Low-Level Linear Memory Management (DRAFT 02/12/06)

Chris Hawblitzel Heng Huang Eric Krupski Edward Wei
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 type-preserving 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, arrays, and cyclic data structures out of the basic linear memory primitives, and then introduces *type sequences* for building regions of nonlinear data. Finally, it presents a Cheney queue type-preserving 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, embedded system code. Often, a small amount of low-level code (say, in a firewall, a network interface device driver, or secure coprocessor[12]) 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 type-preserving 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[18].

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[13][3] and garbage collection. Our implementation of type-preserving garbage collection improves over previous work[17][7] in its Cheney queue implementation, its explicit and space-efficient tag bits, and its low-overhead handling of polymorphism.

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[10], modified to support arbitrary-precision arithmetic, to check arithmetic constraints. The type-checked code is compiled to C++. All the examples in this paper are included with the Clay distribution[1].

Rather than focus on the specifics of Clay, this paper presents the type system as part of a abstract machine based on the standard call-by-value higher-order polymorphic lambda calculus, F^ω, in order to convey the ideas more completely and precisely, and in order to prove soundness. We hope that the type system will contribute to the development of systems for *proof-carrying code* (PCC)[9] and *typed assembly language* (TAL)[8]. 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*[1][4], 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 model of memory, they may be suitable for foundational PCC systems.

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

2. ALIAS TYPES AND LINEAR MEMORY

Our linear memory types are based on a simplified version of alias types[11][15], which 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 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.

Alias types deliberately abstract away the details of how the objects are allocated and where in memory in memory objects reside; 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:

$\{500 \mapsto \text{int}, 501 \mapsto \text{int}, 700 \mapsto \text{int}, 701 \mapsto \text{float}, 702 \mapsto \text{float}\}$

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[14] then suffice to hold the types of multiple memory words. For example, the memory location 500 is described by an object of linear type "$\text{Has}(500, \text{int})$", and the state of the pair of integers at locations 500 and 501 can be stored in a linear pair of type $\wedge\langle \text{Has}(500, \text{int}), \text{Has}(501, \text{int})\rangle$. Throughout the paper, we use $\wedge$ to indicate linear data and $\cdot$ to indicate nonlinear data. Linear data structures may hold nonlinear data (a linear pair type $\wedge\langle \text{int}, \text{int}\rangle$ of nonlinear integers is legal), but nonlinear data structures cannot contain linear data (the nonlinear pair type $\cdot\langle \text{Has}(500, \text{int}), \text{Has}(501, \text{int})\rangle$ is illegal), which ensures that linear data is never aliased. Operations on memory consume values of type $\text{Has}(i, \tau)$ and produce new values of type $\text{Has}(i, \tau')$, so that at all times there is exactly one value of type $\text{Has}(i, \tau)$ for each memory location i. For example, the following function swaps two words of memory, consuming a pair of values of type $\wedge\langle \text{Has}(500, \tau_1), \text{Has}(501, \tau_2)\rangle$ and producing a pair of values of type $\wedge\langle \text{Has}(501, \tau_1), \text{Has}(500, \tau_2)\rangle$:

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

When x_1 has type $\text{Has}(500, \tau_1)$, 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 $\text{Has}(500, \tau_1)$. The expression $\text{store}(500, x_1', y_2)$ consumes x_1' and produces a value of type $\text{Has}(500, \tau_2)$, documenting the change to the type of the value at memory word 500 from τ_1 to τ_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 $\text{Has}(i, \tau)$ is only legal if τ is nonlinear.

2.1 Formalities for the Abstract Machine

While the example above shows how to manipulate values of type $\text{Has}(i, \tau)$, it leaves one question unanswered: what exactly is the value that has type $\text{Has}(i, \tau)$? The syntax for the abstract machine defines a special value, "fact", which is sort of a universal unit value for all types $\text{Has}(i, \tau)$. The fact value carries no run-time information, though, and is not represented at all in a real machine. Stated another way, fact consumes zero bits of memory. We use kinds to distinguish between types with different sizes and linearities: if τ has kind $\overset{\cdot 1}{\text{type}}$, then a value of type τ is nonlinear and has size 1 word; a tuple type $\cdot\langle \tau, \tau, \tau\rangle$ of three one-word τ values has kind $\overset{\cdot 3}{\text{type}}$. The type $\text{Has}(i, \tau)$ has kind $\overset{\wedge 0}{\text{type}}$, which means that it is linear and has size 0.

The abstract machine models memory using a heap M that maps integer memory word addresses to one-word values. The evaluation rules (in appendix A) 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.

For the subset of the language described so far, 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 to be $C \triangleq \Psi; \Gamma$.

2.2 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 "$\overset{\phi i}{\text{type}}$", where $\phi = \wedge$ or $\phi = \cdot$), integers (kind "int"), booleans (kind "bool"), and type operators (kind $K_1 \rightarrow K_2$), so the swap function can be rewritten with type:

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

Following the approach of Xi et al[18], 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 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: $(0 \le \alpha_1 \wedge \alpha_1 \le \alpha_2)$ is a legal boolean type, while $\text{Int}(\alpha_1 + 2 * \alpha_2 + 1)$ is a legal singleton integer type that is the run-time representation of the integer type $\alpha_1 + 2 * \alpha_2 + 1$.

To keep the form of integer and boolean types simple enough for the constraint solver, the kind system bans the kinds int and bool from appearing as the return type of a type operator (e.g. $\text{type} \overset{\cdot 1}{\to} \text{int}$), so that integer and boolean types do not contain operator applications.

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:

$$\forall\alpha_1 : \text{int}.\forall\alpha_2 : \text{int}; \alpha_1 \leq \alpha_2.\forall\beta_1 :\overset{\cdot 1}{\text{type}} .\forall\beta_2 :\overset{\cdot 1}{\text{type}} \ldots.$$

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 $C \triangleq \Psi; \Delta; \Gamma; B$.

