Alias Types for Recursive Data Structures *

David Walker and Greg Morrisett

Cornell University

Abstract

Linear type systems permit programmers to deallocate or explicitly recycle memory, but are severely restricted by the fact that they admit no aliasing. This paper describes a pseudo-linear type system that allows a degree of aliasing and memory reuse as well as the ability to define complex recursive data structures. Our type system can encode conventional linear data structures such as linear lists and trees as well as more sophisticated data structures including cyclic and doubly-linked lists and trees. In the latter cases, our type system is expressive enough to represent pointer aliasing and yet safely permit destructive operations such as object deallocation. We demonstrate the flexibility of our type system by encoding two common space-conscious algorithms: destination-passing style and Deutsch-Schorr-Waite or "link-reversal" traversal algorithms.

1 Introduction

Type-safe programming languages, such as Haskell, Java, and ML, do not give programmers control over memory management. In particular, these languages do not allow programmers to separate allocation and initialization of memory objects, nor do they allow explicit re-use of memory objects. Rather, allocation and initialization of objects are presented to the programmer as an atomic operation, and re-use of memory is achieved "under the covers" through garbage collection. In other words, memory management is achieved by meta-linguistic mechanisms that are largely outside the control of the programmer.

In type-unsafe languages such as C or C++, programmers have control over memory management so they can tailor routines for application-specific constraints, where the time and/or space overheads of general-purpose memory management mechanisms do not suffice. However, such languages have a far more complicated and error-prone programming model. In particular, neither the static type systems, the compilers, nor the run-time systems of these languages prevent the accidental use of uninitialized objects, or the accidental re-use of memory at an incompatible type. Such errors are extremely costly to diagnose and correct.

Our ultimate goal is to provide support for programmer-controlled memory management, without sacrificing type-safety, and without incurring significant overhead. In addition, we hope to discover general typing mechanisms and principles that allow greater lattitude in the design of low-level languages intended for systems applications or as the target of certifying compilers [28, 30]. In this paper, we take a step further towards these goals by developing a type system that gives fine-grained control over memory management, for a rich class of recursively defined datatypes. We demonstrate the power of the type system by showing how we can safely encode two important classes of optimization, destination-passing style and link-reversal traversals of data structures.

1.1 Background

One well-known principle for proving type safety is based upon *type-invariance of memory locations*. Simply put, this property says that, when allocated, a memory object should (conceptually) be stamped with its type, and that the type of the object should not change during evaluation. When this property is maintained, it is straightforward to prove a subject-reduction or type-preservation property (see for example [44, 16]), which is in turn crucial to establishing type-soundness. There are many examples from language design where this principle has been violated and resulted in an unsoundness. For instance, the naive treatment of polymorphic references in an ML-like language, or the covariant treatment of arrays in a Java-like language, both violate this basic principle.

From the type-invariance principle, it becomes clear why most type-safe languages do not support user-level initialization or memory recycling: the type τ of the memory object cannot change, so (1) it must initially have type τ and (2) must continue to have type τ after an evaluation step. Atomic allocation and initialization ensures the first invariant, and the lack of explicit recycling ensures the second. Thus, it appears that some meta-linguistic mechanism is necessary to achieve memory management when the type-invariance principle is employed.

Linear type systems [42, 40] employ a different principle to achieve subject-reduction. In a linear setting, the crucial invariant is that memory objects must have exactly one reference — that is, no object can be aliased. Unlike the traditional approach, the type of a memory object can change over time and thus, explicit initialization and recycling can be performed in the language. Unfortunately, the inability to share objects through aliasing can have a steep cost: Many common and efficient data structures that use sharing or involve cycles cannot be implemented.

In recent previous work, we considered a generalization of linear types that supported a very limited degree of aliasing [36]. Like linear type systems, our alias types supported separation of allocation and initialization, and explicit re-use of memory, but unlike linear approaches, some objects could have more than one reference. To achieve subject reduction, we tracked aliasing in the type system by giving memory objects unique names, and maintained the invariant that the names were unique. We found that alias types unified a number of ad-hoc features in our Typed Assembly

*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.

Language, including the treatment of initialization and control stacks. Furthermore, the alias type constructors were easy to add to our type checker for TALx86 [38].

Unfortunately, the named objects in our alias-type system were restricted to a "second-class" status; though named objects could be passed to and from functions, the type system prevented a programmer from placing these objects in a recursive datatype such as a list or tree. The problem is that our type system did not track aliasing beyond a certain compile-time "frontier", and in this respect, was similar to the k-limiting approaches used in alias analysis [18]. As a result, we could not embed linear datatypes into our language, and the opportunities for user-level memory management were greatly reduced.

In this paper, we extend alias types to cover recursive datatypes in full generality. Our type system is powerful enough to encode linear variants of lists and trees, as well as richer data structures with complex shapes and aliasing relationships, such as cyclic or doubly-linked lists and trees. The critical addition to the type system is a mechanism for combining recursive type operators with first-class store abstractions that represent repeated patterns of aliasing. In this respect, our work is inspired by the more complex approaches to alias and shape analysis that have recently appeared in the literature [11, 12, 33].

The generalization to recursive datatypes opens the door for users or certifying compilers to have far more control over the memory management of complex data structures. To demonstrate this fact, we show how two classes of space optimization can be encoded in a language based on recursive alias types. The first optimization, called *destination-passing style* [41, 22, 7] transforms algorithms that are "tail-recursive modulo allocation" into properly tail-recursive algorithms, thereby avoiding the space overheads of a control stack. The second optimization shows how we can safely encode Deutsch-Schorr-Waite algorithms [35] for traversing a tree using minimal additional space, based on link reversal.

In the following section, we motivate the type structure of the language by introducing a series of type-theoretic abstraction mechanisms that enable suitable approximations of the store. We then show how these constructors may be used to encode a number of common data structures, without losing the ability to explicitly manage memory. Section 3 formalizes these ideas by presenting the syntax and static semantics of a programming language that includes instructions for allocating, deallocating, and overwriting memory objects. Section 4 shows how the destination-passing style and link-reversal optimizations can be safely encoded in the language. Section 5 presents an operational semantics for the language and states a type soundness theorem. We close in Section 6 by discussing some of the limitations of this work and how they might be addressed as well as giving more detail on related research.

2 Types for describing store shapes

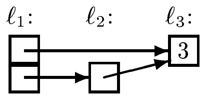
The linear pair $\tau_1 \otimes \tau_2$ captures an extremely valuable memory management invariant: There is only one access path to any value with this type. Consequently, if x has type $\tau_1 \otimes \tau_2$ then once both its components have been extracted, it is safe to reuse x to store new values with incompatible types. Since the only way to access x's data is through x itself, there is no chance that this reuse can introduce inconsistent views of the store and unsoundness into the system.

Unfortunately, the restriction to a single access path makes it impossible to construct a number of important data structures. Our goal is to lift this restriction and yet retain the capacity to reuse or deallocate memory when there is a pointer to it. Our approach is based on the intuition that a linear data structure may be decomposed into two parts, a piece of state and a pointer to that state. Destructive operations such as memory reuse alter only the state component and leave the pointer part unchanged. Consequently, if the goal is to ensure no inconsistencies arise, only the state component need be treated linearly. The pointer may be freely copied, making it possible to construct complex data structures with shared parts. Of course, in order to actually *use* a pointer, there must be some way to relate it to the state it points to. We make this relationship explicit in the type system by introducing locations, ℓ, that contain the state component, and by specializing the type of a pointer to indicate the location it points to. Consider again the linear pair $\tau_1 \otimes \tau_2$. We factor it into two parts:

- A type for the state, called an *aliasing constraint* or *store description*, that takes the form $\{\ell \mapsto \langle\tau_1, \tau_2\rangle\}$. This type states that at location ℓ there exists a memory block containing objects with types τ_1 and τ_2.
- A type for a pointer to the location: $ptr(\ell)$. This type is a *singleton type*—any pointer described by this type is a pointer to the one location ℓ and to no other location.

This simple trick provides a tremendous flexibility advantage over conventional linear type systems because even though constraints may not alias one another, there is no explicit restriction on the way pointer types may be manipulated.

We build complicated data structures by joining a number of aliasing constraints together using the $\otimes$ constructor. For example, the following DAG may be specified by the constraints below.



$$\begin{array}{l}\{\ell_1 \mapsto \langle ptr(\ell_2), ptr(\ell_3)\rangle\} \otimes \\ \{\ell_2 \mapsto \langle ptr(\ell_3)\rangle\} \otimes \\ \{\ell_3 \mapsto \langle int\rangle\}\end{array}$$

In this type, the locations ℓ_1, ℓ_2 and ℓ_3 are necessarily distinct from one another because they all appear on left-hand sides in this collection of constraints. The type system maintains the invariant that if a store is described by constraints $\{\ell_1 \mapsto \tau_1\} \otimes \cdots \otimes \{\ell_n \mapsto \tau_n\}$ then each of the locations ℓ_i must be different from one another. This invariant resembles invariants for the typing context of a standard linear type system. For example, the linear context $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$ implies that the x_i are distinct values with linear types τ_i. However, the analogy is not exact because a linear type system prevents any of the x_i from being used more than once whereas our calculus allows pointers to the locations ℓ_i to be used over and over again and this flexibility makes it possible to represent aliasing: In the type above, there are two paths from location ℓ_1 to location ℓ_3, one direct and one indirect through location ℓ_2.

One other important invariant is that although the $\otimes$ constructor is reminiscent of linear pairs, the ordering of the constraints joined by $\otimes$ is not important: $\{\ell_1 \mapsto \tau_1\} \otimes \{\ell_2 \mapsto \tau_2\}$ is equivalent to $\{\ell_2 \mapsto \tau_2\} \otimes \{\ell_1 \mapsto \tau_1\}$. For the sake of brevity, we often abbreviate $\{\ell_1 \mapsto \tau_1\} \otimes \cdots \otimes \{\ell_n \mapsto \tau_n\}$ with $\{\ell_1 \mapsto \tau_1, \ldots, \ell_n \mapsto \tau_n\}$.

2.1 Abstraction mechanisms

Any particular store can be represented exactly using these techniques[1], even stores containing cyclic data structures. For example, a node containing a pointer to itself may be represented with the type $\{\ell \mapsto \langle ptr(\ell)\rangle\}$. However, the principal difficulty in describing aliasing relationships is not specifying one particular store but being able to specify a class of stores using a single compact representation. We use the following type-theoretic abstraction mechanisms to describe a wide class of pointer-rich data structures.

Location Polymorphism In general, the particular location ℓ that contains an object is inconsequential to the algorithm being executed. The relevant information is the connection between the location ℓ, the contents of the memory residing there, and the pointers $ptr(\ell)$ to that location. Routines that only operate on specific concrete locations are almost useless. If, for example, the dereference function could only operate on a single concrete location ℓ, we would have to implement a different dereference function for every location we allocate in the store! By introducing *location polymorphism*, it is possible to abstract away from the concrete location ℓ using a variable location ρ, but retain the necessary dependencies. We use the meta-variable η to refer to locations generically (either concrete or variable).

