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) 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 correction 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, 16] 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.

Rather than focusing on the specifics of Clay, this paper presents the type system as part of an abstract machine called $\lambda^{low}$, based on the standard call-by-value higher-order polymorphic lambda calculus, $\lambda^c$. 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 $\lambda^{low}$ to many settings. For example, a closure-converted subset of $\lambda^{low}$ mimics C-like languages such as Clay, while a CPS- and closure-converted subset of $\lambda^{low}$ mimics register transfer languages or assembly-languages. Because of this, we believe that $\lambda^{low}$’s type system can contribute to the development of systems for proof-carrying code (PCC) [16] and typed assembly language (TAL) [13]. 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, 16], 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXX-XX-X/XX/XX ...$5.00.

1available 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 mem-
ory 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 strat-
egy 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 ex-
ample, the constraint \( \{ \ell_1 \mapsto (\text{int}, \text{int}) \} \)
indicates that there are two distinct objects at memory lo-
cations \( \ell_1 \) and \( \ell_2 \), 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 ex-
ample, storing a floating point number into the second ele-
ment of \( \ell_2 \) produces a new constraint \( \{ \ell_1 \mapsto (\text{int}, \text{int}) \}, \ell_2 \mapsto
(\text{int}, \text{float}, \text{int}) \} \). Constraints are treated linearly, so that the
old constraint is consumed when the new constraint is pro-
duced, which prevents the program from trying to use the
second field of \( \ell_2 \) as an integer after it has been changed to
a float. In contrast to pure linear types, the program may
keep many pointers to \( \ell_2 \) simultaneously—the type “pointer-
to-\( \ell_2 \)” is nonlinear and may be freely copied. The linearity
of the constraint ensures that any use of a “pointer-to-\( \ell_2 \)”
value respects the current state of \( \ell_2 \).

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 point-
ers. 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 oper-
ations 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 even more sophisticated ways to control aliasing (see
shape types, for example[7]), but most programming lan-
guages 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 life-
time is controlled linearly (or almost linearly). A pointer to
an object O inside a region R is assigned a type “\( \tau \) at R”,
where \( \tau \) describes the data inside O. Loads and stores to a
“\( \tau \) 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 us-
ually 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 advan-
tage 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 re-
gions that the program can deallocate as soon as possible to
minimize memory consumption. Wang and Appel[28] ob-
served 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 writ-
ten entirely with type-safe language features. Such a typed
(or type-preserving) collector offers the generality of tradi-
tional garbage collection without requiring the collector to
be trusted. Unfortunately, the details of the copying pro-
cess are troublesome; the collector needs some way to trave-
sure 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 [26][13][14], but each
new proposal changes the typing rules for regions, often in
ad-hoc and complex ways. Furthermore, the resulting col-
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 \( \lambda^{\text{low}} \).
To make the core language clearer, we delay a couple of fea-
tures (coercion functions and type sequences) until later sec-
tions. Type checking rules for expressions appear through-
out the paper; the complete syntax and semantics of \( \lambda^{\text{low}} \)
are found in [10]. For the portion of the language described
in this subsection, two type environments are needed to type-
check an expression: \( \Gamma \) maps variables to types, and \( \Psi \) maps
integer memory word addresses to types. For convenience,
we often write a combined context \( C \), defined in this sub-
section to be \( \Gamma \triangleq \Psi; \Gamma \). The notation \( C = C_1, C_2 \) indicates
that \( C_1 \) and \( C_2 \) share the same nonlinear assumptions,
but that each of \( C \)’s linear assumptions appears in either \( C_1 \)
or \( C_2 \), but not both. The notation \( C \) 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 \( \tau \) are defined as follows (kinds \( K \), integers \( I \), and booleans \( B \) are defined later in this section):

### Types

\[
\begin{align*}
\tau &= \tau_1 \overset{\phi}{\rightarrow} \tau_2 | \phi(\tau_1) | \text{Mem}(I, \tau) \\
| \alpha | \lambda \alpha : K \cdot \tau_1 \tau_2 | \forall \alpha : K ; B \cdot \tau_1 | \exists \alpha : K ; B \cdot \tau_1 [\mu \alpha : K, \tau_1] \\
| I | | | \text{Int}(I) | | | \text{Bool}(B) | | | \text{if} B \text{ then } \tau_1 \text{ else } \tau_2
\end{align*}
\]

### Expressions

\[
\begin{align*}
e &= i \mid b \mid x \mid e_1 e_2 \mid e \cdot \phi(e) \mid \lambda x : \tau \overset{\phi}{\rightarrow} e \mid e_1 o p e_2 \mid \neg e \\
| \Lambda \alpha : K ; B \cdot v \text{ let } (x) &= e_1 \text{ in } e_2 \text{ if } e_1 \text{ then } e_2 \text{ else } e_3 \\
| \text{pack}(\tau_1, e) &= \exists \alpha : K ; B \cdot \tau_1 \text{ unpack } \alpha, x = e \text{ in } e_2 \\
| \text{roll}(\mu \alpha : K \cdot \tau_1 \cdot \tau_2 \cdot \tau_3) &= \mu \alpha : K \cdot \tau_1 \cdot \tau_2 \cdot \tau_3 (e) \mid \text{fix } x : \tau. v \\
| \text{load}(\epsilon_{\text{ptr}}, \epsilon_{\text{Mem}}) &= \text{store}(\epsilon_{\text{ptr}}, \epsilon_{\text{Mem}}, \epsilon_v) \mid \text{fact}
\end{align*}
\]

