Low-Level Linear Memory Management

Chris Hawblitzel Edward Wei Heng Huang Eric Krupski Lea Wittie
Department of Computer Science, Dartmouth College

ABSTRACT

Efficient low-level systems need more control over memory than safe high-level languages usually provide. As a result, run-time systems are typically written in unsafe languages such as C. This paper extends previous work on linear types, alias types, regions, and typed garbage collection to give type-safe code more control over memory. The approach is truly low-level: memory consists of a single linear array of words, with load and store operations but no built-in notion of an object. The paper constructs lists and arrays out of the basic linear memory primitives, and then introduces *type sequences* for building regions of nonlinear data. It then describes a Cheney queue typed garbage collector, implemented safely over regions.

1. INTRODUCTION

Modern computers rely on the correctness and security of low-level systems, such as garbage collectors, device drivers, and embedded system code. Often, a small amount of low-level code (say, in a firewall, a network interface device driver, or secure coprocessor[22]) is all that stands between a computer system and a hacker trying to break into the system. Even in the absence of malicious outsiders, buggy device drivers often cause annoying system crashes. Given their role as the foundation for higher-level services, one might expect low-level systems to benefit from the static and run-time checks provided by type-safe languages. However, systems programmers usually lean towards assembly language, C, or C++ rather than Java, ML, or Haskell, because they need efficient low-level control of memory. Java's array bounds checking, for example, prevents the buffer overflow attacks that so many C programs are susceptible to, but often imposes extra run-time overhead. Automatic garbage collection prevents dangling pointers, but *implementing* a garbage collector requires the ability to explicitly free heap objects, a privilege not usually granted to safe language programs.

This paper extends previous work on linear types, alias types, regions, and typed garbage collection to give type-safe code more control over memory. It models memory as a linear array of words, indexed by integer memory addresses. Since linearity prevents aliasing, store operations can change the type of a memory word without the danger of a subsequent load operation reading the word with the wrong type. To support integer addressing, the type system includes an arithmetic based on singleton types, in the style of Xi et al[30]. The paper then extends these basic tools with *coercion functions,* which are essentially proofs to modify the types of values, and *type sequences*, which introduce nonlinearity into the system without imposing a specific memory management strategy for the nonlinear data. These are sufficient to implement region-based memory management[24][6] and garbage collection.

To demonstrate the practicality of these ideas, we have developed a safe C-like language called *Clay*, which implements the type system in this paper, along with some heuristic-based type inference and static detection of possible 32-bit integer overflow. The Clay type checker uses the Omega test software[18], modified to support arbitrary-precision arithmetic, to check arithmetic constraints. The type-checked code is compiled to C++. All the examples in this paper, including the garbage collectors and other run-time system code, are included with the Clay distribution[1].

Rather than focusing on the specifics of Clay, this paper presents the type system as part of an abstract machine called λ^{low}, based on the standard call-by-value higher-order polymorphic lambda calculus, F^{ω}. Although it may seem strange to describe low-level operations from within a high-level lambda calculus framework, the generality of the lambda calculus allows us to apply the ideas from λ^{low} to many settings. For example, a closure-converted subset of λ^{low} mimics C-like languages such as Clay, while a CPS- and closure-converted subset of λ^{low} mimics register transfer languages or assembly-languages. Because of this, we believe that λ^{low}'s type system can contribute to the development of systems for *proof-carrying code* (PCC)[16] and *typed assembly language* (TAL)[15]. Since these systems run untrusted code directly on a trusted computer, they are particularly sensitive to bugs or vulnerabilities in the run-time system, and considerable work has focused on moving memory management out of the "trusted computing base" (TCB) for PCC and TAL. The most ambitious effort to reduce the TCB is *foundational PCC*[2][8], which attempts to build an entire PCC/TAL system up from a small number of axioms. Because linear memory types are based on a simple, low-level

[1]available at http://www.cs.dartmouth.edu/~hawblitz/

model of memory, they may be suitable for foundational PCC systems.

2. BACKGROUND

This paper draws from previous research on typed memory management, including linear types, alias types, regions, and typed garbage collection. This section provides a brief background on these techniques.

The primary obstacle to type-safe memory management is dangling pointers (pointers to deallocated memory). In the presence of aliasing (multiple pointers to the same object), it is difficult to prevent dangling pointers. The simplest strategy for safe deallocation is to disallow aliasing completely: if all data structures are trees, rather than arbitrary graphs, then each object has only one pointer to it, and the type system can ensure safety by invalidating the one pointer to an object when the program deallocates the object. In the terminology of linear types [25], a pointer P to an object O is "consumed" when O dies. For example, if the variable x contains the pointer P, then the expression "deallocate(x)" removes x from the scope of the rest of the program, so that any subsequent reference to x fails to type-check.

While pure linear data structures have been implemented in LISP[3] and designed for typed assembly language[4], they tend to be clumsy in practice. Many useful data structures, such as circular and doubly-linked lists, aren't linear. Even for linear data structures, useful operations on the data often violate the linearity. For example, a purely linear program cannot traverse a list while simultaneously holding a pointer to the head of the list, because this would result in two pointers to the list. As a result, programs often make extra copies of data structures, one copy for every pointer to the data structure.

Alias types[21][26] allow limited forms of aliasing rather than prohibiting aliasing completely, so that a program can express some nonlinear data structures. Alias types describe the current state of a set of objects in a *constraint*. For example, the constraint $\{\ell_1 \mapsto \langle\text{int,int}\rangle, \ell_2 \mapsto \langle\text{int,int,int}\rangle\}$ indicates that there are two distinct objects at memory locations ℓ_1 and ℓ_2, one of which is a pair of integers and one of which is a triplet of integers. Allocation, deallocation, loads, and stores alter the state of the current constraint. For example, storing a floating point number into the second element of ℓ_2 produces a new constraint $\{\ell_1 \mapsto \langle\text{int,int}\rangle, \ell_2 \mapsto \langle\text{int,float,int}\rangle\}$. Constraints are treated linearly, so that the old constraint is consumed when the new constraint is produced, which prevents the program from trying to use the second field of ℓ_2 as an integer after it has been changed to a float. In contrast to pure linear types, the program may keep many pointers to ℓ_2 simultaneously—the type "pointer-to-ℓ_2" is nonlinear and may be freely copied. The linearity of the constraint ensures that any use of a "pointer-to-ℓ_2" value respects the current state of ℓ_2.

While alias types are more flexible than pure linear data types, the linear constraint disallows many nonlinear data structures and nonlinear operations on data. In practice, the allowed data structures are "almost linear"—they look like linear data augmented with carefully specified extra pointers. For example, Walker et al[26] express a circular list as a singly-linked list plus one special pointer from the tail to head. However, the authors do not present standard operations on circular lists, such as traversing a list in a circle from the head to tail and back around to the head.

One can extend the power of linear and alias types by adding ever more sophisticated ways to control aliasing (see shape types, for example[7]), but most programming languages allow unrestricted pointer aliasing in the heap, and a compiler from a standard source programming language to typed assembly language or proof-carrying code must deal with this unrestricted aliasing. One strategy is to allow arbitrary nonlinear data inside a *region*[24][6], whose lifetime is controlled linearly (or almost linearly). A pointer to an object O inside a region R is assigned a type "τ at R", where τ describes the data inside O. Loads and stores to a "τ at R" value are only type-correct if R is still alive. When R is destroyed, all the objects in it are deallocated together. Deallocation of individual objects inside a live region is usually disallowed, because arbitrary aliasing is possible inside the region, which means that individual object deallocation could leave dangling pointers.

Any program can be trivially rewritten to take advantage of regions: simply create one big region, allocate all objects in the region, and keep the region alive until the program finishes. This is usually not an efficient strategy, though, so either automated inference[24] or manual region management is necessary to break data up into smaller regions that the program can deallocate as soon as possible to minimize memory consumption. Wang and Appel[28] observed that a program can copy live data from one region into a second region and then deallocate the first region, effectively constructing a copying garbage collector written entirely with type-safe language features. Such a *typed* (or *type-preserving*) collector offers the generality of traditional garbage collection without requiring the collector to be trusted. Unfortunately, the details of the copying process are troublesome; the collector needs some way to traverse the data, and it needs to maintain forwarding pointers from the old region into the new region. Several solutions to these problems have been proposed [28][13][14], but each new proposal changes the typing rules for regions, often in ad-hoc and complex ways. Furthermore, the resulting collectors are often unable to use low-level techniques common in collectors written in C.

The goal of this paper is to develop regions from the ground up, starting with a combination of linear and alias types, in order to provide a stable, flexible, safe, and low-level platform for constructing typed garbage collectors.

3. LINEAR MEMORY

This section describes the syntax and semantics of λ^{low}. To make the core language clearer, we delay a couple of features (coercion functions and type sequences) until later sections. Type checking rules for expressions appear throughout the paper; the complete syntax and semantics of λ^{low} is found in [10]. For the portion of the language described in this subsection, two type environments are needed to type-check an expression: Γ maps variables to types, and Ψ maps integer memory word addresses to types. For convenience, we often write a combined context C, defined in this subsection to be $C \triangleq \Psi;\Gamma$. The notation $C = C_1, C_2$ indicates that C, C_1, and C_2 share the same nonlinear assumptions, but that each of C's linear assumptions appears in either C_1 or C_2, not both. The notation $\dot{C}$ denotes a context with no linear assumptions.

The state of a running program consists of memory M, which maps integer memory word addresses to one-word values, and an expression e. The evaluation rules (in [10]) describe how a machine state (M, e) steps to a new state (M', e'). The load and store expressions are the *only* expressions that read or modify M; even the region and garbage collection examples later in the paper are built entirely on top of simple loads and stores.

