system is constructed from standard type-theoretic building blocks, including linear and singleton types, it is relatively straightforward to implement these ideas in a modern type-directed compiler. Our Typed Assembly Language (TAL) implementation already contained many of the required constructors. In particular, singleton types were already used to enable array bounds check elimination in the style of Xi and Pfenning [28] and run-time type analysis [5], and may soon be used to facilitate static checking of expressive security policies [25]. Moreover TAL, like other F^ω-based type-directed compilers such as TIL(T) [20] and FLINT [18], already had a rich kind structure so adding location and store polymorphism just involved adding two base kinds.

In some ways, our new mechanisms simplify previous work. Previous versions of TAL [12, 11] possessed two separate mechanisms for initializing data structures. Uninitialized heap-allocated data structures were stamped with the type at which they would be used. On the other hand, stack slots could be overwritten with values of arbitrary types. Our new system allows us to treat memory more uniformly. In fact, our new language can encode stack types similar to those described by Morrisett [11] except that activation records are allocated on the heap rather than using a conventional call stack. In this report, we have shown how our new language can compile a simple imperative language in such a way that it explicitly manages its own memory on a stack and uses a display. We believe we could also encode the exception-handling mechanisms described by Morrisett.

The development of our language was inspired by the Calculus of Capabilities [4] (CC). CC provides an alternative to the region-based type system developed by Tofte and Talpin [22]. Because safe region deallocation requires that no aliases be used in the future, CC tracks region aliases. In our new language we adapt CC's techniques to track both object aliases and object type information.

Our work also has close connections with research on alias analyses [6, 19, 17]. Much of that work aims to facilitate program optimizations that require aliasing information in order to be *correct*. However, these optimizations do not necessarily make it harder to check the *safety* of the resulting program. Other work [8, 7] attempts to determine when programs written in unsafe languages, such as C, perform potentially unsafe operations. Our goals are closer to the latter application but differ because we are most interested in compiling *safe* languages and producing low-level code that can be proven safe in a single pass over the program. Moreover, our main result is not to present some new analysis technique, but rather to represent and check the results of analysis, and, in particular, to represent aliasing in low-level compiler-introduced data structures rather than to represent aliasing in source-level data. As we discussed above, the principle advantage of our techniques is that they can be integrated smoothly and efficiently into polymorphic and higher-order type systems such as TAL.

6 Future Work

The language of locations is a flexible framework for reasoning about sharing and destructive operations in a type-safe manner. However, our work to date is only a first step in this area and we are investigating a number of extensions. For instance, functions that do not destructively modify their arguments may be indifferent towards the sort of constraint they require. If we augmented our type system with polymorphism over the sort of constraint (linear or non-linear) then we could preserve the sort of constraint across function calls. Bounded polymorphism provides even more expressive power as it allows linear constraints to be temporarily viewed as non-linear constraints before recovering the linear information. Finally, we are also working on integrating recursive types into the type system. Recursive types would allow us to capture regular repeating structure in the store. We think it will be possible to integrate this collection of features into the language; however we are as yet unsure of the precise form each should take. When combined, we believe these mechanisms will provide us with a safe, but rich and reusable, set of memory abstractions.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag. Volume 528 of *Lecture Notes in Computer Science*.

[3] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In *Thirteenth Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 41–51, Bombay, 1993. In Shyamasundar, ed., Springer-Verlag, LNCS 761.

[4] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, January 1999.

[5] Karl Crary and Stephanie Weirich. Flexible type analysis. In *ACM International Conference on Functional Programming*, pages 233–248, Paris, September 1999.

[6] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *ACM Conference on Programming Language Design and Implementation*, pages 230–241, Orlando, June 1994.

[7] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Detecting memory errors via static pointer analysis (preliminary experience). In *ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, Montreal, June 1998.

[8] David Evans. Static detection of dynamic memory errors. In *ACM Conference on Programming Language Design and Implementation*, Philadelphia, May 1996.

[9] Naoki Kobayashi. Quasi-linear types. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 29–42, San Antonio, January 1999.

[10] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[11] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based Typed Assembly Language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28-52. Springer-Verlag, 1998.

[12] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 3(21):528–569, May 1999.

[13] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[14] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 146–159, Paris, January 1997.

[15] G. D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[16] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Conference Record of the 25th National ACM Conference*, pages 717–740, Boston, August 1972.

[17] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1996.

[18] Z. Shao. An overview of the FLINT/ML compiler. In *Workshop on Types in Compilation*, Amsterdam, June 1997. ACM. Published as Boston College Computer Science Dept. Technical Report BCCS-97-03.

[19] B. Steensgaard. Points-to analysis in linear time. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, January 1996.

[20] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, May 1996.

[21] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.

[22] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[23] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *ACM International Conference on Functional Programming and Computer Architecture*, San Diego, CA, June 1995.

[24] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[25] David Walker. A type system for expressive security policies. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, Boston, January 2000. To appear.

[26] A. K. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4), December 1995.

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

[28] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

A Proof of Type Soundness

This section states and then proves a theorem about the soundness of the type system of our language. We use the syntactic proof technique popularized by Wright and Felleisen [27]. The central lemmas required by the proof technique are Preservation, which states that well-formed programs always step to well-formed programs and Progress, which states that well-formed programs can always take a step (unless they have already halted gracefully using the `halt` instruction).

In order to use this technique, all of the intermediate steps in the computation must type-check. This is not the case for the type system defined in the body of the paper! Intermediate programs may fail to type check because we have defined no relationship between the types *null* and $?\langle\tau_1,\ldots,\tau_n\rangle$, and similarly, between $\langle\tau_1,\ldots,\tau_n\rangle$ and $?\langle\tau_1,\ldots,\tau_n\rangle$. The `tosum` instruction does relate these types locally, but that correspondence is lost when type-checking the remaining instructions. To maintain this information, we introduce a subtyping relationship. Because we have only added rules to the type system, strictly more programs type check. Therefore, type soundness of the new type system implies type soundness of the original system. The new subtying rules follow. The typing rules in their entirety are presented in Appendix B.

$\boxed{\Delta \vdash \tau \leq \tau'}$

$$(\tau\text{-sub-null})\ \frac{\Delta \vdash \langle\tau_1,\ldots,\tau_n\rangle}{\Delta \vdash \mathit{null} \leq ?\langle\tau_1,\ldots,\tau_n\rangle} \qquad (\tau\text{-sub-tuple})\ \frac{}{\Delta \vdash \langle\tau_1,\ldots,\tau_n\rangle \leq ?\langle\tau_1,\ldots,\tau_n\rangle}$$

$$(\tau\text{-sub-eq})\ \frac{\Delta \vdash \tau = \tau'}{\Delta \vdash \tau \leq \tau'} \qquad (\tau\text{-sub-trans})\ \frac{\Delta \vdash \tau \leq \tau'' \qquad \Delta \vdash \tau'' \leq \tau'}{\Delta \vdash \tau \leq \tau'}$$

$\boxed{\Delta \vdash C \leq C'}$

$$(\text{C-sub})\ \frac{\Delta \vdash \tau \leq \tau'}{\Delta \vdash \{\eta \mapsto \tau\}^{\phi} \leq \{\eta \mapsto \tau'\}^{\phi}}\ (\phi = \cdot \text{ or } \phi = \omega)$$

We make use of several notational conveniences in this appendix as well. We let the meta-variable ϕ range over constraint "flags," $\cdot$ (the empty flag that denotes a linear constraint) and ω (for the flag that denotes a non-linear constraint). We let the meta-variable β range over location variables ρ and constraint variables ϵ. We let the meta-variable c range over locations η and constraints C. We use the notation $X = X'$ to denote syntactic equality of objects X and X' up to α-conversion of bound variables. The one exception is a store S where $S = S'$ denotes syntactic equality up to alpha-conversion of bound variables and re-ordering of the elements of the store. Definitional equality is always preceded by a turnstyle and typing context: $\Delta \vdash X = X'$. The notation $S\{\ell \mapsto v\}$ denotes the store S extended with the mapping $\{\ell \mapsto v\}$. It is undefined if ℓ appears in the domain of S.

The rest of the appendix gives a formal statement and proof of the type soundness theorem.

Definition 10 (Stuck State) *A state* (S, ι) *is* stuck *if* $\iota \neq \texttt{halt}$ *and there does not exist another state* (S', ι') *such that* $(S, \iota) \longmapsto (S', \iota')$.

Theorem 11 (Soundness) *If* $\vdash S : C$ *and* $\cdot; C; \cdot \vdash \iota$ *then there is no evaluation sequence* $(S, \iota) \longmapsto \ldots \longmapsto (S', \iota')$ *such that* (S', ι') *is a stuck state.*

Proof:

We prove Soundness by induction on the length of the evaluation sequence. By Progress, the initial state (S, ι) is not stuck. Assume all evaluation sequences of length i, for $i > 0$ do not lead to stuck states. Suppose $(S, \iota) \longmapsto (S'', \iota'') \longmapsto \ldots \longmapsto (S', \iota')$ is an evaluation sequence of length $i + 1$. Since $\vdash (S, \iota)$ and $\cdot; C; \cdot \vdash \iota$, Preservation states that there exists a C'' such that $\vdash S'' : C''$ and $\cdot; C''; \cdot \vdash \iota''$. By induction, (S', ι') is not stuck.

$\Box$

The proofs of Preservation and Progress rely on a number of supplementary lemmas. To aid in reading the proof, we have broken the lemmas into various subsections. The first subsection presents standard lemmas describing well-formedness, substitution, and canonical forms. The second subsection establishes some key properties of constraint equality and subtyping. The third section proves lemmas that relate the store to the constraints. Finally, the fourth section presents the proofs for preservation and progress.

A.1 Standard Lemmas

This section begins with a number of standard substitution lemmas for various kinds of variables (ρ, ϵ and x). In each case the lemma states that substitution preserves well-formedness. The proofs are all by induction on the typing derivation in question, and have been omitted.

Lemma 12 (Type Substitution) *Let* X *and* X' *be one of* τ *and* τ', η *and* η', *or* C *and* C'. *And let* Y *and* Y' *be one of* τ *and* τ', *or* C *and* C'.