### Values

\[
\begin{align*}
v &= i \mid b \mid \Lambda \alpha : K ; B \cdot v \mid \text{pack}(\tau_1, e) \mid \exists \alpha : K ; B \cdot \tau_2 [\phi(v)] \\
| \text{roll}(\mu \alpha : K \cdot \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot (\tau_4) [\phi(v)] &= \lambda x : \tau \overset{\phi}{\rightarrow} e \mid \text{fact}
\end{align*}
\]

Most of the types are borrowed from other languages, as discussed below. The one new type is the the “linear memory type”, \( \text{Mem}(I, \tau) \), 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 \( \{ \ell_1 \leftrightarrow (\tau_1, \tau_2), \ell_2 \leftrightarrow (\tau_3, \tau_4, \tau_5) \} \), the “locations” \( \ell_1 \) and \( \ell_2 \) 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:

\[
\{ 500 \leftrightarrow \tau_1, 501 \leftrightarrow \tau_2, 700 \leftrightarrow \tau_3, 701 \leftrightarrow \tau_4, 702 \leftrightarrow \tau_5 \}
\]

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 “\( \text{Mem}(500, \tau_1) \)”, and the state of the pair of integers at locations 500 and 501 can be stored in a linear tuple of type “\( \text{Mem}(500, \tau_1), \text{Mem}(501, \tau_2) \)”. Following Walker et al[27], the type system includes nonlinear and linear functions \( (\phi \overset{\phi}{\rightarrow} \tau_2) \) and nonlinear and linear tuples \( (\phi(\tau)) \), where \( \phi \) is \( \overset{\phi}{\rightarrow} \) to indicate linear data, and \( \phi \) is \( \phi(\tau) \) 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 \( \overset{\phi}{\rightarrow} (\tau_1, \tau_2) \) containing empty nonlinear tuples is legal), but nonlinear data structures cannot contain linear data (the nonlinear tuple type \( (\tau_1, \tau_2) \) 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 \( \hat{\alpha} \) describes all linear types of size \( n \) words, while \( \hat{\alpha} \) describes all nonlinear types of size \( n \) words. For example, the nonlinear empty tuple type \( \overset{\phi}{\rightarrow} \) has kind \( \hat{\alpha} \).

We use the \( \hat{\alpha} \) in the kind \( \hat{\alpha} \) to restrict operations on memory: values stored in memory must be exactly 1 word long. The type of singleton booleans \( \text{Bool}(\tau) \) has kind \( 1 \), for example, so that a boolean value fits in one word of memory. A triplet of booleans of type \( \overset{\phi}{\rightarrow} (\text{Bool}(\tau), \text{Bool}(\tau), \text{Bool}(\tau)) \), though, has kind \( 3 \), 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):

\[
\begin{align*}
\forall j. (\Phi; \Delta \vdash (\tau_j : n)) &\quad n = \sum_j n_j \\
\Phi; \Delta \vdash \hat{\alpha} : \hat{\alpha} &\quad n = \sum_j n_j \\
\Phi; \Delta \vdash \tau_j : n &\quad n = \sum_j n_j \\
\Phi; \Delta \vdash (\hat{\alpha} : \hat{\alpha}) &\quad n = \sum_j n_j \\
\Phi; \Delta \vdash I : \text{Int} \quad (\Phi; \Delta \vdash \tau : 1) &\quad n = \sum_j n_j \\
\Phi; \Delta \vdash \text{Mem}(I, \tau) : 0
\end{align*}
\]

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

\[
\begin{align*}
\lambda x : \overset{\phi}{\rightarrow} (\text{Mem}(500, \tau_1), \text{Mem}(501, \tau_2)) &\overset{\phi}{\rightarrow} \\
&\text{let } (x_1, x_2) = x \text{ in} \\
&\text{let } (y_1, x_1) = \text{load}(500, x_1) \text{ in} \\
&\text{let } (y_2, x_2) = \text{load}(501, x_2) \text{ in} \\
&\text{let } x''_1 = \text{store}(500, x_1, y_2) \text{ in} \\
&\text{let } x''_2 = \text{store}(501, x, y_1) \text{ in} \\
&\overset{\phi}{\rightarrow}
\end{align*}
\]

When \( x \) has type \( \text{Mem}(500, \tau_1) \), the expression \( \text{load}(500, x_1) \) 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 \( \text{Mem}(501, \tau_2) \). The typing rule for the expression load requires that the first argument \( \epsilon_{\text{ptr}} \) be a singleton integer of type \( \text{Int}(I) \), and the second argument \( \epsilon_{\text{Mem}} \) have type \( \text{Mem}(I, \tau) \) for some \( \tau \).

\[
\begin{align*}
C_1 \vdash \epsilon_{\text{ptr}} : \text{Int}(I) \quad C_2 \vdash \epsilon_{\text{Mem}} : \text{Mem}(I, \tau) \\
C_1, C_2 \vdash \text{load}(\epsilon_{\text{ptr}}, \epsilon_{\text{Mem}}) : \overset{\phi}{\rightarrow} (\tau, \text{Mem}(I, \tau))
\end{align*}
\]

Unlike the more usual type “\( \text{Int} \)”, which is the set of all integers, the singleton type \( \text{Int}(I) \) is the set containing only
one integer \( i \), so that the integer constant 500 has type 
\[ \text{Int}(500); \]
\[ C \vdash i : \text{Int}(i) \]

For the load expression, the \( J \) in \( \text{Int}(J) \) and \( \text{Mem}(J, \tau) \) 
must match, so that load(500, \( x_1 \)) type-checks in the example 
above, but load(4099, \( x_1 \)) would not type-check.