2.3 Lists and Stacks

The singleton integers in the the previous section's swap function act as pointers to the memory words containing values of type β_1 and β_2. Thus, the type system is powerful enough to express pointer-based data structures in the style of recursive alias types[15]. For example, the following type[2] defines a simple linked list terminated by 0, containing data elements of type τ :

$$\begin{aligned}&\mathit{List}\,(\alpha_1 : \text{int}) :\overset{\wedge 0}{\text{type}} \triangleq \text{Union}(\alpha_1 = 0,\\ &\quad {}^\wedge\langle\rangle,\\ &\quad \exists\alpha_2 : \text{int}.{}^\wedge\langle \text{Has}(\alpha_1, \text{Int}(\alpha_2)), \text{Has}(\alpha_2 + 1, \tau), \mathit{List}\,\alpha_2\rangle)\end{aligned}$$

The type $\text{Union}(B, \tau_1, \tau_2)$ contains a τ_1 if B is true and a 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 $\text{Int}(\alpha_2)$ (the pointer to the next element) and τ (the data in the current element). By itself, the $\text{Int}(\alpha_2)$ is just an arbitrary integer, but together with the Has facts in the $(\mathit{List}\,\alpha_2)$ value 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 $\alpha_1 \ldots \alpha_2 - 1$:

$$\begin{aligned}&\mathit{Stack}\,(\alpha_1 : \text{int})\,(\alpha_2 : \text{int}) :\overset{\wedge 0}{\text{type}} \triangleq\\ &\quad \text{Union}(\alpha_1 = \alpha_2,\ {}^\wedge\langle\rangle, {}^\wedge\langle \text{Has}(\alpha_1, \tau), \mathit{Stack}\,(\alpha_1 + 1)\,\alpha_2\rangle)\end{aligned}$$

3. 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 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 than 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 Has fact 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:

$$\begin{aligned}&\mathit{LArray}\,(\alpha_1 : \text{int})\,(\alpha_2 : \text{int})\,(\beta : \text{int} \to \overset{\wedge 0}{\text{type}}) :\overset{\wedge 0}{\text{type}} \triangleq\\ &\quad \text{Union}(\alpha_1 = \alpha_2, {}^\wedge\langle\rangle, {}^\wedge\langle \beta\,\alpha_1,\ \mathit{LArray}\,(\alpha_1 + 1)\,\alpha_2\,\beta\rangle)\end{aligned}$$

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 $\alpha_2 \ldots \alpha_3 - 1$:

$$\begin{aligned}&\text{fix}\,\mathit{split} : \tau_{split}.\Lambda\alpha_1 : \text{int}.\Lambda\alpha_2 : \text{int}.\Lambda\alpha_3 : \text{int}.\\ &\Lambda\beta : \text{int} \to \overset{\wedge 0}{\text{type}}; \alpha_1 \leq \alpha_2 \leq \alpha_3.\\ &\quad \lambda arr : (\mathit{LArray}\,\alpha_1\,\alpha_3) \overset{\cdot,(\alpha_2-\alpha_1)}{\longrightarrow} \text{if}\,\alpha_1 = \alpha_2\\ &\quad \text{then}\,{}^\wedge\langle \text{union}(\text{true}, {}^\wedge\langle\rangle, \tau_{next}, {}^\wedge\langle\rangle), arr\rangle\\ &\quad \text{else let}\,\langle head, tail\rangle = \text{case}(\text{false}, arr)\,\text{in}\\ &\quad\quad \text{let}\,\langle left, right\rangle = \mathit{split}\,(\alpha_1 + 1)\,\alpha_2\,\alpha_3\,\beta\,tail\,\text{in}\\ &\quad\quad {}^\wedge\langle \text{union}(\text{false}, {}^\wedge\langle\rangle, \tau_{next}, {}^\wedge\langle head, left\rangle), right\rangle\\ &\tau_{next} = {}^\wedge\ \langle \beta\,\alpha_1,\ \mathit{LArray}\,(\alpha_1 + 1)\,\alpha_3\,\beta\rangle\\ &\tau_{split} \triangleq \forall\alpha_1 : \text{int}.\forall\alpha_2 : \text{int}.\forall\alpha_3 : \text{int}.\forall\beta : \text{int} \to \overset{\wedge 0}{\text{type}}; \alpha_1 \leq\\ &\alpha_2 \leq \alpha_3.(\mathit{LArray}\,\alpha_1\,\alpha_3) \overset{\cdot,(\alpha_2-\alpha_1)}{\longrightarrow}\\ &\quad {}^\wedge\langle(\mathit{LArray}\,\alpha_1\,\alpha_2), (\mathit{LArray}\,\alpha_2\,\alpha_3)\rangle\end{aligned}$$

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 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. The type system only allows coercion functions to call functions with lower limits

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

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). 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, since it implies that some run-time representation of α_1 and α_2 must 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 "$\text{if}\, B \,\text{then}\, e_1 \,\text{else}\, e_2$".

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 $(LArray\ 0\ 10\,(\lambda\alpha : int.\text{Has}(500+\alpha, \tau)))$, 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 $Has(505, \tau)$ 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 might be more powerful, more amenable to automated proof generation tools, and easier to connect to existing libraries of proofs. The LTT system[2], for example, embeds LF[5] proofs in a separate programming language.

3.1 Cyclic Linear Data Structures

Coercion functions also enable efficient linked data structures. Suppose that instead of being terminated by the condition $\alpha = 0$, the *List* type is terminated by $\alpha = \alpha_A$, where α_A is the first element of the list (this assumes there is at least one element in the list). Then, for example, four elements A,B,C,D can be arranged in a circular list $A \to B \to C \to D \to A$. Stepping forward one element in the list should change the list to $B \to C \to D \to A \to B$, where the list is now terminated by α_B. Although such a step doesn't modify the list in memory, satisfying the type checker requires traversing the list to unpack α_B, α_C, and α_D, and then repacking everything with α_A buried furthest back in the list. This is a perfect job for a coercion function, but with one catch: since the terminating list element is now α_B, not α_A, repacking requires proving $\alpha \neq \alpha_B$ for nodes C, D, and A. Luckily, since Has facts are linear, the existence of two distinct facts $e : \text{Has}(\alpha, \tau)$ and $e_B : \text{Has}(\alpha_B, \tau_B)$ implies that $\alpha \neq \alpha_B$. The special expression "$\text{distinguish}(\alpha, \alpha_B, e, e_B)$" produces an existential proving that $\alpha \neq \alpha_B$.

4. TYPE SEQUENCES AND REGIONS

While the earlier sections argue that linear types can implement many common data structures efficiently, including arrays and simple cyclic data structures, linear data structures are too difficult to use for many applications. Furthermore, compiler-generated proof-carrying code and typed assembly language must support nonlinear data types in the low-level language in order to express the nonlinear data types found in most high-level languages. Unfortunately, nonlinear data structures are harder to deallocate safely, since pointer aliasing makes it difficult to detect dangling pointers. Two well-known approaches to safe nonlinear deallocation are *regions* and *garbage collection*, and we claim that linear memory types can serve as a foundation for both of these approaches. This section describes how to implement regions, and section 5 uses type-preserving garbage techniques[17][7] to build a garbage collector by copying live data from one region to another.

