Alias Types *

Frederick Smith    David Walker    Greg Morrisett
Cornell University

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 [26, 25] 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: \( \text{let } x = (1, 2) \text{ in } \text{let } y = \text{snd}(x) \text{ in } \text{let } z = \text{snd}(x) \text{ in } y + z \). Therefore, \( x \) cannot be given a linear type, and consequently, cannot be deallocated early.

Several authors [26, 9, 3] have extended pure linear type systems to allow greater flexibility. However, most of these efforts have focused on high-level user programming languages and as a result, they have emphasized simple typing rules that programmers can understand and/or typing rules that admit effective type inference techniques. These issues are less important for low-level typed languages designed as compiler intermediate languages [22, 18] or as secure mobile code platforms, such as the Java Virtual Machine [10], Proof-Carrying Code (PCC) [13] or Typed Assembly Language (TAL) [12]. These languages are designed for machine, not human, consumption. On the other hand, because systems such as PCC and TAL make every machine operation explicit and verify that each is safe, the implementation of these systems requires new type-theoretic mechanisms to make efficient use of computer resources.

* 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.
In existing high-level typed 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 [23,27]). In low-level languages that aim to expose the resources 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 \(\langle \text{int}, \text{int} \rangle\) may be created as follows:

\[
\text{malloc } x, 2; \quad (\ast \ x \ \text{has type } \langle \text{junk, junk} \rangle \ast) \\
x[1]:=1; \quad (\ast \ x \ \text{has type } \langle \text{int, junk} \rangle \ast) \\
x[2]:=2; \quad (\ast \ x \ \text{has type } \langle \text{int, int} \rangle \ast)
\]

At each step in this computation, the storage bound to \(x\) takes on a different type ranging from nonsense (indicated by the type \text{junk}) to a fully initialized pair of integers. In this simple example, there are no aliases of the pair and therefore we might be able to use linear types to verify that the code is safe. However, in a more complex example, a compiler might generate code to compute the initial values of the tuple fields between allocation and the initializing assignments. During the computation, a register allocator may be forced to move the uninitialized or partially initialized value \(x\) between stack slots and registers, creating aliases:

If \(x\) is a linear value, one of the pointers shown above would have to be “invalidated” in some way after each move. Unfortunately, assuming the pointer on the stack is invalidated, future register pressure may force \(x\) to be physically copied back onto the stack. Although this additional copy is unnecessary because the register allocator can easily remember that a pointer to the data structure remains on the stack, the limitations of a pure linear type system require it.

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, therefore, should support values whose types change even when those values are aliased.
We have devised a new type system that uses linear reasoning to allow memory reuse at different types, object initialization, safe deallocation, and tracking of sharing in data structures. 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.

We have extended the TAL implementation with the features described in this paper.\(^1\) It was quite straightforward to augment the existing F\(^\omega\)-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.

The Popcorn compiler and TAL implementation demonstrate that the ideas presented in this paper can be integrated with a practical and complete programming language. However, for the sake of clarity, we only present a small fragment of our type system and, rather than formalizing it in the context of TAL, we present our ideas in terms of a more familiar lambda calculus. Section 2 gives an informal overview of how to use aliasing constraints, a notion which extends conventional linear type systems, to admit destructive operations such as object deallocation in the presence of aliasing. Section 3 describes the core language formally, with emphasis on the rules for manipulating linear aliasing constraints. Section 4 extends the language with non-linear aliasing constraints. Finally, Section 5 discusses future and related work.

2 Informal Overview

The main feature of our new type system 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 \(\ell_s\) and the

\(^1\) See http://www.cs.cornell.edu/talc for the latest software release.
second object might occupy location \( \ell_o \). In order to track the flow of pointers to these locations accurately, we reflect locations into the type system: A pointer to a location \( \ell \) is given the singleton type \( \text{ptr}(\ell) \). 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:

\[
\begin{array}{c}
\text{SP} \\
\text{PTR(lo)} \\
\text{INT} \\
\text{BOOL} \\
\text{PTR(lo)} \\
\end{array}
\begin{array}{c}
\text{STACK} \\
\text{PTR(lo)} \\
\text{INT} \\
\end{array}
\begin{array}{c}
\text{RI} \\
\text{PTR(lo)} \\
\text{INT} \\
\end{array}
\]

