A Garbage-Collecting Typed Assembly Language

Chris Hawblitzel
Microsoft Research

Heng Huang
Dartmouth College

Lea Wittie
Bucknell University

Juan Chen
Microsoft Research

Abstract

Typed assembly languages usually support heap allocation safely, but often rely on an external garbage collector to deallocate objects from the heap and prevent unsafe dangling pointers. Even if the external garbage collector is provably correct, verifying the safety of the interaction between TAL programs and garbage collection is nontrivial. This paper introduces a typed assembly language whose type system is expressive enough to type-check a Cheney-queue copying garbage collector, so that ordinary programs and garbage collection can co-exist and interact inside a single typed language. The only built-in types for memory are linear types describing individual memory words, so that TAL programmers can define their own object layouts, method table layouts, heap layouts, and memory management techniques.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic storage manangement

General Terms Languages, Verification

Keywords Typed assembly language, garbage collection

1. Introduction

Operating systems have traditionally protected programs from one another using run-time checking of memory addresses, based on page tables or segments. Many recent projects have used safe languages, such as Java and C#, to replace traditional operating system protection mechanisms. Language-based mechanisms promise more flexible and fine-grained protection than traditional mechanisms, but bring new challenges. In particular, a buggy implementation of a safe language can invalidate the language's safety guarantees and destroy the protection between programs. Since a language's implementation typically consists of a large compiler and large run-time system, there is a large potential for such bugs. Proof-carrying code [18] and typed assembly language [16] eliminate the compiler and some of the run-time system from this "trusted computing base", but typically still require a trusted garbage collector, because the safety of the garbage collector's fundamental operation ("free memory") is difficult to prove in the presence of aliased data structures with complicated types.

This paper presents a typed assembly language called *GTAL*, whose type system is expressive enough to verify the safety of a garbage collector for a simple object-oriented language. In contrast to earlier work on typed garbage collectors [26, 14], GTAL deliberately omits any built-in definitions of heaps, objects, allocation, and deallocation. Instead, it provides programmers with a flat array of memory words, and programmers define their own memory layouts and memory management routines by carving up the array of words to form objects, garbage collection tables, semi-spaces, and so on. This allows programmers to create diverse implementations of memory allocators and deallocators, and to tailor the layout of memory to match the needs of different allocators and deallocators. For example, the garbage collector presented in this paper allocates data contiguously in a semi-space, and prepends each object with a header word pointing to a method table containing the size of the object. The collector uses the size in the table to help scan through a queue of contiguous objects. Other collectors are free to implement different strategies, such as putting objects of the same size on the same page (as done by BiBoP collectors).

In place of built-in types for heaps and objects, GTAL includes a linear logic for encoding new types from individual memory words. The logic is simple, yet powerful enough to encode mutually recursive classes with inheritance, overriding, and polymorphic methods. GTAL's inclusion of a linear logic is similar to the LTT language's inclusion of the linear LF logic [8]. However, while LTT used a linear logic to augment an otherwise standard type system, GTAL goes farther: the logic doesn't augment the type system — the logic *is* the type system. We mean it when we say that programmers encode data types using logical formulas: even the type "int" is encoded in the logic, rather than being built into the type system. Recursive types pose a challenge for such a strictly logic-based approach, since unrestricted recursive types can easily destroy the soundness of a logic. GTAL addresses this challenge using a modal operator that shields the logic from unrolled recursive types.

The rest of the paper is as follows. Section 2 compares GTAL to previous work. Section 3 describes the logic embedded inside GTAL. Section 4 then introduces the typed assembly language, and Section 5 summarizes the proofs of soundness for the logic and the assembly language. Section 6 uses the logic to express heaps and heap objects, and Section 7 describes the GTAL code that garbage collects these objects. (The complete, mechanically-type-checked GTAL code for the garbage collector is available online [10].)

2. Background and related work

GTAL builds on recent work in type systems and proof-carrying code. Wang and Appel [26] observed that by copying live data out of one memory region into another, and then freeing the first region, a programmer can implement a copying garbage collector entirely from safe language primitives. In this approach, the type system tags every pointer with a region annotation, and thereby prevents programs from dereferencing pointers to freed regions. The details of this approach are problematic, though; Wang and Appel required *ad hoc* type system extensions to type forwarding pointers.

Monnier and Shao [15] overcame some of the problems by introducing a language supporting regions, alias types [23], and an embedded proof language. Nevertheless, the regions still require *ad hoc* extensions to type-check Cheney queue scanning [27], and the large number of features in the type system increase the trusted computing base. Furthermore, the region-based approach implements only copying collectors, not mark-sweep collectors.