Nonlinear memory is perfectly safe if it is never deallocated. For example, if a program allocates a float in memory location 300, and it dedicates memory location 300 to holding only floating point numbers for the remainder of the program execution, then it is easy to prove the safety of subsequent loads and stores to word 300, even in the presence of pointer aliasing. While a "never deallocate" strategy is rarely useful for a real physical resource, imagine an abstract resource that supports unlimited allocation and never requires deallocation. Just as our real memory M has an environment Ψ mapping integer memory locations to types, this abstract allocation area will have a mapping F from integers to types. The difference is that the abstract area *only* has an environment: there is no actual memory storage associated with it to store values. The real storage will still be in M, and linear facts of type $\text{Has}(I_1, F(I_2))$ will connect the memory M to the types controlled by the environment F. Since $\text{Has}(I_1, F(I_2))$ is linear, this connection between $M(I_1)$ and $F(I_2)$ can be established and broken at any time; the establishment of a connection corresponds to allocating a word in a region, and the breaking of a set of connections corresponds to deallocating a region.

We refer to F as a *type sequence*, and the allocation of new types in the sequence is controlled by a size-zero linear *generator* of type $\text{Gen}(F, I)$. The expression $\text{define\_fun}(e_{generator}, \tau)$, where $e_{generator}$ has type $\text{Gen}(F, I)$, 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 $\text{Gen}(F, I+1)$
- a nonlinear proof that $F(I) = \tau$, of type $\text{Eq}(F\,I, \tau)$
- a nonlinear proof that I will always be in the domain

of F (since sequences grow but don't shrink), of type $\mathrm{InDomain}(I, F)$.

The nonlinearity of the $\mathrm{Eq}(F\,I, \tau)$ and $\mathrm{InDomain}(I, F)$ 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 $\cdot\langle Int(Base+\alpha), \mathrm{Eq}(F\,\alpha, \tau), \mathrm{InDomain}(\alpha, F)\rangle$. The region itself contains an array of linear facts, each of the type $\mathrm{Has}(Base+\alpha, F\,\alpha)$:

$$
\begin{array}{l}
Region\,(F : \mathrm{int} \rightarrow \overset{\cdot 1}{\mathrm{type}})\,(Base : \mathrm{int})\,(Alloc : \mathrm{int}) \\
\quad (Size : \mathrm{int}) : \overset{\wedge 0}{\mathrm{type}} \triangleq\ {}^\wedge\langle \\
\qquad \mathrm{Gen}(F, Alloc), \\
\qquad (LArray\ 0\ Alloc\,(\lambda\alpha : \mathrm{int}.\mathrm{Has}(Base+\alpha, F\,\alpha))), \\
\qquad (LArray\ Alloc\ Size\,(\lambda\alpha : \mathrm{int}.\tau_{free}))\rangle \\
\tau_{free} \triangleq \exists\beta : \overset{\cdot 1}{\mathrm{type}}.\mathrm{Has}(Base+\alpha, \beta)
\end{array}
$$

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:

- A special expression "$\mathrm{in\_domain}(\alpha, Alloc, e_{InDomain}, e_{generator})$" uses the $\mathrm{InDomain}(\alpha, F)$ value from the pointer together with the $\mathrm{Gen}(F, Alloc)$ value from the region to conclude that $0 \le \alpha < Alloc$.

- Now that α is known to be within the bounds of the $(LArray\,0\,Alloc\ldots)$ array, the element α of the array, of type $\mathrm{Has}(Base+\alpha, F\,\alpha)$, is used to load a value of type $(F\,\alpha)$. Call this loaded value x.

- Since x has type $(F\,\alpha)$ and the $\mathrm{Eq}(F\,\alpha, \tau)$ value from the pointer says that $(F\,\alpha)$ is equivalent to τ, the type checker can conclude that x has type τ. Actually, our current type checker isn't so smart: the program must perform an explicit coercion to coerce x from type $(F\,\alpha)$ to type τ.

Storing a word follows a similar sequence of steps.

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 $(F\,\alpha)$ types is lost. The pointer types $\cdot\langle Int(Base+\alpha), \mathrm{Eq}(F\,\alpha, \tau), \mathrm{InDomain}(\alpha, F)\rangle$ are still legal types, but without the $\mathrm{Has}(Base+\alpha, F\,\alpha)$ 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:

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

Each sequence is assigned an identifier $F_i^{K_i}$, which acts as a type operator of kind $(\mathrm{int} \rightarrow K_i)$. We've been a little slippery about what exactly what 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 $\mathrm{Gen}(F_i^{K_i}, I)$ expressions, which are linear, and other expression containing the type operator $F_i^{K_i}$, which may be nonlinear. The $\mathrm{Gen}(F_i^{K_i}, I)$ expression is only valid in a context where $\phi_i = {}^\wedge$.

5. GARBAGE COLLECTION

This section describes a type-preserving garbage collector implemented using the regions from the previous section. It follows the ideas of Wang et al[17] and Monnier et al[7], but adds several new improvements:

- The garbage collector is a genuine Cheney queue, in which all the collection state is stored in to-space. In contrast to the previously described type-preserving garbage collectors, it does 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. The collector described in this section hides these operations inside coercion functions, which impose no run-time space or time cost.

- The collector supports cyclic data structures, correctly handling mixtures of pointers that may be null and pointer types that don't allow null.

- The collector makes all 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 type-preserving collector.

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.

$$
\begin{array}{l}
\mathit{GcRegion}\,(R : \mathrm{int} \to \mathrm{int} \to \mathrm{int} \overset{\cdot 1}{\to} \mathrm{type})\,(E : \mathrm{int}) \\
\quad (\mathit{Scan} : \mathrm{int})\,(\mathit{Alloc} : \mathrm{int}) :\overset{\wedge 0}{\mathrm{type}} \triangleq {}^{\wedge}\langle \\
\quad\quad \mathrm{Gen}(R\ E, \mathit{Alloc}), \\
\quad\quad (\mathit{LArray}\ 0\ \mathit{Scan}\,(\lambda\alpha : \mathrm{int}.\mathrm{Has}(\mathit{Addr}(E,\alpha), R\,E\,\alpha\,E))), \\
\quad\quad (\mathit{LArray}\ \mathit{Scan}\ \mathit{Alloc}\,(\lambda\alpha : \mathrm{int}. \\
\quad\quad\quad\quad \mathrm{Has}(\mathit{Addr}(E,\alpha), R\ E\ \alpha\,(E-1)))), \\
\quad\quad (\mathit{LArray}\ \mathit{Alloc}\ \mathit{SIZE}\,(\lambda\alpha : \mathrm{int}. \\
\quad\quad\quad\quad \exists\beta :\overset{\cdot 1}{\mathrm{type}} .\mathrm{Has}(\mathit{Addr}(E,\alpha),\beta))))\rangle
\end{array}
$$

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 $(Base+\alpha)$ from the previous definition of regions is changed to $\mathit{Addr}(E,\alpha) \triangleq (BASE+SIZE*(E\,\mathit{mod}\,2)+\alpha)$. 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 $0 \le E_{lo} \le 1$. Since this would add much clutter and little illumination to the presentation below, this paper uses just the single variable E.