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

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 \( \text{deref} \), which retrieves an integer from a reference cell. There are two immediate problems if we demand that code call \( \text{deref} \) when the store has a shape described by \( \{ \ell \mapsto \langle \text{int} \rangle \} \). First, \( \text{deref} \) can only be used to dereference the location \( \ell \), and not, for example, the locations \( \ell' \) or \( \ell'' \). 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 \( \text{deref} \) requires a store \( \{ \rho \mapsto \langle \text{int} \rangle \} \) where \( \rho \) is a location variable instead of some specific location \( \ell \). Second, the constraint \( \{ \ell \mapsto \langle \text{int} \rangle \} \) specifies a store with exactly one location \( \ell \) although we may want to dereference a single integer reference amongst a sea of other heap-allocated objects. Since \( \text{deref} \) 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 \( \text{deref} \) is \( \epsilon \oplus \{ \rho \mapsto \langle \text{int} \rangle \} \) where \( \epsilon \) 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 \( (\rho) \) is not aliased by any other location \( (\rho') \). This invariant is maintained despite the presence of location polymorphism and store polymorphism. Intuitively, because \( \rho \) 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 n words of storage. The new storage is allocated at a fresh location ℓ in the heap and ℓ is substituted for ρ in the remaining instructions. A pointer to ℓ is substituted for x. Both ρ and x are considered bound by this instruction. The `free` instruction deallocates storage. Deallocated storage has type `junk` and the type system prevents any future use of that space.

### Instructions

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Constraints (Initially the constraints ε)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. malloc sp,ρ₁, 2;</td>
<td>ε ⊕ {ρ₁ → ⟨junk, junk⟩} sp : ptr(ρ₁)</td>
</tr>
<tr>
<td>2. sp[1] : = 1;</td>
<td>ε ⊕ {ρ₁ → ⟨int, junk⟩}</td>
</tr>
<tr>
<td>3. malloc r₁,ρ₂, 1;</td>
<td>ε ⊕ {ρ₁ → ⟨int, junk⟩, ρ₂ → ⟨junk⟩} r₁ : ptr(ρ₂)</td>
</tr>
<tr>
<td>4. sp[2] : = r₁;</td>
<td>ε ⊕ {ρ₁ → ⟨int, ptr(ρ₂)⟩, ρ₂ → ⟨junk⟩}</td>
</tr>
<tr>
<td>5. r₁[1] : = 2;</td>
<td>ε ⊕ {ρ₁ → ⟨int, ptr(ρ₂)⟩, ρ₂ → ⟨int⟩}</td>
</tr>
<tr>
<td>6. free r₁;</td>
<td>ε ⊕ {ρ₁ → ⟨int, ptr(ρ₂)⟩, ρ₂ → junk}</td>
</tr>
<tr>
<td>7. free sp;</td>
<td>ε ⊕ {ρ₁ → junk, ρ₂ → junk}</td>
</tr>
</tbody>
</table>

Again, we can intuitively think of sp as the stack pointer and r₁ as a register that holds an alias of an object on the stack. Notice that on line 5, the initialization of r₁ updates the type of the memory at location ρ₂. This has the effect of simultaneously updating the type of r₁ and of sp[1]. Both of these paths are similarly affected when r₁ 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 dereferenced.

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

### 3 The Language of Locations

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

#### 3.1 Values, Instructions, and Programs

A program is a pair of a store (S) and a list of instructions (i). The store maps locations (ℓ) to values (v). Normally, the values held in the store are memory blocks ([r₁, ..., rₙ]), 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 (ptr(ℓ)).

Figure 2 formally defines the operational semantics of the language. The instruction `malloc x,ρ, n`

---

2 Here and elsewhere, the notation X[c₁, ..., cₙ/x₁, ..., xₙ] denotes capture-avoiding substitution of c₁, ..., cₙ for variables x₁, ..., xₙ in X.
\[\ell \in \text{Locations} \quad \rho \in \text{LocationVar} \quad \epsilon \in \text{ConstraintVar} \quad x, f \in \text{ValueVar}\]