Store Polymorphism Any specific routine only operates over a portion of the store. In order to use that routine in multiple contexts, we abstract irrelevant portions of the store using *store polymorphism*. A store described by the constraints $\epsilon \otimes \{\eta \mapsto \tau\}$ contains some store of unknown size and shape ϵ as well as a location η containing objects with type τ. We use the meta-variable C to range over aliasing constraints in general.

Unions Unlike polymorphic types, unions provide users with the abstraction of one of a finite number of choices. A memory block that holds either an integer or a pointer may be encoded using the type $\langle int\rangle \cup \langle ptr(\eta)\rangle$. However, in order to use the contents of the block safely, there must be some way to detect which element of the union the underlying value actually belongs to. There are several ways to perform this test: through a pointer equality test with an object of known type, by descriminating between small integers (including null/0) and pointers, or by distinguishing between components using explicit tags. All of these options will be useful in an implementation, but here we concentrate on the third option (see section 6.1 for further discussion). Hence, the alternatives above will be encoded using the type $\langle \mathcal{S}(1), int\rangle \cup \langle \mathcal{S}(2), ptr(\eta)\rangle$ where $\mathcal{S}(i)$ is another form of singleton type — the type containing only the integer i.

Recursion As yet, we have defined no mechanism for describing regular repeated structure in the store. We use standard recursive types of the form $\mu\alpha.\tau$ to capture this notion. However, recursion by itself is not enough. Consider an attempt to represent a store containing a linked list in the obvious way: $\{\eta \mapsto \mu\alpha.\langle \mathcal{S}(1)\rangle \cup \langle \mathcal{S}(2), \alpha\rangle\}$.[2] An unfolding of this definition results in the type $\{\eta \mapsto \langle \mathcal{S}(1)\rangle \cup \langle \mathcal{S}(2), \langle \mathcal{S}(1)\rangle \cup \langle \mathcal{S}(2), List\rangle\rangle\}$, rather than the type $\{\eta \mapsto \langle \mathcal{S}(1)\rangle \cup \langle \mathcal{S}(2), ptr(\eta')\rangle, \eta' \mapsto \langle \mathcal{S}(1)\rangle \cup \langle \mathcal{S}(2), List\rangle\}$. The former type describes a number of memory blocks flattened into the same location whereas the latter type describes a linked collection of disjoint nodes.

Encapsulation In order to represent linked recursive structures properly, each unfolding must encapsulate its own portion of the store. We use an existential type for this purpose. Hence, a sensible representation for linked lists is

$$\mu\alpha.\langle \mathcal{S}(1)\rangle \cup \exists[\rho{:}\mathtt{Loc} \mid \{\rho \mapsto \alpha\}].\langle \mathcal{S}(2), ptr(\rho)\rangle$$

The existential $\exists[\rho{:}\mathtt{Loc} \mid \{\rho \mapsto \tau_1\}].\tau_2$ may be read "there exists some location ρ, different from all others in the program, such that ρ contains an object of type τ_1, and the value contained in this data structure has type τ_2. More generally, an existential has the form $\exists[\Delta \mid C].\tau$. It abstracts a sequence of type variables with their kinds, Δ, and encapsulates a store described by some constraints C. In our examples, we will omit the kinds from the sequence Δ as they are clear from context. A similar definition gives rise to trees:

$$\mu\alpha.\langle \mathcal{S}(1)\rangle \cup \exists[\rho_1, \rho_2 \mid \{\rho_1 \mapsto \alpha, \rho_2 \mapsto \alpha\}].\langle \mathcal{S}(2), ptr(\rho_1), ptr(\rho_2)\rangle$$

Notice that the existential abstracts a pair of locations and that both locations are bound in the store. From this definition, we can infer that the two subtrees are disjoint. For the sake of contrast, a DAG in which every node has a pair of pointers to a single successor is coded as follows. Here, reuse of the same location variable ρ indicates aliasing.

$$\mu\alpha.\langle \mathcal{S}(1)\rangle \cup \exists[\rho \mid \{\rho \mapsto \alpha\}].\langle \mathcal{S}(2), ptr(\rho), ptr(\rho)\rangle$$

Cyclic lists and trees with leaves that point back to their roots also cause little problem—simply replace the terminal node with a memory block containing a pointer type back to the roots.

$$\begin{aligned} &CircularList = \\ &\quad \{\rho_1 \mapsto \mu\alpha.\langle \mathcal{S}(1), ptr(\rho_1)\rangle \cup \\ &\qquad\qquad \exists[\rho_2 \mid \{\rho_2 \mapsto \alpha\}].\langle \mathcal{S}(2), ptr(\rho_2)\rangle\} \end{aligned}$$

$$\begin{aligned} &CircularTree = \\ &\quad \{\rho_1 \mapsto \mu\alpha.\langle \mathcal{S}(1), ptr(\rho_1)\rangle \cup \\ &\qquad\qquad \exists[\rho_2, \rho_3 \mid \{\rho_2 \mapsto \alpha, \rho_3 \mapsto \alpha\}]. \\ &\qquad\qquad\quad \langle \mathcal{S}(2), ptr(\rho_2), ptr(\rho_3)\rangle\} \end{aligned}$$

Parameterized Recursive Types One common data structure we are unable to encode with the types described so far is the doubly-linked list. Recursive types only "unfold" in one direction, making it easy to represent pointers from a parent "down" to its children, or all the way back up to the top-level store, but much more difficult to represent pointers that point back up from children to their parents, which is the case for doubly-linked lists or trees with pointers back

[1] We cannot represent a store containing a pointer into the middle of a memory block.

[2] Throughout we use the convention that union binds tighter than the recursion operator.

to their parent nodes. Our solution to this problem is to use parameterized recursive types to pass a parent location down to its children. In general, a parameterized recursive type has the form $\texttt{rec}\,\alpha\,(\beta_1{:}\kappa_1,\ldots,\beta_n{:}\kappa_n).\tau$ and has kind $(\kappa_1,\ldots,\kappa_n)\rightarrow\texttt{Type}$. We will continue to use unparameterized recursive types $\mu\alpha.\tau$ in examples and consider them to be an abbreviation for $\texttt{rec}\,\alpha\,().\tau[\alpha\,()/\alpha]$. Once again, kinds will be omitted when they are clear from the context. Trees in which each node has a pointer to its parent may be encoded as follows.

$$\begin{array}{l}\{\rho_{root}\mapsto\langle\mathcal{S}(2),ptr(\rho_L),ptr(\rho_R)\rangle\}\otimes\\ \{\rho_L\mapsto REC\,(\rho_{root},\rho_L)\}\otimes\\ \{\rho_R\mapsto REC\,(\rho_{root},\rho_R)\}\end{array}$$

where

$$\begin{array}{l}REC=\\ \quad\texttt{rec}\,\alpha\,(\rho_{prt},\rho_{curr}).\\ \qquad\langle\mathcal{S}(1),ptr(\rho_{prt})\rangle\cup\\ \qquad\exists[\rho_L,\rho_R\mid\{\rho_L\mapsto\alpha\,(\rho_{curr},\rho_L)\}\otimes\\ \qquad\qquad\qquad\{\rho_R\mapsto\alpha\,(\rho_{curr},\rho_R)\}].\\ \qquad\qquad\qquad\quad\langle\mathcal{S}(2),ptr(\rho_L),ptr(\rho_R),ptr(\rho_{prt})\rangle\end{array}$$

The tree has a root node in location ρ_{root} that points to a pair of children in locations ρ_L and ρ_R, each of which are defined by the recursive type REC. REC has two arguments, one for the location of its immediate parent ρ_{prt} and one for the location of the current node ρ_{curr}. Either the current node is a leaf, in which case it points back to its immediate parent, or it is an interior node, in which case it contains pointers to its two children ρ_L and ρ_R as well as a pointer to its parent. The children are defined recursively by providing the location of the current node (ρ_{curr}) for the parent parameter and the location of the respective child (ρ_L or ρ_R) for the current pointer.

Function Types Functions are polymorphic with type arguments Δ and they express the shape of the store (C) required by the function: $\forall[\Delta\mid C].(\tau_1,\ldots,\tau_n)\rightarrow\mathbf{0}$. The underlying term language will be written in continuation-passing style and therefore functions never return, but instead call another function (the function's continuation). We use the notation "$\rightarrow\mathbf{0}$" to indicate this fact. Continuation-passing style is extremely convenient in this setting because it makes the flow of control explicit in the language and the store shape varies from one control-flow point to the next.

2.2 Summary of Type Structure

Figure 1 defines the formal syntax for the type constructor language. We use β to range over type constructor variables generically. When we want to be more precise, we use ρ, ϵ and α to range over location, store and type variables. A type context Δ is a sequence of bindings of the form $\beta_1{:}\kappa_1,\ldots,\beta_n{:}\kappa_n$ where none of the β_i are repeated. The domain of Δ, denoted $Dom(\Delta)$, is the sequence $\beta_1,\ldots,\beta_n$. The type constructor language itself contains all the types discussed in the previous subsection and one other, the *junk* type. Objects of type *junk* are unuseable and arise during the initialization of data structures. Section 3 contains further explanation.