Before describing the fully general garbage collector, it helps to describe a simpler collector that 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:

$$
\begin{array}{l}
\mathit{GcObject}\,(R : \mathrm{int} \to \mathrm{int} \to \mathrm{int} \overset{\cdot 1}{\to} \mathrm{type})\,(E : \mathrm{int}) \\
\quad (\mathit{This} : \mathrm{int}) :\overset{\cdot 0}{\mathrm{type}} \triangleq \cdot\langle \\
\quad\quad (\mathit{GcWord}\ R\ E\,(\mathit{This}+0)\,(\mathit{GcFwd}\ R)), \\
\quad\quad (\mathit{GcWord}\ R\ E\,(\mathit{This}+1)\,(\mathit{GcPrim}\ \mathrm{float})), \\
\quad\quad (\mathit{GcWord}\ R\ E\,(\mathit{This}+2)\,(\mathit{GcPtr}\ R)), \\
\quad\quad (\mathit{GcWord}\ R\ E\,(\mathit{This}+3)\,(\mathit{GcPtr}\ R))\rangle \\
\mathit{GcWord}\,(R : \mathrm{int} \to \mathrm{int} \to \mathrm{int} \overset{\cdot 1}{\to} \mathrm{type})\,(E : \mathrm{int}) \\
\quad (\mathit{This} : \mathrm{int})\,(A : \mathrm{int} \overset{\cdot 1}{\to} \mathrm{type}) :\overset{\cdot 0}{\mathrm{type}} \triangleq \cdot\langle \\
\quad\quad \mathrm{Eq}(R\ E\ \mathit{This}, A), \\
\quad\quad \mathrm{InDomain}(\mathit{This}, R\ E)\rangle
\end{array}
$$

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 $(A : \mathrm{int} \overset{\cdot 1}{\to} \mathrm{type})$ 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:

$$
\mathit{GcPrim}\,(T :\overset{\cdot 1}{\mathrm{type}})\,(E' : \mathrm{int}) :\overset{\cdot 1}{\mathrm{type}} \triangleq T
$$

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

$$
\begin{array}{l}
\mathit{GcPtr}\,(R : \mathrm{int} \to \mathrm{int} \to \mathrm{int} \overset{\cdot 1}{\to} \mathrm{type})\,(E' : \mathrm{int}) :\overset{\cdot 1}{\mathrm{type}} \triangleq \\
\quad \exists\alpha : \mathrm{int}.\cdot\langle \mathrm{Int}(\mathit{Addr}(E',\alpha)), \\
\quad\quad \mathrm{Union}(\mathit{Addr}(E',\alpha) = 0, \cdot\langle\rangle, \mathit{GcObject}\,R\,E'\,\alpha)\rangle
\end{array}
$$

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:

$$
\begin{array}{l}
\mathit{GcFwd}\,(R : \mathrm{int} \to \mathrm{int} \to \mathrm{int} \overset{\cdot 1}{\to} \mathrm{type})\,(E' : \mathrm{int}) :\overset{\cdot 1}{\mathrm{type}} \triangleq \\
\quad\quad \mathit{GcPtr}\,R\,(E'+1)
\end{array}
$$

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:

$$
\begin{array}{l}
\mathrm{Has}(\mathit{Addr}(5,100), \mathit{GcFwd}\ R\ 5), \ldots \\
\quad\quad \ldots, \mathrm{Has}(\mathit{Addr}(5,103), \mathit{GcPtr}\ R\ 5)
\end{array}
$$

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:

$$
\begin{array}{l}
\mathrm{Has}(\mathit{Addr}(6,0), \mathit{GcFwd}\ R\ 5), \ldots \\
\quad\quad \ldots, \mathrm{Has}(\mathit{Addr}(6,3), \mathit{GcPtr}\ R\ 5)
\end{array}
$$

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:

$$(\mathit{LArray}\,\mathit{Scan}\,\mathit{Alloc}\,(\lambda\alpha : \mathrm{int}.\\ \quad \mathrm{Has}(\mathit{Addr}(E,\alpha), R\ E\ \alpha\,(E-1))))$$

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 $\mathrm{Has}(\mathit{Addr}(6,\alpha), R\ 6\ \alpha\ 6)$ instead of $\mathrm{Has}(\mathit{Addr}(6,\alpha), R\ 6\ \alpha\,(6-1))$, the collector must figure out a way to convert each value of type $(R\ 6\ \alpha\ 5)$ into a value of type $(R\ 6\ \alpha\ 6)$:

- 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 $\mathrm{Has}(\mathit{Addr}(6,1), \mathit{GcPrim}\ \mathrm{float}\ 5)$ is equivalent to $\mathrm{Has}(\mathit{Addr}(6,1), \mathit{GcPrim}\ \mathrm{float}\ 6)$, 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 the full of objects, each of size 4 words, is stored in a nonlinear array of type:

$$(\mathit{NArray}\,(\mathit{Scan} \div 4)\,(\mathit{Alloc} \div 4)\\ \quad (\lambda\alpha : \mathrm{int}.(\mathit{GcObject}\ R\ 6\,(4 * \alpha))))$$

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.