The expression store(500, \( x_1, x_2 \)) consumes \( x_1 \) and 
produces a value of type \( \text{Mem}(500, \tau_2) \), documenting the change 
to the type of the value at memory word 500 from \( \tau_1 \) to \( \tau_2 \):
\[ C_2 \vdash e_{\text{Mem}} : \text{Mem}(J, \tau_1) \]
\[ C_3 \vdash e_v : \tau_2 \]
\[ C_1 \vdash e_{\text{ptr}}, e_{\text{Mem}}, e_v : \text{Int}(I) \]
\[ \text{store}(e_{\text{ptr}}, e_{\text{Mem}}, e_v) : \text{Mem}(I, \tau_2) \]

Notice that the store expression overwrites whatever \( \tau_1 \) 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 
\( \text{Mem}(I, \tau) \) is only legal if \( \tau \) is nonlinear.

While the examples above show how to manipulate values 
of type \( \text{Mem}(i, \tau) \), they leave one question unanswered: 
what exactly is the value that has type \( \text{Mem}(i, \tau) \)? The 
syntax for the abstract machine defines a special value, “fact”, 
which is sort of a universal unit value for all types \( \text{Mem}(i, \tau) \). 
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 \( \tau \):
\[ C, i \mapsto \tau \vdash \text{fact} : \text{Mem}(i, \tau) \]

### 3.1 Integer and Boolean Polymorphism

The swap function described earlier only works for fixed 
types \( \tau_1 \) and \( \tau_2 \), and the particular memory addresses 500 
and 501. The type system allows polymorphism over base 
types (kind \( \hat{n} \)), integers (kind “int”), booleans (kind “bool”), 
and type operators (kind \( \text{K} \rightarrow \text{K}_2 \)), so the swap function 
can be rewritten with type:
\[ \forall \alpha_1 : \text{int}. \forall \alpha_2: \text{int}. \forall \beta_1 : 1. \forall \beta_2 : 1. \]
\[ ^\wedge \text{Int}(\alpha_1). \text{Int}(\alpha_2). \text{Mem}(\alpha_1, \beta_1). \text{Mem}(\alpha_2, \beta_2) \rightarrow^* \]
\[ ^\wedge \text{Mem}(\alpha_2, \beta_1). \text{Mem}(\alpha_1, \beta_2) \]

Following the approach of Xi et al.[30], the singleton integers 
of type \( \text{Int}(\alpha_1) \) and \( \text{Int}(\alpha_2) \) are the one-word run-time 
representations of the integer types \( \alpha_1 \) and \( \alpha_2 \). This avoids the 
need for full-blown dependent types; the run 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: \( (0 \leq \alpha_1 \land \alpha_1 \leq \alpha_2) \) is a legal boolean type, while 
\( \text{Int}(\alpha_1 + 2 \ast \alpha_2 + 1) \) is a legal singleton integer type that is 
the run-time representation of the integer type \( \alpha_1 + 2 \ast \alpha_2 + 1 \).

**arithmetic**

\[ i = \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots \]
\[ b = \text{true} \mid \text{false} \]

\[ I = \alpha \mid i \mid I_1 \oplus I_2 \]
\[ B = \alpha \mid b \mid \neg B \mid B_1 \land B_2 \mid I_1 \oplus I_2 \]
\[ \text{iop} = + \mid - \mid \ast \mid \wedge \mid \vee \]
\[ \text{cmp} = \leq \mid > \mid \leq \mid \geq \mid = \mid \\]
\[ \text{op} = \text{iop} \mid \text{bop} \mid \text{cmp} \]

In addition to the universal polymorphic type \( \forall \alpha : K, \tau \).

The system supports existential polymorphic types \( \exists \alpha : K, \tau \). 
For example, the type \( \exists \alpha : \text{int}. \text{Int}(\alpha) \) is the type of a 
singleton integer, for some unknown integer \( \alpha \); this type 
imitates the traditional type “int” (the type of all integers).

As another example, the type
\[ \exists \alpha : \text{int}. \exists \beta_1 : 1. \exists \beta_2 : 1. \]

\[ \forall \alpha : \text{int}. \forall \alpha_2: \text{int}. \forall \alpha_1 \leq \alpha_2. \forall \beta_1 : 1. \forall \beta_2 : 1. \]

Then (swap 3 6 ...) would be legal, but (swap 10 6 ...) 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 \( \Delta \), so that the combined type checking context is now
\[ C \Delta \Psi; \Delta; \Gamma; B. \]

### 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 \( \alpha \) in \( I_1 \) and \( I_2 \), the types \( I_1 \) and \( I_2 \) simplify to 
equal integer constants. For example, the type system considers 
\( \alpha_1 + 2 \ast \alpha_2 + 1 \) to be equivalent to \( 2 \ast \alpha_2 + 2 + \alpha_1 - 1 \). 
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 “\( \text{if } B \text{ then } \tau_1 \text{ else } \tau_2 \)”, 
which is used to create unions. Conceivably, either of these types 
could have kind int, so that they may be substituted for 
an integer variable \( \alpha \) 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 
\( 2 + (\text{if } \beta \text{ then } 3 \text{ else } 4) + (\alpha (\lambda \gamma: \text{int}. \gamma + 3)) \), 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 
bars the kinds \( \text{int} \) and \( \text{bool} \) from appearing as the return 
type of a type operator (e.g. “int”), so that integer and 
boolean types do not contain operator applications, and the 
type system disallows conditional types from having kind \( \text{int} 
\) or \( \text{bool} \):