Because of the lack of generality and large trusted computing base in current typed assembly languages, many researchers are working on foundational proof-carrying code (FPCC), which in principle can mechanically prove the safety of typed assembly languages and the correctness of their associated run-time systems. Much progress has been made, but just proving the safety of an FPCC system *without* garbage collection [29, 5, 4] is challenging. To add garbage collection to an FPCC system, one must mechanically prove the safety of the TAL, the correctness of the collector, and the correctness of the interaction between the TAL and the collector. As far as we know, this has not been done yet, although enough pieces exist to make it plausible. Birkedal *et al.* [2] formally proved the correctness of a copying collector by showing that the collector produces a copy that is isomorphic to the original live data. Like Birkedal *et al*., we establish and prove invariants for a copying collector, though our invariants stop short of expressing graph isomorphism, which is stronger than necessary to prove the garbage collector's safety. Unlike the collector of Birkedal *et al*., our garbage collector supports header words and GC tables, and our tables can contain embedded code pointers.

The FPCC approach is daunting enough to make consideration of alternatives worthwhile. We follow an approach introduced by Crary and Vanderwaart's LTT system [8] and Shao *et al.*'s TL system [21]. These systems invert the FPCC approach: rather than proving the soundness of a computation language's type system inside a logic, they embed a logic (linear LF for LTT and CIC for TL) inside a computation language's type system. This results in a somewhat larger trusted computing base than FPCC, since both the logic and the computation language must be trusted, but it makes a programmer's job easier, since no foundational proof is required to use the computation language.

Jia *et al.* [13] embed a linear logic inside a typed assembly language to reason about stack allocation. Their logic seems specialized for reasoning about stacks, though — it is not expressive enough to type check our garbage collector. Zhu and Xi's ATS [30] embeds a linear logic in a higher-level language, to support safe reasoning about ephemeral properties of data structures, though they apply ATS to simple data structures, such as arrays, rather than to implementing heaps and garbage collection.

3. A logic

The rest of this paper applies the embedded logic approach of LTT, TL, ATS, and Jia *et al.* to heap objects and garbage collection. The embedded logic must satisfy the following criteria:

- The logic should be general enough to reason about simple propositions, equality, and integer inequality. For example, Section 7 uses integer arithmetic to express a copying garbage collector's from-space and to-space invariants, establishing different invariants for different memory ranges in each space. (LTT and TL, building on well-known, general-purpose logics, satisfy this criterion easily.)
- The logic should support linear reasoning about memory. This lets the collector easily change the types of memory words when it frees and reallocates memory. (LTT, ATS, and Jia *et al*'s approach satisfy this criterion.)
- The logic should support unrestricted recursive specifications, for two reasons. First, the specification of objects in Section 6 is recursive, since objects' classes may refer to each other recursively. Second, the specification of heaps in Section 7 is recursive: heaps contain objects, objects contain header words, header words point to method tables, method tables point to code blocks, and the code blocks take the current heap as an argument. (ATS satisfies this criterion. TL supports "inductive definitions", but these are too restrictive to capture the recursive definitions of sections 6 and 7 directly.)

Given a logic satisfying these criteria, it is straightforward to define the heap and garbage collector data types; the definitions of objects, heaps, and garbage collection in Sections 6 and 7 are lengthy but unsurprising. The proofs about the data types are straightforward enough to be written by hand, without a proof assistant, and mechanically checked. For example, the "heap extension lemma", which proves that extending the heap with a newly allocated value preserves all existing invariants in the heap, is about 200 lines of GTAL's logic. The rest of this section describes GTAL's logic, which combines ideas from LTT, TL, ATS, and Jia *et al.*'s logic in a way that meets the requirements above.

3.1 Proof terms and types

Figure 1 specifies the abstract syntax of GTAL's logic. The logic is based on $F\omega$ [20], with support for linearity [24]. It includes standard function types $\tau_1 \multimap \tau_2$ and pair types $\tau_1 \otimes \tau_2$ from linear logic, type variables A, polymorphic types $\forall A : \kappa.\tau$, existential types $\exists A : \kappa.\tau$, as well as $F\omega$-style functions $\lambda A : \kappa.\tau$ and applications $\tau_1\ \tau_2$ at the type level. Kinds κ include $\mathbb{T}$ for types, $\mathbb{N}$ for numbers, and $\mathbb{R}$ for register names; Figure 2's rules assign kinds to types. Following Xi and Pfenning [28], register names r and numbers n live at the type level, rather than the term level; this simplifies the meta-theory by avoiding full-blown dependent types. Both the term and type levels contain elimination constructs for numbers (the term-level construct implements induction, and the type-level construct implements primitive recursion).