5.1 Garbage Collector Polymorphism

A garbage collector that only copies one type of object could be useful for languages without a static type system, but would not work well for statically typed languages like Java and ML. This section of the paper describes a new version of the *GcObject* type, flexible enough to express many different types of objects:

$$\mathit{GcObject}\,(R : \mathrm{int} \to \mathrm{int} \to \mathrm{int} \overset{\cdot 1}{\to \mathrm{type}})\,(E : \mathrm{int})\\ (\mathit{This} : \mathrm{int})\,(F : \mathrm{int} \to \mathrm{int} \overset{\cdot 1}{\to \mathrm{type}})\,(A : \mathrm{int} \to \mathrm{int} \overset{\cdot 0}{\to \mathrm{type}})\\ (N : \mathrm{int})\,(\mathit{Tag} : \mathrm{int})\,(\mathit{Extra} \overset{\cdot 0}{: \mathrm{type}}) \overset{\cdot 0}{: \mathrm{type}} \triangleq\\ \quad \cdot\langle \tau_{header}, \tau_{fieldlist}, \tau_{fieldarray}, \tau_{coerceback}, \mathit{Extra}\rangle$$

The first difference from the previous section is that the *GcWord* facts are kept in an array rather than in a tuple, so that a *GcObject* may contain N data fields following the header word instead of 3 data fields following the header word:

$$\tau_{fieldarray} \triangleq (\mathit{NArray}\ 0\ N\\ \quad (\lambda\alpha : \mathrm{int}.(\mathit{GcWord}\ R\ E\,(\mathit{This} + \alpha + 1)\,(F\,\alpha))))$$

Furthermore, each field's type is customizable through the type operator F, whose elements can be defined using a type sequence generator. The collector is polymorphic over all N and F, enabling support for any number of fields, each field of any type. However, the collector needs to know *something* about the pointers in the object in order to update them during the collection, so each *GcObject* must contain a header word identifying the number of fields in the object, and a tag bit for each field. A tag bit of 1 indicates a pointer field, while a tag bit of 0 indicates a non-pointer field.

```
...-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       tag bits           |    N    |1|
...-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Alternately, the header word may hold a forwarding pointer. The rightmost bit of 1 specifies a header word containing tag bits, while a rightmost bit of 0 specifies a header word containing a forwarding pointer.

```
...-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          forwarding pointer        |0|
...-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The type for the header word is:

$$\tau_{header} \triangleq (\mathit{GcWord}\ R\ E\,(\mathit{This} + 0)\,(\mathit{GcHdr}\ A\ \mathit{Tag}\ N))$$

where

$$\mathit{GcHdr}\,(A : \mathrm{int} \to \mathrm{int} \overset{\cdot 0}{\to \mathrm{type}})\,(\mathit{Tag} : \mathrm{int})\,(N : \mathrm{int})\\ (E' : \mathrm{int}) \overset{\cdot 1}{: \mathrm{type}} \triangleq \exists H : \mathrm{int}.\exists \mathit{Flag} : \mathrm{int};\ 0 \le \mathit{Flag} \le 1.\\ \quad \cdot\langle \mathrm{Int}(2 * H + \mathit{Flag}), \mathrm{Union}(\mathit{Flag} = 1,\\ \qquad \mathit{Know}(H = 32 * \mathit{Tag} + N),\\ \qquad \exists\alpha : \mathrm{int}; (H \ne 0 \wedge H = \mathit{Addr}(E', \alpha)).\\ \qquad\quad (A\,(E'+1)\,\alpha))\rangle$$

This says that if the flag bit is 1, then the rest of the header word contains N and the tag bits Tag (we define $\mathit{Know}(B)$ to be an abbreviation for $\exists\beta : \mathrm{bool}; B. \cdot \langle\rangle$, an existential whose sole purpose is to record the fact that B is true). If the flag bit is 0, then the rest of the header word is a non-null forwarding pointer to an object at location α in the next region $(E'+1)$, of type $(A\,(E'+1)\,\alpha)$, where A is abstract from the collector's perspective.

So far, the header word tells the collector the there is some Tag containing a collection of bits. The $\tau_{fieldlist}$ value assigns meaning to these bits:

$$\tau_{fieldlist} \triangleq (\mathit{GcFields}\ R\ F\ 0\ N\ \mathit{Tag})$$

where

$$
\begin{array}{l}
GcFields\,(R : \text{int} \to \text{int} \to \text{int} \to \overset{\cdot 1}{\text{type}})\\
(F : \text{int} \to \text{int} \to \overset{\cdot 1}{\text{type}})\,(\alpha : \text{int})\,(N : \text{int})\,(Tag : \text{int}) :\overset{\cdot 0}{\text{type}} \triangleq\\
\quad \text{Union}(\alpha = N, \cdot\langle\rangle,\\
\qquad \exists Tag_{hi} : \text{int}.\exists Tag_{lo} : \text{int}; 0 \le Tag_{lo} \le 1\\
\qquad\quad \wedge Tag = 2 * Tag_{hi} + Tag_{lo}.\\
\qquad \cdot\langle \tau_{fieldtag}, GcFields\ R\ F\,(\alpha+1)\ N\ Tag_{hi}\rangle
\end{array}
$$

The field list contains N elements, each associating a tag bit Tag_{lo} with a single field:

$$
\begin{array}{l}
\tau_{fieldtag} \triangleq \text{Union}(Tag_{lo} = 0,\\
\quad \exists A :\overset{\cdot 1}{\text{type}}\ .\text{Eq}(F\,\alpha, GcPrim\,A),\\
\quad \exists A : \text{int} \to \text{int} \to \overset{\cdot 0}{\text{type}}\ . \cdot \langle \text{Eq}(F\,\alpha, GcPtr\,A),\\
\qquad (GcCoerceForward\ R\ A)\rangle)
\end{array}
$$

The $\tau_{fieldtag}$ type says that if the tag bit is zero, then the field is a nonpointer type, while if the tag bit is one, the field is a pointer type. The $GcPrim$ type is the same as in the previous section; the $GcPtr$ type is slightly modified to use an abstract type operator A instead of the concrete type $GcObject$:

$$
\begin{array}{l}
GcPtr\,(A : \text{int} \to \text{int} \to \overset{\cdot 0}{\text{type}})\,(E' : \text{int}) :\overset{\cdot 1}{\text{type}} \triangleq \exists \alpha : \text{int}. \cdot \langle\\
\quad \text{Int}(Addr(E', \alpha)),\\
\quad \text{Union}(Addr(E', \alpha) = 0, \cdot\langle\rangle, A\ E'\,\alpha)\rangle
\end{array}
$$