A judgement $\Delta\vdash c:\kappa$ states that under type context Δ, the type constructor c is well-formed and has kind κ. Locations have kind `Loc`, aliasing constraints have kind `Store`, and types have kind `Type`. Recursive types have arrow kinds that can be eliminated through constructor application $c\,(c_1,\ldots,c_n)$. The judgement $\Delta\vdash c_1=c_2:\kappa$ states that type constructors c_1 and c_2 are equivalent and well-formed with kind κ. Types are considered equivalent up to alpha-conversion of bound variables and constraints are considered equivalent up to reordering of the elements in the sequence. A recursive type is not considered equal to its unfolding. The formal rules for these judgements are straightforward and they appear in Appendix A.

$$\begin{array}{llcl}
\textit{kinds} & \kappa & ::= & \texttt{Loc}\mid\texttt{Store}\mid\texttt{Type}\mid\\
 & & & (\kappa_1,\ldots,\kappa_n)\rightarrow\texttt{Type}\\
\textit{constructor vars} & \beta & ::= & \rho\mid\epsilon\mid\alpha\\
\textit{constructor ctxts} & \Delta & ::= & \cdot\mid\Delta,\beta{:}\kappa\\
\\
\textit{con's} & c & ::= & \eta\mid C\mid\tau\\
\textit{locations} & \eta & ::= & \rho\mid\ell\\
\textit{store types} & C & ::= & \emptyset\mid C\otimes\{\eta\mapsto\tau\}\mid C\otimes\epsilon\\
\textit{types} & \tau & ::= & \alpha\mid junk\mid int\mid\mathcal{S}(i)\mid ptr(\eta)\mid\\
 & & & \langle\tau_1,\ldots,\tau_n\rangle\mid\tau_1\cup\tau_2\mid\\
 & & & \forall[\Delta\mid C].(\tau_1,\ldots,\tau_n)\rightarrow\mathbf{0}\mid\\
 & & & \exists[\Delta\mid C].\tau\mid\\
 & & & \texttt{rec}\,\alpha\,(\Delta).\tau\mid c\,(c_1,\ldots,c_n)
\end{array}$$

Figure 1: Type Structure: Syntax

We use the notation $A[X/x]$ to denote the capture-avoiding substitution of X for a variable x in A. Occasionally, we use the notation $X[c_1,\ldots,c_n/\Delta]$ to denote capture-avoiding substitution of constructors $c_1,\ldots,c_n$ for the corresponding type variables in $Dom(\Delta)$. Substitution is defined in the stardard way in all cases except for the substitution of constraints in constraints. Substitution of C' for a constraint variable ϵ in C appends the list C' to the list C. We use the notation $C'@C$ to denote the result of appending C to C' (notice that $C\otimes C'$ is not syntactically well-formed). For example,

$$\begin{array}{c}(\emptyset\otimes a_1\otimes\cdots\otimes a_m)@(\emptyset\otimes a_1'\otimes\cdots\otimes a_n')=\\ \emptyset\otimes a_1\otimes\cdots\otimes a_m\otimes a_1'\otimes\cdots\otimes a_n'\end{array}$$

Formally, substitution for constraints is defined as follows.

$$(C\otimes\epsilon)[C'/\epsilon]=(C[C'/\epsilon])@C'$$

We will continue to omit the initial "$\emptyset$" when a constraint is non-empty. For example, we write $\{\eta\mapsto\tau\}$ instead of $\emptyset\otimes\{\eta\mapsto\tau\}$.

3 Term Structure

The term structure is split into three classes: small values, instructions, and coercions. Figure 2 describes the syntax of the language.

3.1 Small Values

Small values include integers (i) and singleton integers $\mathcal{S}(i)$. These two different sorts of integer can be implemented using the same representation. The annotation $\mathcal{S}(\cdot)$ is present only to guide the type checker. Plain integers are given the

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

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

$$\frac{\Delta \vdash \forall[\Delta' \mid C'].(\tau_1, \ldots, \tau_n) \to \mathbf{0} = \tau_f : \mathtt{Type} \quad \Delta\Delta'; C'; \Gamma, f{:}\tau_f, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash \iota}{\Delta; \Gamma \vdash \mathtt{fix} f[\Delta' \mid C'](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).\iota : \tau_f}$$

$$\frac{\Delta; \Gamma \vdash v : \forall[\beta{:}\kappa, \Delta' \mid C'].(\tau_1, \ldots, \tau_n) \to \mathbf{0} \quad \Delta \vdash c : \kappa}{\Delta; \Gamma \vdash v[c] : (\forall[\Delta' \mid C'].(\tau_1, \ldots, \tau_n) \to \mathbf{0})[c/\beta]}$$

$$\frac{\Delta; \Gamma \vdash v : \tau' \quad \Delta \vdash \tau' = \tau : \mathtt{Type}}{\Delta; \Gamma \vdash v : \tau}$$

Figure 3: Static Semantics: Values

$$\begin{array}{llll}
\textit{small values} & v & ::= & x \mid i \mid \mathcal{S}(i) \mid v[c] \mid \\
 & & & \mathtt{fix} f[\Delta \mid C](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).\iota \\
\textit{instructions} & \iota & ::= & \mathtt{new}\ \rho, x, i; \iota \mid \mathtt{free}\ v; \iota \mid \\
 & & & \mathtt{let}\ x = v.i; \iota \mid v_1.i := v_2; \iota \mid \\
 & & & \mathtt{case}\ v\ (\mathtt{inl} \Rightarrow \iota_1 \mid \mathtt{inr} \Rightarrow \iota_2) \mid \\
 & & & v(v_1, \ldots, v_n) \mid \mathtt{halt}\ v \mid \mathtt{coerce}(\gamma); \iota \\
\textit{coercions} & \gamma & ::= & \mathtt{union}_{\tau_1 \cup \tau_2}(\eta) \mid \\
 & & & \mathtt{roll}_{\mathtt{rec}\,\alpha\,(\Delta).\tau\,(c_1, \ldots, c_n)}(\eta) \mid \\
 & & & \mathtt{unroll}(\eta) \mid \\
 & & & \mathtt{pack}_{[c_1, \ldots, c_n \mid C]\mathtt{as}\,\exists[\Delta \mid C].\tau}(\eta) \mid \\
 & & & \mathtt{unpack}\ \Delta\ (\eta)
\end{array}$$

Figure 2: Term Structure: Syntax

general type *int* and singletons are given the specific type $\mathcal{S}(i)$.

Functions are considered small for the purposes of this paper; we will not concern ourselves with the problem of collecting function closures here.[3] Functions may be recursive and contain a specification of the polymorphic variables Δ, the requirements on the store C and the types of the parameters. These preconditions are used to type the instruction sequence that forms the body of the function. The value $v[c]$ denotes type application of the polymorphic function v to type constructor c. We often abbreviate successive type applications $v[c_1] \cdots [c_n]$ by $v[c_1, \ldots, c_n]$. Later, when we give an operational semantics for the language (section 5), we will add other small values, including pointers, but these objects are not manipulated by programmers—they only appear during run time evaluation of programs—and so we omit them for now.

The typing judgements for small values have the form $\Delta; \Gamma \vdash v : \tau$ where Γ is a finite partial map from value variables to small types. The rules are mostly standard and are presented in figure 3.

[3]Programmers may explicitly construct their own closures using the existential types we provide to hide the type of the closure environment [25, 29]. We do not closure convert our code here as it would serve only to complicate the discussion.

3.2 Instructions

Figure 4 presents the typing rules for the instructions. The judgement $\Delta; C; \Gamma \vdash \iota$ states that in type context Δ, a store described by C and value context Γ, the instruction sequence ι is well-formed.

Memory Management Instructions The principle interest of the language is the typing of memory management instructions. Operationally, the $\mathtt{new}\ \rho, x, i$ instruction allocates a memory block of size i at a fresh location and substitutes the location for ρ and a pointer to that location for x in the remaining instructions.[4] This operation is modeled in the type system by extending the store description with a memory type of length i. Initially, the fields of the memory block are filled with uninitialized junk. Once a block has been allocated, it may be operated on by accessor functions $\mathtt{let}\ x = v_1.i$ and $v_1.i := v_2$, which project from or store into the i^{th} field of v_1. The projection operation is well-formed if v_1 is a pointer to some location η and that location contains a object with type $\langle \tau_1, \ldots, \tau_n \rangle$ (where i is less than n). In this case, the remaining instructions ι must be well-formed given the additional assumption that x has type τ_i. The update operation is similar in that v_1 must be a pointer to a location containing a memory block. However, the remaining instructions are verified in a context where the type of the memory block has changed: The i^{th} field has type τ where τ is the type of the object being stored into that location, but is otherwise unconstrained. Although surprising at first, this rule is sound because the constraints behave linearly. Despite the fact that the type of a memory block at a location changes, each location can only appear once in the domain of a store type and therefore there is no opportunity to introduce inconsistencies. Constraints such as $\{\eta \mapsto \tau\} \otimes \{\eta \mapsto \tau'\}$ will never describe a well-formed store. The instruction $\mathtt{free}\ v$ deallocates the memory block pointed to by v. This effect is reflected in the typing rule for free by requiring that the remaining instructions be well-formed in a context C' that does not include the location η.

[4]For the purposes of alpha-conversion, ρ and x are considered bound by this instruction.

$\boxed{\Delta; C; \Gamma \vdash \iota}$

$$\frac{\Delta, \rho{:}\mathtt{Loc}; C \otimes \{\rho \mapsto \overbrace{\langle \mathit{junk}, \ldots, \mathit{junk} \rangle}^{i}\}; \Gamma, x{:}\mathit{ptr}(\rho) \vdash \iota}{\Delta; C; \Gamma \vdash \mathtt{new}\ \rho, x, i; \iota} \quad (x \notin \mathit{Dom}(\Gamma), \rho \notin \mathit{Dom}(\Delta))$$

$$\frac{\Delta; \Gamma \vdash v : \mathit{ptr}(\eta) \quad \Delta \vdash C = C' \otimes \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} : \mathtt{Store} \quad \Delta; C'; \Gamma \vdash \iota}{\Delta; C; \Gamma \vdash \mathtt{free}\ v; \iota}$$

$$\frac{\Delta; \Gamma \vdash v : \mathit{ptr}(\eta) \qquad \Delta \vdash C = C' \otimes \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} : \mathtt{Store} \qquad \Delta; C; \Gamma, x{:}\tau_i \vdash \iota}{\Delta; C; \Gamma \vdash \mathtt{let}\ x = v.i; \iota} \begin{pmatrix} 1 \leq i \leq n \\ x \notin \mathit{Dom}(\Gamma) \end{pmatrix}$$

$$\frac{\begin{array}{cc} \Delta; \Gamma \vdash v_1 : \mathit{ptr}(\eta) & \Delta \vdash C = C' \otimes \{\eta \mapsto \langle \tau_1, \ldots, \tau_i, \ldots, \tau_n \rangle\} : \mathtt{Store} \\ \Delta; \Gamma \vdash v_2 : \tau & \Delta; C' \otimes \{\eta \mapsto \langle \tau_1, \ldots, \tau, \ldots, \tau_n \rangle\}; \Gamma \vdash \iota \end{array}}{\Delta; C; \Gamma \vdash v_1.i := v_2; \iota} \quad (1 \leq i \leq n)$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash v : \mathit{ptr}(\eta) \qquad \Delta \vdash C = C' \otimes \{\eta \mapsto \tau_1 \cup \tau_2\} : \mathtt{Store} \\ \Delta \vdash \tau_1 = \exists[\Delta'_1 \mid C'_1]. \cdots \exists[\Delta'_j \mid C'_j].\langle \mathcal{S}(1), \tau'_1, \ldots, \tau'_k \rangle : \mathtt{Type} \\ \Delta \vdash \tau_2 = \exists[\Delta''_1 \mid C''_1]. \cdots \exists[\Delta''_m \mid C''_m].\langle \mathcal{S}(2), \tau''_1, \ldots, \tau''_n \rangle : \mathtt{Type} \\ \Delta; C' \otimes \{\eta \mapsto \tau_1\}; \Gamma \vdash \iota_1 \qquad \Delta; C' \otimes \{\eta \mapsto \tau_2\}; \Gamma \vdash \iota_2 \end{array}}{\Delta; C; \Gamma \vdash \mathtt{case}\ v\ (\mathtt{inl} \Rightarrow \iota_1 \mid \mathtt{inr} \Rightarrow \iota_2)}$$

$$\frac{\Delta; \Gamma \vdash v : \forall[\cdot \mid C].(\tau_1, \ldots, \tau_n) \to \mathbf{0} \qquad \Delta; \Gamma \vdash v_1 : \tau_1 \qquad \cdots \qquad \Delta; \Gamma \vdash v_n : \tau_n}{\Delta; C; \Gamma \vdash v(v_1, \ldots, v_n)}$$

$$\frac{\Delta; \Gamma \vdash v : \mathit{int}}{\Delta; C; \Gamma \vdash \mathtt{halt}\ v}$$

$$\frac{\Delta; C \vdash \gamma \Longrightarrow \Delta'; C' \quad \Delta'; C'; \Gamma \vdash \iota}{\Delta; C; \Gamma \vdash \mathtt{coerce}(\gamma); \iota}$$