Given basic function and polymorphic types, the type system can encode true (aka "unit"), false (aka "void"), negation, if-and-only-if, choice of A or B (aka additive conjunction, "$A\&B$"), union of A and B (aka additive disjunction, "$A \oplus B$"), and equality $(A = B)$ [12]:

$$\begin{aligned}
&\mathit{True} = \forall A\!:\!\mathbb{T}.A \multimap A \\
&\mathit{False} = \forall A\!:\!\mathbb{T}.A \\
&\mathit{Not} = \lambda A\!:\!\mathbb{T}.A \multimap \mathit{False} \\
&\mathit{Iff} = \lambda A\!:\!\mathbb{T}.\lambda B\!:\!\mathbb{T}.!(A \multimap B)\otimes!(B \multimap A) \\
&\mathit{Choice} = \lambda A\!:\!\mathbb{T}.\lambda B\!:\!\mathbb{T}.\exists X\!:\!\mathbb{T}.X\otimes!(X \multimap A)\otimes!(X \multimap B) \\
&\mathit{Union} = \lambda A\!:\!\mathbb{T}.\lambda B\!:\!\mathbb{T}.\forall C\!:\!\mathbb{T}.\forall X\!:\!\mathbb{T}. \\
&\qquad\qquad C \multimap !(C \multimap A \multimap X) \multimap !(C \multimap B \multimap X) \multimap X \\
&\mathit{Eq}_\kappa = \lambda A\!:\!\kappa.\lambda B\!:\!\kappa.\forall F\!:\!\kappa \to \mathbb{T}.F\ A \multimap F\ B
\end{aligned}$$

As in linear logic, types are linear by default, and are only nonlinear if there is a $!$ symbol in front of them. Thus, a function of type $(!\tau_1) \multimap \tau_2$ may use τ_1 multiple times or not at all, while a function of type $\tau_1 \multimap \tau_2$ (where τ_1 is not of the form $!\tau$) must use its argument exactly once.

$$
\begin{array}{rcll}
\textit{kind} & \kappa & = & \mathbb{T} \mid \mathbb{N} \mid \mathbb{R} \mid \kappa_1 \rightarrow \kappa_2 \\
\textit{number} & n & = & 0 \mid s(n) \\
\textit{register} & r & = & r1 \mid \ \ldots \ \mid rk \\
\textit{location} & \ell & = & n \mid r \\
\textit{type} & \tau & = & !\tau \mid \tau_1 \multimap \tau_2 \mid \tau_1 \otimes \tau_2 \mid \bigcirc\tau \mid \text{rec } \tau \\
 & & \mid & A \mid \forall A{:}\kappa.\tau \mid \exists A{:}\kappa.\tau \mid \lambda A{:}\kappa.\tau \mid \tau_1 \ \tau_2 \\
 & & \mid & 0 \mid s(\tau) \mid \text{elim } \tau_n \ \tau_z \ \tau_s \\
 & & \mid & r \mid \text{Reg } \tau_1 \ \tau_2 \mid \text{Mem } \tau_1 \ \tau_2 \mid \text{Code } \tau_1 \ \tau_2 \\
\textit{pattern} & p & = & x \mid \ !x \ \mid \ !(p) \mid A, p \mid p, p \\
\textit{term} & e & = & x \mid \ !e \mid \text{let}\, p = e_1 \text{ in}\, e_2 \mid \lambda p{:}\tau.e \\
 & & \mid & e_1 \ e_2 \mid e_1, e_2 \mid \lambda A{:}\kappa.e \mid e \ \tau \\
 & & \mid & \text{elim } \tau_n \ \tau_f \ e_z \ e_s \mid \bigcirc e \mid e_1 \bigcirc\!\!<\!< e_2 \mid \text{fact} \\
 & & \mid & \text{pack}[\tau_1, e] \text{ as}\, \tau_2 \mid \text{code}(n)[\tau_1, \ldots, \tau_n] \\
\textit{value} & v & = & !v \mid \lambda p{:}\tau.e \mid v_1, v_2 \mid \lambda A{:}\kappa.e \mid \bigcirc e \mid \text{fact} \\
 & & \mid & \text{pack}[\tau_1, v] \text{ as}\, \tau_2 \mid \text{code}(n)[\tau_1, \ldots, \tau_n] \\
\textit{tvar ctxt} & \Delta & = & \{\} \mid \Delta, A \mapsto \kappa \\
\textit{var ctxt} & \Gamma & = & \{\} \mid \Gamma, x \mapsto \tau \\
\textit{code ctxt} & \Upsilon & = & \{\} \mid \Upsilon, n \mapsto \tau \\
\textit{mem ctxt} & \Psi & = & \{\} \mid \Psi, \ell \mapsto n \\
\textit{ctxt} & C & = & \Delta; \Gamma; \Upsilon; \Psi
\end{array}
$$