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

\[ K = \{ \text{int} \mid \text{bool} \mid \forall \text{bool} \mid K \rightarrow J \} \]

\[ \Phi; \Delta, \alpha : K_0 \vdash \tau : J_0 \]
\[ \Phi; \Delta \vdash \lambda \alpha : K_\alpha, \tau : K_\alpha \rightarrow J_0 \]
\[ \Phi; \Delta \vdash \tau_1 : K_0 \rightarrow J_0 \quad \Phi; \Delta \vdash \tau_2 : K_0 \rightarrow J_0 \]
\[ \Phi; \Delta \vdash B : \text{bool} 
\quad \Phi; \Delta \vdash \tau_1 : J_0 
\quad \Phi; \Delta \vdash \tau_2 : J_0 \]
\[ \Phi; \Delta \vdash \text{if } B \text{ then } \tau_1 \text{ else } \tau_2 : J_0 \]

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:

\[ \Phi; B \vdash (\lambda \alpha : K_\alpha, \tau_0) \tau_1 \equiv [\alpha \mapsto \tau_0] \tau_1 \]
\[ \Phi; B \vdash (\text{if } \text{true} \text{ then } \tau_1 \text{ else } \tau_2) \equiv \tau_1 \]
\[ \Phi; B \vdash (\text{if } \text{false} \text{ then } \tau_1 \text{ else } \tau_2) \equiv \tau_2 \]

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

\[ \exists \alpha : \text{int}^\ast (\text{Int} (\alpha), \text{Mem} (\alpha, \tau_1), \text{Mem} (\alpha + 1, \tau_2)) \]

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 \( \tau \):

\[ \text{List} = \mu \beta : (\text{int} \rightarrow \hat{\beta}). \lambda \alpha_1 : \text{int} \text{if } \alpha_1 = 0 \text{then } ^\ast \{ \} \text{else } \exists \alpha_2 : \text{int}^\ast (\text{Mem} (\alpha_1, \text{Int} (\alpha_2)), \text{Mem} (\alpha_1 + 1, \tau), \beta \alpha_2) \]

For clarity, we’ll often use the \( \hat{\beta} \) symbol to define data types, rather than using the abstract machine’s official, but more cryptic, recursive type \( \mu \alpha : K. \tau \):

\[ \text{List} (\alpha_1 : \text{int}) \hat{\beta} \text{if } \alpha_1 = 0 \text{then } ^\ast \{ \} \text{else } \exists \alpha_2 : \text{int}^\ast (\text{Mem} (\alpha_1, \text{Int} (\alpha_2)), \text{Mem} (\alpha_1 + 1, \tau), \text{List} \alpha_2) \]

The type “\( \text{if } B \text{ then } \tau_1 \text{ else } \tau_2 \)” is equivalent to \( \tau_1 \) if \( B \) is true and to \( \tau_2 \) if \( B \) is false, so the \( \text{List} \) type contains an empty tuple if \( \alpha_1 = 0 \) and a three-word tuple for non-zero \( \alpha_1 \). The first two words of the tuple assert that memory locations \( \alpha_1 \) and \( \alpha_1 + 1 \) contain values of type \( \text{Int} (\alpha_2) \) (the pointer to the next element) and \( \tau \) (the data in the current element). By itself, the \( \text{Int} (\alpha_2) \) is just an arbitrary integer, but together with the \( \text{Mem} \) type in \( \text{List} \alpha_2 \) 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 \( l_1 \ldots l_n \):

\[ \text{Stack} (\alpha_1 : \text{int}) \ (\alpha_2 : \text{int}) \ (\alpha \vdash \hat{\beta} \text{if } \alpha_1 = \alpha_2 \text{then } ^\ast \{ \} \text{else } ^\ast (\text{Mem} (\alpha_1, \tau), \text{Stack} (\alpha_1 + 1, \alpha_2)) \]

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 \( c \) satisfies the following three constraints:

- It has no effect on the memory \( M \)
- 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 \( c \) on a real machine; a compiler can simply erase \( c \) 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:

\[ \text{LArray} (\alpha_1 : \text{int}) \ (\alpha_2 : \text{int}) \ (\beta : \text{int} \rightarrow \hat{\beta}). \hat{\beta} \text{if } \alpha_1 = \alpha_2 \text{then } ^\ast \{ \} \text{else } ^\ast (\beta \alpha_1, \text{LArray} (\alpha_1 + 1, \alpha_2 \beta) \]

The \( \text{LArray} \) type is similar to the \( \text{Stack} \) type, except for extra flexibility it is polymorphic over an operator \( \beta \) so that each array element \( \alpha_1 \) 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 \( \alpha_2 \ldots \alpha_3 - 1 \):

\[ \text{fix } \text{split} : \tau. \text{split}. \lambda \alpha_1 : \text{int}. \lambda \alpha_2 : \text{int}. \lambda \alpha_3 : \text{int}. \lambda \beta : \text{int} \rightarrow \hat{\beta} \]
\[ \quad \alpha_1 \leq \alpha_2 \leq \alpha_3. \lambda \text{arr} : (\text{LArray} \alpha_1 \alpha_3) \ (\alpha_2 \rightarrow \alpha_1) \]
\[ \quad \text{if } \alpha_1 = \alpha_2 \text{ then } ^\ast (\text{roll} \text{LArray} \alpha_1 \alpha_2 \beta (\hat{\beta}), \text{arr}) \]
\[ \quad \text{elselet } (\text{head}, \text{tail}) = \text{unroll(arr)} \]
\[ \quad \text{let } (\text{left}, \text{right}) = \text{split} (\alpha_1 + 1, \alpha_2 \alpha_3 \beta \text{tail} \in \]