Figure 4: Static Semantics: Instructions

As a warm-up example, consider the process of allocating and initializing a pair of pairs, where the deeper pair is aliased. The comments on the right-hand side present a portion of the type checking context after each program point.

$$\begin{array}{ll}
\mathtt{new}\ \rho_x, x, 2; & \%\ x{:}\mathit{ptr}(\rho_x) \\
& \%\ \{\rho_x \mapsto \langle \mathit{junk}, \mathit{junk} \rangle\} \\
\\
\mathtt{new}\ \rho_y, y, 2; & \%\ x{:}\mathit{ptr}(\rho_x), y{:}\mathit{ptr}(\rho_y) \\
& \%\ \{\rho_x \mapsto \langle \mathit{junk}, \mathit{junk} \rangle\} \otimes \\
& \%\ \{\rho_y \mapsto \langle \mathit{junk}, \mathit{junk} \rangle\} \\
\\
x.1 := y; & \%\ x{:}\mathit{ptr}(\rho_x), y{:}\mathit{ptr}(\rho_y) \\
& \%\ \{\rho_x \mapsto \langle \mathit{ptr}(\rho_y), \mathit{junk} \rangle\} \otimes \\
& \%\ \{\rho_y \mapsto \langle \mathit{junk}, \mathit{junk} \rangle\} \\
\\
x.2 := y; & \%\ x{:}\mathit{ptr}(\rho_x), y{:}\mathit{ptr}(\rho_y) \\
& \%\ \{\rho_x \mapsto \langle \mathit{ptr}(\rho_y), \mathit{ptr}(\rho_y) \rangle\} \otimes \\
& \%\ \{\rho_y \mapsto \langle \mathit{junk}, \mathit{junk} \rangle\} \\
\vdots &
\end{array}$$

At each update operation, the type checker verifies that x has a pointer type and modifies the type of x's memory block

$$\boxed{\Delta; C \vdash \gamma \Longrightarrow \Delta'; C'}$$

$$\frac{\Delta \vdash C = C' \otimes \{\eta \mapsto \tau_i\} : \mathtt{Store} \qquad \Delta \vdash \tau_1 : \mathtt{Type} \qquad \Delta \vdash \tau_2 : \mathtt{Type}}{\Delta; C \vdash \mathtt{union}_{\tau_1 \cup \tau_2}(\eta) \Longrightarrow \Delta; C' \otimes \{\eta \mapsto \tau_1 \cup \tau_2\}} \text{ (for } i = 1 \text{ or } 2)$$

$$\frac{\Delta \vdash \tau = (\mathtt{rec}\,\alpha\,(\Delta').\tau')\,(c_1, \ldots, c_n) : \mathtt{Type} \qquad \Delta \vdash C = C' \otimes \{\eta \mapsto \tau'[\mathtt{rec}\,\alpha\,(\Delta').\tau'/\alpha][c_1, \ldots, c_n/\Delta']\} : \mathtt{Store}}{\Delta; C \vdash \mathtt{roll}_\tau(\eta) \Longrightarrow \Delta; C' \otimes \{\eta \mapsto \tau\}}$$

$$\frac{\Delta \vdash C = C' \otimes \{\eta \mapsto \tau\} : \mathtt{Store} \qquad \Delta \vdash \tau = (\mathtt{rec}\,\alpha\,(\Delta').\tau')\,(c_1, \ldots, c_n) : \mathtt{Type}}{\Delta; C \vdash \mathtt{unroll}(\eta) \Longrightarrow \Delta; C' \otimes \{\eta \mapsto \tau'[\mathtt{rec}\,\alpha\,(\Delta').\tau'/\alpha][c_1, \ldots, c_n/\Delta']\}}$$

$$\frac{\begin{array}{c}\Delta' = \beta_1{:}\kappa_1, \ldots, \beta_n{:}\kappa_n \qquad \cdot \vdash c_i : \kappa_i \quad (\text{for } 1 \leq i \leq n) \\ \Delta \vdash C = C'' \otimes \{\eta \mapsto \tau[c_1, \ldots, c_n/\Delta']\} \otimes C'[c_1, \ldots, c_n/\Delta'] : \mathtt{Store}\end{array}}{\Delta; C \vdash \mathtt{pack}_{[c_1, \ldots, c_n \mid C'[c_1, \ldots, c_n/\Delta']]\,\mathtt{as}\,\exists[\Delta' \mid C'].\tau}(\eta) \Longrightarrow \Delta; C'' \otimes \{\eta \mapsto \exists[\Delta' \mid C'].\tau\}}$$

$$\frac{\Delta \vdash C = C'' \otimes \{\eta \mapsto \exists[\Delta' \mid C'].\tau\} : \mathtt{Store}}{\Delta; C \vdash \mathtt{unpack}\,\Delta'\,(\eta) \Longrightarrow \Delta, \Delta'; C'' \otimes \{\eta \mapsto \tau\} \,@\, C'}$$

Figure 5: Static Semantics: Coercions

accordingly. The interesting aspect of this example is that after the fourth instruction in the sequence, there are three aliases to the second memory block: the variable y and the two components of x. We can see this is true, simply by counting the number of occurences of $ptr(\rho_y)$ in the type checking context. Each occurence must alias the others. All three aliases are accurately tracked in the type system and any of them may be used. When we are finished with the data structure, we may deallocate it:

```
free y;   % x:ptr(ρx), y:ptr(ρy)
          % {ρx ↦ ⟨ptr(ρy), ptr(ρy)⟩}

free x;   % x:ptr(ρx), y:ptr(ρy)
          % ∅
```

After deallocation, we are left with two dangling pointers, one to the deallocated location ρ_x and a second to the deallocated location ρ_y. Fortunately, the type checker prevents these pointers from being derefenced. For example, if the next instruction in the sequence was the projection $\mathtt{let}\ z = x.1$, it will fail since there is no constraint C' such that $\emptyset = C' \otimes \{\rho_x \mapsto \langle\tau_1, \ldots, \tau_n\rangle\}$.

Control-flow Instructions The typing of the case expression is somewhat unusual. Operationally, case checks the first field of the memory block in the location pointed to by a value v. If the first field is a 1, execution continues with the first instruction sequence, and if it is a 2, execution continues with the second instruction sequence. However the memory type constructor $\langle\cdots\rangle$ will not be the top-most type constructor (otherwise, the case would be unnecessary). The type system expects a union type to be top-most and each alternative may contain some number (possibly zero) of existential quantifiers to abstract the store encapsulated in that alternative. The underlying memory value must have either tag 1 or tag 2 in its first field.

Because the language has been defined in continuation-passing style, all instruction sequences are either terminated by a function call $v(v_1, \ldots, v_n)$ or a call to the terminal continuation `halt`, which requires an integer argument. Function calls are well-formed if the polymorphic function v has been fully instantiated, the constraints in the current context equal the constraints required by the function, and the argument types match the types of the function parameters.

3.3 Coercions

The last instruction $\mathtt{coerce}(\gamma)$ applies a typing coercion to the store. Coercions, unlike the other instructions are for type-checking purposes only. Intuitively, coercions may be erased before executing a program and the run-time behaviour will not be affected. The judgement form $\Delta; C \vdash \gamma \Longrightarrow \Delta'; C'$ indicates that a coercion is well-formed, extends the type context to Δ', and produces new store constraints C'. These judgements are presented in figure 5.

Each coercion operates on a particular store location η. The `union` coercion lifts the object at η into a union type and the `roll`/`unroll` coercions witness the isomorphism between a recursive type and its unfolding. The coercion $\mathtt{pack}_{[c_1, \ldots, c_n \mid C'[c_1, \ldots, c_n/\Delta']]\,\mathtt{as}\,\exists[\Delta' \mid C'].\tau}(\eta)$ introduces an existential type by hiding the type constructors $c_1, \ldots, c_n$ and encapsulating the store described by $C'[c_1, \ldots, c_n/\Delta']$. The `unpack` coercion eliminates an existential type, augments the current constraints with the encapsulated C', and extends the type context Δ with Δ', the hidden type constructors.

4 Applications

In this section, we show how our language can be used to encode two common programming patterns, the destination-passing style pattern, which constructs data structures efficiently and the Deutsch-Schorr-Waite or "link-reversal" patterns, which traverse data structures using minimal additional space.

4.1 Destination-Passing Style

The *destination-passing style* (DPS) transformation detects a certain form of "almost-tail-recursive" function and automatically transforms it into an efficient properly tail-recursive function. The transformation improves many functional programs significantly, leading a number researchers to study the problem in depth [41, 22, 7, 24]. Our contribution is to provide a type system that can be used in a type-preserving compiler and is capable of verifying that the code resulting from the transformation is safe.

Append is the canonical example of a function suitable for DPS:

```
fun append (xs,ys) =
  case xs of
    [] -> ys
  | hd :: tl -> hd :: append (tl,ys)
```

Here, the second-last operation in the second arm of the case is a function call and the last operation constructs a cons cell. If the two operations were inverted, we would have an efficient tail-recursive function. In DPS, the function allocates a cons cell before the recursive call and passes the partially uninitialized value to the function, which computes its result and fills in the uninitialized part of the data structure. If the input list `xs` is linear, it will not be used in the future. In this case, it is possible to further optimize the program by reusing the input list cells for the output list. Our example performs both of these optimizations.