1. *If* $\rho, \Delta \vdash X$ *and* $\cdot \vdash \eta''$ *then* $\Delta \vdash X[\eta''/\rho]$
2. *If* $\epsilon, \Delta \vdash X$ *and* $\cdot \vdash C''$ *then* $\Delta \vdash X[C''/\epsilon]$
3. *If* $\rho, \Delta \vdash X = X'$ *and* $\cdot \vdash \eta''$ *then* $\Delta \vdash X[\eta''/\rho] = X'[\eta''/\rho]$
4. *If* $\epsilon, \Delta \vdash X = X'$ *and* $\cdot \vdash C''$ *then* $\Delta \vdash X[C''/\epsilon] = X'[C''/\epsilon]$
5. *If* $\rho, \Delta \vdash Y \leq Y'$ *and* $\cdot \vdash \eta''$ *then* $\Delta \vdash Y[\eta''/\rho] \leq Y'[\eta''/\rho]$
6. *If* $\epsilon, \Delta \vdash Y \leq Y'$ *and* $\cdot \vdash C''$ *then* $\Delta \vdash Y[C''/\epsilon] \leq Y'[C''/\epsilon]$

Lemma 13 (ρ-substitution) *Let $C' = C[\ell/\rho]$, $\Gamma' = \Gamma[\ell/\rho]$, $\iota' = \iota[\ell/\rho]$, $v' = v[\ell/\rho]$, and $\tau' = \tau[\ell/\rho]$.*

1. *If $\rho, \Delta; C; \Gamma \vdash \iota$ then $\Delta; C'; \Gamma' \vdash \iota'$*
2. *If $\rho, \Delta; \Gamma \vdash v : \tau$ then $\Delta; \Gamma' \vdash v' : \tau'$*

Lemma 14 (ϵ-substitution) *Let $C' = C[C''/\epsilon]$, $\Gamma' = \Gamma[C''/\epsilon]$, $\iota' = \iota[C''/\epsilon]$, $v' = v[C''/\epsilon]$, and $\tau' = \tau[C''/\epsilon]$.*

1. *If $\cdot \vdash C''$ and $\epsilon, \Delta; C; \Gamma \vdash \iota$ then $\Delta; C'; \Gamma' \vdash \iota'$*
2. *If $\cdot \vdash C''$ and $\epsilon, \Delta; \Gamma \vdash v : \tau$ then $\Delta; \Gamma' \vdash v' : \tau'$*

Lemma 15 (x-substitution) *If $\Delta; \Gamma \vdash v : \tau$ then*

1. *If $\Delta; C; x : \tau, \Gamma \vdash \iota$ then $\Delta; C; \Gamma \vdash \iota[v/x]$*
2. *If $\Delta; x : \tau, \Gamma \vdash v' : \tau'$ then $\Delta; \Gamma \vdash v'[v/x] : \tau'$*

The next group of lemmas describes a series of standard derived rules about the well-formedness of one object given the well-formedness of another object.

Lemma 16

1. *If $\Delta \vdash \tau$ or $\Delta \vdash \tau'$ and $\Delta \vdash \tau = \tau'$ then $\Delta \vdash \tau$ and $\Delta \vdash \tau'$*
2. *If $\Delta \vdash \eta$ or $\Delta \vdash \eta'$ and $\Delta \vdash \eta = \eta'$ then $\Delta \vdash \eta$ and $\Delta \vdash \eta'$*
3. *If $\Delta \vdash C$ or $\Delta \vdash C'$ and $\Delta \vdash C = C'$ then $\Delta \vdash C$ and $\Delta \vdash C'$*
4. *If $\Delta \vdash C$ or $\Delta \vdash C'$ and $\Delta \vdash C \leq C'$ then $\Delta \vdash C$ and $\Delta \vdash C'$*

Proof:

Parts 1, 2, and 3 are proven by a simultaneous induction on the height of the typing derivations. Part 4 is also proven by induction on the typing derivation and follows from part 3.

□

Lemma 17 *If $\Delta; \Gamma \vdash v : \tau$ and for all $\tau' \in Rng(\Gamma)$, $\Delta \vdash \tau'$ then $\Delta \vdash \tau$.*

Proof:

The proof is by induction on the typing derivation for values. All cases but rules v-ρ and v-ϵ are immediate or follow directly from the induction hypothesis. The rules v-ρ and v-ϵ employ the Type Substitution Lemma (Lemma 12) after using the induction hypothesis.

□

Lemma 18 *Assuming well-formed types, if $\cdot \vdash \tau' \leq ?\langle \tau_1, \ldots, \tau_n \rangle$ then τ' is one of null, $\langle \tau_1', \ldots, \tau_n' \rangle$, or $?\langle \tau_1', \ldots, \tau_n' \rangle$ where $\cdot \vdash \tau_i' = \tau_i$ for $1 \leq i \leq n$.*

Proof: By induction on the subtyping derivation.

Lemma 19 *For any types τ_1 and τ_2 such that $\cdot \vdash \tau_1 \leq \tau_2$, one of the following must hold:*

1. $\cdot \vdash \tau_1 = \tau_2$
2. $\tau_1 = null$ *and* $\tau_2 = ?\langle \tau_1', \ldots, \tau_n' \rangle$
3. $\tau_1 = \langle \tau_1', \ldots, \tau_n' \rangle$. *and* $\tau_2 = ?\langle \tau_1'', \ldots, \tau_n'' \rangle$

Proof:

The proof proceeds by induction on the height of the derivation $\cdot \vdash \tau_1 \leq \tau_2$.

□

Lemma 20 (Canonical Forms) *If $\cdot; \cdot \vdash v : \tau$ then*

$$
\begin{array}{lll}
\tau = ptr(\ell) & \textit{implies} & v = \mathtt{ptr}(\ell) \\
\tau = junk & \textit{implies} & v = \mathtt{junk} \\
\tau = null & \textit{implies} & v = \mathtt{null} \\
\tau = \langle \tau_1, \ldots, \tau_n \rangle & \textit{implies} & v = \langle v_1, \ldots, v_n \rangle \\
\tau = \forall[\Delta'; C'].(\tau_1, \ldots, \tau_n) \rightarrow 0 & \textit{implies} & v = (\mathtt{fix}\, f[\Delta; C; \Gamma].\iota)[c_1, \ldots, c_m] \textit{ and} \\
 & & \Delta = \beta_1, \ldots, \beta_{m+k} = \beta_1, \ldots, \beta_m, \Delta' \textit{ and} \\
 & & \Gamma = x_1{:}\tau_1', \ldots, x_n{:}\tau_n' \\
\tau = ?\langle \tau_1, \ldots, \tau_n \rangle & \textit{implies} & v = \mathtt{null} \textit{ or } v = \langle v_1, \ldots, v_n \rangle
\end{array}
$$

Proof:

The proof proceeds by induction on the height of the typing derivation.

- *ptr*(ℓ)
  Only the rules v-ℓ and v-sub can generate a conclusion of this form. If rule v-ℓ is the last rule in the derivation then inspection of the rule reveals that $v = \texttt{ptr}(\ell)$. If rule v-sub is the last rule in the derivation then we know that $\cdot \vdash \tau' \leq ptr(\ell)$ and that $\cdot; \cdot \vdash v : \tau'$. By inspection of the rules for equality and subtyping, we can conclude that τ' is $ptr(\ell)$. By induction, we can conclude that $v = \texttt{ptr}(\ell)$.
- *junk*
  Only v-junk and v-sub have conclusions of this form. The case for v-junk is immediate. The case for v-sub proceeds similarly to the argument for *ptr*(ℓ) above.
- *null*
  Only v-null and rlev-sub have conclusions of this form. The case for v-junk is immediate. The case for v-sub proceeds similarly to the argument for *ptr*(ℓ) above.
- $\langle \tau_1, \ldots, \tau_n \rangle$
  Only v-tuple and v-sub have conclusions of this form. The case for v-tuple is immediate. The case for v-sub proceeds similarly to the argument for *ptr*(ℓ) above.
- $\forall[\Delta'; C'].(\tau_1, \ldots, \tau_n) \rightarrow 0$
  Multiple rules can generate types of this form: v-fix, v-ρ, v-ϵ, and v-sub. The case for v-fix is immediate. The case for v-sub proceeds similarly to the argument for *ptr*(ℓ). Consider the case for rule v-ρ (rule v-ϵ is similar):

  $$\frac{\cdot \vdash \eta \qquad \cdot; \cdot \vdash v : \forall[\rho, \Delta''; C''].(\tau_1', \ldots, \tau_n') \rightarrow 0}{\cdot; \cdot \vdash v[\eta] : (\forall[\Delta''; C''].(\tau_1', \ldots, \tau_n') \rightarrow 0)[\eta/\rho]}$$

  By induction, we have that:

  $$\begin{array}{l} v = (\texttt{fix}\, f[\Delta; C; \Gamma].\iota)[c_1, \ldots, c_m] \text{ and} \\ \Delta = \beta_1, \ldots, \beta_{m+k} = \beta_1, \ldots, \beta_m, \rho, \Delta' \text{ and} \\ \Gamma = x_1{:}\tau_1, \ldots, x_n{:}\tau_n \end{array}$$

  Thus our result follows trivially. In particular: $v[\eta] = (\texttt{fix}\, f[\Delta; C; \Gamma].\iota)[c_1, \ldots, c_m][\eta]$
- $?\langle \tau_1, \ldots, \tau_n \rangle$
  The rules with conclusions of this form are v-opt, and v-sub. If v-opt is used then by the induction hypothesis $v = \langle v_1, \ldots, v_n \rangle$, giving the desired result. v-sub follows because the only subtype of $?\langle \tau_1, \ldots, \tau_n \rangle$, are either *null* or $\langle \tau_1, \ldots, \tau_n \rangle$.

□

Lemma 21 (Value Types) *If* $\cdot; \cdot \vdash v : \tau$ *then*

$v = \texttt{ptr}(\ell)$	*implies*	$\tau = ptr(\ell)$
$v = \texttt{junk}$	*implies*	$\tau = junk$
$v = \texttt{null}$	*implies*	$\tau = null$ *or there exists* $\tau_1, \ldots, \tau_n$ *such that* $\tau = ?\langle \tau_1, \ldots, \tau_n \rangle$
$v = \langle v_1, \ldots, v_n \rangle$	*implies*	*for* $1 \leq i \leq n$ *there exists* τ_i *such that* $\cdot; \cdot \vdash v_i : \tau_i$ *and* $\tau = \langle \tau_1, \ldots, \tau_n \rangle$ *or* $\tau = ?\langle \tau_1, \ldots, \tau_n \rangle$
$v = (\texttt{fix}\, f[\Delta; C; \Gamma].\iota)[c_1, \ldots, c_m]$ *where* $\Delta = \beta_1, \ldots, \beta_{m+k}$ *and* $\Gamma = x_1{:}\tau_1, \ldots, x_n{:}\tau_n$	*implies*	$\Delta; C; \Gamma \vdash \iota$ *and* $\tau = \mathcal{S}(\forall[\beta_{m+1}, \ldots, \beta_{m+k}; C].(\tau_1, \ldots, \tau_n) \rightarrow 0)$ *and* $\mathcal{S}$ *is the substitution* $[c_1, \ldots, c_m/\beta_1, \ldots, \beta_m]$

Proof:

For each different value, the proof is by induction on the height of the typing derviation for values. For the values $\texttt{ptr}(\ell)$ and $\texttt{junk}$ there are two typing rules that apply — one for the value itself and the v-sub rule. If the height of the derivation is 1, we must have used the first rule and we have our result immediately. If the height is greater than one, the last rule must be v-sub. Inspection of the equality and subtyping rules reveals that v-sub preserves the shape of the type in question. Hence, by induction we have our result.