The expressions e, values v, and types τ are defined as follows (kinds K, integers I, and booleans B are defined later in this section):

---

types

$$\begin{array}{l}\tau = \tau_1 \xrightarrow{\phi} \tau_2 \,|\, \phi\langle\overrightarrow{\tau}\rangle \,|\, \mathrm{Mem}(I,\tau)\\ \;|\, \alpha \,|\, \lambda\alpha : K.\tau \,|\, \tau_1\,\tau_2 \,|\, \forall\alpha : K; B.\tau \,|\, \exists\alpha : K; B.\tau \,|\, \mu\alpha : K.\tau\\ \;|\, I \,|\, B \,|\, \mathrm{Int}(I) \,|\, \mathrm{Bool}(B) \,|\, \mathrm{if}\, B \,\mathrm{then}\, \tau_1 \,\mathrm{else}\, \tau_2\end{array}$$

expressions

$$\begin{array}{l}e = i \,|\, b \,|\, x \,|\, e_1\, e_2 \,|\, e\, \tau \,|\, \phi\langle\overrightarrow{e}\rangle \,|\, \lambda x : \tau \xrightarrow{\phi} e \,|\, e_1 \,\mathrm{op}\, e_2 \,|\, \neg e\\ \;|\, \Lambda\alpha : K; B.v \,|\, \mathrm{let}\,\langle\overrightarrow{x}\rangle = e_1 \,\mathrm{in}\, e_2 \,|\, \mathrm{if}\, e_1 \,\mathrm{then}\, e_2 \,\mathrm{else}\, e_3\\ \;|\, \mathrm{pack}[\tau_1, e] \,\mathrm{as}\, \exists\alpha : K; B.\tau_2 \,|\, \mathrm{unpack}\,\alpha, x = e_1 \,\mathrm{in}\, e_2\\ \;|\, \mathrm{roll}[(\mu\alpha : K.\tau_0)\tau_1 \cdots \tau_n](e) \,|\, \mathrm{unroll}(e) \,|\, \mathrm{fix}\, x : \tau.\mathrm{v}\\ \;|\, \mathrm{load}(e_{\mathrm{ptr}}, e_{\mathrm{Mem}}) \,|\, \mathrm{store}(e_{\mathrm{ptr}}\,, e_{\mathrm{Mem}}, \mathrm{e_v}) \,|\, \mathrm{fact}\end{array}$$

values

$$\begin{array}{l}v = i \,|\, b \,|\, \Lambda\alpha : K; B.v \,|\, \mathrm{pack}[\tau_1, v] \,\mathrm{as}\, \exists\alpha : K; B.\tau_2 \,|\, \phi\langle\overrightarrow{v}\rangle\\ \;|\, \mathrm{roll}[(\mu\alpha : K.\tau_0)\tau_1 \cdots \tau_n](v) \,|\, \lambda x : \tau \xrightarrow{\phi} e \,|\, \mathrm{fact}\end{array}$$

Most of the types are borrowed from other languages, as discussed below. The one new type is the the "linear memory type", $\mathrm{Mem}(I, \tau)$, which is loosely based on alias types. As discussed in the previous section, alias types maintain a linear context that maps locations to types. Alias types deliberately abstract away the details of how the objects are allocated and where in memory in memory objects reside: in the context $\{\ell_1 \mapsto \langle\tau_1,\tau_2\rangle,\, \ell_2 \mapsto \langle\tau_3,\tau_4,\tau_5\rangle\}$, the "locations" ℓ_1 and ℓ_2 are opaque abstractions, not integers. Linear memory types, on the other hand, are designed to implement object allocation from the ground up, and therefore they must expose the details of the underlying memory. Rather than mapping opaque locations to types, linear memory types map *integer word addresses* to the types of individual memory words. If the two objects above reside at memory locations 500 and 700, then five distinct mappings describe the state of the objects:
$\{500 \mapsto \tau_1, 501 \mapsto \tau_2, 700 \mapsto \tau_3, 701 \mapsto \tau_4, 702 \mapsto \tau_5\}$

Our examples do not require some of the more advanced features of alias types, such as explicit store polymorphism and nonlinear constraints. Because of this, it is easiest to express the type of each memory word as an individual first-class linear value, rather than as a member of a constraint set. Traditional linear data structuring mechanisms[25] then suffice to hold the types of multiple memory words. For example, the memory location 500 is described by an object of linear type "$\mathrm{Mem}(500, \tau_1)$", and the state of the pair of integers at locations 500 and 501 can be stored in a linear tuple of type $^\wedge\langle\mathrm{Mem}(500, \tau_1), \mathrm{Mem}(501, \tau_2)\rangle$.

Following Walker et al[27], the type system includes nonlinear and linear functions ($\tau_1 \xrightarrow{\phi} \tau_2$) and nonlinear and linear tuples ($\phi\langle\overrightarrow{\tau}\rangle$), where ϕ is $^\wedge$ to indicate linear data, and ϕ is $\cdot$ to indicate nonlinear data. A linear functions must be called exactly once, while a nonlinear function can be called arbitrarily often. Linear tuples are consumed when their fields are extracted. Linear data structures may hold nonlinear data (a linear tuple type $^\wedge\langle\cdot\langle\rangle, \cdot\langle\rangle\rangle$ containing empty nonlinear tuples is legal), but nonlinear data structures cannot contain linear data (the nonlinear tuple type $\cdot\langle^\wedge\langle\rangle,^\wedge\langle\rangle\rangle$ containing empty linear tuples is illegal), which ensures that linear data is never aliased. To enforce this restriction, we assign kinds to types that distinguish between linear and nonlinear types. The kind $\hat{n}$ describes all linear types of size n words, while $\dot{n}$ describes all nonlinear types of size n words. For example, the nonlinear empty tuple type $\cdot\langle\rangle$ has kind $\dot{0}$.

We use the n in the kind $\overset{\phi}{n}$ to restrict operations on memory: values stored in memory must be exactly 1 word long. The type of singleton booleans $\mathrm{Bool}(\tau)$ has kind $\dot{1}$, for example, so that a boolean value fits in one word of memory. A triplet of booleans of type $\cdot\langle\mathrm{Bool}(\tau), \mathrm{Bool}(\tau), \mathrm{Bool}(\tau)\rangle$, though, has kind $\dot{3}$, and is too large to fit into a single memory word. The kinding rules for tuple types and linear memory types illustrate the size and linearity rules for data (the environment $\Phi; \Delta$ is explained later):

$$\frac{\forall j.(\Phi; \Delta \vdash \tau_j : \dot{n}_j) \qquad n = \sum_j n_j}{\Phi; \Delta \vdash \cdot\langle\overrightarrow{\tau}\rangle : \dot{n}}$$

$$\frac{\forall j.(\Phi; \Delta \vdash \tau_j : \overset{\phi_j}{n_j}) \qquad n = \sum_j n_j}{\Phi; \Delta \vdash\, ^\wedge\langle\overrightarrow{\tau}\rangle : \hat{n}}$$

$$\frac{\Phi; \Delta \vdash I : \mathrm{int} \quad \Phi; \Delta \vdash \tau : \dot{1}}{\Phi; \Delta \vdash \mathrm{Mem}(I, \tau) : \overset{\wedge}{0}}$$

The type $\mathrm{Mem}(I, \tau)$ has kind $\overset{\wedge}{0}$, so that it is linear and occupies no space. Operations on memory consume values of type $\mathrm{Mem}(I, \tau)$ and produce new values of type $\mathrm{Mem}(I, \tau')$, so that at all times there is exactly one value of type $\mathrm{Mem}(i, \tau)$ for each memory location i. For example, the following function swaps two words of memory, consuming a pair of values of type $^\wedge\langle\mathrm{Mem}(500, \tau_1), \mathrm{Mem}(501, \tau_2)\rangle$ and producing a pair of values of type $^\wedge\langle\mathrm{Mem}(501, \tau_1), \mathrm{Mem}(500, \tau_2)\rangle$:

$$\begin{array}{l}\lambda x :\, ^\wedge\langle\mathrm{Mem}(500, \tau_1), \mathrm{Mem}(501, \tau_2)\rangle \xrightarrow{\cdot}\\ \quad \mathrm{let}\,\langle x_1, x_2\rangle = x \,\mathrm{in}\\ \quad \mathrm{let}\,\langle y_1, x_1'\rangle = \mathrm{load}(500, x_1) \,\mathrm{in}\\ \quad \mathrm{let}\,\langle y_2, x_2'\rangle = \mathrm{load}(501, x_2) \,\mathrm{in}\\ \quad \mathrm{let}\, x_1'' = \mathrm{store}(500, x_1', y_2) \,\mathrm{in}\\ \quad \mathrm{let}\, x_2'' = \mathrm{store}(501, x_2', y_1) \,\mathrm{in}\, ^\wedge\langle x_2'', x_1''\rangle\end{array}$$

When x_1 has type $\mathrm{Mem}(500, \tau_1)$, the expression $\mathrm{load}(500, x_1)$ consumes x_1 (so that x_1 is no longer in scope in the remainder of the function), and produces a pair whose first element contains the contents of memory word 500, and whose second element has type $\mathrm{Mem}(500, \tau_1)$. The typing rule for the expression $\mathrm{load}(e_{\mathrm{ptr}}, e_{\mathrm{Mem}})$ requires that the first argument e_{ptr} be a *singleton integer* of type $\mathrm{Int}(I)$, and the second argument e_{Mem} have type $\mathrm{Mem}(I, \tau)$ for some τ.

$$\frac{C_1 \vdash e_{\mathrm{ptr}} : \mathrm{Int}(I) \quad C_2 \vdash e_{\mathrm{Mem}} : \mathrm{Mem}(I, \tau)}{C_1, C_2 \vdash \mathrm{load}(e_{\mathrm{ptr}}, e_{\mathrm{Mem}}) :\, ^\wedge\langle\tau, \mathrm{Mem}(I, \tau)\rangle}$$

Unlike the more usual type "int", which is the set of all integers, the singleton type $\mathrm{Int}(I)$ is the set containing only one integer I, so that the integer constant 500 has type Int(500):

$$\dot{C} \vdash i : \mathrm{Int}(i)$$

For the load expression, the I in $\mathrm{Int}(I)$ and $\mathrm{Mem}(I, \tau)$ must match, so that $\mathrm{load}(500, x_1)$ type-checks in the example above, but $\mathrm{load}(499, x_1)$ would not type-check.

The expression $\mathrm{store}(500, x_1', y_2)$ consumes x_1' and produces a value of type $\mathrm{Mem}(500, \tau_2)$, documenting the change to the type of the value at memory word 500 from τ_1 to τ_2:

$$\frac{\begin{array}{c} C_2 \vdash e_{\mathrm{Mem}} : \mathrm{Mem}(I, \tau_1) \quad C_3 \vdash e_{\mathrm{v}} : \tau_2 \\ C_1 \vdash e_{\mathrm{ptr}} : \mathrm{Int}(I) \end{array}}{C_1, C_2, C_3 \vdash \mathrm{store}(e_{\mathrm{ptr}}, e_{\mathrm{Mem}}, e_{\mathrm{v}}) : \mathrm{Mem}(I, \tau_2)}$$

Notice that the store expression overwrites whatever τ_1 value was previously at memory location 500, while the load expression makes a copy of a value from memory. Because only nonlinear values may be discarded and copied, the type $\mathrm{Mem}(I, \tau)$ is only legal if τ is nonlinear.