Before presenting the code for the optimized function, we will need to define a number of abbreviations. Such abbreviations not only aid readability, but also help compress typing information in a compiler [14]. First, recall the type of integer lists $List$ and their unrolling $List'$:

$$
\begin{array}{lcl}
List & = & \mu\alpha.\langle\mathcal{S}(1)\rangle \cup \exists[\rho \mid \{\rho \mapsto \alpha\}].\langle\mathcal{S}(2), int, ptr(\rho)\rangle \\
List' & = & \langle\mathcal{S}(1)\rangle \cup \exists[\rho \mid \{\rho \mapsto List\}].\langle\mathcal{S}(2), int, ptr(\rho)\rangle
\end{array}
$$

Given these list definitions, it will be useful to define the following composite coercion.

$$
\begin{array}{l}
\texttt{rollList}\,\rho_1\,\texttt{packing}\,\rho_2 = \\
\quad \texttt{pack}_{[\rho_2 \mid \{\rho_2 \mapsto List\}]\texttt{as}\,\exists[\rho_2 \mid \{\rho_2 \mapsto List\}].\langle\mathcal{S}(2), int, ptr(\rho_2)\rangle}(\rho_1); \\
\quad \texttt{union}_{List'}(\rho_1); \\
\quad \texttt{roll}_{List}(\rho_1)
\end{array}
$$

This coercion operates on a portion of the store with shape $\{\rho_1 \mapsto \langle\mathcal{S}(2), int, ptr(\rho_2)\rangle\} \otimes \{\rho_2 \mapsto List\}$. It packs up ρ_2 into an existential around ρ_1, lifts the resultant object up to a union type and finally rolls it up, producing a store with the shape $\{\rho_1 \mapsto List\}$.

The function $append'$, presented in figure 6, implements the inner loop of the optimized append function. A wrapper function must check for the case that the input list is empty. If not, it passes two pointers to the beginning of the first list (aliases of one another) to $append'$ for parameters $prev$ and $start$. It also passes a pointer to the second element in that list for parameter xs and a pointer to the second list for parameter ys. Notice that the contents of location ρ_s are not described by the aliasing constraints. On the first iteration of the loop ρ_s is an alias of ρ_p and on successive iterations, it abstracted by ϵ. However, these facts are not explicit in the type structure and therefore ρ_s cannot be used during any iteration of the loop ($cont$ will be aware that ρ_s equals ρ_p and may use the resultant list).

$$
\begin{array}{l}
\texttt{fix}\,append'\,[\epsilon, \rho_{xs}, \rho_{ys}, \rho_p, \rho_s \mid \\
\quad \epsilon \otimes \{\rho_p \mapsto \langle\mathcal{S}(2), int, ptr(\rho_{xs})\rangle, \rho_{xs} \mapsto List, \rho_{ys} \mapsto List\}]. \\
\quad (xs : ptr(\rho_{xs}), ys : ptr(\rho_{ys}), prev : ptr(\rho_p), start : ptr(\rho_s), \\
\qquad cont : \tau_c[\epsilon, \rho_p, \rho_s]). \\
\quad \texttt{unroll}(\rho_{xs}); \\
\quad \texttt{case}\,xs \\
\qquad (\ \texttt{inl} \Longrightarrow \\
\qquad\qquad \texttt{free}\,xs; \qquad\qquad\qquad\quad \texttt{\% 1.} \\
\qquad\qquad prev.3 := ys; \qquad\qquad\quad\ \ \texttt{\% 2.} \\
\qquad\qquad \texttt{rollList}\,\rho_p\,\texttt{packing}\,\rho_{ys}; \quad \texttt{\% 3.} \\
\qquad\qquad cont(start) \\
\qquad \mid \texttt{inr} \Longrightarrow \\
\qquad\qquad \texttt{unpack}\,\rho_{tl}\,(\rho_{xs}); \qquad\quad\ \texttt{\% 4.} \\
\qquad\qquad \texttt{let}\ tl = xs.3; \qquad\qquad\quad \texttt{\% 5.} \\
\qquad\qquad append' \\
\qquad\qquad\quad [\epsilon \otimes \{\rho_p \mapsto \langle\mathcal{S}(2), int, ptr(\rho_{xs})\rangle\}, \rho_{tl}, \rho_{ys}, \rho_{xs}, \rho_s] \\
\qquad\qquad\quad (tl, ys, xs, start, cont')) \\
\\
\texttt{where}\ \ \tau_c[\epsilon, \rho_p, \rho_s] = \forall[\cdot \mid \epsilon \otimes \{\rho_p \mapsto List\}].(ptr(\rho_s)) \to \mathbf{0}
\end{array}
$$

Figure 6: Optimized Append

The first place to look to understand this code is at the aliasing constraints, which act as a loop invariant. Reading the constraints in the type from left to right reveals that the function expects a store with some unknown part (ϵ) as well as a known part. The known part contains a cons cell at location ρ_p that is linked to a $List$ in location ρ_{xs}. Independent of either of these objects is a third location, ρ_{ys}, which also contains a $List$.

The first instruction in the function unrolls the recursive type of the object at ρ_{xs} to reveal that it is a union and can be eliminated by a case statement. In the first branch of the case, xs must point to null. The code frees the null cell, resulting in a store at program point `1` that can be described by the constraints $\epsilon \otimes \{\rho_p \mapsto \langle\mathcal{S}(2), int, ptr(\rho_{xs})\rangle\} \otimes \{\rho_{ys} \mapsto List\}$. Observe that the cons cell at ρ_p contains a dangling pointer to memory location ρ_{xs}, the location that has just been freed and no longer appears in the constraints. Despite the dangling pointer, the code is perfectly safe: The typing rules prevent the pointer from being used.

Next, the second list ys is banged into the cons cell at ρ_p. Hence, at program point `2`, the store has a shape described by $\epsilon \otimes \{\rho_p \mapsto \langle\mathcal{S}(2), int, ptr(\rho_{ys})\rangle\} \otimes \{\rho_{ys} \mapsto List\}$. The type of the cons cell at ρ_p is different here than at `1`, reflecting the new link structure of store. The tail of the cell no longer points to location ρ_{xs}, but to ρ_{ys} instead. After packing and rolling using the composite coercion, the store can be described by $\epsilon \otimes \{\rho_p \mapsto List\}$. This shape equals the shape expected by the continuation (see the definition of τ_c), so the function call is valid.

In the second branch of the case, xs must point to a cons cell. The existential containing the tail of the list is unpacked and at program point 4, the store has shape $\epsilon \otimes \{\rho_p \mapsto \langle \mathcal{S}(2), int, ptr(\rho_{xs})\rangle\} \otimes \{\rho_{xs} \mapsto \langle \mathcal{S}(2), int, ptr(\rho_{tl})\rangle\} \otimes \{\rho_{tl} \mapsto List\} \otimes \{\rho_{ys} \mapsto List\}$. It is now possible to project the tail of xs. To complete the loop, the code uses polymorphic recursion. At the end of the second branch, the constraint variable ϵ for the next iteration of the loop is instantiated with the current ϵ and the contents of location ρ_p, hiding the previous node in the list. The location variables ρ_{xs} and ρ_p are instantiated to reflect the shift to the next node in the list. The locations ρ_{ys} and ρ_s are invariant around the loop and therefore are instantiated with themselves.

The last problem is how to define the continuation $cont'$ for the next iteration. The function should be tail-recursive, so we would like to use the continuation $cont$. However, close inspection reveals that the next iteration of append requires a continuation with type $\tau_c[\epsilon \otimes \{\rho_p \mapsto \langle \mathcal{S}(2), int, ptr(\rho_{xs})\rangle\}, \rho_{xs}, \rho_s]$ but that the continuation $cont$ has type $\tau_c[\epsilon, \rho_p, \rho_s]$. The problem is that this iteration of the recursion has unrolled and unpacked the recursive data structure pointed to by xs, but before "returning" by calling the continuation, the list must be packed and rolled back up again. Therefore, the appropriate definition of $cont'$ is $cont \circ (\texttt{rollList}\,\rho_p\,\texttt{packing}\,\rho_{xs})$. Once the continuation packs ρ_{xs} and rolls the contents of location ρ_p into a $List$, the constraints satisfy the requirements of the continuation $cont$. Semantically, $cont'$ is equivalent to the following function.

$\quad\texttt{fix}\,\_[\cdot \mid \epsilon \otimes \{\rho_p \mapsto \langle \mathcal{S}(2), int, ptr(\rho_{xs})\rangle\}\{\rho_{xs} \mapsto List\}]$
$\quad\quad(start{:}ptr(\rho_s)).$
$\quad\quad\texttt{rollList}\,\rho_p\,\texttt{packing}\,\rho_{xs};$
$\quad\quad cont(start)$

However, because coercions can be erased before running a program, it is simple to arrange for $cont'$ to be implemented by $cont$.

4.2 Deutsch-Schorr-Waite Algorithms

Deutsch-Schorr-Waite or "link reversal" algorithms, are well-known algorithms for traversing data structures while incurring minimal additional space overhead. These algorithms were first developed for executing the mark phase of a garbage collector [35]. During garbage collection, there is little or no extra space available for storing control information, so minimizing the overhead of the traversal is a must. Recent work by Sobel and Friedman [37] has shown how to automatically transform certain continuation-passing style programs, those generated by *anamorphisms* [23], into link-reversal algorithms. Here we give an example how to encode a link-reversal algorithm in our calculus.

For this application, we will use the definition of trees from section 2.

$$\begin{array}{l} Tree = \\ \quad \mu\alpha.\langle \mathcal{S}(1)\rangle \cup \exists[\rho_L, \rho_R \mid \{\rho_L \mapsto \alpha, \rho_R \mapsto \alpha\}]. \\ \quad\quad\quad\quad \langle \mathcal{S}(2), ptr(\rho_L), ptr(\rho_R)\rangle \end{array}$$

$$\begin{array}{l} Tree' = \\ \quad \langle \mathcal{S}(1)\rangle \cup \exists[\rho_L, \rho_R \mid \{\rho_L \mapsto Tree, \rho_R \mapsto Tree\}]. \\ \quad\quad\quad\quad \langle \mathcal{S}(2), ptr(\rho_L), ptr(\rho_R)\rangle \end{array}$$