For the value $(\texttt{fix}\, f[\beta_1, \ldots, \beta_{m+k}; C; x_1{:}\tau_1, \ldots, x_n{:}\tau_n].\iota)[c_1, \ldots, c_m]$, the rules v-fix, v-$\rho$, v-$\epsilon$, v-sub might have appeared last. If rule v-fix or v-sub appeared last then the argument given for the case of $\texttt{ptr}(\ell)$ applies here as well. If rule v-ρ or rule v-ϵ appeared last then the result follows by induction.

For the values $\texttt{null}$ and $\langle \tau_1, \ldots, \tau_n \rangle$, there are also two rules that apply. Consider the case for the value $\texttt{null}$ (the case for $\langle \tau_1, \ldots, \tau_n \rangle$ is similar). Once again, the base case (v-null) is trivial. For rule v-sub, we have:

$$\frac{\Delta; \Gamma \vdash \texttt{null} : \tau' \qquad \Delta \vdash \tau' \leq \tau}{\Delta; \Gamma \vdash \texttt{null} : \tau}$$

By induction, we know that τ' is either *null* or $?\langle \tau_1, \ldots, \tau_n \rangle$. By inspection of the equality and subtyping rules, we can deduce that the shapes of these types are preserved and therefore that τ is *null* or $?\langle \tau_1, \ldots, \tau_n' \rangle$.

$\square$

Lemma 22 *If* $\cdot;\cdot \vdash v : \tau$, $\cdot;\cdot \vdash v : \tau'$, *and* $\cdot;\cdot \vdash v : \tau''$; *and furthermore* $\cdot \vdash \tau' \leq \tau$, *and* $\cdot \vdash \tau'' \leq \tau$, *then either* $\cdot \vdash \tau' \leq \tau''$ *or* $\cdot \vdash \tau'' \leq \tau'$.

Proof:

The proof proceeds by cases given the possibilities in Lemma 19. If any two of the three types are equal the proof is complete by transitivity of subtyping. In the remaining cases τ must be an option type (let τ be $?\langle \tau_1, \ldots, \tau_n \rangle$), and τ' and τ'' must be one of *null* or $\langle \tau_1', \ldots, \tau_n' \rangle$.

By Lemma 20 and $\cdot;\cdot \vdash v : ?\langle \tau_1, \ldots, \tau_n \rangle$, $v = \texttt{null}$ or $v = \langle v_1, \ldots, v_n \rangle$.

- $v = \texttt{null}$
  By Lemma 21, τ' and τ'' are each either an option type or *null*. If both are *null* then the proof is complete. If one of them is an option type then it must be equal to τ by Lemma 19.
- $v = \langle v_1, \ldots, v_n \rangle$
  By Lemma 21, τ' and τ'' are each either a tuple type or an option type. If one of them is an option type the argument is the same as in the previous case. If both are tuple types then by Lemma 18, through their common ancestor τ, and the definition of equality on tuples, $\cdot \vdash \tau' = \tau''$.

$\square$

A.2 Properties of Constraints

Preservation relies on the aliasing constraints C remaining a faithful description of the store across operations such as `free` and `malloc`. To prove that this relationship is maintained throughout execution, we must reason about how aliasing constraints are related as types change.

To do so we define a notion of substitution which uniformly changes the type associated with a location (η) in a constraint (C) to a new type (τ). The constraint $[[C]]_{\eta\mapsto\tau}$ is C with every primitive constraint $\{\eta \mapsto \tau'\}^{\phi}$ replaced by $\{\eta \mapsto \tau\}^{\phi}$.

Definition 23 $[[C]]_{\eta\mapsto\tau}$ *is a constraint* C*, defined as follows.*

$$
\begin{array}{rcll}
[[\emptyset]]_{\eta\mapsto\tau} & = & \emptyset & \\
[[\epsilon]]_{\eta\mapsto\tau} & = & \epsilon & \\
[[\{\eta \mapsto \tau'\}^{\phi}]]_{\eta\mapsto\tau} & = & \{\eta \mapsto \tau\}^{\phi} & \quad \textit{(for } \phi = \cdot \textit{ or } \omega\textit{)} \\
[[\{\eta' \mapsto \tau'\}^{\phi}]]_{\eta\mapsto\tau} & = & \{\eta' \mapsto \tau'\}^{\phi} & \quad \textit{(if } \eta' \neq \eta\textit{)} \\
[[C_1 \oplus C_2]]_{\eta\mapsto\tau} & = & [[C_1]]_{\eta\mapsto\tau} \oplus [[C_2]]_{\eta\mapsto\tau} &
\end{array}
$$

The following definitions provide us with several useful abstractions we will use to prove facts about constraints. $\mathtt{Id}_\tau$ is the set of types equal to τ and $\mathtt{Sup}_\tau$ is the set of supertypes of τ. If $\{\eta \mapsto \tau\}^{\phi}$ appears in C then τ and all types equal to it are in the "image" of η; in other words, they are in $\mathtt{Im}(C, \eta)$. The set $\mathtt{S}_{\eta,T}(C)$ contains all the constraints generated by replacing occurences of $\{\eta \mapsto \tau\}^{\phi}$ with $\{\eta \mapsto \tau'\}$ where τ' is in the set of types T.

Definition 24 $\mathtt{Id}_\tau$ *and* $\mathtt{Sup}_\tau$ *are sets of types, defined as follows.*

$$
\begin{array}{lcl}
\mathtt{Id}_\tau & = & \{\tau' \mid \cdot \vdash \tau = \tau'\} \\
\mathtt{Sup}_\tau & = & \{\tau' \mid \cdot \vdash \tau \leq \tau'\}
\end{array}
$$

Definition 25 $\mathtt{Im}(C, \eta)$ *is a set of types, defined as follows.*

$$
\begin{array}{lcll}
\mathtt{Im}(\emptyset, \eta) & = & \emptyset & \\
\mathtt{Im}(\epsilon, \eta) & = & \emptyset & \\
\mathtt{Im}(\{\eta \mapsto \tau\}^{\phi}, \eta) & = & \mathtt{Id}_\tau & \\
\mathtt{Im}(\{\eta \mapsto \tau\}^{\phi}, \eta') & = & \emptyset & \quad \textit{(if } \eta \neq \eta'\textit{)} \\
\mathtt{Im}(C_1 \oplus C_2, \eta) & = & \mathtt{Im}(C_1, \eta) \cup \mathtt{Im}(C_2, \eta) &
\end{array}
$$

Definition 26 $\mathtt{S}_{\eta,T}(C)$ *is a set of constraints, defined as follows.*

$$
\begin{array}{lcll}
\mathtt{S}_{\eta,T}(\emptyset) & = & \emptyset & \\
\mathtt{S}_{\eta,T}(\epsilon) & = & \epsilon & \\
\mathtt{S}_{\eta,T}(\{\eta \mapsto \tau'\}^{\phi}) & = & \{\ \{\eta \mapsto \tau\}^{\phi} \mid \tau \in T\ \} & \\
\mathtt{S}_{\eta,T}(\{\eta' \mapsto \tau'\}^{\phi}) & = & \{\ \{\eta' \mapsto \tau'\}^{\phi}\ \} & \quad \textit{(if } \eta \neq \eta'\textit{)} \\
\mathtt{S}_{\eta,T}(C_1 \oplus C_2) & = & \{\ C_1' \oplus C_2' \mid C_1' \in \mathtt{S}_{\eta,T}(C_1) \textit{ and } C_2' \in \mathtt{S}_{\eta,T}(C_2)\ \} &
\end{array}
$$

The following lemmas specify properties of the interactions between $[[C]]_{\eta\mapsto\tau}$, $\mathtt{Im}(C,\eta)$ and $\mathtt{S}_{\eta,T}(C)$.

Lemma 27 *For any constraint C and type τ such that $\cdot\vdash\tau$,*

1. $C\in\mathtt{S}_{\eta,\mathtt{Id}_\tau}(C)$ *if and only if* $\mathtt{Im}(C,\eta)\subseteq\mathtt{Id}_\tau$.
2. $C\in\mathtt{S}_{\eta,\mathtt{Sup}_\tau}(C)$ *if and only if* $\mathtt{Im}(C,\eta)\subseteq\mathtt{Sup}_\tau$.

Proof: Both directions can be shown by induction on the structure of C.

Lemma 28 *For all locations η and constraints C_1 and C_2,*

1. *If* $\cdot\vdash C_1=C_2$ *then* $\mathtt{Im}(C_1,\eta)=\mathtt{Im}(C_2,\eta)$.
2. *If* $\cdot\vdash C_1\leq C_2$ *then for all* $\tau\in\mathtt{Im}(C_2,\eta)$ *there exists a type* $\tau'\in\mathtt{Im}(C_1,\eta)$ *such that* $\cdot\vdash\tau'\leq\tau$.

Proof:

Part 1 is by induction on the height of the equality derivation. Part 2 is by induction on the height of the subtyping derivation.

□

Lemma 29 *For all locations η, types τ, and constraints C_1 and C_2,*

1. *If* $\cdot\vdash C_1=C_2$ *then* $\cdot\vdash[[C_1]]_{\eta\mapsto\tau}=[[C_2]]_{\eta\mapsto\tau}$.
2. *If* $\cdot\vdash C_1\leq C_2$ *then* $\cdot\vdash[[C_1]]_{\eta\mapsto\tau}\leq[[C_2]]_{\eta\mapsto\tau}$.

Proof:

Part 1 is by induction on the height of the equality derivation. Part 2 is by induction on the height of the subtyping derivation.

□

Lemma 30 *For all locations η, types τ such that $\cdot\vdash\tau$, and constraints C, if $C_1\in\mathtt{S}_{\eta,\mathtt{Id}_\tau}(C)$ and $C_2\in\mathtt{S}_{\eta,\mathtt{Sup}_\tau}(C)$ then $\cdot\vdash C_1\leq C_2$.*

Proof:

By induction on the structure of C. The only non-trivial case is $C = \{\eta \mapsto \tau\}^\phi$. In this case, $C_1 = \{\eta \mapsto \tau'\}^\phi$ for some τ' such that (1) $\cdot \vdash \tau = \tau'$ and $C_2 = \{\eta \mapsto \tau''\}^\phi$ for some τ'' such that (2) $\cdot \vdash \tau \leq \tau''$. By rule C-sub, we have $\cdot \vdash C_1 \leq C_2$ since (1) and (2) imply that $\cdot \vdash \tau' \leq \tau''$.

□

Lemma 31 *For all constraints C_1 and C_2 such that $\cdot \vdash C_1 \leq C_2$, and all types τ such that $\cdot \vdash \tau$. If $C_1 \in \mathtt{S}_{\eta,\mathtt{Id}_\tau}(C_1)$ then $C_2 \in \mathtt{S}_{\eta,\mathtt{Sup}_\tau}(C_2)$.*

Proof:

By Lemma 28, for each $\tau'' \in \mathtt{Im}(C_2, \eta)$ there is a $\tau' \in \mathtt{Im}(C_1, \eta)$ such that (1) $\cdot \vdash \tau' \leq \tau''$. Now, we can use our assumption and Lemma 27, part 1, to conclude $\mathtt{Im}(C_1, \eta) = \mathtt{Id}_\tau$ and therefore that (2) $\cdot \vdash \tau = \tau'$. By (1) and (2), $\cdot \vdash \tau \leq \tau''$ and we conclude that $\mathtt{Im}(C_2, \eta) \subseteq \mathtt{Sup}_\tau$. Hence we have the result via Lemma 27, part 2.