While the examples above show how to manipulate values of type $\mathrm{Mem}(i, \tau)$, they leave one question unanswered: what exactly is the value that has type $\mathrm{Mem}(i, \tau)$? The syntax for the abstract machine defines a special value, "fact", which is sort of a universal unit value for all types $\mathrm{Mem}(i, \tau)$. The fact value carries no run-time information, though, and is not represented at all in a real machine. The type-checking rule states that fact is well-typed if the current context contains exactly one linear assumption, saying that memory location i holds a value of type τ:

$$\dot{C}, i \mapsto \tau \vdash \mathit{fact} : \mathrm{Mem}(i, \tau)$$

3.1 Integer and Boolean Polymorphism

The swap function described earlier only works for fixed types τ_1 and τ_2, and the particular memory addresses 500 and 501. The type system allows polymorphism over base types (kind $\overset{\phi}{n}$), integers (kind "int"), booleans (kind "bool"), and type operators (kind $K_1 \to K_2$), so the swap function can be rewritten with type:

$$\begin{array}{l} \forall \alpha_1 : \mathrm{int}.\forall \alpha_2 : \mathrm{int}.\forall \beta_1 : \dot{1}\ .\forall \beta_2 : \dot{1}\ . \\ \quad {}^\wedge\langle \mathrm{Int}(\alpha_1), \mathrm{Int}(\alpha_2), \mathrm{Mem}(\alpha_1, \beta_1), \mathrm{Mem}(\alpha_2, \beta_2)\rangle \overset{\cdot}{\longrightarrow} \\ \quad {}^\wedge\langle \mathrm{Mem}(\alpha_2, \beta_1), \mathrm{Mem}(\alpha_1, \beta_2)\rangle \end{array}$$

Following the approach of Xi et al[30], the singleton integers of type $\mathrm{Int}(\alpha_1)$ and $\mathrm{Int}(\alpha_2)$ are the one-word run-time representations of the integer types α_1 and α_2. This avoids the need for full-blown dependent types; the type system only needs to deal with a subset of integer arithmetic that can be solved easily by a standard constraint solver: universally quantified variables, addition, subtraction, comparison, and multiplication by constants. We use the symbol I to refer to integer types, and the symbol B to refer to boolean types: $(0 \leq \alpha_1 \wedge \alpha_1 \leq \alpha_2)$ is a legal boolean type, while $\mathrm{Int}(\alpha_1 + 2 * \alpha_2 + 1)$ is a legal singleton integer type that is the run-time representation of the integer type $\alpha_1 + 2 * \alpha_2 + 1$.

arithmetic

$i = \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots$
$b = \mathrm{true} \mid \mathrm{false}$
$I = \alpha \mid i \mid I_1\ \mathrm{iop}\ I_2$
$B = \alpha \mid b \mid \neg B \mid B_1\ \mathrm{bop}\ B_2 \mid I_1\ \mathrm{cmp}\ I_2$
$\mathrm{iop} = + \mid - \mid *$ $\qquad \mathrm{bop} = \wedge \mid \vee$
$\mathrm{cmp} = < \mid > \mid \leq \mid \geq \mid = \mid \neq$
$\mathrm{op} = \mathrm{iop} \mid \mathrm{bop} \mid \mathrm{cmp}$

In addition to the universal polymorphic type $\forall \alpha : K.\tau$, the type system supports existential polymorphic types $\exists \alpha : K.\tau$. For example, the type $\exists \alpha : \mathrm{int}.\mathrm{Int}(\alpha)$ is the type of a singleton integer, for some unknown integer α; this type imitates the traditional type "int" (the type of all integers). As another example, the type
$\exists \alpha : \mathrm{int}.{}^\wedge\langle \mathrm{Int}(\alpha), \mathrm{Mem}(\alpha, \tau_1), \mathrm{Mem}(\alpha + 1, \tau_2)\rangle$
says that some unknown integer α is an address where two words of type τ_1 and τ_2 are stored, and also provides the singleton integer $\mathrm{Int}(\alpha)$ needed to load and store from these memory addresses; this type imitates a pointer to a pair of τ_1 and τ_2.

Universal and existential polymorphic types may contain constraints that must be satisfied when types are substituted in for type variables. For example, if swap has the following type:

$$\forall \alpha_1 : \mathrm{int}.\forall \alpha_2 : \mathrm{int}; \alpha_1 \leq \alpha_2.\forall \beta_1 : \dot{1}\ .\forall \beta_2 : \dot{1}\ \ldots.$$

Then $(\mathit{swap}\ 3\ 6\ \ldots)$ would be legal, but $(\mathit{swap}\ 10\ 6\ \ldots)$ would be illegal. When type checking the body of the swap function, the fact that $\alpha_1 \leq \alpha_2$ is kept inside an environment B, while the kinds of the type variables are kept in an environment Δ, so that the combined type checking context is now $C \triangleq \Psi; \Delta; \Gamma; B$.

3.2 Type Equivalence

Following Xi's approach, we define two integer types I_1 and I_2 to be equivalent ($I_1 \equiv I_2$) if our constraint solver can show that for all substitutions of integer constants i for integer variables α in I_1 and I_2, the types I_1 and I_2 simplify to equal integer constants. For example, the type system considers $\alpha_1 + 2 * \alpha_2 + 1$ to be equivalent to $2 * \alpha_2 + 2 + \alpha_1 - 1$. However, our language contains two types that do not interact gracefully with Xi's approach: type operator application "$\tau_1\, \tau_2$", which is used for polymorphic abstract data types, and the conditional type "$\mathrm{if}\, B\, \mathrm{then}\, \tau_1\, \mathrm{else}\, \tau_2$", which is used to create unions[2]. Conceivably, either of these types could have kind int, so that they may be substituted for an integer variable α inside an integer type I during type-checking (the type-checking rule for type application $e\,\tau$ performs this sort of substitution, for instance). If this happens, the constraint solver may be faced with an integer type like $2 + (\mathrm{if}\, \beta\, \mathrm{then}\, 3\, \mathrm{else}\, 4) + (\alpha\, (\lambda\gamma : \mathrm{int}.\gamma + 3))$, which is difficult to deal with. Nevertheless, we valued the convenience of Xi's approach in our implementation (and in the examples in this paper) enough to resort to a hack: the kind system bans the kinds int and bool from appearing as the return type of a type operator (e.g. $\dot{1} \to \mathrm{int}$), so that integer and boolean types do not contain operator applications, and the type system disallows conditional types from having kind int or bool:

[2] Our technical report and our implementation use a different type for unions that requires more explicit programmer annotation, in order to simplify the implementation's type inference algorithm. The conditional type in this paper is more elegant, though.

kinds:

$$K = J \,|\, \mathrm{int} \,|\, \mathrm{bool} \qquad J = \overset{\phi}{n} \;|\, K \rightarrow J$$

$$\frac{\Phi; \Delta, \alpha : K_a \vdash \tau : J_b}{\Phi; \Delta \vdash \lambda\alpha : K_a.\tau : K_a \rightarrow J_b}$$

$$\frac{\Phi; \Delta \vdash \tau_f : K_a \rightarrow J_b \quad \Phi; \Delta \vdash \tau_a : K_a}{\Phi; \Delta \vdash \tau_f \tau_a : J_b}$$

$$\frac{\Phi; \Delta \vdash B : \mathrm{bool} \quad \Phi; \Delta \vdash \tau_1 : J \quad \Phi; \Delta \vdash \tau_2 : J}{\Phi; \Delta \vdash \mathrm{if}\, B \,\mathrm{then}\, \tau_1 \,\mathrm{else}\, \tau_2 : J}$$

Once these type and kind restrictions are in place, the type equivalence rules for type application and conditional types no longer interfere with the type equivalence rules for integers and booleans:

$$\Phi; B \vdash (\lambda\alpha : K.\tau_b)\tau_a \equiv [\alpha \mapsto \tau_a]\tau_b$$

$$\Phi; B \vdash (\mathrm{if\,true\,then}\, \tau_1 \,\mathrm{else}\, \tau_2) \equiv \tau_1$$

$$\Phi; B \vdash (\mathrm{if\,false\,then}\, \tau_1 \,\mathrm{else}\, \tau_2) \equiv \tau_2$$

In a typed assembly language or proof-carrying code system, it might make more sense to require the program to provide proofs of arithmetic equivalence using fundamental arithmetic axioms, rather than to rely on a constraint solver[5][23]; in this case, we wouldn't need to hack the kind system.

3.3 Recursive Types, Lists, and Stacks

As described earlier, an existential type like

$$\exists\alpha : \mathrm{int}.\,{}^\wedge\langle \mathrm{Int}(\alpha), \mathrm{Mem}(\alpha, \tau_1), \mathrm{Mem}(\alpha + 1, \tau_2)\rangle$$

can imitate a pointer to data in memory. When this idea is combined with recursive types, the type system is powerful enough to express pointer-based data structures in the style of recursive alias types[26]. For example, the following type defines a simple linked list terminated by 0, containing data elements of type τ:

$$\begin{aligned}&List = \mu\beta : (\mathrm{int} \rightarrow \hat{0}).\lambda\alpha_1 : \mathrm{int}.\mathrm{if}\, \alpha_1 = 0 \,\mathrm{then}\, {}^\wedge\langle\rangle \,\mathrm{else} \\ &\quad \exists\alpha_2 : \mathrm{int}.\,{}^\wedge\langle \mathrm{Mem}(\alpha_1, \mathrm{Int}(\alpha_2)), \mathrm{Mem}(\alpha_1 + 1, \tau), \beta\, \alpha_2\rangle\end{aligned}$$

For clarity, we'll often use the $\triangleq$ symbol to define data types, rather than using the abstract machine's official, but more cryptic, recursive type $\mu\alpha : K.\tau$:

$$\begin{aligned}&List\,(\alpha_1 : \mathrm{int}) : \hat{0} \triangleq \mathrm{if}\, \alpha_1 = 0 \,\mathrm{then}\, {}^\wedge\langle\rangle \,\mathrm{else} \\ &\quad \exists\alpha_2 : \mathrm{int}.\,{}^\wedge\langle \mathrm{Mem}(\alpha_1, \mathrm{Int}(\alpha_2)), \mathrm{Mem}(\alpha_1 + 1, \tau), List\, \alpha_2\rangle\end{aligned}$$

The type "$\mathrm{if}\, B \,\mathrm{then}\, \tau_1 \,\mathrm{else}\, \tau_2$" is equivalent to τ_1 if B is true and to τ_2 if B is false, so the $List$ type contains an empty tuple if $\alpha_1 = 0$ and a three-word tuple for non-zero α_1. The first two words of the tuple assert that memory locations α_1 and $\alpha_1 + 1$ contain values of type $\mathrm{Int}(\alpha_2)$ (the pointer to the next element) and τ (the data in the current element). By itself, the $\mathrm{Int}(\alpha_2)$ is just an arbitrary integer, but together with the Mem type in $List\, \alpha_2$ it may be used to load the fields of the next list element. Based on this approach, the language easily supports tree-like data structures. For example, free lists (or lists of free lists of various size objects) provide a simple memory management strategy for linear data.