5
Extensions to $\lambda$-low for coercion functions

\[
\text{limit} = \ell / (\infty)
\]

\[
\text{types}
\]

\[
\tau = \ldots \rightarrow \tau_1 \rightarrow_{\phi, \text{limit}} \tau_2
\]

\[
\text{expressions}
\]

\[
e = \ldots \mid \lambda x : \tau \rightarrow_{\phi, \text{limit}} \epsilon \text{ if } B \text{ then } e_1 \text{ else } e_2 \mid \text{coerce}(e)
\]

\[
\text{values}
\]

\[
v = \ldots \mid \lambda x : \tau \rightarrow_{\phi, \text{limit}} \epsilon
\]

It’s easy to define other coercions for the $\text{LArray}$ type: combining two adjacent arrays into a single array, constructing and destructing empty arrays, and constructing and destructing single-element arrays. These coercions suffice to implement familiar operations to get and set array elements. For example, given an expression of type

\[
(\text{LArray} \ 0 \ 10 (\lambda x : \text{int} \cdot \text{Mem}(500 + \alpha, \tau)), \text{describing an array of 10 elements of type } \tau \text{ stored in memory words } 500\ldots509, \text{ element 5 of the array is accessed by splitting the array into three arrays } 0..4, 5..5, \text{ and 6..9, retrieving the } \text{Mem}(505, \tau) \text{ fact from the single-element array } 5..5, \text{ performing a load or store to memory word 505, and then recombining the three arrays into the original array } 0..9).
\]