□

The following "cardinality" lemmas will be used to conclude that for any C such that $\vdash S : C$ and for any ℓ occuring in a linear constraint in C, ℓ occurs exactly once on the left-hand side of any constraint in C.

Definition 32 *$|C|^\phi_\ell$ is an integer, defined as follows.*

$$
\begin{array}{rcll}
|\emptyset|^\phi_\ell & = & 0 & \\
|\epsilon|^\phi_\ell & = & 0 & \\
|\{\ell \mapsto \tau\}^\phi|^\phi_\ell & = & 1 & \\
|\{\ell' \mapsto \tau'\}^{\phi'}|^\phi_\ell & = & 0 & \quad \textit{(if } \ell' \neq \ell \textit{ or } \phi' \neq \phi\textit{)} \\
|C_1 \oplus C_2|^\phi_\ell & = & |C_1|^\phi_\ell + |C_2|^\phi_\ell &
\end{array}
$$

Lemma 33 *For any constraints C_1 and C_2, and any location ℓ,*

1. *If $\cdot \vdash C_1 = C_2$ then $|C_1|^\cdot_\ell = |C_2|^\cdot_\ell$.*
2. *If $\cdot \vdash C_1 \leq C_2$ then $|C_1|^\cdot_\ell \geq |C_2|^\cdot_\ell$.*

Proof:

Part 1 is proven by induction on the height of the derivation of $\cdot \vdash C_1 = C_2$. Part 2 is proven using Part 1 for the case C-sub-eq, and a similar induction argument.

□

Lemma 34 *For any constraints C_1 and C_2,*

1. *If $\cdot \vdash C_1 = C_2$ and $|C_1|_\ell^\omega = 0$ or $|C_1|_\ell^\omega = 0$ then $|C_1|_\ell^\omega = 0$ and $|C_2|_\ell^\omega = 0$.*
2. *If $\cdot \vdash C_1 \leq C_2$, $|C_1|_\ell^\cdot = |C_2|_\ell^\cdot$ and $|C_1|_\ell^\omega = 0$ then $|C_2|_\ell^\omega = 0$.*

Proof:

Both parts are proven by induction on the height of the derivation. Most cases are trivial. The exception is the case for rule C-sub-trans in part 2. In this case, we have:

$$\frac{\Delta \vdash C_1 \leq C_3 \qquad \Delta \vdash C_3 \leq C_2}{\Delta \vdash C_1 \leq C_2}$$

and the facts that (1) $|C_1|_\ell^\cdot = |C_2|_\ell^\cdot$ and $|C_1|_\ell^\omega = 0$. By Lemma 33, part 2, we know that $|C_1|_\ell^\cdot \geq |C_3|_\ell^\cdot \geq |C_2|_\ell^\cdot$. However, from (1), we can deduce that these cardinalities are in fact equal: $|C_1|_\ell^\cdot = |C_3|_\ell^\cdot = |C_2|_\ell^\cdot$. Therefore, we can use induction on the first sub-derivation to conclude that $|C_3|_\ell^\omega = 0$, and subsequently, by induction on the second sub-derivation, we have our result, $|C_2|_\ell^\omega = 0$.

□

A.3 Relating Stores and Constraints

Lemma 35 *If $\vdash S : C$ then $\cdot \vdash C$.*

Proof:

By induction on the height of the derivation $\vdash S : C$. Assume the final rule in the derivation is S-base. By inspection of the rule, we can see there are no free variables in C and therefore that $\cdot \vdash C$. If, on the other hand, we assume rule S-sub was the final rule in the derivation:

$$\frac{\vdash S : C' \qquad \cdot \vdash C' \leq C}{\vdash S : C}$$

then by induction, we have $\cdot \vdash C'$. Further, by Lemma 16, we can conclude $\cdot \vdash C$.

□

The following lemma proves to be very useful. Basically it says that any derivation of $\vdash S : C$ need have at most one application of S-base and one application of S-sub.

Lemma 36 *If $\vdash S : C$ then there is a C_{base} such that $\vdash S : C_{base}$ is derivable from the rule* S-base *alone, and $\cdot \vdash C_{base} \leq C$.*

Proof: By induction on the derivation of $\vdash S : C$, using transitivity of subtyping on constraints (C-sub-trans).

Lemma 37 *If $\vdash S\{\ell \mapsto v\} : C$ then there exists a τ such that $\cdot;\cdot \vdash v : \tau$.*

Proof: By induction on the typing derivation $\vdash S\{\ell \mapsto v\} : C$.

Lemma 38 *If $\ell \notin Dom(S)$, $\vdash S : C$ and $\cdot;\cdot \vdash v : \tau$ then $\vdash S\{\ell \mapsto v\} : C \oplus \{\ell \mapsto \tau\}$.*

Proof:

The proof is by induction on the derivation of $\vdash S : C$. Case S-base is true by definition. For case S-sub we assume:

$$\frac{\vdash S : C' \qquad \cdot \vdash C' \leq C}{\vdash S : C}$$

Therefore, by induction, we know that $\vdash S \oplus \{\ell \mapsto v\} : C' \oplus \{\ell \mapsto \tau\}$. Using the fact that $\cdot \vdash C' \leq C$ and the congruence rules for subtyping, we can conclude that $\cdot \vdash C' \oplus \{\ell \mapsto \tau\} \leq C \oplus \{\ell \mapsto \tau\}$ completing the proof.

□

The following lemma states simply that if two values have the same type, then we can type the store in the same way regardless of which value is in the store.

Lemma 39 *If $\vdash S\{\ell \mapsto v\} : C$ and $\cdot;\cdot \vdash v : \tau$ and $\cdot;\cdot \vdash v' : \tau$ then $\vdash S\{\ell \mapsto v'\} : C$.*

Proof: By induction on the typing derivation $\vdash S\{\ell \mapsto v\} : C$.

The following lemma formally encapsulates the central idea of the paper. The same location can be used to store two values of different types. We do not need to "stamp" a location with a single type. Moreover, the types of each store are identical except for modification at a single location.

Lemma 40 (Update) *If $\vdash S\{\ell \mapsto v\} : C \oplus \{\ell \mapsto \tau\}$ and $\cdot;\cdot \vdash v' : \tau'$ then $\vdash S\{\ell \mapsto v'\} : C \oplus \{\ell \mapsto \tau'\}$.*

Proof:

To simplify the presentation we separate the proof into two parts. First, we prove that (1) $\vdash S\{\ell \mapsto v'\} : [[C \oplus \{\ell \mapsto \tau\}]]_{\ell \mapsto \tau'}$. Second, we prove that (2) $[[C \oplus \{\ell \mapsto \tau\}]]_{\ell \mapsto \tau'} = C \oplus \{\ell \mapsto \tau'\}$ – equality here is syntactic equality. Together, these two facts imply the result.

By Lemma 36, there is a constraint, C_{base}, such that $\vdash S : C_{base}$ is derivable using rule S-base only and $\cdot \vdash C_{base} \leq C \oplus \{\ell \mapsto \tau\}$. Now, parts (1) and (2):

1. By rule S-base applied to $S\{\ell \mapsto v'\}$ we can conclude that $\vdash S\{\ell \mapsto v'\} : [[C_{base}]]_{\ell \mapsto \tau'}$. Next, by Lemma 29, $\cdot \vdash [[C_{base}]]_{\ell \mapsto \tau'} \leq [[C \oplus \{\ell \mapsto \tau\}]]_{\ell \mapsto \tau'}$. Finally, by rule S-sub we obtain that $\vdash S\{\ell \mapsto v'\} : [[C \oplus \{\ell \mapsto \tau\}]]_{\ell \mapsto \tau'}$.
2. Note that $|C_{base}|^{\omega}_{\ell} = 0$ and $|C_{base}|^{\cdot}_{\ell} = 1$ by the definition of S-base. Furthermore, by Lemma 33, $|C \oplus \{\ell \mapsto \tau\}|^{\cdot}_{\ell} \leq |C_{base}|^{\cdot}_{\ell} \leq 1$. But inspection of the definition of $|\cdot|^{\cdot}_{\ell}$ reveals that $|C \oplus \{\ell \mapsto \tau\}|^{\cdot}_{\ell} \geq 1$ and therefore $|C \oplus \{\ell \mapsto \tau\}|^{\cdot}_{\ell}$ is, in fact, equal to 1. Therefore, we can apply Lemma 34 and conclude that $|C \oplus \{\ell \mapsto \tau\}|^{\omega}_{\ell} = 0$. These two facts together means that there is exactly one occurence of ℓ in $C \oplus \{\ell \mapsto \tau\}$ so there must be no occurences in C. Therefore $[[C \oplus \{\ell \mapsto \tau\}]]_{\ell \mapsto \tau'} = C \oplus [[\{\ell \mapsto \tau\}]]_{\ell \mapsto \tau'} = C \oplus \{\ell \mapsto \tau'\}$.

□

The next lemma describes a property of the store that we require for the progress proof: If a location appears in a constraint then it also appears in the store and contains a value of an appropriate type.

Lemma 41 (Well-formed store) *If $\vdash S : C \oplus \{\ell \mapsto \tau\}^{\phi}$ for $\phi = w$ or $\cdot$, then $S = S'\{\ell \mapsto v\}$ and furthermore $\cdot;\cdot \vdash v : \tau$*

Proof:

By Lemma 36, there is a C_{base} derived using S-base alone, such that $\cdot \vdash C_{base} \leq C \oplus \{\ell \mapsto \tau\}^{\phi}$. By Lemma 28, there is a τ' in $\mathtt{Im}(C_{base}, \ell)$ such that (1) $\cdot \vdash \tau' \leq \tau$.

By inspection of the definition of $\mathtt{Im}(\cdot, \cdot)$, the constraint $\{\ell \mapsto \tau''\}$ is in C_{base} for some τ'' and (2) $\cdot \vdash \tau'' = \tau'$. By inspection of the rule S-base, there can only be 1 occurence of ℓ in $Dom(S)$ and there must exist a v such that $S = S\{\ell \mapsto v\}$. Moreover, $\cdot;\cdot \vdash v : \tau''$. By (1) and (2) and inspection of the rules for subtyping, we can conclude $\cdot \vdash \tau'' \leq \tau$. Consequently, by use of the rule v-sub, we have our result: $\cdot;\cdot \vdash v : \tau$.