By allowing simple integer arithmetic, the type system expresses more than just traditional link-based structures. For example, a slight modification of the linked list example produces a stack type, held in a contiguous sequence of memory locations $\alpha_1 \ldots \alpha_2 - 1$:

$$\begin{aligned}&Stack\,(\alpha_1 : \mathrm{int})\,(\alpha_2 : \mathrm{int}) : \hat{0} \triangleq \mathrm{if}\, \alpha_1 = \alpha_2 \,\mathrm{then}\, {}^\wedge\langle\rangle \,\mathrm{else} \\ &\quad {}^\wedge\langle \mathrm{Mem}(\alpha_1, \tau), Stack\,(\alpha_1 + 1)\, \alpha_2\rangle\end{aligned}$$

4. COERCION FUNCTIONS

The previous section's stack type is fine for simple pushes and pops, but it does not support constant-time random access into the middle of the stack. Getting to the middle of the stack is hard to do: the program must unroll the data type n times to get to the nth element, performing tuple extractions with each unroll. Since $O(n)$ abstract machine operations are required, there's no obvious way to perform a constant time access of an inner element. And yet, unroll operations and tuple extractions for size-zero data aren't supposed to impose any run-time cost; they are purely compile-time coercions. What is needed is a way to combine n such coercions into a single $O(1)$ time operation. Suppose that an expression e satisfies the following three constraints:

- It has no effect on the memory M
- If it evaluates to a value, then that value's type has size 0
- It is guaranteed to terminate (by evaluating to a value in a finite time)

Then there is no reason to execute e on a real machine; a compiler can simply erase e in the same way that it erases the pack/unpack and roll/unroll coercions when compiling typed code to untyped machine language code. Such an expression can still perform many coercion operations, such as traversing a stack data structure to retrieve a Mem value from deep inside the stack, but now with no run-time cost, since the expression is erased by the compiler. For example, consider the following data type for arrays:

$$\begin{aligned}&LArray\,(\alpha_1 : \mathrm{int})\,(\alpha_2 : \mathrm{int})\,(\beta : \mathrm{int} \rightarrow \hat{0}) : \hat{0} \triangleq \\ &\quad \mathrm{if}\, \alpha_1 = \alpha_2 \,\mathrm{then}\, {}^\wedge\langle\rangle \,\mathrm{else}\, {}^\wedge\langle \beta\, \alpha_1,\; LArray\,(\alpha_1 + 1)\, \alpha_2\, \beta\rangle\end{aligned}$$

The $LArray$ type is similar to the $Stack$ type, except for extra flexibility it is polymorphic over an operator β so that each array element α_1 can have data of a different type $(\beta\, \alpha_1)$. The following coercion function splits an array into two adjacent arrays, the left containing elements $\alpha_1 \ldots \alpha_2 - 1$ and the right containing elements $\alpha_2 \ldots \alpha_3 - 1$:

```
fix $split : \tau_{split}.\Lambda\alpha_1 : \mathrm{int}.\Lambda\alpha_2 : \mathrm{int}.\Lambda\alpha_3 : \mathrm{int}.\Lambda\beta : \mathrm{int} \rightarrow \hat{0}$
  $;\alpha_1 \le \alpha_2 \le \alpha_3.\, \lambda arr : (LArray\, \alpha_1\, \alpha_3) \overset{\cdot,(\alpha_2 - \alpha_1)}{\longrightarrow}$
    if $\alpha_1 = \alpha_2$
    then $^\wedge\langle \mathrm{roll}[LArray\, \alpha_1\, \alpha_2\, \beta](^\wedge\langle\rangle), arr\rangle$
    else let $\langle head, tail\rangle = \mathrm{unroll}(arr)$ in
          let $\langle left, right\rangle = split\,(\alpha_1 + 1)\, \alpha_2\, \alpha_3\, \beta\, tail$ in
```

$$\wedge\langle \mathrm{roll}[LArray\,\alpha_1\,\alpha_2\,\beta](\wedge\langle head,\, left\rangle), right\rangle$$

$$\begin{aligned}\tau_{split} \triangleq\ & \forall\alpha_1 : \mathrm{int}.\forall\alpha_2 : \mathrm{int}.\forall\alpha_3 : \mathrm{int}.\\ & \forall\beta : \mathrm{int} \to \hat{0}; \alpha_1 \leq \alpha_2 \leq \alpha_3.(LArray\,\alpha_1\,\alpha_3) \overset{\cdot,(\alpha_2-\alpha_1)}{\longrightarrow}\\ & \wedge\langle (LArray\,\alpha_1\,\alpha_2),\, (LArray\,\alpha_2\,\alpha_3)\rangle\end{aligned}$$

In the case where $\alpha_1 = \alpha_2$, the left array is empty and the right array is the whole array. Otherwise, the function pops the head of the array, recurses, and pushes the head back onto the left array returned from the recursive call.

For the compiler to omit a call to this coercion function at run-time, it must know that it satisfies the three conditions above: no stores to memory, return type of size 0, and definite termination. Since the language already supports integer arithmetic, an easy (though heavy-handed) way to ensure termination is to annotate the type of each coercion function ($\lambda x : \tau \overset{\phi,limit}{\longrightarrow} e$) with a nonnegative integer *limit*:

$$\frac{\Phi;\Delta \vdash \tau_1 : \overset{\phi_1}{n_1} \quad \Phi;\Delta \vdash \tau_2 : \overset{\phi_2}{n_2} \quad (\Phi;\Delta \vdash limit : \mathrm{int} \wedge n = 0)\ or\ (limit = \infty \wedge n = 1)}{\Phi;\Delta \vdash \tau_1 \overset{\phi,limit}{\longrightarrow} \tau_2 : \overset{\phi}{n}}$$

The type system only allows coercion functions to call functions with lower limits than their own limit. The recursive call above is legal because the caller has limit $(\alpha_2 - \alpha_1)$, as indicated in its annotation, while the called function $(split\,(\alpha_1 + 1)\,\alpha_2\,\alpha_3\,\beta)$ has limit $(\alpha_2 - (\alpha_1 + 1))$, which is smaller than $(\alpha_2 - \alpha_1)$. The current limit is stored in the context C, whose definition is extended to be $C = \Psi;\Delta;\Gamma;B;limit$ where *limit* is either an integer type I (for coercion functions) or the symbol ∞ (for normal functions; we usually omit the ∞ annotation on function types). In a context with an integer limit, expressions are not allowed to store anything to memory, although loads are allowed.

The *split* function performs one rather unrealistic operation: it tests to see whether two type variables α_1 and α_2 are equal to each other. This *intentional type analysis* would be problematic for a compiler to implement for ordinary functions: if types are erased before run-time, α_1 and α_2 won't be available to use in the run-time equality test. Since coercion functions won't actually be executed at run-time, though, the language lets them use a boolean type test expression "$\mathrm{if}\,B\,\mathrm{then}\,e_1\,\mathrm{else}\,e_2$".

Extensions to λ^{low} *for coercion functions*

$\mathrm{limit} = I \,|\, \infty$

types

$\tau = \ldots \,|\, \tau_1 \overset{\phi,\mathrm{limit}}{\longrightarrow} \tau_2$

expressions

$e = \ldots \,|\, \lambda x : \tau \overset{\phi,\mathrm{limit}}{\longrightarrow} e \,|\, \mathrm{if}\,B\,\mathrm{then}\,e_1\,\mathrm{else}\,e_2 \,|\, \mathrm{coerce}(e)$

values

$v = \ldots \,|\, \lambda x : \tau \overset{\phi,\mathrm{limit}}{\longrightarrow} e$

It's easy to define other coercions for the $LArray$ type: combining two adjacent arrays into a single array, constructing and deconstructing empty arrays, and constructing and deconstructing single-element arrays. These coercions suffice to implement familiar operations to get and set array elements. For example, given an expression of type $(LArray\ 0\ 10\,(\lambda\alpha : int.\mathrm{Mem}(500 + \alpha, \tau)))$, describing an array of 10 elements of type τ stored in memory words 500...509, element 5 of the array is accessed by splitting the array into three arrays 0...4, 5...5, and 6...9, retrieving the $\mathrm{Mem}(505, \tau)$ fact from the single-element array 5...5, performing a load or store to memory word 505, and then recombining the three arrays into the original array 0...9.

Viewed from another perspective (courtesy of the Curry-Howard isomorphism), the *split* function takes a proof as an argument (say, a proof that memory words 500...509 all contain values of type τ) and returns two proofs as a result. The language of coercion functions is then nothing more than a strongly normalizing proof language, although it happens to look suspiciously similar to the original programming language. In the abstract machine and in Clay, the only differences between the proof and programming languages is the treatment of stores (disallowed in the proof language), intentional type analysis (disallowed in the programming language), and function calls (restricted in the proof language). This similarity has the advantage that programmers need not learn a second language to implement "proofs", and the programming and proof languages interact seamlessly. On the other hand, a specialized, dedicated proof language would be more elegant, powerful, more amenable to automated proof generation tools, and easier to connect to existing libraries of proofs. The LTT system[5], for example, embeds LF[9] proofs in a separate programming language. The TL system[20] also embeds proofs in programs.