\begin{align*}
\text{locations} & : \eta := \ell | \rho \\
\text{constraints} & : C := \emptyset | \epsilon \rightarrow \tau | C_1 \cup C_2 \\
\text{types} & : \tau := \text{int} | \text{junk} | \text{ptr}(\eta) | \langle \tau_1, \ldots, \tau_n \rangle | \forall [\Delta; C].(\tau_1, \ldots, \tau_n) \rightarrow 0 \\
\text{value ctxs} & : \Gamma ::= \cdot | \Gamma; x: \tau \\
\text{type ctxs} & : \Delta ::= \cdot | \Delta, \rho | \Delta, \epsilon \\
\text{values} & : v ::= x | i | \text{junk} | \text{ptr}(\ell) | \langle v_1, \ldots, v_n \rangle | \text{fix} \ f[\Delta; C; \Gamma] \cdot i \ | v[\eta] \ | v[C] \\
\text{instructions} & : t ::= \text{malloc} \ x, \rho; \ n | x = v[i]; t | v[i] := v'; t | \text{free} \ v; t | v(v_1, \ldots, v_n) \ | \text{halt} \\
\text{stores} & : S ::= \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \\
\text{programs} & : P ::= (S, t)
\end{align*}

Fig. 1. Language of Locations: Syntax

Allocates an uninitialized memory block (filled with junk) of size \(n\) at a new location \(\ell\), and binds \(x\) to the pointer \(\text{ptr}(\ell)\). The location variable \(\rho\), bound by this instruction, is the static representation of the dynamic location \(\ell\). The instruction \(x = v[i]\) binds \(x\) to the \(i\)th component of the memory block pointed to by \(v\) in the remaining instructions. The instruction \(v[i] := v'\) stores \(v'\) in the \(i\)th component of the block pointed to by \(v\). The final memory management primitive, \text{free} \(v\), deallocates the storage pointed to by \(v\). If \(v\) is the pointer \(\text{ptr}(\ell)\) then deallocation is modeled by updating the store \((S)\) so that the location \(\ell\) maps to \text{junk}.

The program \((\{\}, \text{malloc} \ x, \rho; 2; x[1] := 3; x[2] := 5; \text{free} \ x; \text{halt})\) allocates, initializes and finally deallocates a pair of integers. Its evaluation is shown below:

\[
\begin{array}{ll}
\text{Store} & \text{Instructions} \\
\{\} & \text{malloc} \ x, \rho; n (*\text{ allocate new location}\ \ell, * ) \\
 & (*\text{ substitute}\ \text{ptr}(\ell), \ell \text{ for}\ x, \rho \ *) \\
\{\ell \mapsto \langle \text{junk}, \text{junk} \rangle\} & \text{ptr}(\ell)[1] := 3 (*\text{ initialize field}\ 1 \ *) \\
\{\ell \mapsto \langle 3, \text{junk} \rangle\} & \text{ptr}(\ell)[2] := 5 (*\text{ initialize field}\ 2 \ *) \\
\{\ell \mapsto \langle 3, 5 \rangle\} & \text{free} \ \text{ptr}(\ell) (*\text{ free storage}\ *) \\
\{\ell \mapsto \text{junk}\} & \\
\end{array}
\]

A sequence of instructions \((t)\) ends in either a \text{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 \text{continuation-passing style} (CPS) transformation \([14, 15]\). It is possible to define a direct-style language, but doing so 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 \text{fix} \(f[\Delta; C; \Gamma]\.\). These functions are recursive \((f\ may\ appear\ in\ t)\). The context \((\Delta; C; \Gamma)\) specifies a pre-condition.
that must be satisfied before the function can be invoked. The type context \( \Delta \) 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 \( \Gamma \) assigns types to free variables in \( \iota \).

To call a polymorphic function, code must first instantiate the type variables in \( \Delta \) using the value 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).

\[
(S, \text{malloc } x, \rho, n; \iota) \quad \mapsto \quad (S\{\ell \mapsto \{\text{junk}_1, \ldots, \text{junk}_n\}\}, \iota[\ell/\rho][\text{ptr}(\ell)/x])
\]

where \( \ell \not\in S \)

\[
(S\{\ell \mapsto v\}, \text{freeptr}(\ell); \iota) \quad \mapsto \quad (S\{\ell \mapsto \text{junk}\}, \iota)
\]

\[
(S\{\ell \mapsto v\}, \text{ptr}(\ell)[i]=v'; \iota) \quad \mapsto \quad (S\{\ell \mapsto \{v_1, \ldots, v_{i-1}, v', v_{i+1}, \ldots, v_n\}\}, \iota)
\]

if \( v = \{v_1, \ldots, v_n\} \) and \( 1 \leq i \leq n \)