□

Lemma 42 (Store-subtype) *If $\vdash S\{\ell \mapsto v\} : C \oplus \{\ell \mapsto \tau\}^{\phi}$ and $\cdot;\cdot \vdash v : \tau'$ where $\cdot \vdash \tau' \leq \tau$ then $\vdash S\{\ell \mapsto v\} : C \oplus \{\ell \mapsto \tau'\}^{\phi}$*

Proof:

In the following let

$$\begin{array}{ll} C' & \text{represent} \quad C \oplus \{\ell \mapsto \tau\}^{\phi} \\ C'' & \text{represent} \quad C \oplus \{\ell \mapsto \tau'\}^{\phi} \end{array}$$

By Lemma 36, there is a constraint C_{base} derived from S-base such that $\cdot \vdash C_{base} \leq C'$. By Lemma 28, there is a $\tau'' \in \mathtt{Im}(C_{base}, \ell)$ such that $\cdot \vdash \tau'' \leq \tau$. By inspection of the rule S-base, ℓ only occurs once in C_{base} and $\cdot;\cdot \vdash v : \tau''$. Therefore, (1) $\mathtt{Im}(C_{base}, \ell) = \mathtt{Id}_{\tau''}$.

Without loss of generality, assume that (2) $\{\ell \mapsto \tau''\}$ appears in C_{base} (rather than some other constraint $\{\ell \mapsto \tau'''\}$ where τ''' is equal to τ''). Now, because $\cdot;\cdot \vdash v : \tau''$, we can apply Lemma 22 to conclude that either $\cdot \vdash \tau'' \leq \tau'$ or $\cdot \vdash \tau' \leq \tau''$.

1. $\cdot \vdash \tau'' \leq \tau'$

   This case is proven in two parts. In the first part we show that $\cdot \vdash C_{base} \leq [[C'']]_{\ell \mapsto \tau''}$. In the second part we show that $\cdot \vdash [[C'']]_{\ell \mapsto \tau''} \leq C''$. From these two results and the rule S-sub, we can conclude that $\vdash S\{\ell \mapsto v\} : C''$. This is the desired result by the definition of C''.

   (a) Proof of $\cdot \vdash C_{base} \leq [[C'']]_{\ell \mapsto \tau''}$.
   By Lemma 29, $\cdot \vdash [[C_{base}]]_{\ell \mapsto \tau''} \leq [[C']]_{\ell \mapsto \tau''}$. By fact (2), $[[C_{base}]]_{\ell \mapsto \tau''}$ is syntactically equal to C_{base}. Therefore, $\cdot \vdash C_{base} \leq [[C']]_{\ell \mapsto \tau''}$.
   Using the definition of $[[C']]_{\ell \mapsto \tau''}$, we show that $[[C']]_{\ell \mapsto \tau''}$ is syntactically equal to $[[C'']]_{\ell \mapsto \tau''}$.

$$\begin{array}{rcl} [[C']]_{\ell \mapsto \tau''} & = & [[C]]_{\ell \mapsto \tau''} \oplus [[\{\ell \mapsto \tau\}^{\phi}]]_{\ell \mapsto \tau''} \\ & = & [[C]]_{\ell \mapsto \tau''} \oplus \{\ell \mapsto \tau''\}^{\phi} \\ & = & [[C]]_{\ell \mapsto \tau''} \oplus [[\{\ell \mapsto \tau'\}^{\phi}]]_{\ell \mapsto \tau''} \\ & = & [[C'']]_{\ell \mapsto \tau''} \end{array}$$

   From the above equalities and $\cdot \vdash C_{base} \leq [[C']]_{\ell \mapsto \tau''}$, we can conclude that $\cdot \vdash C_{base} \leq [[C'']]_{\ell \mapsto \tau''}$.

   (b) Proof of $\cdot \vdash [[C'']]_{\ell \mapsto \tau''} \leq C''$
   By Lemma 27, part 1, and fact (1), $C_{base} \in \mathtt{S}_{\ell, \mathtt{Id}_{\tau''}}(C_{base})$. Now using $\cdot \vdash C_{base} \leq C'$ once again, and Lemma 31, we have that (3) $C' \in \mathtt{S}_{\ell, \mathtt{Sup}_{\tau''}}(C')$.
   By the definition of C', fact (3) can be rewritten as

$$C \oplus \{\ell \mapsto \tau\}^{\phi} \in \mathtt{S}_{\ell, \mathtt{Sup}_{\tau''}}(C \oplus \{\ell \mapsto \tau\}^{\phi})$$

   By the definition of $\mathtt{S}_{\ell, \mathtt{Sup}_{\tau''}}(C')$, we can conclude that $C \in \mathtt{S}_{\ell, \mathtt{Sup}_{\tau''}}(C)$. Furthermore because $\{\ell \mapsto \tau'\}^{\phi} \in \mathtt{S}_{\ell, \mathtt{Sup}_{\tau''}}(\{\ell \mapsto \tau'\}^{\phi})$, it follows that

$$C'' \in \mathtt{S}_{\ell, \mathtt{Sup}_{\tau''}}(C'')$$

   Inspection of the definitions shows that (4) $[[C'']]_{\ell \mapsto \tau''} \in \mathtt{S}_{\ell, \mathtt{Id}_{\tau''}}(C'')$. Together facts (3) and (4) allow us to apply Lemma 30 to conclude that $\cdot \vdash [[C'']]_{\ell \mapsto \tau''} \leq C''$.

2. $\cdot \vdash \tau' \leq \tau''$

   By inspection of the rule S-base, we can conclude that $\vdash S\{\ell \mapsto v\} : C'_{base}$ where $C'_{base} = [[C_{base}]]_{\ell \mapsto \tau'}$. Since $\cdot \vdash \tau' \leq \tau''$, we can prove that $\cdot \vdash C'_{base} \leq C_{base}$ by use of the rules C-cong and C-sub. Therefore, by transitivity, $\cdot \vdash C'_{base} \leq C'$ and consequently, we have the derivation:

$$\frac{\vdash S : C'_{base} \qquad \cdot \vdash C'_{base} \leq C'}{\vdash S : C'}$$

   where the type $\{\ell \mapsto \tau'\}^{\phi}$ appears in C'_{base}. Obviously (by reflexivity), $\cdot \vdash \tau' \leq \tau'$. Hence, we can reason as in case 1 above and obtain the result: $\vdash S\{\ell \mapsto v\} : C \oplus \{\ell \mapsto \tau'\}^{\phi}$

□

A.4 Preservation and Progress

Lemma 43 (Preservation) *If* $\vdash S : C$, $\cdot; C; \cdot \vdash \iota$, *and* $(S, \iota) \longmapsto (S', \iota')$ *then there is a* C' *such that* $\vdash S' : C'$, *and* $\cdot; C'; \cdot \vdash \iota'$

Proof:

The proof proceeds by cases on the structure of ι. Each case begins by stating the operational rule (OR) that must have been used given the shape of ι. It also states the relevant parts of the typing derivation for the instruction (TD) specialized to the particular case and the store (STD) being considered.

To simplify the presentation, whenever judgements of the form $\cdot;\cdot \vdash \mathtt{ptr}(\ell) : \mathit{ptr}(\eta)$ occur in the typing derivation (TD), we have implicitly used Lemma 20 to conclude that $\eta = \ell$.

- $\mathtt{malloc}\, x, \rho, n; \iota$

  OR:

  $$(S, \mathtt{malloc}\, x, \rho, n; \iota) \quad \longmapsto \quad (S\{\ell \mapsto \langle \mathtt{junk}_1, \ldots, \mathtt{junk}_n \rangle\}, \iota[\ell/\rho][\mathtt{ptr}(\ell)/x]) \qquad \text{where } \ell \notin S$$

  TD:

  $$\frac{\rho; C \oplus \{\rho \mapsto \langle \mathit{junk}_1, \ldots, \mathit{junk}_n \rangle\}; x{:}\mathit{ptr}(\rho) \vdash \iota}{\cdot; C; \cdot \vdash \mathtt{malloc}\, x, \rho, n; \iota}$$

  STD:

  $$\vdash S : C$$

  First, we show the new store $S' = S\{\ell \mapsto \langle \mathtt{junk}_1, \ldots, \mathtt{junk}_n \rangle\}$ can be typed by $C' = C \oplus \{\ell \mapsto \langle \mathit{junk}_1, \ldots, \mathit{junk}_n \rangle\}$. Using typing rules v-tuple and v-junk, we can conclude

  $$\cdot;\cdot \vdash \langle \mathtt{junk}_1, \ldots, \mathtt{junk}_n \rangle : \langle \mathit{junk}_1, \ldots, \mathit{junk}_n \rangle \qquad (1)$$

  Therefore, since $\ell \notin S$, by Lemma 38, we know $\vdash S' : C'$.

  Second, we need to prove the new instruction sequence is well-formed. From TD and Lemma 13 (ρ-substitution) we can conclude:

  $$\cdot; C \oplus \{\ell \mapsto \langle \mathit{junk}_1, \ldots, \mathit{junk}_n \rangle\}; x : \mathit{ptr}(\ell) \vdash \iota[\ell/\rho]$$

  By (1) above, $\langle \mathtt{junk}_1, \ldots, \mathtt{junk}_n \rangle$ is well-formed. Therefore, we can apply our second substitution lemma (Lemma 15), and obtain the final result:

  $$\cdot; C \oplus \{\ell \mapsto \langle \mathit{junk}_1, \ldots, \mathit{junk}_n \rangle\}; \cdot \vdash \iota[\ell/\rho][\mathtt{ptr}(\ell)/x]$$

- $\mathtt{mknull}\, x, \rho; \iota$

  Similar to the case for `malloc`.

- $x{=}\mathtt{ptr}(\ell)[i]; \iota$

  OR:

  $$(S\{\ell \mapsto \langle v_1, \ldots, v_n \rangle\}, x{=}\mathtt{ptr}(\ell)[i]; \iota) \quad \longmapsto \quad (S\{\ell \mapsto \langle v_1, \ldots, v_n \rangle\}, \iota[v_i/x]) \qquad \text{if } 1 \le i \le n$$

  TD:

  $$\frac{\cdot;\cdot \vdash \mathtt{ptr}(\ell) : \mathit{ptr}(\ell) \qquad \cdot \vdash C = C' \oplus \{\ell \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^{\phi} \qquad \cdot; C; x{:}\tau_i \vdash \iota}{\cdot; C; \cdot \vdash x{=}\mathtt{ptr}(\ell)[i]; \iota} \ (\ 1 \le i \le n\ )$$

STD:

$$\vdash S : C$$

The resulting store is the same as the initial store, so the first requirement, $\vdash S : C$, is trivially satisfied.

As for the second requirement, using STD, the subtyping condition from TD and the store typing rule S-sub, we can conclude $\vdash S : C' \oplus \{\ell \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^\omega$. Consequently, by Lemma 41, $S = S'\{\ell \mapsto v\}$ and $\cdot; \cdot \vdash v : \langle \tau_1, \ldots, \tau_n \rangle$. By Canonical Forms, $v = \langle v_1, \ldots, v_n \rangle$ and also (1) $\cdot; \cdot \vdash v_i : \tau_i$.

Now, from TD

$$\cdot; C; x{:}\tau_i \vdash \iota$$

Using this fact and (1), we can apply Lemma 15 (Substitution) and obtain the required result:

$$\cdot; C; \cdot \vdash \iota[v_i/x]$$

- $\texttt{ptr}(\ell)[i]{:=}v; \iota$

  Let $v_{res} = \langle v_1, \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_n \rangle$ in the following.

