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 [25, 24] 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: `let` $x = \langle 1, 2\rangle$ `in let` $y = fst(x)$ `in let` $z = snd(x)$ `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 [25], for example, introduced a new let-form `let!` $(x)\ y = e_1$ `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 type inference techniques. These

⋆ 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.

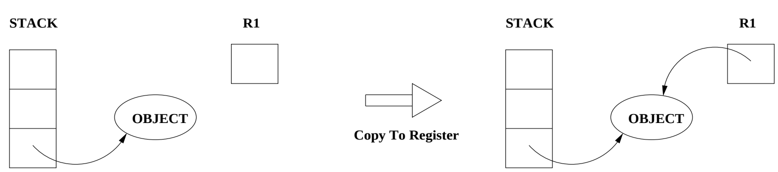
issues are not as much of a concern for low-level typed languages designed as compiler intermediate languages [21, 18] or as secure mobile code platforms, such as the Java Virtual Machine [10], Proof-Carrying Code [13] or Typed Assembly Language (TAL) [12]. These languages are designed for machine, not 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 [22, 27], 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:

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

   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.

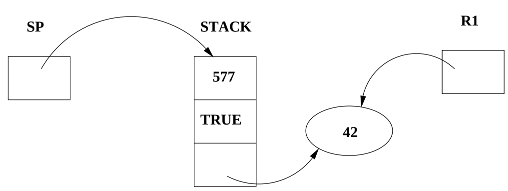
Pointer aliasing and data sharing also occur naturally in other data structures introduced by a compiler. For example, compilers often use a top-of-stack pointer and a frame pointer, both of which point to the same data structure. Compiling a language like Pascal using displays [1] generalizes this problem 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

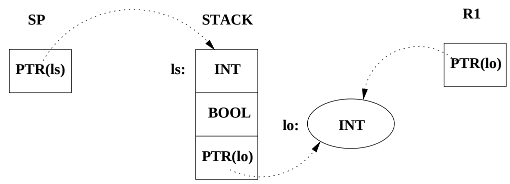
We have devised a new type system that is capable of tracking sharing in data structures and that admits operations for memory reuse at different types, object initialization, and deallocation. This paper formalizes the type system and provides a theoretical foundation for safely integrating operations 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.	$\texttt{malloc}\, sp, \rho_1, 2;$	$\epsilon \oplus \{\rho_1 \mapsto \langle junk, junk \rangle\}$	$sp : ptr(\rho_1)$
2.	$sp[1]{:=}1;$	$\epsilon \oplus \{\rho_1 \mapsto \langle int, junk \rangle\}$	
3.	$\texttt{malloc}\, r_1, \rho_2, 1;$	$\epsilon \oplus \{\rho_1 \mapsto \langle int, junk \rangle, \rho_2 \mapsto \langle junk \rangle\}$	$r_1 : ptr(\rho_2)$
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.	$\texttt{free}\, r_1;$	$\epsilon \oplus \{\rho_1 \mapsto \langle int, ptr(\rho_2) \rangle, \rho_2 \mapsto junk\}$	
7.	$\texttt{free}\, sp;$	$\epsilon \oplus \{\rho_1 \mapsto junk, \rho_2 \mapsto junk\}$	

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.[2] 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. Finally, in Section 4 and Section 5, 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] See `http://www.cs.cornell.edu/talc` for the latest software release.

$\rho \in$ LabelVar $\quad \epsilon \in$ ConstraintVar $\quad \ell \in$ Locations $\quad x, f \in$ ValueVar

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

Fig. 1. Language of Locations: Syntax

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 $\texttt{malloc}\, x, \rho, n$ 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, $\texttt{free}\, v$, deallocates the 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` $\texttt{ptr}(\ell), \ell$ `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 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.

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

Fig. 2. Language of Locations: Operational Semantics

2.2 Type Constructors

There are three kinds of type constructors: locations[3] (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 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:

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

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

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

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:

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

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

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:

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

but,

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

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 $\epsilon \oplus \{\rho_1 \mapsto \langle int \rangle, \rho_2 \mapsto \langle int \rangle\}$:

$\texttt{fix}\, \mathit{foo}[\Delta; C; x{:}\mathit{ptr}(\rho_1), y{:}\mathit{ptr}(\rho_2), \mathit{cont}{:}\forall[\cdot;\epsilon].(\mathit{int})\to 0].$

$\texttt{free}\ x;$	`(* constraints =` $\epsilon \oplus \{\rho_2 \mapsto \langle int \rangle\}$	`*)`
$z{=}y[0];$	`(* ok because` $y : \mathit{ptr}(\rho_2)$ `and` $\{\rho_2 \mapsto \langle int \rangle\}$	`*)`
$\texttt{free}\ y;$	`(* constraints =` ϵ	`*)`
$\mathit{cont}(z)$	`(* 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 $\{\ell \mapsto \langle int \rangle\}$. 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 $\{\ell \mapsto \langle int \rangle\}$ do not equal the pre-condition $\emptyset \oplus \{\ell \mapsto \langle int \rangle, \ell \mapsto \langle int \rangle\}$.

Figure 3 contains the typing rules for values and instructions.

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 1 (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 [28]. Due to space considerations, we cannot include the proof in this paper. It appears in a companion technical report [19].

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:

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

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

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

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

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

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

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

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

$$\frac{\Delta, \rho; C \oplus \{\rho \mapsto \langle junk_1, \ldots, junk_n \rangle\}; \Gamma, x{:}ptr(\rho) \vdash \iota}{\Delta; C; \Gamma \vdash \texttt{malloc}\ x, \rho, n; \iota} \ (x \notin \Gamma, \rho \notin \Delta)$$

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

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

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

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

Fig. 3. Language of Locations: Value and Instruction Typing

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 $\forall[\rho_1, \rho_2, \epsilon; \{\rho_1 \mapsto \langle int\rangle\}^\omega \oplus \{\rho_2 \mapsto \langle int\rangle\}^\omega \oplus \epsilon].(ptr(\rho_1), ptr(\rho_2))\rightarrow 0$ and consider this code:

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

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

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

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

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

Fig. 4. Language of Locations: Non-linear Constraints

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.

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 $ptr(\eta)$, we check the first branch (ι_1) with the constraint $\{\eta \mapsto null\}^\phi$ 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 = {η ↦ ⟨ptr(η′)⟩, η′ ↦ ?⟨int⟩}, sp:ptr(η)           *)
r1=sp[1];               (* r1:ptr(η′)                              *)
ifnull r1 then halt     (* null check                              *)
else ···                (* constraints = {η ↦ ⟨ptr(η′)⟩}⊕{η′ ↦ ⟨int⟩}^ω *)
```

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 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 [29] and run-time type analysis [5], and may soon be used to facilitate static checking of expressive security policies [26]. Moreover TAL, like other F^ω-based type-directed compilers such as TIL(T) [21] 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. 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.

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 [23]. Because safe region deallocation requires that no

Syntax:

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

Operational semantics:

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

Static Semantics:

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

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

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

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

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

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, 20, 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.

5 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. Frederick Smith, David Walker, and Greg Morrisett. Alias types. Technical Report TR99-1773, Cornell University, October 1999.
20. B. Steensgaard. Points-to analysis in linear time. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, January 1996.
21. 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.
22. Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.
23. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
24. 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.
25. 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.
26. David Walker. A type system for expressive security policies. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, Boston, January 2000. To appear.

27. A. K. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4), December 1995.
28. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
29. 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.