\[
(S\{\ell \mapsto v\}, x=\text{ptr}(\ell)[i]; \iota) \quad \mapsto \quad (S\{\ell \mapsto v\}, \iota[v_i/x])
\]

if \( v = \{v_1, \ldots, v_n\} \) and \( 1 \leq i \leq n \)

\[
(S, v(v_1, \ldots, v_n)) \quad \mapsto \quad (S, \{c_1, \ldots, c_m/\beta_1, \ldots, \beta_m\}[v_1, \ldots, v_n/f, x_1, \ldots, x_n])
\]

if \( v = v'[c_1, \ldots, c_m] \) and \( v' = \text{fix } f[\Delta; C; x_1:\tau_1, \ldots, x_n:\tau_n]; \iota \)

and \( \text{Dom}(\Delta) = \beta_1, \ldots, \beta_m \) (where \( \beta \) ranges over \( \rho \) and \( \epsilon \))

\[\text{Fig. 2. Language of Locations: Operational Semantics}\]

### 3.2 Type Constructors

There are three kinds of type constructors: locations\(^3\) (\( \eta \)), types (\( \tau \)), and aliasing constraints (\( C \)). The simplest types are the base types, which we have chosen to be integers (\( \text{int} \)). A pointer to a location \( \eta \) is given the singleton type \( \text{ptr}(\eta) \).

The only value in the type \( \text{ptr}(\eta) \) is the pointer \( \text{ptr}(\eta) \), so if \( v_1 \) and \( v_2 \) both have type \( \text{ptr}(\eta) \), 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 \( \text{ptr}(\eta) \) 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 \( \eta \) containing a value of type \( \tau \). 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 \( \{\eta \mapsto \tau, \eta' \mapsto \tau'\} \).

\(^3\) We use the meta-variable \( \ell \) to denote concrete locations, \( \rho \) to denote location variables, and \( \eta \) to denote either.
3.3 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 judgement $\vdash S : C$. Intuitively, whenever a location $\ell$ contains a value $v$ of type $\tau$, the constraints should specify that location $\ell$ maps to $\tau$ (or an equivalent type $\tau'$). Formally:

$$\vdash \ell : \tau_1, \ldots, \ell_n : \tau_n$$

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

$$\vdash S : C' \quad \vdash C' = C$$

**Instruction Typing** Instructions are type checked in a context $\Delta; C; \Gamma$. The judgement $\Delta; C; \Gamma \vdash \iota$ states that the instruction sequence is well-formed. A related judgement, $\Delta; C; \Gamma \vdash v : \tau$, ensures that the value $v$ is well-formed and has type $\tau$.4

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:

$$\Delta; C = C' \oplus \{ \eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle \} \quad \Delta; C; x : \tau_i \vdash \iota \quad \text{(} x \not\in \Gamma \text{)} \\
\Delta; C; \Gamma \vdash v[i] : \tau_i \quad (1 \leq i \leq n)$$

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 $C' \oplus \{ \eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle \}$. (Constraint equality uses $\Delta$ to denote the free type variables that may appear on the right-hand side.) 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 constraints $C$.

Next, examine the rule for the assignment operation:

$$\Delta; C = C' \oplus \{ \eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle \} \quad \Delta; C' = C' \oplus \{ \eta \mapsto \tau \} \quad \Delta; \Gamma \vdash \iota \quad \text{(} 1 \leq i \leq n \text{)}$$

where $\tau_{\text{after}}$ 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 $\eta$. The type of the contents of $\eta$ 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 $C' \oplus \{ \eta \mapsto \tau_{\text{after}} \}$.

4 The subscripts on $\vdash_\iota$ and $\vdash_i$ are used to distinguish judgement forms and for no other purpose.
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 $\tau'$ and the contents of the location $\eta$ originally has type $\langle \tau_1, \ldots, \tau_n \rangle$, then $\{ \eta \mapsto \tau_{\text{after}} \}$ describes the contents of the location $\eta$ after the update accurately. However, we must avoid a situation in which $C'$ continues to hold an outdated type for the contents of the location $\eta$. 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 $p_1$ and the current constraints are $\{ p_1 \mapsto \tau, p_2 \mapsto \tau \}$. We cannot determine whether or not $p_1$ and $p_2$ are aliases and therefore whether the final constraint set should be $\{ p_1 \mapsto \tau', p_2 \mapsto \tau' \}$ or $\{ p_1 \mapsto \tau', p_2 \mapsto \tau \}$.