The abstract type A allows each pointer to point to a different type of object. Again, though, the collector must know *something* about A in order to copy and scan it. Therefore, the $\tau_{fieldtag}$ pairs the $GcPtr$ with a coercion function that transforms the A into a $GcObject$:

$$
\begin{array}{l}
GcCoerceForward\,(R : \text{int} \to \text{int} \to \text{int} \to \overset{\cdot 1}{\text{type}})\\
\ (A : \text{int} \to \text{int} \to \overset{\cdot 0}{\text{type}}) :\overset{\cdot 0}{\text{type}} \triangleq\\
\quad \forall E' : \text{int}.\forall \alpha : \text{int}.(A\,E\,\alpha) \xrightarrow{\cdot,0}\\
\qquad \exists F : \text{int} \to \text{int} \to \overset{\cdot 1}{\text{type}}\ .\\
\qquad \exists N : \text{int}; 0 \le N < 32.\\
\qquad \exists Tag : \text{int}; 0 \le Tag.\\
\qquad \exists Extra :\overset{\cdot 0}{\text{type}}\ .\\
\qquad\quad (GcObject\ R\ E\ \alpha\ F\ A\ N\ Tag\ Extra)
\end{array}
$$

Since the coercion function is not called at run-time, it has size 0 and can therefore fit into the $\tau_{fieldtag}$ without adding space. After coercing the A object into a $GcObject$, the collector copies the object and later scans it. After copying the object, the collector must coerce the new to-space $GcObject$ object back into an A in order to produce a new ($GcPtr\,A$) into to-space. Each $GcObject$ therefore contains a reverse coercion closure, consisting of some closure environment data of arbitrary type $Extra$ and a coercion function:

$$
\begin{array}{l}
\tau_{coerceback} \triangleq \forall E' : \text{int}.\forall \alpha : \text{int}.\\
\quad (GcObject\ R\ E\ \alpha\ F\ A\ N\ Tag\ Extra) \xrightarrow{\cdot,0} (A\,E\,\alpha)
\end{array}
$$

For example, consider a user-defined type:

$$
\begin{array}{l}
Foo\,(R : \text{int} \to \text{int} \to \text{int} \to \overset{\cdot 1}{\text{type}})\,(X : \text{int})\,(E : \text{int})\\
(This : \text{int}) :\overset{\cdot 0}{\text{type}} \triangleq \exists Y : \text{int}. \cdot \langle\\
\quad (GcWord\,R\,E\,(This+0)\,(GcHdr\,(Foo\,R\,X)\,0\,2)),\\
\quad (GcWord\,R\,E\,(This+1)\,(GcPrim\,(\text{Int}(X+Y)))),\\
\quad (GcWord\,R\,E\,(This+2)\,(GcPrim\,(\text{Int}(X-Y))))\rangle
\end{array}
$$

The coercion function for $A = (Foo\,R\,X)$ takes the Foo object, unpacks Y, forms a type sequence for the fields:

$$
\begin{array}{l}
F(0) = (GcPrim\,(\text{Int}(X+Y)))\\
F(1) = (GcPrim\,(\text{Int}(X-Y)))
\end{array}
$$

and creates a $(GcObject\,R\,E\,This\,F\,(Foo\,R\,X)\,2\,0\,Extra_{Foo})$, where $Extra_{Foo}$ contains the information needed to reconstitute a Foo object from a copied $GcObject$:

$$
\begin{array}{l}
Extra_{Foo} \triangleq \exists Y : \text{int}. \cdot \langle\\
\quad \text{Eq}(F\,0, GcPrim\,(\text{Int}(X+Y)))\\
\quad \text{Eq}(F\,1, GcPrim\,(\text{Int}(X-Y)))\rangle
\end{array}
$$

Since the unpacking and packing of Y occurs inside a coercion function, it incurs no run-time cost.

5.2 Features, Optimizations and Inefficiencies

The Clay implementation of the collector supports a couple additional features, which we discuss briefly:

- The definition of $GcPtr$ assumed that all pointers could be null. In Java, this might be the case, but languages like ML often know that a pointer is never null, and shouldn't waste time doing a null pointer check when traversing a pointer. Therefore, an extra flag is added to $GcPtr$ that allows or prohibits nullness: $GcPtr\,(A : \text{int} \to \text{int} \to \overset{\cdot 0}{\text{type}})\,(E' : \text{int})\,(MayBe0 : \text{bool}) :\overset{\cdot 1}{\text{type}} \triangleq \exists \alpha : \text{int};\ MayBe0 \vee (\alpha \ne 0). \cdot \langle \ldots \rangle$ The $MayBe0$ flag is held abstract from the garbage collector, so that the collector cannot set a never-null word to null.

- The definition of $GcHdr$ assumed that any header word could be a forwarding pointer. In fact, only header words in from-space may be forwarding pointers, so there is no reason to require the collector to perform a run-time test of the header flag bit when scanning the to-space queue. Therefore, the Clay implementation modifies the $LArray$s to pass in a flag to each word allowing or disallowing forwarding pointers. Regions start with forwarding pointers disabled, and before each garbage collection, a coercion function modifies the flag in each $LArray$ element to enable the forwarding pointer for each word.

There are still a couple inefficiencies that the type system imposes on the collector. First, when scanning to-space, the collector must store the header word in each object after loading it in order to update the type of the header word, even though the store doesn't change the bits in memory. Second, for each to-space object allocation, the collector must check that there is enough free space for the object, even though to-space should be no larger than from-space. The first problem can probably be solved by changing the linear array holding the Cheney queue to track header words more precisely, but the second problem stems from an invariant too deep for the type system to capture.

6. CONCLUSIONS AND RELATED 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 data structures, ranging from simple lists to Cheney queues of objects with efficient header words. Some might complain that we've cheated by relying a proof language for the coercion functions: of course anyone can prove a garbage collector safe if they have a whole proof system at their disposal and enough time on their hands! In fact, in the Clay implementations of the examples in this paper, the "proofs" are all very short functions, and they constitute only a small fraction of the total code. Plus, the proof language is only a minor variation of the programming language. A more pressing concern is the number of little coercions (pack, unpack, split, combine, type equality coercions, etc.) required in the programming portion of the language to set up simple reads and writes of memory, which can make low-level Clay programming very tedious. We've tried to factor out common sequences of operations inside inline Clay functions, but it would be nice to have a smarter type system that removes some of the burden from the programmer in a more general way.

In terms of ease of use, alias types have an advantage over linear memory types in that the alias type contexts are updated implicitly during loads and stores, rather than as explicit arguments to the load and store operations. A similar technique might work for linear memory types, although this would require matching integer type pointer addresses to integer constraint elements, which might be more computationally intensive than matching symbolic locations.

The previous section discussed advantages of the linear memory type garbage collector over existing type-preserving garbage collectors. However, we must admit that in some ways, the existing approaches are still ahead of us: Wang et al measured the performance of their collector in a real ML back-end, and Monnier et al developed ideas for supporting generational collection. Hopefully, the near future will bring a synthesis of the best features of all of this work.