5. NONLINEAR DATA STRUCTURES

The previous sections described types for linear data structures, such as linear lists and linear arrays. Later sections will describe extensions to the type system for building regions of nonlinear data structures. Interestingly, even without extending the type system, linear memory types can express nonlinear data structures. Consider the following C code that creates a circular list with 3 elements, and calls a function to get the 5th list element, which it then splices into the beginning of the list:

```
struct List {
  struct List *next;
};
struct List *nth(int n, struct List *x) {
  if(n==0) return x;
  else return nth(n-1, x->next);
}
struct List a;
struct List b;
struct List c;
a.next = &b;
b.next = &c;
c.next = &a;
a.next = nth(5, &a);
```

This code creates enough unpredictable aliases to thwart a direct representation of the list with alias types or linear memory types. However, rather than making the list a linear type, we can encode the list as a nonlinear function that fetches data from a linear area of memory represented by an abstract type β. The linear area of memory β acts much like a region, while the nonlinear type $List\,\beta$ acts like a nonlinear pointer into the region (something like the type "List at β" in standard region terminology). As long as β stays around, any $\mathit{List}\,\beta$ value can be used to load and store from the region:

$$\begin{aligned}&\mathit{List}\,(\beta : \overset{\wedge}{0}) : \dot{1} \triangleq \exists\alpha : \text{int}. \cdot \langle \text{Int}(\alpha),\\ &\qquad \beta \xrightarrow{\cdot,0} \wedge\langle \text{Mem}(\alpha, \mathit{List}\,\beta),\ \text{Mem}(\alpha, \mathit{List}\,\beta) \xrightarrow{\wedge,0} \beta\rangle\rangle\end{aligned}$$

Each list value contains one singleton integer $\text{Int}(\alpha)$, which is the address of the list element. To get the data from the list element, the program calls the nonlinear function of type $\beta \xrightarrow{\cdot,0} \wedge\langle\text{Mem}(\alpha, \mathit{List}\,\beta), ...\rangle$. This function consumes the linear "region" β, and returns a linear memory type describing the list element's data (in this example, the element contains only one word, the field "next"). With the linear memory type, the program is free to load and store the element's data. When it is finished loading and storing, it calls the linear $\text{Mem}(\alpha, \mathit{List}\,\beta) \xrightarrow{\wedge,0} \beta$ function to relinquish the linear memory type and reconstruct the region β.

The $\beta \xrightarrow{\cdot,0} \wedge\langle\text{Mem}(\alpha, \mathit{List}\,\beta), ...\rangle$ function in $\mathit{List}\,\beta$ manipulates linear data, but is not linear itself; it is merely a middleman that passes linear data back and forth between the region and the program. Thus, the $\mathit{List}\,\beta$ type is nonlinear, and the program can freely copy and discard lists. For example, here is an implementation of function "nth":

```
$nth = \Lambda\beta : \overset{\wedge}{0}\ .\text{fix}$
$f : (\forall\alpha : \text{int}.\text{Int}(\alpha) \xrightarrow{\cdot} (\mathit{List}\,\beta) \xrightarrow{\cdot} \beta \xrightarrow{\cdot} \wedge\langle \mathit{List}\,\beta,\ \beta\rangle).$
    $\Lambda\alpha : \text{int}.\lambda n : \text{Int}(\alpha) \xrightarrow{\cdot} \lambda x : \mathit{List}\,\beta \xrightarrow{\cdot} \lambda r : \beta \xrightarrow{\cdot}$
    $\text{if}\, n = 0\, \text{then} \wedge\langle x, r\rangle\ \text{else}$
    $\text{let}\,\langle x', r'\rangle = \mathit{getnext}\,\beta\, x\, r\, \text{in}$
    $f\,(\alpha - 1)\,(n - 1)\,x'\,r'$
```

The nth function in λ^{low} is similar to the C implementation, except that it takes a third argument r of type β, and returns r along with the desired list element (in the tuple $\wedge\langle x, r\rangle$) when the function terminates. The r value is needed to load the next field from the list x, as implemented in the following helper function:

```
$\mathit{getnext} = \Lambda\beta : \overset{\wedge}{0}\ .\lambda x : \mathit{List}\,\beta \xrightarrow{\cdot} \lambda r : \beta \xrightarrow{\cdot}$
    $\text{unpack}\,\alpha, y = \text{unroll}(x)\,\text{in}$
    $\text{let}\,\langle \mathit{yptr}, \mathit{yf}\rangle = y\,\text{in}$
    $\text{let}\,\langle \mathit{mem}, \mathit{zf}\rangle = \mathit{yf}\,r\,\text{in}$
    $\text{let}\,\langle x', \mathit{mem}'\rangle = \text{load}(\mathit{yptr}, \mathit{mem})\,\text{in}$
    $\wedge\langle x', \mathit{zf}\,\mathit{mem}'\rangle$
```

The $\mathit{getnext}$ function extracts the list address yptr and the list function yf, which when called yields the linear memory value mem, of type $\text{Mem}(\alpha, \mathit{List}\,\beta)$. With the address and linear memory value, $\mathit{getnext}$ loads the next field from memory and returns it, along with the linear β value. It's easy to define a $\mathit{setnext}$ function in the same way, using a store instead of a load:

```
$\mathit{setnext} =$
$\Lambda\beta : \overset{\wedge}{0}\ .\lambda x : \mathit{List}\,\beta \xrightarrow{\cdot} \lambda x' : \mathit{List}\,\beta \xrightarrow{\cdot} \lambda r : \beta \xrightarrow{\cdot}$

    $\text{unpack}\,\alpha, y = \text{unroll}(x)\,\text{in}$
    $\text{let}\,\langle \mathit{yptr}, \mathit{yf}\rangle = y\,\text{in}$
    $\text{let}\,\langle \mathit{mem}, \mathit{zf}\rangle = \mathit{yf}\,r\,\text{in}$
    $\mathit{zf}\,(\text{store}(\mathit{yptr}, \mathit{mem}, x'))$
```

With these functions, the C code

```
a.next = nth(5, &a);
```

is implemented with a function:

```
$\mathit{fifth} = \Lambda\beta : \overset{\wedge}{0}\ .\lambda a : \mathit{List}\,\beta \xrightarrow{\cdot} \lambda r : \beta \xrightarrow{\cdot}$
    $\text{let}\,\langle b, r'\rangle = \mathit{nth}\,\beta\,5\,5\,a\,r\,\text{in}$
    $\mathit{setnext}\,\beta\,a\,b\,r'$
```

All that remains is to allocate and initialize the circular list. For this, we need a concrete type for β, and we choose a type that describes three words of memory, at addresses 1,2, and 3, each containing a list. Let's call this type Three:

$$\begin{aligned}\mathit{Three} : \dot{0} \triangleq\ \wedge\langle&\text{Mem}(1, \mathit{List}\ \mathit{Three}),\\ &\text{Mem}(2, \mathit{List}\ \mathit{Three}),\\ &\text{Mem}(3, \mathit{List}\ \mathit{Three})\rangle\end{aligned}$$

To create a list of type $\mathit{List}\ \mathit{Three}$, we need to build a pair of an address and a function. Let's start with address 2, corresponding to the object "b" from the C code:

$$\begin{aligned}&b = \text{roll}[\mathit{List}\ \mathit{Three}](\\ &\qquad \text{pack}[2, \cdot\langle 2, f_b\rangle]\ \text{as}\ \exists\alpha : \text{int}. \cdot \langle\text{Int}(\alpha), \mathit{Three} \xrightarrow{\cdot,0} ...\rangle)\end{aligned}$$

The function f_b must extract a $\text{Mem}(2, \mathit{List}\ \mathit{Three})$ value from a Three value, and return this value along with a function that reconstructs the original Three value:

```
$f_b = \lambda r : \mathit{Three} \xrightarrow{\cdot,0}$
  $\text{let}\,\langle m_1, m_2, m_3\rangle = r\,\text{in}$
  $\wedge\langle m_2,\ \lambda m_2' : \text{Mem}(2, \mathit{List}\ \mathit{Three}) \xrightarrow{\wedge,0} \wedge\langle m_1, m_2', m_3\rangle\rangle$
```

In a similar way, we can construct a value a for address 1 and a value c for address 3. Finally, we implement the C code for initialization

```
a.next = &b;
b.next = &c;
c.next = &a;
```

using three store operations, assuming we are given memory words 1, 2, and 3 initialized to some arbitrary junk types τ_1, τ_2, and τ_3, which we overwrite:

```
$\lambda m :\ \wedge\langle\text{Mem}(1, \tau_1), \text{Mem}(2, \tau_2), \text{Mem}(3, \tau_3)\rangle \xrightarrow{\cdot}$
    $\text{let}\,\langle m_1, m_2, m_3\rangle = m\,\text{in}$
    $\text{let}\,\langle m_1', m_2', m_3'\rangle =\ \wedge\langle$
            $\text{store}(1, m_1, b),$
            $\text{store}(2, m_2, c),$
            $\text{store}(3, m_3, a)\rangle\,\text{in}$
    $\mathit{fifth}\ \mathit{Three}\ a \wedge\langle m_1', m_2', m_3'\rangle$
```

We chose an example based on a single, minimal list type for simplicity. The techniques above extend to more complex data types and mixtures of data types, though. Consider a type

```
struct LTree {
  struct List *list;
  struct LTree *left;
  struct LTree *right;
};
```

This is represented as:

$$\begin{aligned}&\mathit{LTree}\,(\beta : \overset{\wedge}{0}) : \dot{1} \triangleq\\ &\qquad \exists\alpha : \text{int}. \cdot \langle\text{Int}(\alpha), \beta \xrightarrow{\cdot,0} \wedge\langle\tau_{mem},\ \tau_{mem} \xrightarrow{\wedge,0} \beta\rangle\rangle\\ &\tau_{mem} =\ \wedge\langle\text{Mem}(\alpha, \mathit{List}\,\beta),\\ &\qquad\quad \text{Mem}(\alpha + 1, \mathit{LTree}\,\beta),\\ &\qquad\quad \text{Mem}(\alpha + 2, \mathit{LTree}\,\beta)\rangle\end{aligned}$$