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 $p_1$ and $p_2$ 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 $\tau$ is syntactic up to $\alpha$-conversion of bound variables and modulo equality on constraints. Therefore:

$$
\Delta \vdash \{ p_1 \mapsto \langle \text{int} \rangle \} \oplus \{ p_2 \mapsto \langle \text{bool} \rangle \} = \{ p_2 \mapsto \langle \text{bool} \rangle \} \oplus \{ p_1 \mapsto \langle \text{int} \rangle \}
$$

but,

$$
\Delta \not\vdash \{ p_1 \mapsto \langle \text{int} \rangle \} \oplus \{ p_2 \mapsto \langle \text{bool} \rangle \} = \{ p_1 \mapsto \langle \text{int} \rangle \} \oplus \{ p_1 \mapsto \langle \text{int} \rangle \} \oplus \{ p_2 \mapsto \langle \text{bool} \rangle \}
$$

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 $\cdot \vdash v' : \tau'$ 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 $\ell$ 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 $\Delta$ is $\{ \rho_1, \rho_2, \epsilon \}$ and the constraints $C$ are $\epsilon \oplus \{ \rho_1 \mapsto \langle \text{int} \rangle, \rho_2 \mapsto \langle \text{int} \rangle \}$:

```
fix foo|\Delta; C; x:ptr(\rho_1), y:ptr(\rho_2), cont:\forall \epsilon . (\\epsilon \rightarrow \text{int}) \rightarrow \text{int} .
free z; (* constraints = \epsilon \oplus \{ \rho_2 \mapsto \langle \text{int} \rangle \} *)
z=y[0]; (* ok because y:ptr(\rho_2) and \{ \rho_2 \mapsto \langle \text{int} \rangle \} *)
free y; (* constraints = \epsilon *)
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 \( \text{foo} \) is called in a state where \( p_1 \) and \( p_2 \) 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 \( \text{foo} \) can never be called from such a state.

Suppose that the store currently contains a single integer reference: \( \{ \ell \mapsto \langle \text{int} \rangle \} \). This store can be described by the constraints \( \{ \ell \mapsto \langle \text{int} \rangle \} \). If the programmer attempts to instantiate both \( p_1 \) and \( p_2 \) with the same label \( \ell \), the function call \( \text{foo}(\ell, \ell, \emptyset | \text{ptr}(\ell)) \) will fail to type check because the constraints \( \{ \ell \mapsto \langle \text{int} \rangle \} \) do not equal the pre-condition \( \emptyset \oplus \{ \ell \mapsto \langle \text{int} \rangle, \ell \mapsto \langle \text{int} \rangle \} \).

Figure 3 contains the typing rules for values and instructions. Note that the judgement \( \Delta \vdash_{\forall f} \tau \) indicates that \( \Delta \) contains the free type variables in \( \tau \).

### 3.4 Soundness

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

**Theorem 1 (Soundness)** If \( \vdash S : C \) and \( C; \cdot \vdash \iota \) and \( (S, \iota) \mapsto \ldots \mapsto (S', \iota') \), then \( (S', \iota') \) is not a stuck state.

We prove soundness syntactically in the style of Wright and Felleisen [28]. The proof appears in the companion technical report [19].

### 4 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 \( \text{add} \) has type \( \forall [p_1, p_2, e; \langle \text{int} \rangle \mapsto \langle \text{int} \rangle]^\omega \oplus \langle \text{int} \rangle \mapsto \langle \text{int} \rangle \) and consider this code:

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Constraints (Initially ( \emptyset ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc ( x, p_1, 1 );</td>
<td>( C_1 = { p \mapsto \langle \text{junk} \rangle }, x : \text{ptr}(p) )</td>
</tr>
<tr>
<td>( x[0] := 3 );</td>
<td>( C_2 = { p \mapsto \langle \text{int} \rangle } )</td>
</tr>
<tr>
<td>( \text{add}[p, p, 0](x, x) )</td>
<td>( C_2 \leq { p \mapsto \langle \text{int} \rangle }^\omega = { p \mapsto \langle \text{int} \rangle }^\omega \oplus { p \mapsto \langle \text{int} \rangle }^\omega \oplus \emptyset )</td>
</tr>
</tbody>
</table>

Typing rules for non-linear constraints are presented in Figure 4.
Fig. 3. Language of Locations: Value and Instruction Typing
\[
\frac{\Delta; \Gamma ; v : \text{ptr}(\eta) \quad \Delta \vdash C = C' \oplus \{ \eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle \}^\omega}{\Delta; \Gamma, x : \tau_i, \ i \vdash x = v[i]; \ i} \quad \left( x \notin \Gamma \right)
\]

\[
\frac{\Delta; \Gamma ; v : \text{ptr}(\eta) \quad \Delta \vdash C = C' \oplus \{ \eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle \}^\omega \quad \Delta \vdash \tau' = \tau_i \quad \Delta; \Gamma, v[i] : \tau \vdash v[i] = v'[i]}{\Delta; \Gamma ; v[i] : \tau \vdash v[i] = v'[i]} \quad (1 \leq i \leq n)
\]

\[
\frac{\Delta; \Gamma ; v : \forall i : C', (\tau_1, \ldots, \tau_n) \rightarrow \alpha \quad \Delta \vdash C \leq C'}{\Delta; \Gamma, v_1 : \tau_1 \quad \ldots \quad \Delta; \Gamma, v_n : \tau_n \vdash S; C' \vdash C' \leq C \quad \vdash S; C}
\]

4.1 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.

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

The \texttt{ifnull v then i1 else i2} construct tests an option to determine whether it is null or not. Assuming \( v \) has type \texttt{ptr(\eta)} \( \), we check the first branch \( (i_1) \) with the constraint \( \{ \eta \mapsto \text{null} \}^\rho \) and the second branch with the constraint \( \{ \eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle \}^\rho \) where \( \langle \tau_1, \ldots, \tau_n \rangle \) is the appropriate non-null variant. As before, imagine that \( sp \) is the stack pointer, which contains an integer option.