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 $\tau$) 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:

```c
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 $\beta$. The linear area of memory $\beta$ acts much like a region, while the nonlinear type $\text{List}{\beta}$ 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 List β value can be used to load and store from the region:

\[
List (β : 0) : 1 Δ \exists α : \text{int} . \langle \text{Int}(α), \text{List}(β), \text{Mem}(α, List(β)) \rangle
\]

Each list value contains one singleton integer Int(α), which is the address of the list element. To get the data from the list element, the program calls the nonlinear function of type \( β 0 β ^ {Δ} β \) (Mem(α, List(β), ...)). 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 Mem(α, List(β)) \( β 0 β ^ {Δ} β \) function to relinquish the linear memory type and reconstruct the region β.

The function \( β 0 β ^ {Δ} β ^ {Δ} β \) function in List β 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 List β type is non-linear, and the program can freely copy and discard lists. For example, here is an implementation of function “nth”:

\[
\text{nth} = \text{List} : 0 . \text{fix}
\]

\[
f : (\forall α : \text{int}. \text{Int}(α) \rightarrow (\text{List} β) β \rightarrow β \rightarrow (\text{List} β, β)) .
\]

\[
\text{List} : (\forall α : \text{int}. β α β λ x : \text{List} β \rightarrow λ r : β \rightarrow (β x r) \rightarrow f (α - 1) (n - (n - 1) x r');
\]

The nth function in λ\text{low} is similar to the C implementation, except that it takes a third argument η of type β, and returns r along with the desired list element (in the tuple ^{(x, r)} 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:

\[
\text{getnext} = \text{List} : 0 . \text{fix}
\]

\[
f : (\forall α : \text{int}. \lambda α : \text{List} β \rightarrow λ r : β \rightarrow (λ x : \text{List} β \rightarrow λ r : β \rightarrow \text{unpack} α, y = \text{unroll}(x) \in
\]

\[
\text{let} (\text{yptr}, yf) = y \text{in}
\]

\[
\text{let} (\text{mem}, zf) = yf r \text{in}
\]

\[
\text{let} (\text{x'}, mem') = \text{load}(\text{yptr}, \text{mem}) \in
\]

\[
^{(x', z f \text{mem'})}
\]

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

\[
\text{setnext} = \text{List} : 0 . \text{fix}
\]

\[
f : (\forall α : \text{int}. \lambda α : \text{List} β \rightarrow λ x : \text{List} β \rightarrow λ r : β \rightarrow (λ x : \text{List} β \rightarrow λ r : β \rightarrow \text{unpack} α, y = \text{unroll}(x) \in
\]

\[
\text{let} (\text{yptr}, yf) = y \text{in}
\]

\[
\text{let} (\text{mem}, zf) = yf r \text{in}
\]

\[
\text{zf} (\text{store}(\text{ypr}, \text{mem}, x'))
\]

With these functions, the C code is implemented with a function:

\[
fifth = \text{List} : 0 . \lambda α : \text{List} β \rightarrow λ r : β \rightarrow
\]

\[
\text{let} (\text{r', r}) = \text{nth} β 5 5 r \in
\]

\[
\text{setnext} β a b r'
\]

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:

\[
\text{Three} : 0 Δ \rightarrow (\text{Mem}(1, \text{List} Three), \text{Mem}(2, \text{List} Three), \text{Mem}(3, \text{List} Three))
\]

To create a list of type List 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:

\[
b = \text{roll}[\text{List} Three(2 \rightarrow \langle \text{Mem}(1, \text{List} Three), \text{Mem}(2, \text{List} Three), \text{Mem}(3, \text{List} Three) \rangle)
\]

\[
\text{pack}[2, \left< f_0 \right>] \text{as } α : \text{int} . \left< \text{Int}(α), \text{List} β \rightarrow β \rightarrow \text{Mem}(α, \text{List}(β), \ldots) \right>
\]

\[
\text{The function } f_0 \text{ must extract a Mem}(2, \text{List} Three) \text{ value from a Three value, and return this value along with a function that reconstructs the original Three value:
}\]

\[
f_0 = \lambda r : \text{Three} \rightarrow β \rightarrow
\]

\[
\text{let} (m_1, m_2, m_3) = r \text{ in}
\]

\[
^{(m_2, \lambda m'_2 : \text{Mem}(2, \text{List} Three) \rightarrow \lambda m_1, m_2, m_3)}
\]

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:

\[
\text{a} = \text{nth} (5, \&a);
\]

\[
\text{b} = \text{nth} (5, \&c);
\]

\[
\text{c} = \text{nth} (5, \&a);
\]

using three store operations, we assume we are given memory words 1, 2, and 3 initialized to some arbitrary junk types t_1, t_2, and t_3, which we overwrite:

\[
\lambda m : (\langle \text{Mem}(1, t_1), \text{Mem}(2, t_2), \text{Mem}(3, t_3) \rangle \rightarrow
\]

\[
\text{let} (m_1', m_2', m_3') = \langle \text{store}(1, m_1, b),
\]

\[
\text{store}(2, m_2, c),
\]

\[
\text{store}(3, m_3, a) \rangle \text{ in
}\]

\[
\text{fifth Three a} \rightarrow (m_1', m_2', m_3')
\]

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

\[
\text{struct LTree} \{
\text{struct List *list;}
\text{struct LTree *left;}
\text{struct LTree *right;}
\}
\]

This is represented as:

\[
\text{LTree} : (β : 0) : Δ \exists α : \text{int} . \langle \text{Int}(α), β β ^ {Δ} β \text{mem} \rangle
\]

\[
\text{τ}_{\text{mem}} = \langle \text{Mem}(α, \text{List}(β), \text{Mem}(α + 1, \text{LTree}(β), \text{Mem}(α + 2, \text{LTree}(β)))}\rangle
\]
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:

Three : n △ (Mem(1, List Three),
Mem(2, List Three),
Mem(3, List Three))

Four : n △ (Mem(1, List Four),
Mem(2, List Four),
Mem(3, List Four),
Mem(4, List Four))

Unfortunately, the type Three is embedded in the types of all the existing lists. It’s not clear how to coerce a list of type List Three to types 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 τ 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 underscore τ to start with, by leaving an abstract hole "?” in the type:

τ =

^⟨Mem(1, List τ), ^⟨Mem(2, List τ), ^⟨Mem(3, List τ), ?)⟩⟩

then we can later specify the hole "?” to be a particular type, such as Mem(4, List τ), 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:

\( (L\text{Array} 0 N (\lambda I : \text{int}.\text{Mem}(I, F I))) \)

As F is filled in, the region’s array grows to include more and more linear memory types Mem(I, F I).

Extensions to \( \lambda \text{low} \) for type sequences types

\[
\tau = \ldots | F^K | \text{Gen}(\tau, I) | \text{Eq}(\tau_1, \tau_2) | \text{InDomain}(I, \tau)
\]