6. TYPE SEQUENCES AND REGIONS

The previous section's encoding of nonlinear data structures in "regions" suffers from a serious limitation: the regions cannot grow dynamically. Suppose we wanted to add a fourth list to the region defined by $Three$. We'd need to change the region type from $Three$ to $Four$:

$$\begin{aligned} Three :\dot{0}\triangleq\wedge\ &\langle \mathrm{Mem}(1, List\ Three), \\ &\mathrm{Mem}(2, List\ Three), \\ &\mathrm{Mem}(3, List\ Three)\rangle \\ Four :\dot{0}\triangleq\wedge\ &\langle \mathrm{Mem}(1, List\ Four), \\ &\mathrm{Mem}(2, List\ Four), \\ &\mathrm{Mem}(3, List\ Four), \\ &\mathrm{Mem}(4, List\ Four)\rangle \end{aligned}$$

Unfortunately, the type $Three$ is embedded in the types of all the existing lists. It's not clear how coerce a list of type $List\ Three$ to type $List\ Four$. This section combines the intuition behind the previous section's encoding with a new language mechanism called *type sequences* to encode regions that grow dynamically. Type sequences allow a program to refine an existing type τ at run-time; a list of type $List\,\tau$ stays valid as τ is refined, so that the program can add new data to a region without needing to coerce the old data to a different type. We need to be very careful about our notion of type refinement, though: changing an existing type arbitrarily is certainly unsafe. However, if we underspecify τ to start with, by leaving an abstract hole "?" in the type:

$$\tau = \\ \wedge\langle \mathrm{Mem}(1, List\,\tau),\ \wedge\langle \mathrm{Mem}(2, List\,\tau),\ \wedge\langle \mathrm{Mem}(3, List\,\tau), ?\rangle\rangle\rangle$$

then we can later specify the hole "?" to be a particular type, such as $\mathrm{Mem}(4, List\,\tau)$, as long as we don't try to specify "?" in two different, conflicting ways. We use linearity to ensure that a hole is filled in at most once.

Our current formulation of type sequences is engineered (perhaps over-engineered) towards implementing regions. We define a *type sequence* to be not just a single hole, but an infinite vector of holes. The holes are filled in one at a time, in order. For example, if F is a type sequence, then $(F\,0)$ is specified first, followed by $(F\,1)$, then $(F\,2)$, and so on.

The key properties of type sequences are:

- Large, dynamic namespace: F supplies an unbounded number of names for types: $(F\,0)$, $(F\,1)$, $(F\,2)$, ..., and F grows at run-time as new elements are added to the sequence.
- Indelibility: once a type τ is assigned to a name $(F\,I)$, then $(F\,I)$ will always refer to τ, and no other type.

Using these properties, we can define a region as a mapping from memory words to elements of a type sequence:

$$(LArray\ 0\ N\,(\lambda I : \mathrm{int}.\mathrm{Mem}(I, F\,I)))$$

As F is filled in, the region's array grows to include more and more linear memory types $\mathrm{Mem}(I, F\,I)$.

Extensions to λ^{low} for type sequences

types

$\tau = \ldots \mid \mathrm{F}^K \mid \mathrm{Gen}(\tau, I) \mid \mathrm{Eq}(\tau_1, \tau_2) \mid \mathrm{InDomain}(I, \tau)$

expressions

$e = \ldots \mid \mathrm{make\_eq}(\tau) \mid \mathrm{apply\_eq}(\tau, e_1, e_2) \mid \mathrm{new\_seq}(J)$
$\mid \mathrm{discard\_seq}(e) \mid \mathrm{define\_seq}(e, \tau) \mid \mathrm{in\_domain}(I_1, I_2, e_1, e_2)$

The extensions to λ^{low} for handling type sequences are shown above. First, we define an equality type $\mathrm{Eq}(\tau_1, \tau_2)$, where a value of type $\mathrm{Eq}(\tau_1, \tau_2)$ is evidence that τ_1 is equivalent to τ_2. The make_eq(τ) expression creates a new equality value of type $\mathrm{Eq}(\tau, \tau)$, for any type τ, and the expression apply_eq(τ_f, e_{eq}, e_f) uses an equality value e_{eq} of type $\mathrm{Eq}(\tau_a, \tau_b)$ to substitute τ_b for τ_a in selected locations inside the type of e_f, effectively coercing e_f to a different, but equivalent, type:

$$\frac{\dot{C}\vdash \tau : K}{\dot{C}\vdash \mathrm{make\_eq}(\tau) : \mathrm{Eq}(\tau, \tau)}$$

$$\frac{C_1, C_2 \vdash \tau_f : K \to J \quad C_1 \vdash e_{eq} : \mathrm{Eq}(\tau_a, \tau_b) \\ C_1, C_2 \vdash \tau_a : K \quad C_1, C_2 \vdash \tau_b : K \quad C_2 \vdash e_f : \tau_f\tau_a}{C_1, C_2 \vdash \mathrm{apply\_eq}(\tau_f, e_{eq}, e_f) : \tau_f\tau_b}$$

The new_seq(J) expression creates a new type sequence F of kind int $\to J$ for any non-integer, non-boolean kind J:

$$\dot{C}\vdash \mathrm{new\_seq}(J) : \exists F : (\mathrm{int} \to J).\mathrm{Gen}(F, 0)$$

The allocation of new types in the sequence is controlled by a size-zero linear *generator* of type $\mathrm{Gen}(F, I)$. The expression define_seq(e_{gen}, τ), where e_{gen} has type $\mathrm{Gen}(F, I)$, adds a new type τ to the sequence. It consumes the old generator and returns three values, all of size zero:

- a new generator, of type $\mathrm{Gen}(F, I + 1)$
- a nonlinear proof that $(F\,I) = \tau$, of type $\mathrm{Eq}(F\,I, \tau)$
- a nonlinear proof that I will always be in the domain of F (since sequences grow but don't shrink), of type $\mathrm{InDomain}(I, F)$.

$$\frac{C\vdash e{:}\mathrm{Gen}(\tau_f, I) \quad C\vdash \tau_f{:}\mathrm{int}\to J \quad C\vdash \tau_a{:}J}{C\vdash \mathrm{define\_seq}(e, \tau_a){:}\wedge\langle \mathrm{Gen}(\tau_f, I{+}1), \mathrm{Eq}(\tau_f I, \tau_a), \mathrm{InDomain}(I, \tau_f)\rangle}$$

The $\mathrm{Eq}(F\,I, \tau)$ type is used to substitute τ for $(F\,I)$ and vice-versa, via the apply_eq(τ_f, e_{eq}, e_f) expression described above. The $\mathrm{InDomain}(I_1, F)$ type, when combined with the current generator of type $\mathrm{Gen}(F, I_2)$ produces evidence that $0 \le I_1 < I_2$, which is useful when I_1 is an index into an array of length I_2:

$$\frac{C_1, C_2 \vdash I_1 : \mathrm{int} \quad C_1, C_2 \vdash I_2 : \mathrm{int} \\ C_1 \vdash e_1 : \mathrm{InDomain}(I_1, \tau_f) \quad C_2 \vdash e_2 : \mathrm{Gen}(\tau_f, I_2)}{C_1, C_2\vdash \mathrm{in\_domain}(I_1, I_2, e_1, e_2){:}\wedge\langle \mathrm{Know}(0{\le}I_1 \wedge I_1{<}I_2), \mathrm{Gen}(\tau_f, I_2)\rangle}$$

Here, $\mathrm{Know}(B)$ is an abbreviation for $\exists\alpha : \mathrm{bool}; B. \cdot\langle\rangle$.

When the program is finished adding elements to a sequence, it may use the discard_seq(e) expression to discard the linear generator:

$$\frac{C \vdash e : \mathrm{Gen}(\tau, I)}{C \vdash \mathrm{discard\_seq}(e) : \cdot\langle\rangle}$$

The nonlinearity of the $\text{Eq}(F\,I, \tau)$ and $\text{InDomain}(I, F)$ values is the basis for building nonlinear data structures. Suppose a region consists of a single block of memory, starting at address *Base* and containing *Size* words (for clarity, we'll often use italicized, capitalized letters and words for type variables rather than Greek letters). Then a nonlinear pointer to a type τ in the region is a triplet of type $\cdot\langle Int(Base + I), \text{Eq}(F\,I, \tau), \text{InDomain}(I, F)\rangle$. The region itself contains an array of linear facts, each of the type $\text{Mem}(Base + I, F\,I)$:

$$\begin{array}{l} Region\,(F : \text{int} \rightarrow \dot{1})\,(Base : \text{int})\,(Alloc : \text{int}) \\ \quad (Size : \text{int}) : \hat{0} \triangleq\ \wedge\langle \\ \qquad \text{Gen}(F, Alloc), \\ \qquad (LArray\ 0\ Alloc\,(\lambda I : \text{int}.\text{Mem}(Base + I, F\,I))), \\ \qquad (LArray\ Alloc\ Size\,(\lambda I : \text{int}.\tau_{free}))\rangle \\ \tau_{free} \triangleq \exists\beta : \dot{1}\ .\text{Mem}(Base + I, \beta) \end{array}$$

The first element of the *Region* tuple is the sequence generator, the second is the array of allocated memory, which grows over time, and the third is the array of free memory, which shrinks over time. Given a region and a pointer, loading the pointed-to word consists of three steps:

- use the $\text{InDomain}(I, F)$ value from the pointer together with the $\text{Gen}(F, Alloc)$ value from the region to conclude that $0 \leq I < Alloc$.
- Now that I is known to be within the bounds of the $(LArray\,0\,Alloc\,\ldots)$ array, use element I of the array, of type $\text{Mem}(Base + I, F\,I)$, to load a value of type $(F\,I)$. Call this loaded value x.
- Finally, use the $\text{Eq}(F\,I, \tau)$ value from the pointer says that coerce x from type $(F\,I)$ to type τ.

Storing a word follows a similar sequence of steps. Notice that the middle step (using array element I to perform a load) is very similar to the operation performed by the *getnext* function from the previous section; both approaches temporarily borrow a linear memory type from a region, use it for a load, and then return the linear memory type to the region.

Each new allocation in the region grows the region's allocation array and shrinks the region's free array. When the program is done using the region, it merges the allocation array back into the free array, and the connection between memory and the $(F\,I)$ types is lost. The pointer types $\cdot\langle \text{Int}(Base + I), \text{Eq}(F\,I, \tau), \text{InDomain}(I, F)\rangle$ are still legal types, but without the $\text{Mem}(Base + I, F\,I)$ facts in the allocation array, they are no longer useful.