7. REFERENCES

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

[2] 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.

[3] 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.

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

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

[6] Heng Huang and Chris Hawblitzel. Proofs of soundness and strong normalization for linear memory types. Technical Report TR2002-437, Dartmouth College, 2002.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

Appendix A: Formal Properties of the Abstract Machine

Due to space constraints, not all evaluation, kinding, and type checking rules are listed here. See the technical report[6] for complete listings, as well as proofs of safety (for all well-typed expressions) and strong normalization (for expressions that type-check with a $limit \neq \infty$). The rules follow notational conventions from Wadler[14] and Walker et al[16].

abbreviations

$\forall\alpha : K.\tau \triangleq \forall\alpha : K; \text{true}.\tau$
$\exists\alpha : K.\tau \triangleq \exists\alpha : K; \text{true}.\tau$
$\text{Know}(B) \triangleq \exists\alpha : \text{bool}; B. \cdot\langle\rangle$
$\text{let}\, x = e_1 \,\text{in}\, e_2 \triangleq \text{let}\, \langle x\rangle =^{\wedge} \langle e_1\rangle \,\text{in}\, e_2$
$\tau_1 \xrightarrow{\phi} \tau_2 \triangleq \tau_1 \xrightarrow{\phi,\infty} \tau_2$

linearity: $\quad \phi = \cdot \mid \wedge$

kinds: $\quad K = J \mid \text{int} \mid \text{bool} \qquad J = \overset{\phi i}{\text{type}} \mid K \to J$

arithmetic

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

types

$\text{limit} = I \mid \infty$
$\tau = \tau_1 \xrightarrow{\phi,limit} \tau_2 \mid \langle\vec{\tau}\rangle \mid \alpha \mid I \mid B \mid \lambda\alpha : K.\tau \mid \forall\alpha : K; B.\tau$
$\mid \exists\alpha : K; B.\tau \mid \tau_1\,\tau_2 \mid \mu\alpha : K.\tau \mid \text{Int}(I) \mid \text{Bool}(B)$
$\mid \text{Union}(B, \tau_1, \tau_2) \mid \text{Has}(I, \tau) \mid \text{Gen}(\tau, I) \mid \text{Eq}(\tau_1, \tau_2)$
$\mid \text{F}^K \mid \text{InDomain}(I, \tau)$

expressions

$e = i \mid b \mid x \mid e_1\, e_2 \mid e\tau \mid \langle\vec{e}\rangle \mid \lambda x : \tau \xrightarrow{\phi,\text{limit}} e \mid e_1 \,\text{op}\, e_2 \mid \neg e$
$\mid \Lambda\alpha : K; B.v \mid \text{let}\, \langle\vec{x}\rangle = e_1 \,\text{in}\, e_2 \mid \text{union}(b, \tau_1, \tau_2, e)$
$\mid \text{if}\, e_1 \,\text{then}\, e_2 \,\text{else}\, e_3 \mid \text{if}\, B \,\text{then}\, e_1 \,\text{else}\, e_2 \mid \text{case}(b, e)$
$\mid \text{pack}[\tau_1, e] \,\text{as}\, \exists\alpha : K; B.\tau_2 \mid \text{unpack}\, \alpha, x = e_1 \,\text{in}\, e_2$
$\mid \text{roll}[(\mu\alpha : K.\tau_0)\tau_1 \cdots \tau_n](e) \mid \text{unroll}(e) \mid \text{fix}\, x : \tau.\text{v}$
$\mid \text{load}(e_{\text{ptr}}, e_{\text{Has}}) \mid \text{store}(e_{\text{ptr}}, e_{\text{Has}}, \text{e}_{\text{v}}) \mid \text{coerce}(e)$
$\mid \text{distinguish}(I_1, I_2, e_1, e_2) \mid \text{make\_eq}(\tau)$
$\mid \text{apply\_eq}(\tau, e_1, e_2) \mid \text{new\_fun}(K) \mid \text{discard\_fun}(e)$
$\mid \text{define\_fun}(e, \tau) \mid \text{in\_domain}(I_1, I_2, e_1, e_2) \mid \text{fact}$
$v = i \mid b \mid \Lambda\alpha : K; B.v \mid \text{pack}[\tau_1, v] \,\text{as}\, \exists\alpha : K; B.\tau_2$
$\mid \text{roll}[(\mu\alpha : K.\tau_0)\tau_1 \cdots \tau_n](v) \mid \lambda x : \tau \xrightarrow{\phi,\text{limit}} e$
$\mid \langle\vec{v}\rangle \mid \text{union}(b, \tau_1, \tau_2, v) \mid \text{fact}$

environments $(C = \Psi; \Phi; \Delta; \Gamma; B; \text{limit})$

$M = \{1 \mapsto v_1, \cdots, n \mapsto v_n\}$
$\Psi = \{1 \mapsto \tau_1, \cdots, n \mapsto \tau_n\}$
$\Phi = \{F_1^{K_1} \overset{\phi_1}{\mapsto} \delta_1, \cdots, F_n^{K_n} \overset{\phi_n}{\mapsto} \delta_n\}$
$\delta = \{0 \mapsto \tau_0, \cdots, n \mapsto \tau_n\}$
$\Delta = \{\alpha_1 \mapsto K_1, \cdots, \alpha_n \mapsto K_n\}$
$\Gamma = \{x_1 \mapsto \tau_1 \cdots, x_n \mapsto \tau_n\}$

selected evaluation rules

$(M, \text{load}(i, fact)) \to (M, ^{\wedge}\langle M(i), fact\rangle)$
$(M, \text{store}(i, fact, v)) \to ([i \mapsto v]M, fact)$
$\frac{e \to e'}{(M,e) \to (M,e')}$
$(\lambda x : \tau \xrightarrow{\phi,I} e_1)v_2 \to \text{coerce}([x \mapsto v_2]e_1)$
$(\lambda x : \tau \xrightarrow{\phi,\infty} e_1)v_2 \to [x \mapsto v_2]e_1$
$\text{coerce}(v) \to v$
$\text{fix}\, x : \tau.\text{v} \to [x \mapsto \text{fix}\, x : \tau.\text{v}]\text{v}$
$\text{case}(b, \text{union}(b, \tau_1, \tau_2, v)) \to v$
$\text{make\_eq}(\tau) \to fact$
$\text{apply\_eq}(\tau, fact, v) \to v$
$\text{new\_fun}(K) \to \text{pack}[F^K, fact] \,\text{as}\, \exists\alpha : \text{int} \mapsto K.\text{FunGen}(\alpha, 0)$, where F^K is fresh
$\text{discard\_fun}(fact) \to \cdot\langle\rangle$
$\text{define\_fun}(fact, \tau) \to^{\wedge} \langle fact, fact, fact\rangle$