  OR:

  $$(S\{\ell \mapsto \langle v_1, \ldots, v_n \rangle\}, \texttt{ptr}(\ell)[i]{:=}v; \iota) \quad \mapsto \quad (S\{\ell \mapsto v_{res}\}, \iota)$$

  STD:

  $$\vdash S\{\ell \mapsto \langle v_1, \ldots, v_n \rangle\} : C$$

  There are two possible typing derivations of this term: one for $\phi = \omega$, and one for $\phi = \cdot$. Hence we have split this case into two parts.

  - $\phi = \omega$

    TD:

    $$\frac{\begin{array}{cc}\cdot; \cdot \vdash \texttt{ptr}(\ell) : \mathit{ptr}(\ell) & \cdot; \cdot \vdash v : \tau_i \\ \cdot \vdash C = C' \oplus \{\ell \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^\omega & \cdot; C; \cdot \vdash \iota\end{array}}{\cdot; C; \cdot \vdash \texttt{ptr}(\ell)[i]{:=}v; \iota} \; (1 \leq i \leq n)$$

    First, we prove $\vdash S\{\ell \mapsto v_{res}\} : C$. By Lemma 37 there exists a type τ such that (1) $\cdot; \cdot \vdash \langle v_1, \ldots, v_n \rangle : \tau$. By Lemma 21, τ has the form $\langle \tau_1, \ldots, \tau_n \rangle$ and (2) $\cdot; \cdot \vdash v_j : \tau_j$ for $1 \leq j \leq n$. Now, by the typing judgement TD, we have $\cdot; \cdot \vdash v : \tau_i$. Therefore, using this fact and (2), we can conclude by inspection of the rule v-tuple that (3) $\cdot; \cdot \vdash v_{res} : \langle \tau_1, \ldots, \tau_n \rangle$. Finally, by (1) and (3) and STD, we can conclude that $\vdash S\{\ell \mapsto v_{res}\} : C$.

    Our second obligation, to prove $\cdot; C; \cdot \vdash \iota$, follows immediately from TD.

  - $\phi = \cdot$

    Let $\tau_{res} = \langle \tau_1, \ldots, \tau_{i-1}, \tau, \tau_{i+1}, \ldots, \tau_n \rangle$ in the following.

    TD:

    $$\frac{\begin{array}{c}\cdot; \cdot \vdash \texttt{ptr}(\ell) : \mathit{ptr}(\ell) \qquad \cdot; \cdot \vdash v : \tau \qquad \cdot \vdash C = C' \oplus \{\ell \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} \\ \cdot; C' \oplus \{\ell \mapsto \tau_{res}\}; \cdot \vdash \iota\end{array}}{\cdot; C; \cdot \vdash \texttt{ptr}(\ell)[i]{:=}v; \iota} \; (1 \leq i \leq n)$$

Our first proof obligation is to show that $\vdash S\{\ell \mapsto v_{res}\} : C' \oplus \{\ell \mapsto \tau_{res}\}$. Using STD, the equality $\cdot \vdash C = C' \oplus \{\ell \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}$ from TD, and the rules S-sub and C-sub-eq, we can prove $\vdash S\{\ell \mapsto \langle v_1, \ldots, v_n \rangle\} : C' \oplus \{\ell \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}$. Now, by Lemma 21 and v-tuple, we have that $\cdot; \cdot \vdash v_{res} : \tau_{res}$. Therefore we can apply Lemma 40 to conclude that $\vdash S\{\ell \mapsto v_{res}\} : C' \oplus \{\ell \mapsto \tau_{res}\}$.

The second proof obligation, $\cdot; C' \oplus \{\ell \mapsto \tau_{res}\}; \cdot \vdash \iota$, follows directly from TD.

- $\texttt{free}\,\texttt{ptr}(\ell); \iota'$

  Similar to assignment with $\phi = \cdot$.

- $\texttt{tosum}\, v, ?\langle \tau_1, \ldots, \tau_n \rangle; \iota'$

  OR:

$$(S, \texttt{tosum}\, v, ?\langle \tau_1, \ldots, \tau_n \rangle; \iota') \quad \mapsto \quad (S, \iota)$$

There are two possible typing rules that could have been used: either i-tosum1 or i-tosum2. Here we coalesce them into one rule where τ either has the shape *null* (corresponding to i-tosum1) or $\langle \tau_1, \ldots, \tau_n \rangle$ (corresponding to i-tosum2).

TD:

$$\frac{\begin{array}{cc} \cdot; \cdot \vdash v : \mathit{ptr}(\eta) & \cdot \vdash C = C' \oplus \{\eta \mapsto \tau\}^{\phi} \\ \cdot \vdash ?\langle \tau_1, \ldots, \tau_n \rangle & \cdot; C' \oplus \{\eta \mapsto ?\langle \tau_1, \ldots, \tau_n \rangle\}^{\phi}; \cdot \vdash \iota \end{array}}{\cdot; C; \cdot \vdash \texttt{tosum}\, v, ?\langle \tau_1, \ldots, \tau_n \rangle; \iota'}$$

STD:

$$\vdash S : C$$

From the judgement $\cdot; \cdot \vdash v : \mathit{ptr}(\eta)$, we can conclude there exists an ℓ such that $\eta = \ell$ and $v = \texttt{ptr}(\ell)$ using Lemma 17 and Lemma 21.

Given these facts, we will proceed to prove our first obligation: $\vdash S : C' \oplus \{\ell \mapsto ?\langle \tau_1, \ldots, \tau_n \rangle\}$. By STD, we have (1) $\vdash S : C$. By TD, we have (2) $\cdot \vdash C = C' \oplus \{\ell \mapsto \tau\}^{\phi}$. From (2) and rule C-sub-eq, we have (3) $\cdot \vdash C \leq C' \oplus \{\ell \mapsto \tau\}^{\phi}$. Because $\cdot \vdash \tau \leq ?\langle \tau_1, \ldots, \tau_n \rangle$ (recall that τ is either *null* or $\langle \tau_1, \ldots, \tau_n \rangle$), and through the use of rules C-sub-cong and C-sub, we can conclude (4) $\cdot \vdash C' \oplus \{\ell \mapsto \tau\}^{\phi} \leq C' \oplus \{\ell \mapsto ?\langle \tau_1, \ldots, \tau_n \rangle\}^{\phi}$. By C-sub-trans and (4), $\cdot \vdash C \leq C' \oplus \{\ell \mapsto ?\langle \tau_1, \ldots, \tau_n \rangle\}^{\phi}$.

The second proof obligation follows directly from TD: $\cdot; C' \oplus \{\ell \mapsto ?\langle \tau_1, \ldots, \tau_n \rangle\}^{\phi}; \cdot \vdash \iota$

- $\texttt{ifnull}\,\texttt{ptr}(\ell)\,\texttt{then}\,\iota_1\,\texttt{else}\,\iota_2$

  There are two operational rules that may have been applied, o-ifnull1 or o-ifnull2. Here, we show preservation when the first branch of the if is taken. The other alternative is similar.

  OR:

$$(S\{\ell \mapsto \texttt{null}\}, \texttt{ifnull}\,\texttt{ptr}(\ell)\,\texttt{then}\,\iota_1\,\texttt{else}\,\iota_2) \longmapsto (S\{\ell \mapsto \texttt{null}\}, \iota_1)$$

The constraints that appear in the typing judgement may either be linear constraints or non-linear constraints. The case for linear constraints follows using an argument similar to the arguments for rules i-a1 and i-free. Thus, we assume that the constraints in the judgement are non-linear constraints:

TD:

$$\frac{\begin{array}{cc} \cdot; \cdot \vdash \texttt{ptr}(\ell) : \mathit{ptr}(\ell) & \cdot \vdash C = C' \oplus \{\ell \mapsto ?\langle \tau_1, \ldots, \tau_n \rangle\}^{\omega} \\ \cdot; C' \oplus \{\ell \mapsto \mathit{null}\}^{\omega}; \cdot \vdash \iota_1 & \cdot; C' \oplus \{\ell \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^{\omega}; \cdot \vdash \iota_2 \end{array}}{\cdot; C; \cdot \vdash \texttt{ifnull}\,\texttt{ptr}(\ell)\,\texttt{then}\,\iota_1\,\texttt{else}\,\iota_2}$$

STD:

$$\vdash S\{\ell \mapsto \texttt{null}\} : C$$

Now, our first obligation is to prove that $\vdash S\{\ell \mapsto \texttt{null}\} : C' \oplus \{\ell \mapsto \mathit{null}\}^\omega$. From TD and STD, using rules S-sub and C-sub-eq, we can conclude that $\vdash S\{\ell \mapsto \texttt{null}\} : C' \oplus \{\ell \mapsto ?\langle\tau_1, \ldots, \tau_n\rangle\}^\omega$. From Lemma 42 we conclude that $\vdash S\{\ell \mapsto \texttt{null}\} : C' \oplus \{\ell \mapsto \mathit{null}\}^\omega$.

The second obligation is to show that $\cdot; C' \oplus \{\ell \mapsto \mathit{null}\}^\omega; \cdot \vdash \iota_1$, which is immediate from the typing derivation TD.

- $v(v_1, \ldots, v_n)$

  OR:

$$(S, v(v_1, \ldots, v_n)) \mapsto (S, \iota[c_1 \ldots, c_m/\beta_1, \ldots, \beta_m][v', v_1, \ldots, v_n/f, x_1, \ldots, x_n]$$

  where $v = v'[c_1, \ldots, c_m]$ and $v' = \texttt{fix}\, f[\beta_1, \ldots, \beta_m; C''; x_1{:}\tau_1, \ldots, x_n : \tau_n].\iota$

  TD:

$$\frac{\begin{array}{c}\cdot;\cdot \vdash v : \forall[\cdot; C''].(\tau_1, \ldots, \tau_n)\rightarrow 0 \qquad \cdot \vdash C \leq C' \\ \cdot;\cdot \vdash v_1 : \tau_1 \qquad \cdots \qquad \cdot;\cdot \vdash v_n : \tau_n\end{array}}{\cdot; C; \cdot \vdash v(v_1, \ldots, v_n)}$$

  STD:

$$\vdash S : C$$

  The first proof obligation is to show that $\vdash S : C'$. This is trivial using STD, the typing derivation above which states that $\cdot \vdash C \leq C'$ and the rule S-sub.

  The second proof obligation is to show that the function body type checks:

$$\cdot; C'; \cdot \vdash \iota[c_1 \ldots, c_m/\beta_1, \ldots, \beta_m][v', v_1, \ldots, v_n/f, x_1, \ldots, x_n]$$

  By Lemma 21, v' has an appropriate function type and its body type-checks under the assumptions in its precondition. By induction on the number of type applications and repeated use of the ρ-, ϵ- and x-substitution lemmas, the body type checks under C' and we have our result.

□

Lemma 44 (Progress) *If* $\vdash S : C$ *and* $\cdot; C; \cdot \vdash \iota$ *then either* $\iota = \texttt{halt}$ *or* $(S, \iota) \longmapsto (S', \iota')$ *for some* (S', ι').