At this point, the program can use the free array to allocate other objects. In particular, it can use the free array to create a new region in place of the old region. Although it would be possible to continue using the old generator, appending new types on the end of the old sequence, it seems easier to discard the old generator and create a new sequence, with a new generator. Therefore, the abstract machine environment tracks the state of a *set* of type sequences, which grows as new sequences are allocated:

$$\begin{array}{rll} \Phi = & \{ & F_1^{K_1} \overset{\phi_1}{\mapsto} \{0 \mapsto \tau_{1,0}, 1 \mapsto \tau_{1,1}, \ldots\}, \\ & & F_2^{K_2} \overset{\phi_2}{\mapsto} \{0 \mapsto \tau_{2,0}, 1 \mapsto \tau_{2,1}, \ldots\}, \\ & & \ldots\} \end{array}$$

Each sequence is assigned an identifier $F_i^{K_i}$, which acts as a type operator of kind $(\text{int} \rightarrow K_i)$. We've been a little slippery about what exactly a sequence is in the abstract machine up to this point; now we can state more precisely that a sequence identifier is a type operator, and type variables (such as the F used in the region example above) may be bound to sequence identifiers. The new (and final) abstract machine environment is defined as $C = \Psi; \Phi; \Delta; \Gamma; B; limit$. One tricky point: each $F_i^{K_i}$ in Φ must suffice to type-check both $\text{Gen}(F_i^{K_i}, I)$ expressions, which are linear, and other expressions containing the type operator $F_i^{K_i}$, which may be nonlinear. The $\text{Gen}(F_i^{K_i}, I)$ expression is only valid in a context $\{..., F_i^{K_i} \overset{\phi_i}{\mapsto} ...\}$ where $\phi_i = \wedge$. Proofs of the following theorems are found in [10]:

Preservation: If $C \vdash (M, e : \tau)$ and (M, e) steps to (M', e'), then there is some C' so that $C' \vdash (M', e' : \tau)$

Progress: If $\Psi; \Phi; \emptyset; \emptyset; true; limit \vdash (M, e : \tau)$, and e is not a value, then (M, e) steps

Strong normalization: If $\Psi; \Phi; \emptyset; \emptyset; true; i \vdash (M, e : \tau)$, then (M, e) steps to a value (M, v) in a finite number of steps

Erasure: If $\Psi; \Phi; \emptyset; \emptyset; true; limit \vdash (M, e : \tau)$, then (i) If (M, e) steps to (M', e'), then erase$((M, e))$ steps to erase$((M', e'))$ in zero or one steps, (ii) If erase(e) is a value, then (M, e) steps to some (M', v) such that erase$((M, e))$=erase$((M', v))$ in zero or more steps, (iii) If erase$((M, e))$ steps to some (M'_{erase}, e'_{erase}), then (M, e) steps in one or more steps to some (M', e') that erases to (M'_{erase}, e'_{erase}). Our definition of erase$((M, e))$ erases types and calls to coercion functions, just as the real Clay compiler does.

7. SIMPLE COPYING COLLECTION

We have used λ^{low}'s type system to implement several typed garbage collectors, including an incremental mark-sweep collector and two polymorphic copying collectors. The collectors follow the ideas of Wang et al[28] and Monnier et al[13], but adds several new improvements:

- The copying collectors are genuine Cheney queues, in which all the collection state is stored in to-space. In contrast to the previously described typed garbage collectors, our collectors do not need an auxiliary stack to implement a recursive descent through the live data.
- The approach of Wang et al requires that the heap have a monomorphic type, which imposes inefficiencies on the compiled code. The approach of Monnier et al uses intentional type analysis to handle polymorphism, but this still leaves a subtle inefficiency in the collector: the collector must perform run-time analysis and processing to pack, unpack, roll, and unroll expressions. In addition, there must be some sort of run-time tag bits or tag words to identify existentials, rolled types, and so on. λ^{low} can hide these operations inside coercion functions, which impose no run-time space or time cost.
- The collectors supports cyclic data structures, correctly handling mixtures of pointers that may be null and pointer types that don't allow null.
- The collectors make all data layout and tag information explicit, down to the last bit. This demonstrates

that it is possible to implement efficient representations of header words and forwarding pointers in a typed collector.

More information about these collectors, including complete implementations in Clay, is available in Wei[29] and at the URL http://www.cs.dartmouth.edu/~hawblitz/. The technical report[10] also contains a translation from λ^C, a continuation-passing-style, closure-converted intermediate language developed for typed assembly language by Morrisett et al[15], to λ^{low}, using one of our copying collectors for memory management. Since [15] already provides a translation from a variant of Girard's System-F (the polymorphic lambda calculus) to λ^C, this forms a complete translation from a high-level polymorphic language to λ^{low}. The details of the collectors are beyond the scope of this paper, though; for brevity's sake, this section describes only a simple monomorphic Cheney queue typed garbage collector, based on the regions from the previous section.

A Cheney queue collector starts with a root pointer into a region of memory (from-space), and makes a breadth-first traverse through all the data reachable from that root. The collector blindly copies each traversed object into the to-space region, even though the object still contains pointers that point to from-space, and then later goes back to fix up the pointers so that to-space objects points to other to-space objects. After all live data is traversed, the collector deallocates the from-space region and, for the next collection, allocates a new to-space region (which is usually just the old from-space region, recycled).

Ideally, the type of each pointer in from-space would say that it points to another object in from-space, and the type of each pointer in to-space would say that it points to another object in to-space; in this case, each object need only concern itself with its own region. Unfortunately, the blind copy leaves temporary objects in the to-space Cheney queue pointing back to from-space. Furthermore, the collector tags the copied from-space object with a forwarding pointer into to-space, so that it won't attempt to copy the same object more than once. Since there are pointers into to-space from from-space and vice-versa, the type of each object must be aware not only of its own region, but of the previous and next regions.

To equip the objects with information about multiple regions, we build a type sequence of type sequences: the outer type sequence R contains one type sequence $(R\,E)$ for each region number E (the E is supposed to stand for "epoch"), and each $(R\,E\,I)$ describes the Ith word of memory in region number E. If region E is from-space, then a from-space object can name the type of the Ith word in to-space using the type $(R\,(E+1)\,I)$. To add even more flexibility, each word descriptor $(R\,E\,I)$ is actually a type operator that takes a region number as an argument: $(R\,E\,I\,E')$ is the type of the word stored in region E, word I, configured with the extra information E'. This extra information describes the types of the pointers contained in an object: even though the object *lives* in region E, it might *point* to a different region E'. This captures the state of the objects in the Cheney queue, which have been copied to region E, but still point to region $E-1$. The type of region E now contains three arrays: one for the finished objects that point to region E, one for the Cheney queue objects that still point to region $E-1$, and one for the free space.

$$\begin{aligned}
&\mathit{GcRegion}\,(R:K_R)\,(E:\text{int})\,(\mathit{Scan}:\text{int})\,(\mathit{Alloc}:\text{int}) :\hat{0} \triangleq \\
&\quad {}^\wedge\langle \text{Gen}(R\ E, \mathit{Alloc}), \\
&\quad (\mathit{LArray}\ 0\,\mathit{Scan}\,(\lambda I:\text{int}.\text{Mem}(\mathit{Addr}(E,I), R\,E\,I\,E))), \\
&\quad (\mathit{LArray}\ \mathit{Scan}\ \mathit{Alloc}\,(\lambda I:\text{int}. \\
&\qquad\qquad \text{Mem}(\mathit{Addr}(E,I), R\ E\ I\,(E-1)))), \\
&\quad (\mathit{LArray}\ \mathit{Alloc}\ \mathit{SIZE}\,(\lambda I:\text{int}. \\
&\qquad\qquad \exists\beta:\dot{1}\ .\text{Mem}(\mathit{Addr}(E,I),\beta))))\rangle \\
&K_R \triangleq \text{int}\rightarrow\text{int}\rightarrow\text{int}\rightarrow\dot{1}
\end{aligned}$$

Before the types get too complicated, we introduce a simplifying assumption: all regions have the same size $SIZE$, all even numbered regions use memory words $BASE\ldots(BASE+SIZE-1)$, and all odd numbered regions use memory words $(BASE+SIZE)\ldots(BASE+2*SIZE-1)$. The $(Base+I)$ from the previous definition of regions is changed to $Addr(E,I) \triangleq (BASE+SIZE*(E\,mod\,2)+I)$. The type language doesn't yet contain a *mod* operator, so the real Clay implementation of the collector uses two variables E_{hi} and E_{lo} to form a region number $2*E_{hi}+E_{lo}$, where $0 \leq E_{lo} \leq 1$. Since this would add much clutter and little illumination to the presentation below, this paper uses just the single variable E.

The simple collector in this section only supports one object type, *GcObject*, defined to contain a forwarding pointer, one floating point data field, and two possibly-null pointers to other *GcObject*s. The *GcObject* type describes the nonlinear type of each of the 4 words of the object:

$$\begin{aligned}
&\mathit{GcObject}\,(R:K_R)\,(E:\text{int})\,(\mathit{This}:\text{int}) :\dot{0} \triangleq \cdot\langle \\
&\qquad (\mathit{GcWord}\ R\ E\,(\mathit{This}+0)\,(\mathit{GcFwd}\ R)), \\
&\qquad (\mathit{GcWord}\ R\ E\,(\mathit{This}+1)\,(\mathit{GcPrim}\ \text{float})), \\
&\qquad (\mathit{GcWord}\ R\ E\,(\mathit{This}+2)\,(\mathit{GcPtr}\ R)), \\
&\qquad (\mathit{GcWord}\ R\ E\,(\mathit{This}+3)\,(\mathit{GcPtr}\ R))\rangle \\
&\mathit{GcWord}\,(R:K_R)\,(E:\text{int})\,(\mathit{This}:\text{int})\,(A:\text{int}\rightarrow\dot{1}) :\dot{0} \triangleq \cdot\langle \\
&\qquad \text{Eq}(R\ E\ \mathit{This}, A), \\
&\qquad \text{InDomain}(\mathit{This}, R\ E)\rangle
\end{aligned}$$