The code for the algorithm appears in figure 7. The trick to the algorithm is that when recursing into the left subtree, it uses space normally reserved for a pointer to that subtree to point back to the parent node. Similarly, when recursing into the right subtree, it uses the space for the

```
% Traverse a tree node
letrec walk[ε, ρ1, ρ2 | ε ⊗ {ρ1 ↦ Tree}]
          (t : ptr(ρ1), up : ptr(ρ2), cont : τc[ε, ρ1, ρ2]).
  unroll(ρ1);
  case t of
     ( inl ⟹
        union_Tree'(ρ1);
        roll_Tree(ρ1);
        cont(t, up)
     | inr ⟹
        unpack ρL, ρR (ρ1);
        % store cont in tag position
        t.1 := cont;
        let left = t.2;
        % store parent pointer as left subtree
        t.2 := up;
        walk[ε⊗
              {ρ1 ↦ ⟨τc[ε, ρ1, ρ2], ptr(ρ2), ptr(ρR)⟩}⊗
              {ρR ↦ Tree}, ρL, ρ1]
            (left, t, rwalk[ε, ρ1, ρ2, ρL, ρR]))

% Walk the right-hand subtree
and rwalk[ε, ρ1, ρ2, ρL, ρR | ε⊗
        {ρ1 ↦ ⟨τc[ε, ρ1, ρ2], ptr(ρ2), ptr(ρR)⟩}⊗
        {ρL ↦ Tree}⊗
        {ρR ↦ Tree}]
        (left : ptr(ρL), t : ptr(ρ1)).
  let up = t.2;
  % restore left subtree
  t.2 := left;
  let right = t.3;
  % store parent pointer as right subtree
  t.3 := up;
  walk[ε⊗
        {ρ1 ↦ ⟨τc[ε, ρ1, ρ2], ptr(ρL), ptr(ρ2)⟩}⊗
        {ρL ↦ Tree}, ρR, ρ1]
      (right, t, finish[ε, ρ1, ρ2, ρL, ρR])

% Reconstruct tree node and return
and finish[ε, ρ1, ρ2, ρL, ρR | ε⊗
        {ρ1 ↦ ⟨τc[ε, ρ1, ρ2], ptr(ρL), ptr(ρ2)⟩}⊗
        {ρL ↦ Tree}⊗
        {ρR ↦ Tree}]
        (right : ptr(ρR), t : ptr(ρ1)).
  let up = t.3;
  % restore right subtree
  t.3 := right;
  let cont = t.1;
  % restore tag
  t.1 := S(2);
  pack_{ρL,ρR}(ρ1);
  union_Tree'(ρ1);
  roll_Tree(ρ1);
  cont(t, up)

where τc[ε, ρ1, ρ2] =
    ∀[· | ε ⊗ {ρ1 ↦ Tree}].(ptr(ρ1), ptr(ρ2)) → 0
```

Figure 7: Deutsch-Schorr-Waite tree traversal with constant space overhead

right pointer. In both cases, it uses the tag field of the data structure to store a continuation that knows what to do next (recurse into right subtree or follow the parent pointers back up the tree). Before ascending back up out of the tree, the algorithm restores the link structure to a proper tree shape and the type system checks this is done properly. Notice that all of the functions and continuations are closed, so there is no stack hiding in the closures.

5 Operational Semantics and Type Soundness

In this section, we define the syntax and static semantics of the values manipulated at run-time, including pointers, memory blocks and the store and give an operational semantics for the language. The type system is sound with respect to this semantics.

5.1 Run-time Values

First, we extend the class of small values to include the junk object $\mathtt{junk}$ and pointers $\mathtt{ptr}(\ell)$. Next, we define a class of stored values (s) that include memory blocks $\langle v_1,\ldots,v_n\rangle$ and witnessed values $\varsigma(s)$. Witnessed values are introduced by coercions. There is one witness for each of the $\mathtt{roll}$, $\cup$ and $\mathtt{pack}$ coercions.

$$\begin{array}{lcll} \textit{small values} & v & ::= & \cdots \mid \mathtt{junk} \mid \mathtt{ptr}(\ell) \\ \textit{stored values} & s & ::= & \langle v_1,\ldots,v_n\rangle \mid \varsigma(s) \\ \textit{witnesses} & \varsigma & ::= & \mathtt{union}_{\tau_1\cup\tau_2} \mid \\ & & & \mathtt{pack}_{[c_1,\ldots,c_n \mid S]\mathtt{as}\exists[\Delta\mid C].\tau} \mid \\ & & & \mathtt{roll}_{(\mathtt{rec}\,\alpha\,(\Delta).\tau)\,(c_1,\ldots,c_n)} \end{array}$$

The well-formedness of junk and pointers is established using the same judgement form as other values. Stored values use the judgement $\vdash s : \tau$. Since stored values only appear at run time, when type and value variables have been substituted away, they are always checked in an empty context. Figure 8 formalizes these two judgements.

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

$$\frac{}{\Delta;\Gamma\vdash \mathtt{ptr}(\ell) : \mathit{ptr}(\ell)}$$

$$\frac{}{\Delta;\Gamma\vdash \mathtt{junk} : \mathit{junk}}$$

$$\boxed{\vdash s:\tau}$$

$$\frac{\cdot;\cdot\vdash v_1:\tau_1 \quad \cdots \quad \cdot;\cdot\vdash v_n:\tau_n}{\vdash \langle v_1,\ldots,v_n\rangle : \langle \tau_1,\ldots,\tau_n\rangle}$$

$$\frac{\cdot\vdash \tau_1\cup\tau_2 : \mathtt{Type} \qquad \vdash s:\tau_1 \quad \text{or} \quad \vdash s:\tau_2}{\vdash \mathtt{union}_{\tau_1\cup\tau_2}(s) : \tau_1\cup\tau_2}$$

$$\frac{\begin{array}{c}\cdot\vdash \tau = (\mathtt{rec}\,\alpha\,(\Delta).\tau')\,(c_1,\ldots,c_n) : \mathtt{Type} \\ \vdash s : \tau'[\mathtt{rec}\,\alpha\,(\Delta).\tau'/\alpha][c_1,\ldots,c_n/\Delta]\end{array}}{\vdash \mathtt{roll}_\tau(s) : \tau}$$

$$\frac{\begin{array}{c}\Delta = \beta_1{:}\kappa_1,\ldots,\beta_n{:}\kappa_n \qquad \cdot\vdash c_i:\kappa_i \quad (\text{for } 1\le i\le n) \\ \vdash S : C[c_1,\ldots,c_n/\Delta] \qquad \vdash s:\tau[c_1,\ldots,c_n/\Delta]\end{array}}{\vdash \mathtt{pack}_{[c_1,\ldots,c_n\mid S]\mathtt{as}\exists[\Delta\mid C].\tau}(s) : \exists[\Delta\mid C].\tau}$$

Figure 8: Static Semantics: Run-time Values

5.2 Store and Program Typing

The pack coercion encapsulates a portion of the store, S, which is a finite partial mapping from concrete locations to stored values. We treat stores equivalent up to reordering of their elements and use the notation $S\{\ell\mapsto s\}$ to denote the extension of S with the mapping $\{\ell\mapsto s\}$. The notation is undefined if $\ell\in \mathit{Dom}(S)$. The store well-formedness judgement is written $\vdash S : C$ and is given below.

$$\frac{\begin{array}{c} S = \{\ell_1\mapsto s_1,\ldots,\ell_n\mapsto s_n\} \\ \cdot\vdash C = \{\ell_1\mapsto \tau_1,\ldots,\ell_n\mapsto\tau_n\} : \mathtt{Store} \\ \vdash s_1:\tau_1 \quad \cdots \quad \vdash s_n:\tau_n \end{array}}{\vdash S : C}$$

There are no duplicate locations in the domain of a store (otherwise, it would not be a finite partial map). However, we will require a stronger property of stores to prove that program evaluation cannot get stuck. Informally, there can be no duplication of locations in the domain of the store or in any encapsulated store. We call this property *Global Uniqueness*.

Definition 1 (Global Uniqueness) $\mathtt{GU}(S)$ *if and only if there are no duplicate locations in* $\mathtt{L}(S)$.

Definition 2 (Global Store Locations) $\mathtt{L}(S)$ *is the multi-set given by the following definition.*

$$\begin{array}{l} \mathtt{L}(\{\ell_1\mapsto s_1,\ldots,\ell_n\mapsto s_n\}) \\ \quad = \{\ell_1,\ldots,\ell_n\}\uplus \mathtt{L}(s_1)\uplus\cdots\uplus \mathtt{L}(s_n) \\ \mathtt{L}(\mathtt{pack}_{[c_1,\ldots,c_n\mid S]\mathtt{as}\tau}(s)) \\ \quad = \mathtt{L}(S)\uplus \mathtt{L}(s) \end{array}$$

$$\begin{array}{l} \mathtt{L}(x) = \mathtt{L}(x_1)\uplus\cdots\uplus \mathtt{L}(x_n) \\ \quad \textit{for any other term construct } x \\ \quad \textit{where } x_1,\ldots,x_n \textit{ are the subcomponents of } x. \end{array}$$

A program is a store paired with an instruction stream. A program is well-formed, written $\vdash (S,\iota)$, under the following circumstances.

Definition 3 (Well-formed Program) $\vdash (S,\iota)$ *iff*

1. *The store adheres to global uniqueness* $\mathtt{GU}(S)$*.*
2. *There exists constraints C such that $\vdash S : C$.*
3. *The instructions are well-formed with the given constraints: $\cdot;C;\cdot\vdash\iota$.*

5.3 Operational Semantics

The small-step operational semantics for the language is given by a function $P\longmapsto_P P'$. The majority of the operational rules are entirely standard and formalize the intuitive rules described earlier in the paper. The operational rule for the coerce instruction depends upon a separate semantics for coercions that has the form $S\longmapsto_\gamma S',\theta$ where θ is a

$\boxed{(S,\iota) \longmapsto_P (S,\iota)}$

$$
\begin{array}{lcl}
(S, \texttt{new}\ \rho, x, i; \iota) & \longmapsto_P & (S\{\ell \mapsto s\}, \iota[\ell/\rho][\texttt{ptr}(\ell)/x]) \\
\quad \text{where } \ell \notin S, \iota \text{ and } s = \overbrace{\langle \texttt{junk}, \ldots, \texttt{junk} \rangle}^{i} & & \\
(S\{\ell \mapsto \langle v_1, \ldots, v_n \rangle\}, \texttt{free ptr}(\ell); \iota) & \longmapsto_P & (S, \iota) \\
(S\{\ell \mapsto \langle v_1, \ldots, v_n \rangle\}, \texttt{let}\ x = \texttt{ptr}(\ell).i; \iota) & \longmapsto_P & (S\{\ell \mapsto \langle v_1, \ldots, v_n \rangle\}, \iota[v_i/x]) \\
\quad \text{where } 1 \leq i \leq n & & \\
(S\{\ell \mapsto \langle v_1, \ldots, v_i, \ldots, v_n \rangle\}, \texttt{ptr}(\ell).i := v'; \iota) & \longmapsto_P & (S\{\ell \mapsto \langle v_1, \ldots, v', \ldots, v_n \rangle\}, \iota) \\
\quad \text{where } 1 \leq i \leq n & & \\
(S\{\ell \mapsto s\}, \texttt{case ptr}(\ell)\,(\texttt{inl} \Rightarrow \iota_1 \mid \texttt{inr} \Rightarrow \iota_2)) & \longmapsto_P & (S\{\ell \mapsto s'\}, \iota_i)
\end{array}
$$

$$
\begin{array}{rrcl}
 & i & = & 1 \text{ or } 2 \\
\text{where} & s & = & \texttt{union}_{\tau_1 \cup \tau_2}(\varsigma_1(\cdots \varsigma_m(\langle \mathcal{S}(i), v_1, \ldots, v_n \rangle) \cdots)) \\
 & s' & = & \varsigma_1(\cdots \varsigma_m(\langle \mathcal{S}(i), v_1, \ldots, v_n \rangle) \cdots)
\end{array}
$$

$$
\begin{array}{lcl}
(S, v(v_1, \ldots, v_n)) & \longmapsto_P & (S, \theta(\iota))
\end{array}
$$

$$
\begin{array}{rrcl}
 & v & = & v'[c_1, \ldots, c_m] \\
\text{where} & v' & = & \texttt{fix} f[\Delta \mid C](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).\iota \\
 & \theta & = & [c_1, \ldots, c_m/\Delta][v'/f][v_1, \ldots, v_n/x_1, \ldots, x_n]
\end{array}
$$

$$
\begin{array}{lcl}
(S, \texttt{coerce}(\gamma); \iota) & \longmapsto_P & (S', \theta(\iota)) \\
\quad \text{where } \gamma(S) \longmapsto_\gamma S', \theta & &
\end{array}
$$

Figure 9: Operational Semantics: Programs

substitution of type constructors for type constructors variables. Inspection of these rules reveals that coercions do not alter the association between locations and memory blocks; they simply insert witnesses that alter the typing derivation so that it is possible to prove a type soundness result. The rules for program and coercion operational semantics may be found in figures 9 and 10.