Proof:

The proof is by a case analysis on the shape of ι. The Well-formed Stores Lemma (Lemma 41) in conjunction with the Canonical Forms Lemmas (Lemma 20) ensure that the store allows the triggered rule to evaluate. (In what follows we write CF for the Canonical Forms Lemma and WFS for the Well-formed Stores Lemma)

- $\texttt{malloc}\,x, \rho, n; \iota$
  o-malloc always applies.
- $x{=}v[i]; \iota$
  By static semantics rule i-let,

$$\cdot;\cdot \vdash v : \mathit{ptr}(\eta) \qquad \cdot \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^{\phi}$$

  By Lemma 17, there are no free location variables and therefore $\eta = \ell$ for some ℓ. By CF, $v = \texttt{ptr}(\ell)$. By (S-sub), $\vdash S : C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^{\phi}$. By WFS, $S = S'\{\ell \mapsto v'\}$ for some S' and $\cdot;\cdot \vdash v' : \langle \tau_1, \ldots, \tau_n \rangle$. By CF, $v' = \langle v_1, \ldots, v_n \rangle$. Therefore, operational rule o-let applies.
- $v[i]{:=}v'; \iota$
  This instruction must have been type-checked under either i-a1 or i-a2. In either case we have that:

$$\cdot;\cdot \vdash v : \mathit{ptr}(\eta) \qquad \cdot \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^{\phi}$$

  In exactly the same way as in the previous case, we conclude that $v = \texttt{ptr}(\ell)$ and $S = S'\{\ell \mapsto \langle v_1, \ldots, v_n \rangle\}$. Therefore o-a applies.
- $\texttt{free}\,v; \iota$
  By the rule i-free,

$$\cdot;\cdot \vdash v : \mathit{ptr}(\eta) \qquad \cdot \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}$$

  As in the previous cases, we conclude $\eta = \ell$, $v = \texttt{ptr}(\ell)$, and $S = S'\{\ell \mapsto v'\}$. Consequently rule o-free applies.
- $v(v_1, \ldots, v_n)$
  Although this is the most complicated instruction, this step is easy. From i-app we have:

$$\begin{array}{c} \cdot;\cdot \vdash v : \forall[\cdot; C'].(\tau_1, \ldots, \tau_n) \rightarrow 0 \qquad \cdot \vdash C \leq C' \\ \cdot;\cdot \vdash v_1 : \tau_1 \qquad \cdots \qquad \cdot;\cdot \vdash v_n : \tau_n \end{array}$$

  By CF, $v = (\texttt{fix}\,f[\Phi].\iota)[c_1, \ldots, c_n]$. This suffices to apply o-app.
- $\texttt{halt}$
  The result holds trivially.
- $\texttt{mknull}\,x, \rho; \iota$
  o-mknull always applies.
- $\texttt{tosum}\,v, ?\langle \tau_1, \ldots, \tau_n \rangle$
  o-tosum always applies.
- $\texttt{ifnull}\,v\,\texttt{then}\,\iota_1\,\texttt{else}\,\iota_2$
  By i-ifnull,

$$\cdot;\cdot \vdash v : \mathit{ptr}(\eta) \qquad \cdot \vdash C = C' \oplus \{\eta \mapsto ?\langle \tau_1, \ldots, \tau_n \rangle\}^{\phi}$$

  As before, we may conclude $\eta = \ell$. By WFS, $S = S'\{\ell \mapsto v'\}$ and $\cdot;\cdot \vdash v' : ?\langle \tau_1, \ldots, \tau_n \rangle$. By CF, $v = \texttt{ptr}(\ell)$ and either $v' = \texttt{null}$ or $v' = \langle \tau_1, \ldots, \tau_n \rangle$.
  In the first case o-ifnull1 applies and in the second case o-ifnull2 applies.

□

B Complete rules

This appendix contains the complete description of the language of locations.

B.1 Syntax

$$\begin{array}{llcl}
 & \rho & \in & \text{LabelVar} \\
 & \epsilon & \in & \text{CapVar} \\
 & b & \in & \text{BaseType} \\
 & \ell & \in & \text{Locations} \\
 & & & \\
\textit{locations} & \eta & ::= & \ell \mid \rho \\
\textit{constraints} & C & ::= & \emptyset \mid \epsilon \mid \{\eta \mapsto \tau\} \mid \{\eta \mapsto \tau\}^{\omega} \mid C_1 \oplus C_2 \\
\textit{types} & \tau & ::= & \mathit{int} \mid \mathit{junk} \mid \mathit{ptr}(\eta) \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \forall[\Delta; C].(\tau_1, \ldots, \tau_n) \rightarrow 0 \mid \\
 & & & ?\langle \tau_1, \ldots, \tau_n \rangle \mid \mathit{null} \\
\textit{value contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\textit{constructor contexts} & \Delta & ::= & \cdot \mid \Delta, \rho \mid \Delta, \epsilon \\
 & & & \\
\textit{values} & v & ::= & x \mid i \mid \texttt{junk} \mid \texttt{ptr}(\ell) \mid \langle v_1, \ldots, v_n \rangle \mid \texttt{fix}\, f[\Phi].\iota \mid v[\eta] \mid v[C] \mid \texttt{null} \\
\textit{instructions} & \iota & ::= & \texttt{malloc}\, x, \rho, n; \iota \mid x{=}v[i]; \iota \mid v[i]{:=}v'; \iota \mid \texttt{free}\, v; \iota \mid v(v_1, \ldots, v_n) \mid \\
 & & & \texttt{halt} \mid \texttt{mknull}\, x, \rho; \iota \mid \texttt{tosum}\, v, ?\langle \tau_1, \ldots, \tau_n \rangle \mid \\
 & & & \texttt{ifnull}\, v\, \texttt{then}\, \iota_1\, \texttt{else}\, \iota_2 \\
\textit{stores} & S & ::= & \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \\
\textit{programs} & P & ::= & (S, \iota)
\end{array}$$

B.2 Static Semantics

$\boxed{\Delta \vdash \tau \quad \Delta \vdash \eta \quad \Delta \vdash C}$

$$\text{(wf-type)}\ \frac{\mathrm{FV}(\tau) \subseteq \Delta}{\Delta \vdash \tau} \qquad \text{(wf-loc)}\ \frac{\mathrm{FV}(\eta) \subseteq \Delta}{\Delta \vdash \eta} \qquad \text{(wf-con)}\ \frac{\mathrm{FV}(C) \subseteq \Delta}{\Delta \vdash C}$$

$\boxed{\Delta \vdash \tau_1 = \tau_2 \quad \Delta \vdash \eta_1 = \eta_2}$

Standard equality up to alpha-conversion of bound variables modulo equality on constraints.

Rules omitted.

$\boxed{\Delta \vdash \tau \leq \tau'}$

$$(\tau\text{-sub-null})\ \frac{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle}{\Delta \vdash \mathit{null} \leq ?\langle \tau_1, \ldots, \tau_n \rangle} \qquad (\tau\text{-sub-tuple})\ \frac{}{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle \leq ?\langle \tau_1, \ldots, \tau_n \rangle}$$

$$(\tau\text{-sub-eq})\ \frac{\Delta \vdash \tau = \tau'}{\Delta \vdash \tau \leq \tau'} \qquad (\tau\text{-sub-trans})\ \frac{\Delta \vdash \tau \leq \tau'' \qquad \Delta \vdash \tau'' \leq \tau'}{\Delta \vdash \tau \leq \tau'}$$

$\boxed{\Delta \vdash C_1 = C_2}$

$$(\text{C-reflex})\ \frac{}{\Delta \vdash C = C} \qquad (\text{C-trans})\ \frac{\Delta \vdash C = C' \qquad \Delta \vdash C' = C''}{\Delta \vdash C = C''} \qquad (\text{C-symm})\ \frac{\Delta \vdash C' = C}{\Delta \vdash C = C'}$$

$$(\text{C-}\emptyset)\ \frac{}{\Delta \vdash C \oplus \emptyset = C} \qquad (\text{C-comm})\ \frac{}{\Delta \vdash C_1 \oplus C_2 = C_2 \oplus C_1}$$

$$(\text{C-assoc})\ \frac{}{\Delta \vdash C_1 \oplus (C_2 \oplus C_3) = (C_1 \oplus C_2) \oplus C_3} \qquad (\text{C-}\omega)\ \frac{}{\Delta \vdash \{\eta \mapsto \tau\}^\omega = \{\eta \mapsto \tau\}^\omega \oplus \{\eta \mapsto \tau\}^\omega}$$

$$(\text{C-atom})\ \frac{\Delta \vdash \tau = \tau'}{\Delta \vdash \{\eta \mapsto \tau\}^\phi = \{\eta \mapsto \tau'\}^\phi}\ (\phi = \omega \text{ or } \cdot) \qquad (\text{C-cong})\ \frac{\Delta \vdash C_1 = C_1' \qquad \Delta \vdash C_2 = C_2'}{\Delta \vdash C_1 \oplus C_2 = C_1' \oplus C_2'}$$

$\boxed{\Delta \vdash C_1 \leq C_2}$

$$(\text{C-sub-eq})\ \frac{\Delta \vdash C = C'}{\Delta \vdash C \leq C'} \qquad (\text{C-sub-}\omega)\ \frac{}{\Delta \vdash \{\eta \mapsto \tau\} \leq \{\eta \mapsto \tau\}^\omega}$$

$$(\text{C-sub-forget})\ \frac{}{\Delta \vdash C \leq \emptyset}$$

$$(\text{C-sub-trans})\ \frac{\Delta \vdash C \leq C' \qquad \Delta \vdash C' \leq C''}{\Delta \vdash C \leq C''} \qquad (\text{C-sub-cong})\ \frac{\Delta \vdash C_1 \leq C_1' \qquad \Delta \vdash C_2 \leq C_2'}{\Delta \vdash C_1 \oplus C_2 \leq C_1' \oplus C_2'}$$

$$(\text{C-sub})\ \frac{\Delta \vdash \tau \leq \tau'}{\Delta \vdash \{\eta \mapsto \tau\}^\phi \leq \{\eta \mapsto \tau'\}^\phi}$$