The abbreviation *GcWord* describes the type A of a single word *This* in a region E, where the type A is actually a type operator $(A:\text{int}\rightarrow\dot{1})$ that takes a region number E' as an argument. For non-pointer data types, like float, the E' argument is irrelevant, and the A operator always returns the same type:

$$\mathit{GcPrim}\,(T:\dot{1})\,(E':\text{int}) :\dot{1} \triangleq T$$

Pointer types, on the other hand, use the E' to identify the region containing the pointed-to object:

$$\begin{aligned}
&\mathit{GcPtr}\,(R:K_R)\,(E':\text{int}) :\dot{1} \triangleq \exists\alpha:\text{int}.\cdot\langle\text{Int}(\mathit{Addr}(E',\alpha)), \\
&\qquad \text{if}\,\mathit{Addr}(E',\alpha)=0\,\text{then}\ \cdot\langle\rangle\ \text{else}\,\mathit{GcObject}\,R\,E'\,\alpha\rangle
\end{aligned}$$

A non-null pointer is a pair of the singleton integer containing the real memory address of the target object, and a *GcObject* describing the state of the target object. A null pointer is just the singleton integer Int(0). Forwarding pointers are just like ordinary pointers, except that they point into the next region:

$$\mathit{GcFwd}\,(R:K_R)\,(E':\text{int}) :\dot{1} \triangleq \mathit{GcPtr}\,R\,(E'+1)$$

Garbage collection starts with to-space entirely free. Suppose that from-space is region number 5, to-space is region number 6, and root from-space pointer points to a *GcObject* in words $100\ldots103$ of region 5. Then the root pointer has type $(\mathit{GcPtr}\,R\,5)$, and from-space has four linear facts:

$$\begin{aligned}&\mathrm{Mem}(\mathit{Addr}(5,100), \mathit{GcFwd}\ R\ 5), \ldots\\ &\quad \ldots, \mathrm{Mem}(\mathit{Addr}(5,103), \mathit{GcPtr}\ R\ 5)\end{aligned}$$

The collector copies the root object to to-space with load and store expressions; the root object goes into the front of the Cheney queue, at words $0\ldots3$ of region 6:

$$\begin{aligned}&\mathrm{Mem}(\mathit{Addr}(6,0), \mathit{GcFwd}\ R\ 5), \ldots\\ &\quad \ldots, \mathrm{Mem}(\mathit{Addr}(6,3), \mathit{GcPtr}\ R\ 5)\end{aligned}$$

The types of the words in to-space are initially exactly the same as the types in from-space; the collector really does perform a blind copy from one-space to the other. Since each Has fact is a mixture of region number 6 and and region number 5, to-space's linear array takes care of the mismatch between the 6 and the 5:

$$\begin{aligned}&(\mathit{LArray}\,\mathit{Scan}\,\mathit{Alloc}\,(\lambda I : \mathrm{int}.\\ &\quad \mathrm{Mem}(\mathit{Addr}(E,I), R\ E\ I\ (E-1))))\end{aligned}$$

At this point, $\mathit{Scan} = 0$, $\mathit{Alloc} = 4$, $E = 6$, and $E - 1 = 5$, while the type sequence $(R\ 6)$ is defined as:

$$((R\ 6\ 0) = (\mathit{GcFwd}\ R)), \ldots, ((R\ 6\ 3) = (\mathit{GcPtr}\ R))$$

The collector constructs a $(\mathit{GcObject}\ R\ 6\ 0)$ with this sequence, which in turn is used to form a to-space pointer of type $(\mathit{GcPtr}\ R\ 6)$. In from-space, the root object's forwarding pointer has type $(\mathit{GcFwd}\ R\ 5)$, which is equivalent to $(\mathit{GcPtr}\ R\ 6)$, allowing the forwarding pointer to point to the to-space object.

Next, the collector starts to scan the Cheney queue, shifting words from the Cheney queue linear array to the finished object linear array. Since the finished object linear array uses facts of the form $\mathrm{Mem}(\mathit{Addr}(6,I), R\ 6\ I\ 6)$ instead of $\mathrm{Mem}(\mathit{Addr}(6,I), R\ 6\ I\ (6-1))$, the collector must figure out a way to convert each value of type $(R\ 6\ I\ 5)$ into a value of type $(R\ 6\ I\ 6)$:

- The collector sets the forwarding pointer to null with a store operation; it is easy to build a null pointer to any region number.
- For the float value, the collector merely observes that $\mathrm{Mem}(\mathit{Addr}(6,1), \mathit{GcPrim}\ \mathrm{float}\ 5)$ is equivalent to $\mathrm{Mem}(\mathit{Addr}(6,1), \mathit{GcPrim}\ \mathrm{float}\ 6)$, since *GcPrim* ignores the region number argument. No loads or stores are necessary.
- For each of the two pointers, the collector must convert a $(\mathit{GcPtr}\,R\,5)$ to a $(\mathit{GcPtr}\,R\,6)$. This is exactly the problem that the collector solved when it copied the root object. Thus, the collector just makes a blind copy of whatever from-space objects the $(\mathit{GcPtr}\,R\,5)$ pointers point to.

The collector continues scanning and copying until the Cheney queue is empty. As it progresses, it must keep track of the types of the objects in the Cheney queue, so that it can repeatedly pop the front object off the queue and process it. The fact that the queue is full of objects, each of size 4 words, is stored in a nonlinear array of type:

$$\begin{aligned}&(\mathit{NArray}\,(\mathit{Scan} \div 4)\,(\mathit{Alloc} \div 4)\\ &\qquad (\lambda\alpha : \mathrm{int}.(\mathit{GcObject}\ R\ 6\,(4 * \alpha))))\end{aligned}$$

NArray is exactly like *LArray*, except that it is nonlinear and holds nonlinear size-0 elements. As was the case with *mod*, the abstract machine doesn't actually have a division operation, but it simplifies the presentation to use one. The real Clay implementation changes the definitions to make *Scan* and *Alloc* object indices rather than word offsets, so that it multiplies them by 4 to compute the word offset rather than dividing by 4 to compute the object index.

8. RELATED WORK

The previous section compared our collectors to some other typed garbage collectors. Particularly close to our work, although not as low-level, is Monnier et al's second region calculus and collector[14]. The region calculus extends ordinary regions with a type operator that the program updates linearly at run-time; the types of the object are filtered by this type operator. The linear control over the type operator is similar to the way our linear arrays control the types of words in a region.

Separation logics[12][19] share our goal of using techniques derived from linear logic to describe data structures, although as far as we know, separation logics haven't yet been used to build regions and garbage collectors. In [10], we add an expression to λ^{low} that deduces $I_1 \neq I_2$ if a value of type $\mathrm{Mem}(I_1, \tau_1)$ and a value of type $\mathrm{Mem}(I_2, \tau_2)$ exist simultaneously; this is similar to an axiom in [19], and we've used it to develop a usable linear circular list type. Walker[1] explored a variant of separation logic that describes memory location adjacency, rather than treating locations as integers. It would be interesting to see how much of our λ^{low} and Clay code could be rewritten with adjacency rules rather than integer arithmetic rules.

Igarashi and Kobayashi[11] developed a hybrid memory management strategy combining garbage collection and linear data. However, the collector was not written in a safe language; rather, the focus was on making sure that linear data deallocation and traditional garbage collection did not interfere with each other. Petersen et al[17] also assume an existing garbage collector, but use an ordered variation of linearity to expose the memory allocation process to high level programs in a safe way.

9. CONCLUSIONS AND FUTURE WORK

This paper has demonstrated that linear types, augmented with support for simple arithmetic types, coercion functions, and type sequences, are sufficient to express a variety of type-safe data structures, ranging from simple lists to mark-sweep heaps to Cheney queues of nonlinear objects. Many issues remain. First, λ^{low} forces programs to contain many explicit coercions (pack, unpack, split, combine, type equality coercions, etc.) to set up simple loads and stores, making programming tedious. Second, perhaps our definition of type sequences could be simplified. For example, support for individual "holes" in types might prove as powerful as direct support for infinite vectors of holes. Third, our proof language is something of a hack, but it isn't yet clear what to replace it with. In particular, we're concerned that recursive types and type sequences will invalidate the strong normalization property of a typical proof language based on the Curry-Howard correspondence. On the other hand, if the nonlinear data structures from section 5 could be rewritten without recursive types, then maybe we could achieve growable regions with neither recursive types nor type sequences.

10. REFERENCES

[1] Amal Ahmed and David Walker. The logical approach to stack typing. In *003 ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2003.

[2] Andrew W. Appel. Foundational proof-carrying code. In *Logic in Computer Science*, 2001.

[3] Henry G. Baker. Lively linear Lisp — 'Look Ma, no garbage!'. *ACM SIGPLAN Notices*, 27(9):89–98, 1992.

[4] James Cheney and Greg Morrisett. A linearly typed assembly language. Technical report.

[5] Karl Crary and Joseph C. Vanderwaart. An expressive, scalable type theory for certified code. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 191–205. ACM Press, 2002.

[6] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275. ACM Press, 1999.

[7] Pascal Fradet and Daniel Le Metayer. Shape types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–39. ACM Press, 1997.

[8] N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof carrying-code, 2002.

[9] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[10] Heng Huang, Lea Wittie, and Chris Hawblitzel. Formal properties of linear memory types. Technical Report TR2003-468, Dartmouth College, 2003.

[11] A. Igarashi and N. Kobayashi. Garbage collection based on a linear type system, 2000.

[12] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[13] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 81–91. ACM Press, 2001.

[14] Stefan Monnier and Zhong Shao. Typed regions. Technical Report YALEU/DCS/TR-1242, Department of Computer Science, Yale University, 2002.

[15] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. volume 21, pages 527–568. ACM Press, 1999.

[16] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, 1996. USENIX.

[17] L. Petersen, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout, 2003.

[18] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM Press, 1991.

[19] J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002.

[20] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *ACM Symposium on Principles of Programming Languages*, 2002.

[21] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *In European Symposium on Programming*, 2000.

[22] S.W. Smith and S.H. Weingart. Building a high-performance, programmable secure coprocessor. In *Computer Networks (Special Issue on Computer Network Security)*, volume 31, pages 831–860, 1999.

[23] David Teller and Zhong Shao. Algorithm-independent framework for verifying integer constraints. Technical Report YALEU/DCS/TR-1195, Department of Computer Science, Yale University, 2000.

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

[25] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[26] David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071, 2001.

[27] David Walker and Kevin Watkins. On regions and linear types (extended abstract). In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 181–192. ACM Press, 2001.

[28] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 166–178. ACM Press, 2001.

[29] Ed Wei. Using low level linear memory management for type-preserving mark-sweep garbage collector (undergraduate thesis). Technical Report TR2003-465, Dartmouth College, 2003.

[30] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 249–257. ACM Press, 1998.