5.4 Type Soundness

We now have all the pieces necessary to state and prove that execution of a program in our language "can't get stuck." A *stuck program* is a program that is not in the terminal configuration $\texttt{halt}\ i$ and for which no operational rule applies.

Theorem 4 (Type Soundness)
If $\vdash (S,\iota)$ *and* $(S,\iota) \longmapsto_P^* (S',\iota')$ *then* (S',ι') *is not stuck.*

The proof itself uses standard Subject Reduction and Progress lemmas in the style popularized by Wright and Felleisen [44] and is mostly mechanical. Due to space limitations, it has not been included. See Walker's thesis [43] for details.

6 Discussion

Our alias type system has intermediate expressiveness when compared with other frameworks for reasoning about aliasing. It is more powerful than simple k-limiting approaches [18] as it provides the ability to represent recursive data structures, but much less powerful than recent approaches based on Hoare logic [32, 17]. In the following sections we explain some of the limitations of this work and discuss related research in more depth.

6.1 Limitations

There are at least three significant limitations of the work described in this paper:

1. A lack of *may-alias* constraints.
2. Limited support for union types.
3. Limited logic (no implication, disjunction, negation or equality predicates) for compile-time reasoning about store shapes.

The first limitation refers to the fact that no element in a store type may alias any other element of the store type. This property is what makes deallocation safe, but it also prevents us from writing many useful functions. For example, it is impossible to write a function of two or more arguments where each argument may or may not alias the others. It is also not possible to write a general graph or DAG in this language. In earlier work with Fred Smith [36], we describe how to add may-alias constraints to a simpler language and we have done this in our Typed Assembly Language implementation [38]. The disadvantage of these may-alias constraints is that destructive object manipulation, including deallocation, is disallowed.

The second limitation refers to the fact that the introduction and elimination rules for union types limit a programmer's choice of data layout. Unions always have the form:

$\boxed{\gamma(S) \longmapsto_\gamma S', \theta}$

$$
\begin{array}{lll}
\texttt{union}_{\tau_1 \cup \tau_2}(\ell)(S\{\ell \mapsto s\}) & \longmapsto_\gamma & S\{\ell \mapsto \texttt{union}_{\tau_1 \cup \tau_2}(s)\}, [\,] \\
\texttt{roll}_\tau(\ell)(S\{\ell \mapsto s\}) & \longmapsto_\gamma & S\{\ell \mapsto \texttt{roll}_\tau(s)\}, [\,] \\
\texttt{unroll}(\ell)(S\{\ell \mapsto \texttt{roll}_\tau(s)\}) & \longmapsto_\gamma & S\{\ell \mapsto s\}, [\,] \\
\texttt{pack}_{[c_1,\ldots,c_n|C]\texttt{as}\,\tau}(\ell)(S\{\ell \mapsto s\}S') & \longmapsto_\gamma & S\{\ell \mapsto \texttt{pack}_{[c_1,\ldots,c_n|S']\texttt{as}\tau}(s)\}, [\,] \\
\quad \text{where } C = \{\ell_1 \mapsto \tau_1, \ldots, \ell_m \mapsto \tau_m\} \text{ and } S' = \{\ell_1 \mapsto s_1, \ldots, \ell_m \mapsto s_m\} & & \\
\texttt{unpack}\,\Delta\,(\ell)(S\{\ell \mapsto \texttt{pack}_{[c_1,\ldots,c_n|S']\texttt{as}\exists[\Delta|C].\tau}(s)\}) & \longmapsto_\gamma & SS'\{\ell \mapsto s\}, [c_1, \ldots, c_n/\Delta]
\end{array}
$$

Figure 10: Operational Semantics: Coercions

$$
\begin{array}{c}
\exists[\Delta_1 \mid C_1]. \cdots \exists[\Delta_j \mid C_j].\langle \mathcal{S}(1), \tau_1, \ldots, \tau_k \rangle \\
\cup \\
\exists[\Delta'_1 \mid C'_1]. \cdots \exists[\Delta'_l \mid C'_l].\langle \mathcal{S}(2), \tau'_1, \ldots, \tau'_m \rangle
\end{array}
$$

The only way to eliminate a union is to case on the first component of the memory block. There are many other ways to represent sum types: We may want to separate the union tag from the data structure itself or we may not want a tag at all, relying instead on pointer equality tests or the ability to differentiate between nil and a pointer. In general, to separate the tag from the union type requires more dependency mechanisms than we have in this language. However, we can accomodate some additional data type representations by simply adding a new introduction and elimination forms—the type structure need not change at all. For instance an option type that is either a nil object or reference to τ may be encoded as $\mathcal{S}(0) \cup \langle \tau \rangle$. A test for zero eliminates this form of union. Regardless of which additional choices we make here, we cannot represent completely undescriminated unions; we must use some dynamic test to determine which element of a union a value inhabits before using that value.

The third limitation provides some interesting possibilities for future research. The type system described in this paper may thought of as a specialization of Xi and Pfenning's work [46, 45] to the problem of memory management. The store types are simply a linear conjunction of points-to predicates. Since the logic of stores contains no implication, negation or quantifiers and limited value dependencies, there are many properties of graphs that we cannot express. For example, our type system cannot encode "balanced" trees. It is worth investigating whether we can follow Xi and Pfenning's program and integrate a full first-order logic into the type system. Xi and Pfenning have already shown that they can encode complex invariants such as the red-black tree invariant. The disadvantage of this approach is that we would have to integrate a theorem prover into our language in order to decide the validity of the logical formulae.

6.2 Related Work

Our type system builds upon foundational work by other groups on syntactic control of interference [31], linear logic [13] and linear type systems in functional programming languages [20, 42, 1, 15, 3, 8, 40].

Our research also has much in common with efforts to define program logics for reasoning about aliasing [6, 9, 26, 32, 17]. In particular, if we view propositions as types, there are striking similarities with recent work by Reynolds [32] who builds on earlier research by Burstall [6]. Reynolds' logic employs a "spatial conjunction," which, like our $\otimes$ operator, joins two operands that depend upon disjoint portions of the store. Updating a single memory cell can alter at most one of the propositions joined by Reynolds' conjunction, making it possible to state simple Hoare-logic rules for memory allocation, dereference and update. Ishtiaq and O'Hearn [17] have further analyzed Reynolds' rules in the context of the logic of bunched implications. They give a slightly different set of Hoare rules and are able to prove that their rules generate weakest preconditions. They also introduce an operation for safe object deallocation. As mentioned earlier, one practical difference between these program logics and our type system is that for our system there is no need to implement a theorem prover to check the safety of programs. Consequently, we have found it straightforward to integrate alias types with our Typed Assembly Language implementation [38].

There are also similarities between our research and work on alias analysis techniques for imperative languages [18, 21, 11, 12, 33]. Our type system appears most closely related to the shape analysis developed by Sagiv, Reps, and Wilhelm (SRW) [33], which has also been used to develop sophisticated pointer logics [34, 4]. Although the precise relationship is currently unknown to us, it is clear that several of the key features that make SRW shape analysis more effective than similar alias analyses can be expressed in our type system. More specifically:

1. Unlike some other analyses, SRW shape nodes do not contain information about concrete locations or the site where the node was allocated. Our type system drops information about concrete locations using location polymorphism.

2. SRW shape nodes are named with the set of program variables that point to that node. Our type system can only label a node with a single name, but we are able to express the fact that a set of program variables point to that node using the same singleton type for each program variable in the set.

3. SRW shape nodes may be flagged as unshared. Linear types account for unshared shape nodes.

4. A single SRW summary node describes many memory blocks, but through the process of *materialization* a summary node may split off a new, separate shape node. Summary nodes may be represented as recursive types in our framework and materialization can be explained by the process of unrolling and unpacking a recursive and existential type.

One of the advantages to our approach is that our language makes it straightforward to create dependencies between functions and data using store or location polymorphism. For example, in our implementation of the Deutsch-Schorr-Waite algorithm, we manipulate continuations that know how to reconstruct a well-formed tree from the current heap structure and we are able to express this dependence in the type system. Explicit manipulation of continuations is necessary in sufficiently low-level typed languages such as Typed Assembly Language [27].

Several other authors have considered alternatives to pure linear type systems that increase their flexibility. For example, Kobayashi [19] extends standard linear types with data-flow information and Minamide [24] uses a linear type discipline to allow programmers to manipulate "data structures with a hole." Minamide's language allows users to write programs that are compiled into destination-passing style. However, Minamide's language is still quite high-level; he does not show how to verify explicit pointer manipulation. Moreover, neither of these type systems provide the ability to represent cyclic data structures.

Tofte, Talpin, and others [39, 5, 2] have explored the use of region-based memory management. In their work, objects are allocated into one of several *regions* of memory. When a region is deallocated, all the objects in that region are deallocated too. Region-based memory management performs extremely well in many circumstances, but unlike systems based on linear types, space is not, in general, reused on a per-object basis. Moreover, regions cannot be encapsulated inside recursive data structures. Recently, Crary, Walker and Morrisett [10] investigated an alternative region type system that reasons about aliasing between regions using similar technology as we use here to reason about aliasing between individual objects. We believe that some of the techniques developed in this paper will make it possible to capture regions in recursive data structures and we are eager to investigate a combined region-alias type framework that can take advantage of both forms of typed memory management.

Acknowledgements

Fred Smith worked with us on the predecessor to this research and the many stimulating discussions we had together contributed to the current paper. Neal Glew and the anonymous reviewers for TIC'00 made many helpful comments on an earlier draft of this paper.

References

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM Conference on Programming Language Design and Implementation*, pages 174–185, La Jolla, California, 1995.