expressions

\[
e = \ldots | \text{make_eq}(\tau) | \text{apply_eq}(\tau, e_1, e_2) | \text{new_seq}(J) | \text{discard_seq}(e) | \text{define_seq}(e, \tau) | \text{in_domain}(I_1, I_2, e_1, e_2)
\]

The extensions to \( \lambda \text{low} \) for handling type sequences are shown above. First, we define an equality type Eq(τ₁, τ₂), where a value of type Eq(τ₁, τ₂) is evidence that τ₁ is equivalent to τ₂. The make_eq(τ) expression creates a new equality value of type Eq(τ, τ), for any type τ, and the expression apply_eq(τ, e₁, e₂) uses an equality value e₁ of type Eq(τ₁, τ₂) to substitute e₁ for τ₁ in selected locations inside the type of e₂, effectively coercing e₂ to a different, but equivalent, type:

\[
C \vdash \tau : K
\]

\[
C \vdash \text{make_eq}(\tau) : \text{Eq}(\tau, \tau)
\]

\[
C_1, C_2 \vdash \tau_1 : K \rightarrow J \quad C_1 \vdash \text{eq}_0 : \text{Eq}(\tau_0, \tau_0)
\]

\[
C_1, C_2 \vdash \tau_0 : K \quad C_1, C_2 \vdash \tau_0 : K \quad C_2 \vdash e_f : \tau_1 \tau_0
\]

\[
C_1, C_2 \vdash \text{apply_eq}(\tau_f, e_0, e_f) : \tau_f \tau_0
\]

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

\[
C \vdash \text{new_seq}(J) : \exists F : (\text{int} \rightarrow J).\text{Gen}(F, 0)
\]

The allocation of new types in the sequence is controlled by a size-zero linear generator of type Gen(F, I). The expression define_seq(e_gen, τ), where e_gen has type 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 Gen(F, I + 1)
- a nonlinear proof that (F I) = τ, of type Eq(F I, τ)
- a nonlinear proof that I will always be in the domain of F (since sequences grow but don’t shrink), of type InDomain(I, F).

\[
C \vdash \text{Gen}(\tau_f, I) \quad C \vdash \tau_f \rightarrow J \quad C \vdash \tau_0 \rightarrow J
\]

\[
C \vdash \text{define_seq}(\tau_0, \tau_f) \wedge (\text{Gen}(\tau_f, I + 1), \text{Eq}(\tau_f, \tau_0), \text{InDomain}(I, \tau_f))
\]

The Eq(F I, τ) type is used to substitute τ for (F I) and vice-versa, via the apply_eq(τ_f, e_0, e_f) expression described above. The InDomain(I, F) type, when combined with the current generator of type Gen(F, I₂) produces evidence that 0 ≤ I₁ < I₂, which is useful when I₁ is an index into an array of length I₂:

\[
C_1, C_2 \vdash I_1 \vdash \text{int} \quad C_1, C_2 \vdash I_2 \vdash \text{int}
\]

\[
C_1 \vdash e_1 : \text{InDomain}(I_1, \tau_f) \quad C_2 \vdash e_2 : \text{Gen}(\tau_f, \tau_0)
\]

\[
C_1, C_2 \vdash \text{in_domain}(I_1, I_2, e_1, e_2) \wedge (\text{Know}(0 < I_1 < I_2), \text{Gen}(\tau_f, I_2))
\]

Here, Know(B) is an abbreviation for ∃a : bool; B : a.

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

\[
C \vdash e : \text{Gen}(\tau, I)
\]

\[
C \vdash \text{discard_seq}(e) : \langle \rangle
\]
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 \( \text{Base} \) and containing \( \text{Size} \) 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 \( \tau \) in the region is a triplet of type \( \langle \text{Int}(\text{Base} + I), \text{Eq}(F, I, \tau), \text{InDomain}(I, F) \rangle \). The region itself contains an array of linear facts, each of the type \( \text{Mem}(\text{Base} + I, F, I) \):

\[
\begin{align*}
\text{Region}(F : \text{int} \to 1)(\text{Base} : \text{int})(\text{Alloc} : \text{int}) \\
(\text{Size} : \text{int}) : 0 \triangleq ( \\
\text{Gen}(F, \text{Alloc}), \\
(\text{IArray Alloc} (\lambda \lambda : \text{int} \cdot \text{Mem}(\text{Base} + I, F, I))), \\
(\text{IArray Alloc Size} (\lambda \lambda : \text{int} \cdot \tau_{\text{free}})) \\
\tau_{\text{free}} \triangleq 3 \beta : 1 \cdot \text{Mem}(\text{Base} + I, \beta)
\end{align*}
\]

The first element of the \( \text{Region} \) 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 \( \text{Gen}(F, \text{Alloc}) \) value from the region to conclude that \( 0 \leq I < \text{Alloc} \).
- Now that \( I \) is known to be within the bounds of the \((\lambda\text{alloc} \Rightarrow \text{Alloc} \ldots)\) array, use element \( I \) of the array, of type \( \text{Mem}(\text{Base} + I, F, I) \), 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 \( x \) from type \( (F, I) \) to type \( \tau \).

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 \text{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 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 \( \langle \text{Int}(\text{Base} + I), \text{Eq}(F, I, \tau), \text{InDomain}(I, F) \rangle \) are still legal types, but without the \( \text{Mem}(\text{Base} + I, F, I) \) 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:

\[
\Phi = \{ \quad F_1^K, \phi_1 \vdash \{ 0 \mapsto \tau_1, 0, 1 \mapsto \tau_1, 1, \ldots \}, \\
F_2^K, \phi_2 \vdash \{ 0 \mapsto \tau_2, 0, 1 \mapsto \tau_2, 1, \ldots \}, \\
\ldots \}
\]

Each sequence is assigned an identifier \( F_i^K \), which acts as a type operator of kind \((\text{int} \to K_i) \). 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 \( C = \Psi \cdot \Phi \cdot \Delta ; I ; B; \text{limit} \). One tricky point: each \( F_i^K \) in \( \Phi \) must suffice to type-check both \( \text{Gen}(F_i^K, I) \) expressions, which are linear, and other expressions containing the type operator \( F_i^K \), which may be nonlinear. The \( \text{Gen}(F_i^K, I) \) expression is only valid in a context \( \langle ..., F_i^K, \phi_i \vdash \ldots \rangle \) where \( \phi_i = \emptyset \). 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 \( \Psi ; \Phi ; \Psi ; \Psi ; \text{true}; \text{limit} \vdash (M, e : \tau) \), and \( e \) is not a value, then \((M, e)\) steps.