```plaintext
(* constraints = \{ \eta \mapsto \langle \text{ptr(\eta')} \rangle, \eta' \mapsto ?(\text{int}) \}, sp:ptr(\eta) *)

r1 = sp[1];
(* r1:ptr(\eta') *)
ifnull r1 then halt (* null check *)
else ... (* constraints = \{ \eta \mapsto \langle \text{ptr(\eta')} \rangle \}^\omega \{ \eta' \mapsto ?(\text{int}) \}^\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.
These additional features of our language are also proven sound in the companion technical report [19].

**Syntax:**

```
types \( \tau ::= \ldots \mid \mathit{?}\langle \tau_1, \ldots, \tau_n \rangle \mid \mathit{null} \)
values \( v ::= \ldots \mid \mathit{null} \)
instructions \( \iota ::= \ldots \mid \mathit{mknul} x, \rho; \iota \mid \mathit{tosum} v, \mathit{?}\langle \tau_1, \ldots, \tau_n \rangle \mid \mathit{ifnull} v \mathit{then} \iota_1 \mathit{else} \iota_2 \)
```

**Operational semantics:**

\[
\begin{align*}
(S, \mathit{mknul} x, \rho; \iota) & \quad \mapsto (S[\ell \mapsto \mathit{null}], \iota[\ell/\rho][\mathit{ptr}(\ell)/x]) \\
(S, \mathit{tosum} v, \mathit{?}\langle \tau_1, \ldots, \tau_n \rangle; \iota) & \quad \mapsto (S, \iota) \\
(S[\ell \mapsto \mathit{null}],
\mathit{ifnull} \mathit{ptr}(\ell) \mathit{then} \iota_1 \mathit{else} \iota_2) & \quad \mapsto (S[\ell \mapsto \mathit{null}], \iota_1) \\
(S[\ell \mapsto \langle v_1, \ldots, v_n \rangle],
\mathit{ifnull} \mathit{ptr}(\ell) \mathit{then} \iota_1 \mathit{else} \iota_2) & \quad \mapsto (S[\ell \mapsto \langle v_1, \ldots, v_n \rangle], \iota_2)
\end{align*}
\]