selected kinding rules

$$\frac{\Phi; \Delta \vdash \tau_1 : \overset{\phi_1 i_1}{\text{type}} \quad \Phi; \Delta \vdash \tau_2 : \overset{\phi_2 i_2}{\text{type}} \quad (\Phi; \Delta \vdash limit : \text{int} \wedge i = 0) \;or\; (limit = \infty \wedge i = 1)}{\Phi; \Delta \vdash \tau_1 \xrightarrow{\phi,limit} \tau_2 : \overset{\phi i}{\text{type}}}$$

$$\Phi, \text{F}^K \overset{\cdot}{\mapsto} \delta; \Delta \vdash \text{F}^K : \text{int} \to K$$

$$\frac{\Phi; \Delta \vdash I : \text{int}}{\Phi; \Delta \vdash \text{Int}(I) : \overset{\cdot 1}{\text{type}}}$$

$$\frac{\Phi; \Delta \vdash I : \text{int} \quad \Phi; \Delta \vdash \tau : \overset{\cdot 1}{\text{type}}}{\Phi; \Delta \vdash \text{Has}(I, \tau) : \overset{\wedge 0}{\text{type}}}$$

selected type checking rules

$$\frac{\Psi = \Psi_{spare}, \Psi_e \qquad \Phi = \dot{\Phi}_{spare}, \Phi_e \quad \forall i \in dom(\Psi).(\varnothing; \dot{\Phi}; \varnothing; \varnothing; true; \infty \vdash M(i) : \Psi(i))) \quad \Psi_e; \Phi_e; \Delta; \Gamma; B; \text{limit} \vdash e : \tau}{\Psi; \Phi; \Delta; \Gamma; B; \text{limit} \vdash (M, e : \tau)}$$

$$\frac{\Psi; \Phi; \Delta; \Gamma; B; I_2 \vdash e : \tau \quad (\text{limit}_1 = \infty \wedge \tau : \overset{\phi 0}{\text{type}}) \text{ or } (\text{limit}_1 = I_1 \wedge B \vdash I_1 > I_2)}{\Psi; \Phi; \Delta; \Gamma; B; \text{limit}_1 \vdash \text{coerce}(e) : \tau}$$

$$\frac{\overset{\phi}{\Phi}; \Delta \vdash \tau_1 : K \quad \overset{\phi}{\Psi}; \overset{\phi}{\Phi}; \Delta; \overset{\phi}{\Gamma}, x : \tau_1; B; \text{limit}_2 \vdash e : \tau_2 \quad (\text{limit}_2 = \infty) \text{ or } (\overset{\phi}{\Phi}; \Delta \vdash \text{limit}_2 : \text{int} \quad B \vdash \text{limit}_2 \geq 0)}{\overset{\phi}{\Psi}; \overset{\phi}{\Phi}; \Delta; \overset{\phi}{\Gamma}; B; \text{limit}_1 \vdash \lambda x : \tau_1 \xrightarrow{\phi,\text{limit}_2} e : \tau_1 \xrightarrow{\phi,\text{limit}_2} \tau_2}$$

$$\frac{C_1, C_2 = \Psi; \Phi; \Delta; \Gamma; B; \text{limit}_C \quad C_1 \vdash e_1 : \tau_a \xrightarrow{\phi,\text{limit}_f} \tau_b \quad C_2 \vdash e_2 : \tau_a \quad (\text{limit}_C = \text{limit}_f = \infty) \text{ or } (B \vdash \text{limit}_f < \text{limit}_C) \text{ or } (\text{limit}_C = \infty,\ \text{limit}_f = I,\ \Phi; \Delta \vdash \tau_b : \overset{\phi_b 0}{\text{type}})}{C_1, C_2 \vdash e_1\, e_2 : \tau_b}$$

$$\frac{C \vdash \tau : \overset{\phi i}{\text{type}} \quad C, x : \tau \vdash \text{v} : \tau}{C \vdash (\text{fix}\, x : \tau.\text{v}) : \tau}$$

$$\dot{C}, i \mapsto \tau \vdash fact : \text{Has}(i, \tau)$$

$$\dot{C}, F^K \overset{\wedge}{\mapsto} \text{fun} \vdash fact : \text{Gen}(F^K, i)$$

$$\dot{C} \vdash fact : \text{Eq}(\tau, \tau)$$

$$\frac{i \in dom(\text{fun})}{\dot{C}, F^K \overset{\cdot}{\mapsto} \text{fun} \vdash fact : \text{InDomain}(i, F^K)}$$

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

$$\frac{C = C_1, C_2, C_3 = \Psi; \Phi; \Delta; \Gamma; B; \infty \quad C_2 \vdash e_{\text{Has}} : \text{Has}(I, \tau_1) \quad C_3 \vdash e_{\text{v}} : \tau_2 \quad C_1 \vdash e_{\text{ptr}} : \text{Int}(I) \quad C \vdash \tau_2 : \overset{\cdot 1}{\text{type}}}{C \vdash \text{store}(e_{\text{ptr}}, e_{\text{Has}}, e_{\text{v}}) : \text{Has}(I, \tau_2)}$$

$$\frac{C_1, C_2 \vdash I_1 : \text{int} \quad C_1, C_2 \vdash I_2 : \text{int} \quad C_1 \vdash e_1 : \text{Has}(I_1, \tau_1) \quad C_2 \vdash e_2 : \text{Has}(I_2, \tau_2)}{C_1, C_2 \vdash \text{distinguish}(I_1, I_2, e_1, e_2) : ^{\wedge}\langle\text{Know}(I_1 \neq I_2), \text{Has}(I_1, e_1), \text{Has}(I_2, e_2)\rangle}$$

$$\frac{C \vdash \tau_f : K \to J \quad C \vdash e_1 : \text{Eq}(\tau_a, \tau_b) \quad C \vdash \tau_a : K \quad C \vdash \tau_b : K \quad C \vdash e_2 : \tau_f \tau_a}{C \vdash \text{apply\_eq}(\tau_f, e_1, e_2) : \tau_f \tau_b}$$

$$\dot{C} \vdash \text{new\_fun}(J) : \exists\alpha : \text{int} \to J.\text{Gen}(\alpha, 0)$$

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

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