**Strong normalization:** If \( \Psi ; \Phi ; \Psi ; \Psi ; \text{true}; \text{limit} \vdash (M, e : \tau) \), then (i) \((M, e)\) steps to \((M', e')\), then erase\((M, e)\) steps to erase\((M', e')\) in zero or one steps, (ii) if \( e \) is a value, then \((M, e)\) steps to some \((M', e')\) such that erase\((M', e')\) is erase\((M', e')\) in zero or more steps, (iii) if \( \text{erase}(M, e)\) steps to some \((M_{\text{erase}}, e_{\text{erase}})\), then \((M, e)\) steps in one or more steps to some \((M', e')\) that erase\((M_{\text{erase}}, e_{\text{erase}})\). 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 \( \lambda_{\text{low}} \)'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 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. \( \lambda_{\text{low}} \) can hide these operations inside coercions, 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 Weil[29] and at the URL http://www.cs.dartmouth.edu/~hawblitzl/. The technical report[10] also contains a translation from $\lambda^C$, a continuation-passing-style, closure-converted intermediate language developed for typed assembly language by Morrisett et al[13], to $\lambda^{low}$, using one of our copy collectors for memory management. Since $\lambda^{low}$ already provides a translation from a variant of Girard’s System-F (the polymorphic lambda calculus) to $\lambda^C$, this forms a complete translation from a high-level polymorphic language to $\lambda^{low}$. The details of the collector 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 $(RE)$ for each region number $E$ (the $E$ is supposed to stand for “epoch”), and each $(REI)$ describes the $i$th word of memory in region number $E$. If region $E$ is from-space, then a from-space object can name the type of the $i$th word in to-space using the type $(REI + 1)I$. To add even more flexibility, each word descriptor $(REI)$ is actually a type operator that takes a region number as an argument: $(REI + 1)I$ in 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 objects 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.

$$\text{GcRegion}(R : K_R)(E : \text{int}) (\text{Scan} : \text{int}) (\text{Alloc} : \text{int}) \Downarrow 0 \triangleleft$$

$\langle \text{Gen}(R, E, \text{Alloc}),$

$\langle \text{Array 0 Scan}(M : \text{int}. \text{Mem}(\text{Addr}(E, I), R E I E)),$

$\langle \text{Array Scan Alloc}(M I : \text{int},$

$\langle \text{Mem}(\text{Addr}(E, I), R E I (E - 1))),$

$\langle \text{Array Alloc SIZE}(M I : \text{int}),$

$\langle \exists \beta : 1. \text{Mem}(\text{Addr}(E, I), \beta)) \rangle$ 

$$K_R \Downarrow \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow 1$$

Before the types get too complicated, we introduce a simplifying assumption: all regions have the same $\text{SIZE}$, all even numbered regions use memory words $\text{BASE} \ldots (\text{BASE} + \text{SIZE} - 1)$, and all odd numbered regions use memory words $(\text{BASE} + \text{SIZE}) \ldots (\text{BASE} + 2* \text{SIZE} - 1)$. The $(\text{Base} + I)$ from the previous definition of regions is changed to $\text{Addr}(E, I) \triangleleft (\text{BASE} + \text{SIZE}* (E \text{mod} 2) + I)$. The type language doesn’t yet contain a $\text{mod}$ operator, so the real Clay implementation of the collector uses two variables $E_h$ and $E_o$ to form a region number $2 * E_h + E_o$, where $0 \leq E_o \leq 1$. 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, $\text{GcObject}$, defined to contain a forwarding pointer, one floating point data field, and two possibly-null pointers to other $\text{GcObjects}$. The $\text{GcObject}$ type describes the non-linear type of each of the 4 words of the object:

$$\text{GcObject}(R : K_R)(E : \text{int}) (\text{This} : \text{int}) 0 \triangleleft \langle$$

$\langle \text{GcWord} R E (E + 0) (\text{GcFwd} R),$$

$\langle \text{GcWord} R E (E + 1) (\text{GcPrim} \text{float}),$$

$\langle \text{GcWord} R E (E + 2) (\text{GcPtr} R),$$

$\langle \text{GcWord} R E (E + 3) (\text{GcPtr} R) \rangle$$

$$\text{GcWord}(R : K_R)(E : \text{int})(\text{This} : \text{int})(A : \text{int} \rightarrow 1) 0 \triangleleft \langle$$

$\langle \text{Eq}(R E \text{This}, A),$$

$\langle \text{InDomain}(R E) \rangle$$

The abbreviation $\text{GcWord}$ describes the type $A$ of a single word $\text{This}$ in a region $E$, where the type $A$ is actually a type operator $(A : \text{int} \rightarrow 1)$ 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:

$$\text{GcPrim}(T : 1)(E’ : \text{int}) 1 \triangleleft T$$

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

$$\text{GcPtr}(R : K_R)(E’ : \text{int}) 1 \triangleleft 3 \alpha : \text{int} : \langle \text{Int}(\text{Addr}(E’, \alpha)),$$

$\langle \text{If}(\text{Addr}(E’, \alpha) = 0 \text{then} : \langle \text{else} \text{GcObject} R E’ \alpha) \rangle$$

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

$$\text{GcFwd}(R : K_R)(E’ : \text{int}) 1 \triangleleft \text{GcPtr} R (E’ + 1)$$

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 $\text{GcObject}$ in words 100...103 of region 5. Then the root pointer has type $(\text{GcPtr} R 5)$, and from-space has four linear facts:
Mem(Addr(5, 100), GcFwd R 5), . . . , Mem(Addr(5, 103), GcPtr R 5)

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 . . . 3 of region 6:

Mem(Addr(6, 0), GcFwd R 5), . . . , Mem(Addr(6, 3), GcPtr R 5)

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 at region number 5, to-space’s linear array takes care of the mismatch between the 6 and the 5:

(LArray Scan Alloc (λA : int. Mem(Addr(E, I), R E I (E − 1))))

At this point, Scan = 0, Alloc = 4, E = 6, and E − 1 = 5, while the type sequence (R 6) is defined as:

((R 6 0) = (GcFwd R)), . . . , (R 6 3) = (GcPtr R))

The collector constructs a (GcObject R 6 0) with this sequence, which in turn is used to form a to-space pointer of type (GcPtr R 6). In from-space, the root object’s forwarding pointer has type (GcFwd R 5), which is equivalent to (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 Mem(Addr(6, I), R 6 I 6) instead of Mem(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 Mem(Addr(6, I), GcPrim float 5) is equivalent to Mem(Addr(6, I), GcPrim 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 (GcPtr R 5) to a (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 (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:

(NArray(Scan ÷ 4)(Alloc ÷ 4) (λA : int.(GcObject R 6 (4 + A))))

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 $\lambda_{low}$ that deduces $I_1 \neq I_2$ if a value of type Mem$I_1, \tau_1$ and a value of type 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 $\lambda_{low}$ 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, $\lambda_{low}$ 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 grow-
able regions with neither recursive types nor type sequences.

10. REFERENCES