**Static Semantics:**

\[
\begin{align*}
\Delta; \Gamma \vdash \mathit{null} : \mathit{null} & \quad \frac{\Delta, \rho; C \oplus \{ \rho \mapsto \mathit{null} \}; \Gamma, x: \mathit{ptr}(\rho) \vdash \iota}{\Delta; \Gamma, x, \rho; \iota} \quad (x \not\in \Gamma, \rho \not\in \Delta)
\end{align*}
\]

\[
\begin{align*}
\Delta; \Gamma \vdash v : \mathit{ptr}(\eta) & \quad \frac{\Delta; C = C' \oplus \{ \eta \mapsto \mathit{null} \}^\phi}{\Delta; C' \oplus \{ \eta \mapsto \mathit{null} \}^\phi; \Gamma \vdash \iota}
\end{align*}
\]

\[
\begin{align*}
\Delta; \Gamma \vdash \mathit{tosum} v, \mathit{?}\langle \tau_1, \ldots, \tau_n \rangle; \iota & \quad \frac{\Delta; \Gamma \vdash v : \mathit{ptr}(\eta)}{\Delta; C = C' \oplus \{ \eta \mapsto \mathit{null} \}^\phi; \Gamma \vdash \iota}
\end{align*}
\]

\[
\begin{align*}
\Delta; \Gamma \vdash \mathit{ifnull} v \mathit{then} \iota_1 \mathit{else} \iota_2 & \quad \frac{\Delta; C' \oplus \{ \eta \mapsto \mathit{null} \}^\phi; \Gamma \vdash \iota_1 \quad \Delta; C' \oplus \{ \eta \mapsto \mathit{null} \}^\phi; \Gamma \vdash \iota_2}{\Delta; C; \Gamma \vdash \mathit{ifnull} v \mathit{then} \iota_1 \mathit{else} \iota_2}
\end{align*}
\]

**Fig. 5.** Language of Locations: Extensions for option types

5 Related and Future Work

Our research extends previous work on linear type systems [26] and syntactic control of interference [16] by allowing both aliasing and safe deallocation. Several authors [26,3,9] have explored alternatives to pure linear type systems to
allow greater flexibility. Wadler [26], for example, introduced a new let-form
let! (x) y = e1 in e2 that permits the variable x to be used as a non-linear
value in e1 (i.e. it can be used many times, albeit in a restricted fashion) and
then later used as a linear value in e2. We believe we can encode similar behavior
by extending our simple subtyping with bounded quantification. For instance, if
a function f requires some collection of aliasing constraints e that are bounded
above by \{\rho_1 \mapsto \langle \text{int} \rangle\} \oplus \{\rho_2 \mapsto \langle \text{int}\rangle\} \omega, then f may be called with a single
linear constraint \{\rho \mapsto \langle \text{int} \rangle\} (instantiating both \rho_1 and \rho_2 with \rho and e with 
\{\rho \mapsto \langle \text{int} \rangle\}). The constraints may now be used non-linearly within the body
of f. Provided f expects a continuation with constraints \epsilon, its continuation will
retain the knowledge that \{\rho \mapsto \langle \text{int} \rangle\} is linear and will be able to deallocate
the storage associated with \rho when it is called. However, we have not yet imple-
mented this feature.

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. 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
et al. [11] except that activation records are allocated on the heap rather than
using a conventional call stack. The companion technical report [19] shows how
to compile a simple imperative language in such a way that it allocates and
deletes its own stack frames.

This research is also related to other work on type systems for low-level
languages. Work on Java bytecode verification [20,8] also develops type systems
that allows locations to hold values of different types. However, the Java bytecode
type system is not strong enough to represent aliasing as we do here.

The development of our language was inspired by the Calculus of Capa-
bilities (CC) [4]. CC provides an alternative to the region-based type system
developed by Toft and Talpin [24]. Because safe region deallocation requires that no aliases be used in the future, CC tracks region aliases. In our new lan-
guage 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 [5,21,
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 [7,6] 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 a new analysis technique,
but rather a sound system for representing and checking the results of analysis, and, in particular, for representing aliasing in low-level compiler-introduced data structures rather than for representing aliasing in source-level data.

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. In particular, we are working on integrating recursive types into the type system as they would allow us to capture regular repeating structure in the store. When we have completed this task, we believe our aliasing constraints will provide us with a safe, but rich and reusable, set of memory abstractions.

Acknowledgements

This work arose in the context of implementing the Typed Assembly Language compiler. We are grateful for the many stimulating discussions that we have had on this topic with Karl Crary, Neal Glew, Dan Grosman, Dexter Kozen, Stephanie Weinrich, and Steve Zdancevic. Sophia Drossopoulou, Kathleen Fisher, Andrew Myers, and Anne Rogers gave helpful comments on a previous draft of this work.

References