[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] Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for describing linked data structures. In *European Symposium on Programming*, pages 2–19, Amsterdam, March 1999.

[5] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg, January 1996.

[6] Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, pages 23–50, Edinburgh, 1972. Edinburgh University Press.

[7] Perry Cheng and Chris Okasaki. Destination-passing style and generational garbage collection. Unpublished., November 1996.

[8] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, March 1996.

[9] Stephen Cook and Derek Oppen. An assertion language for data structures. In *Second ACM Symposium on Principles of Programming Languages*, pages 160–166, New York, 1975. ACM Press.

[10] 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.

[11] 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.

[12] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 1–15, St. Petersburg Beach, Florida, January 1996.

[13] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[14] Dan Grossman and Greg Morrisett. Scalable certification of native code: Experience from compiling to TALx86. Technical Report TR2000-1783, Cornell University, February 2000.

[15] Juan C. Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Symposium on Logic in Computer Science*, pages 333–343, Philadelphia, June 1990. IEEE Computer Society Press.

[16] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994.

[17] Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. Preliminary draft, March 2000.

[18] Neil D. Jones and Steven Muchnick, editors. *Flow analysis and optimization of Lisp-like structures*. Prentice-Hall, 1981.

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

[20] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[21] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *ACM Conference on Programming Language Design and Implementation*, pages 24–31, June 1988.

[22] James Richard Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California at Berkeley, May 1989. Available as Berkeley technical report UCB/CSD 89/502.

[23] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In *ACM Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, Berlin, 1991. Springer-Verlag.

[24] Y. Minamide. A functional representation of data structures with a hole. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 75–84, San Diego, January 1998.

[25] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, January 1996.

[26] Bernhard Möller. Towards pointer algebra. *Science of Computer Programming*, 21:57–90, 1993.

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

[28] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998.

[29] 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.

[30] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM Conference on Programming Language Design and Implementation*, pages 333 – 344, Montreal, June 1998.

[31] John C. Reynolds. Syntactic control of interference. In *Fifth ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, 1978.

[32] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Symposium in Celebration of the Work of C. A. R. Hoare*, 2000. To appear.

[33] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[34] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 105–118, San Antonio, January 1999.

[35] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[36] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Berlin, March 2000. Springer-Verlag.

[37] Johnathan Sobel and Daniel Friedman. Recycling continuations. In *ACM International Conference on Functional Programming*, pages 251–260, Baltimore, September 1998.

[38] TALx86. See http://www.cs.cornell.edu/talc for an implementation of Typed Assembly Language based on Intel's IA32 architecture.

[39] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, January 1994.

[40] 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.

[41] Philip Wadler. *Listlessness is Better than Laziness*. PhD thesis, Carnegie Mellon University, August 1985. Available as Carnegie Mellon University technical report CMU-CS-85-171.

[42] 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.

[43] David Walker. *Typed Memory Management*. PhD thesis, Cornell University, January 2001. To appear.

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

[45] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1999.

[46] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, TX, January 1999.

A Type Well-formedness and Equality

$\boxed{\Delta \vdash c : \kappa}$

$$\frac{}{\Delta \vdash \beta : \Delta(\beta)}$$

$$\frac{}{\Delta \vdash \ell : \mathtt{Loc}}$$

$$\frac{}{\Delta \vdash \emptyset : \mathtt{Store}}$$

$$\frac{\Delta \vdash C : \mathtt{Store} \quad \Delta \vdash \eta : \mathtt{Loc} \quad \Delta \vdash \tau : \mathtt{Type}}{\Delta \vdash C \otimes \{\eta \mapsto \tau\} : \mathtt{Store}}$$

$$\frac{\Delta \vdash C : \mathtt{Store} \quad \Delta \vdash \epsilon : \mathtt{Store}}{\Delta \vdash C \otimes \epsilon : \mathtt{Store}}$$

$$\frac{}{\Delta \vdash \mathit{junk} : \mathtt{Type}}$$

$$\frac{}{\Delta \vdash \mathit{int} : \mathtt{Type}}$$

$$\frac{}{\Delta \vdash \mathcal{S}(i) : \mathtt{Type}}$$

$$\frac{\Delta \vdash \eta : \mathtt{Loc}}{\Delta \vdash \mathit{ptr}(\eta) : \mathtt{Type}}$$

$$\frac{\Delta \vdash \tau_1 : \mathtt{Type} \quad \cdots \quad \Delta \vdash \tau_n : \mathtt{Type}}{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle : \mathtt{Type}}$$

$$\frac{\Delta \vdash \tau_1 : \mathtt{Type} \quad \Delta \vdash \tau_2 : \mathtt{Type}}{\Delta \vdash \tau_1 \cup \tau_2 : \mathtt{Type}}$$

$$\frac{\begin{array}{c}\Delta, \Delta' \vdash C : \mathtt{Store} \\ \Delta, \Delta' \vdash \tau_1 : \mathtt{Type} \quad \cdots \quad \Delta, \Delta' \vdash \tau_n : \mathtt{Type} \\ (\mathit{Dom}(\Delta) \cap \mathit{Dom}(\Delta') = \emptyset)\end{array}}{\Delta \vdash \forall[\Delta' \mid C].(\tau_1, \ldots, \tau_n) \to \mathbf{0} : \mathtt{Type}}$$

$$\frac{\begin{array}{c}\Delta, \Delta' \vdash C : \mathtt{Store} \qquad \Delta, \Delta' \vdash \tau : \mathtt{Type} \\ (\mathit{Dom}(\Delta) \cap \mathit{Dom}(\Delta') = \emptyset)\end{array}}{\Delta \vdash \exists[\Delta' \mid C].\tau : \mathtt{Type}}$$

$$\frac{\begin{array}{c}\Delta, \alpha{:}(\mathit{Dom}(\Delta')) \to \mathtt{Type}, \Delta' \vdash \tau : \mathtt{Type} \\ (\mathit{Dom}(\Delta) \cap \mathit{Dom}(\Delta') = \emptyset)\end{array}}{\Delta \vdash \mathtt{rec}\,\alpha\,(\Delta').\tau : (\mathit{Dom}(\Delta')) \to \mathtt{Type}}$$

$$\frac{\Delta \vdash c : (\kappa_1, \ldots, \kappa_n) \to \mathtt{Type} \quad \Delta \vdash c_1 : \kappa_1 \quad \cdots \quad \Delta \vdash c_n : \kappa_n}{\Delta \vdash c\,(c_1, \ldots, c_n) : \mathtt{Type}}$$

$\boxed{\Delta \vdash a_1 = a_2 : \mathtt{Atom}}$

$$\frac{\Delta \vdash \eta : \mathtt{Loc} \quad \Delta \vdash \tau = \tau' : \mathtt{Type}}{\Delta \vdash \{\eta \mapsto \tau\} = \{\eta \mapsto \tau'\} : \mathtt{Atom}}$$

$$\frac{\Delta \vdash \epsilon : \mathtt{Store}}{\Delta \vdash \epsilon = \epsilon : \mathtt{Atom}}$$

$\boxed{\Delta \vdash c_1 = c_2 : \kappa}$

$$\frac{\Delta \vdash c : \kappa}{\Delta \vdash c = c : \kappa}$$

$$\frac{\Delta \vdash c_2 = c_1 : \kappa}{\Delta \vdash c_1 = c_2 : \kappa}$$

$$\frac{\Delta \vdash c_1 = c_2 : \kappa \quad \Delta \vdash c_2 = c_3 : \kappa}{\Delta \vdash c_1 = c_3 : \kappa}$$

$$\frac{\begin{array}{c}\Delta \vdash a_1 = a_1'' : \mathtt{Atom} \qquad \cdots \qquad \Delta \vdash a_n = a_n'' : \mathtt{Atom} \\ a_1', \ldots, a_n' \text{ is a permutation of } a_1'', \ldots, a_n''\end{array}}{\Delta \vdash \emptyset \otimes a_1 \otimes \cdots \otimes a_n = \emptyset \otimes a_1' \otimes \cdots \otimes a_n' : \mathtt{Store}}$$

$$\frac{\Delta \vdash \tau_1 = \tau_1' : \mathtt{Type} \quad \cdots \quad \Delta \vdash \tau_n = \tau_n' : \mathtt{Type}}{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle = \langle \tau_1', \ldots, \tau_n' \rangle : \mathtt{Type}}$$

$$\frac{\Delta \vdash \tau_1 = \tau_1' : \mathtt{Type} \quad \Delta \vdash \tau_2 = \tau_2' : \mathtt{Type}}{\Delta \vdash \tau_1 \cup \tau_2 = \tau_1' \cup \tau_2' : \mathtt{Type}}$$

$$\frac{\begin{array}{c}\Delta, \Delta' \vdash C = C' : \mathtt{Store} \\ \Delta, \Delta' \vdash \tau_1 = \tau_1' : \mathtt{Type} \\ \vdots \\ \Delta, \Delta' \vdash \tau_n = \tau_n' : \mathtt{Type} \\ (\mathit{Dom}(\Delta) \cap \mathit{Dom}(\Delta') = \emptyset)\end{array}}{\begin{array}{l}\Delta \vdash \forall[\Delta' \mid C].(\tau_1, \ldots, \tau_n) \to \mathbf{0} = \\ \qquad \forall[\Delta' \mid C'].(\tau_1', \ldots, \tau_n') \to \mathbf{0} : \mathtt{Type}\end{array}}$$

$$\frac{\begin{array}{c}\Delta, \Delta' \vdash C = C' : \mathtt{Store} \qquad \Delta, \Delta' \vdash \tau = \tau' : \mathtt{Type} \\ (\mathit{Dom}(\Delta) \cap \mathit{Dom}(\Delta') = \emptyset)\end{array}}{\Delta \vdash \exists[\Delta' \mid C].\tau = \exists[\Delta' \mid C'].\tau' : \mathtt{Type}}$$

$$\frac{\begin{array}{c}\Delta, \alpha{:}(\mathit{Dom}(\Delta')) \to \mathtt{Type}, \Delta' \vdash \tau = \tau' : \mathtt{Type} \\ (\mathit{Dom}(\Delta) \cap (\{\alpha\} \cup \mathit{Dom}(\Delta')) = \emptyset)\end{array}}{\Delta \vdash \mathtt{rec}\,\alpha\,(\Delta').\tau = \mathtt{rec}\,\alpha\,(\Delta').\tau' : (\mathit{Dom}(\Delta')) \to \mathtt{Type}}$$

$$\frac{\begin{array}{c}\Delta \vdash c = c' : (\kappa_1, \ldots, \kappa_n) \to \mathtt{Type} \\ \Delta \vdash c_1 = c_1' : \kappa_1 \quad \cdots \quad \Delta \vdash c_n = c_n' : \kappa_n\end{array}}{\Delta \vdash c\,(c_1, \ldots, c_n) = c'\,(c_1', \ldots, c_n') : \mathtt{Type}}$$