$\boxed{\Delta; \Gamma \vdash v : \tau}$

$$(\text{v-int})\ \frac{}{\Delta; \Gamma \vdash i : \mathit{int}} \qquad (\text{v-junk})\ \frac{}{\Delta; \Gamma \vdash \texttt{junk} : \mathit{junk}} \qquad (\text{v-null})\ \frac{}{\Delta; \Gamma \vdash \texttt{null} : \mathit{null}}$$

$$(\text{v-var})\ \frac{}{\Delta; \Gamma \vdash x : \Gamma(x)} \qquad (\text{v-}\ell)\ \frac{}{\Delta; \Gamma \vdash \texttt{ptr}(\ell) : \mathit{ptr}(\ell)}$$

$$\text{(v-tuple)}\ \frac{\Delta;\Gamma \vdash v_1 : \tau_1 \qquad \cdots \qquad \Delta;\Gamma \vdash v_n : \tau_n}{\Delta;\Gamma \vdash \langle v_1,\ldots,v_n\rangle : \langle \tau_1,\ldots,\tau_n\rangle}$$

$$\text{(v-fix)}\ \frac{\begin{array}{c}\Delta \vdash \forall[\Delta';C'].(\tau_1,\ldots,\tau_n)\to 0 \\ \Delta,\Delta';C';\Gamma,f{:}\forall[\Delta';C'].(\tau_1,\ldots,\tau_n)\to 0, x_1{:}\tau_1,\ldots,x_n{:}\tau_n \vdash \iota\end{array}}{\Delta;\Gamma \vdash \mathtt{fix}\, f[\Delta';C';x_1{:}\tau_1,\ldots,x_n{:}\tau_n].\iota : \forall[\Delta';C'].(\tau_1,\ldots,\tau_n)\to 0}\ (f,x_1,\ldots,x_n \notin \Gamma)$$

$$\text{(v-}\rho\text{)}\ \frac{\Delta \vdash \eta \qquad \Delta;\Gamma \vdash v : \forall[\rho,\Delta';C'].(\tau_1,\ldots,\tau_n)\to 0}{\Delta;\Gamma \vdash v[\eta] : \forall[\Delta';C'].(\tau_1,\ldots,\tau_n)\to 0[\eta/\rho]}$$

$$\text{(v-}\epsilon\text{)}\ \frac{\Delta \vdash C' \qquad \Delta;\Gamma \vdash v : \forall[\epsilon,\Delta;C''].(\tau_1,\ldots,\tau_n)\to 0}{\Delta;\Gamma \vdash v[C'] : \forall[\Delta;C''].(\tau_1,\ldots,\tau_n)\to 0[C'/\epsilon]}$$

$$\text{(v-sub)}\ \frac{\Delta;\Gamma \vdash v : \tau' \qquad \Delta \vdash \tau' \le \tau}{\Delta;\Gamma \vdash v : \tau}$$

$\boxed{\Phi \vdash \iota}$

$$\text{(i-malloc)}\ \frac{\Delta,\rho;C \oplus \{\rho \mapsto \langle \mathtt{junk}_1,\ldots,\mathtt{junk}_n\rangle\};\Gamma,x{:}ptr(\rho) \vdash \iota}{\Delta;C;\Gamma \vdash \mathtt{malloc}\, x,\rho,n;\iota}\ (x \notin \Gamma, \rho \notin \Delta)$$

$$\text{(i-free)}\ \frac{\Delta;\Gamma \vdash v : ptr(\eta) \qquad \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1,\ldots,\tau_n\rangle\} \qquad \Delta;C' \oplus \{\eta \mapsto junk\};\Gamma \vdash \iota}{\Delta;C;\Gamma \vdash \mathtt{free}\, v;\iota}$$

$$\text{(i-a1)}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash v : ptr(\eta) \qquad \Delta;\Gamma \vdash v' : \tau' \qquad \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1,\ldots,\tau_n\rangle\} \\ \Delta;C' \oplus \{\eta \mapsto \langle \tau_1,\ldots,\tau_{i-1},\tau',\tau_{i+1},\ldots,\tau_n\rangle\};\Gamma \vdash \iota\end{array}}{\Delta;C;\Gamma \vdash v[i]{:=}v';\iota}\ (1 \le i \le n)$$

$$\text{(i-a2)}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash v : ptr(\eta) \qquad \Delta;\Gamma \vdash v' : \tau_i \\ \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1,\ldots,\tau_n\rangle\}^{\omega} \qquad \Delta;C;\Gamma \vdash \iota\end{array}}{\Delta;C;\Gamma \vdash v[i]{:=}v';\iota}\ (1 \le i \le n)$$

$$\text{(i-let)}\ \frac{\Delta;\Gamma \vdash v : ptr(\eta') \qquad \Delta \vdash C = C' \oplus \{\eta' \mapsto \langle \tau_1,\ldots,\tau_n\rangle\}^{\phi} \qquad \Delta;C;\Gamma,x{:}\tau_i \vdash \iota}{\Delta;C;\Gamma \vdash x{=}v[i];\iota}\ \begin{pmatrix} x \notin \Gamma \\ 1 \le i \le n\end{pmatrix}$$

$$\text{(i-app)}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash v : \forall[\cdot;C'].(\tau_1,\ldots,\tau_n)\to 0 \qquad \Delta \vdash C \le C' \\ \Delta;\Gamma \vdash v_1 : \tau_1 \qquad \cdots \qquad \Delta;\Gamma \vdash v_n : \tau_n\end{array}}{\Delta;C;\Gamma \vdash v(v_1,\ldots,v_n)}$$

$$\text{(i-mknull)}\ \frac{\Delta,\rho;C \oplus \{\rho \mapsto null\};\Gamma,x{:}ptr(\rho) \vdash \iota}{\Delta;C;\Gamma \vdash \mathtt{mknull}\, x,\rho;\iota}\ (x \notin \Gamma, \rho \notin \Delta)$$

$$\text{(i-tosum1)}\ \frac{\Delta;\Gamma \vdash v : \mathit{ptr}(\eta) \qquad \Delta \vdash C = C' \oplus \{\eta \mapsto \mathit{null}\}^{\phi} \qquad \Delta \vdash ?\langle\tau_1,\ldots,\tau_n\rangle \qquad \Delta; C' \oplus \{\eta \mapsto ?\langle\tau_1,\ldots,\tau_n\rangle\}^{\phi};\Gamma \vdash \iota}{\Delta; C;\Gamma \vdash \mathtt{tosum}\, v, ?\langle\tau_1,\ldots,\tau_n\rangle; \iota}$$

$$\text{(i-tosum2)}\ \frac{\Delta;\Gamma \vdash v : \mathit{ptr}(\eta) \qquad \Delta \vdash C = C' \oplus \{\eta \mapsto \langle\tau_1,\ldots,\tau_n\rangle\}^{\phi} \qquad \Delta; C' \oplus \{\eta \mapsto ?\langle\tau_1,\ldots,\tau_n\rangle\}^{\phi};\Gamma \vdash \iota}{\Delta; C;\Gamma \vdash \mathtt{tosum}\, v, ?\langle\tau_1,\ldots,\tau_n\rangle; \iota}$$

$$\text{(i-ifnull)}\ \frac{\Delta;\Gamma \vdash v : \mathit{ptr}(\eta) \qquad \Delta \vdash C = C' \oplus \{\eta \mapsto ?\langle\tau_1,\ldots,\tau_n\rangle\}^{\phi} \qquad \Delta; C' \oplus \{\eta \mapsto \mathit{null}\}^{\phi};\Gamma \vdash \iota_1 \qquad \Delta; C' \oplus \{\eta \mapsto \langle\tau_1,\ldots,\tau_n\rangle\}^{\phi};\Gamma \vdash \iota_2}{\Delta; C;\Gamma \vdash \mathtt{ifnull}\, v\ \mathtt{then}\ \iota_1\ \mathtt{else}\ \iota_2}$$

$\boxed{\vdash S : C}$

$$\text{(S-sub)}\ \frac{\vdash S : C' \qquad \vdash C' \leq C}{\vdash S : C}$$

$$\text{(S-base)}\ \frac{\cdot;\cdot \vdash v_1 : \tau_1 \qquad \cdots \qquad \cdot;\cdot \vdash v_n : \tau_n}{\vdash \{\ell_1 \mapsto v_1,\ldots,\ell_n \mapsto v_n\} : \{\ell_1 \mapsto \tau_1\} \oplus \ldots \oplus \{\ell_n \mapsto \tau_n\}}\ (\text{for } 1 \leq i \leq n, \ell_i \text{ distinct})$$

B.3 Operational Semantics

(o-malloc)	$(S, \mathtt{malloc}\, x, \rho, n; \iota)$	$\longmapsto$	$(S\{\ell \mapsto \langle \mathtt{junk}_1,\ldots,\mathtt{junk}_n\rangle\}, \iota')$
	where $\ell \notin S$		
	and $\iota' = \iota[\ell/\rho][\mathtt{ptr}(\ell)/x]$		
(o-free)	$(S\{\ell \mapsto \langle v_1,\ldots,v_n\rangle\}, \mathtt{free}\, \mathtt{ptr}(\ell); \iota)$	$\longmapsto$	$(S\{\ell \mapsto \mathtt{junk}\}, \iota)$
(o-a)	$(S\{\ell \mapsto \langle v_1,\ldots,v_n\rangle\}, \mathtt{ptr}(\ell)[i]{:=}v; \iota)$	$\longmapsto$	$(S\{\ell \mapsto \langle v_1,\ldots,v_{i-1},v,v_{i+1},\ldots,v_n\rangle\}, \iota)$
	if $1 \leq i \leq n$		
(o-let)	$(S\{\ell \mapsto \langle v_1,\ldots,v_n\rangle\}, x{=}\mathtt{ptr}(\ell)[i]; \iota)$	$\longmapsto$	$(S\{\ell \mapsto \langle v_1,\ldots,v_n\rangle\}, \iota[v_i/x])$
	if $1 \leq i \leq n$		
(o-app)	$(S, v(v_1,\ldots,v_n))$	$\longmapsto$	(S, ι')
	if $v = v'[c_1,\ldots,c_m]$		
	and $v' = \mathtt{fix}\, f[\Delta; C; x_1{:}\tau_1,\ldots,x_n{:}\tau_n].\iota$		
	and $\iota' = \iota[c_1,\ldots,c_m/\beta_1,\ldots,\beta_m][v',v_1,\ldots,v_n/f,x_1,\ldots,x_n]$		
	and $\mathit{Dom}(\Delta) = \beta_1,\ldots,\beta_m$ (where β ranges over ρ and ϵ)		
(o-mknull)	$(S, \mathtt{mknull}\, x, \rho; \iota)$	$\longmapsto$	$(S\{\ell \mapsto \mathtt{null}\}, \iota[\ell/\rho][\mathtt{ptr}(\ell)/x])$
	where $\ell \notin S$		
(o-tosum)	$(S, \mathtt{tosum}\, v, ?\langle\tau_1,\ldots,\tau_n\rangle; \iota)$	$\longmapsto$	(S, ι)
(o-ifnull1)	$(S\{\ell \mapsto \mathtt{null}\},$ $\mathtt{ifnull}\ \mathtt{ptr}(\ell)\ \mathtt{then}\ \iota_1\ \mathtt{else}\ \iota_2)$	$\longmapsto$	$(S\{\ell \mapsto \mathtt{null}\}, \iota_1)$
(o-ifnull2)	$(S\{\ell \mapsto \langle v_1,\ldots,v_n\rangle\},$ $\mathtt{ifnull}\ \mathtt{ptr}(\ell)\ \mathtt{then}\ \iota_1\ \mathtt{else}\ \iota_2)$	$\longmapsto$	$(S\{\ell \mapsto \langle v_1,\ldots,v_n\rangle\}, \iota_2)$