Figure 1. Proof term and type syntax

$$
\frac{\Delta \vdash \tau : \mathbb{T}}{\Delta \vdash !\tau : \mathbb{T}}
\qquad
\frac{\Delta \vdash \tau_1 : \mathbb{T} \qquad \Delta \vdash \tau_2 : \mathbb{T}}{\Delta \vdash \tau_1 \multimap \tau_2 : \mathbb{T}}
\qquad
\Delta \vdash r : \mathbb{R}
$$

$$
\frac{\Delta \vdash \tau_1 : \mathbb{T} \qquad \Delta \vdash \tau_2 : \mathbb{T}}{\Delta \vdash \tau_1 \otimes \tau_2 : \mathbb{T}}
\qquad
\Delta \vdash 0 : \mathbb{N}
\qquad
\frac{\Delta \vdash \tau : \mathbb{N}}{\Delta \vdash s(\tau) : \mathbb{N}}
$$

$$
\frac{\Delta \vdash \tau_n : \mathbb{N} \qquad \Delta \vdash \tau_z : \kappa \qquad \Delta \vdash \tau_s : \mathbb{N} \rightarrow \kappa \rightarrow \kappa}{\Delta \vdash \text{elim } \tau_n \ \tau_z \ \tau_s : \kappa}
$$

$$
\Delta, A : \kappa \vdash A : \kappa
\qquad
\frac{\Delta, A : \kappa \vdash \tau : \mathbb{T}}{\Delta \vdash \forall A{:}\kappa.\tau : \mathbb{T}}
\qquad
\frac{\Delta, A : \kappa \vdash \tau : \mathbb{T}}{\Delta \vdash \exists A{:}\kappa.\tau : \mathbb{T}}
$$

$$
\frac{\Delta, A : \kappa_a \vdash \tau : \kappa_b}{\Delta \vdash \lambda A{:}\kappa_a.\tau : \kappa_a \rightarrow \kappa_b}
\qquad
\frac{\Delta \vdash \tau_f : \kappa_a \rightarrow \kappa_b \qquad \Delta \vdash \tau_a : \kappa_a}{\Delta \vdash \tau_f \ \tau_a : \kappa_b}
$$

$$
\frac{\Delta \vdash \tau : \kappa}{\Delta \vdash \bigcirc\tau : \kappa}
\qquad
\frac{\Delta \vdash \tau : \kappa \rightarrow \kappa}{\Delta \vdash \text{rec } \tau : \kappa}
\qquad
\frac{\Delta \vdash \tau_r : \mathbb{R} \qquad \Delta \vdash \tau_n : \mathbb{N}}{\Delta \vdash \text{Reg } \tau_r \ \tau_n : \mathbb{T}}
$$

$$
\frac{\Delta \vdash \tau_m : \mathbb{N} \qquad \Delta \vdash \tau_n : \mathbb{N}}{\Delta \vdash \text{Mem } \tau_m \ \tau_n : \mathbb{T}}
\qquad
\frac{\Delta \vdash \tau_m : \mathbb{N} \qquad \Delta \vdash \tau : \mathbb{T}}{\Delta \vdash \text{Code } \tau_m \ \tau : \mathbb{T}}
$$

Figure 2. Kinding rules

The typing rules for terms, shown in Figures 4 and 5, enforce the linearity requirements by distinguishing between linear variable bindings and nonlinear variable bindings [24]. The environment Γ either binds a variable to a type linearly ($x : \tau$), in which case x must be used exactly once, or nonlinearly ($!x : \tau$), in which case x may be used more than once or not at all. Pattern matching introduces new variables into the environment. Figure 4 shows the typing rules for patterns: the judgment $\vdash p : \tau \Longrightarrow \Delta; \Gamma$ says that the pattern p, matching a value of type τ, introduces a set of type variables Δ and value variables Γ into the environment. The pattern x introduces a linear binding $x : \tau$ into Γ, and the pattern $!x$ introduces a nonlinear binding $!x : \tau$ into Γ. The pattern $!(p)$ discards a "!" operator, while patterns (p_1, p_2) and $[A, p]$ unpack pairs and existential types. For example, the term $\lambda(x, !y) : \tau_x \otimes !\tau_y.(x, y, y)$ binds x linearly and y nonlinearly; this term has type $\tau_x \otimes !\tau_y \multimap \tau_x \otimes \tau_y \otimes \tau_y$. The term $\lambda(x, !y) : \tau_x \otimes !\tau_y.(x, x, y)$, on the other hand, is ill-typed because it uses the linearly bound variable x twice. The notation $\Gamma = \Gamma_1, \Gamma_2$ indicates that Γ and Γ_1 and Γ_2 have identical nonlinear assumptions, but that $\Gamma' s$ linear assumptions are split between Γ_1 and Γ_2. The notation $\Gamma = !\Gamma_1$ indicates that Γ and Γ_1 have identical nonlinear assumptions, but that Γ has no linear assumptions. The typing rule that concludes $!e : !\tau$, for example, requires that e type-check in a purely nonlinear environment ($!\Gamma \vdash e : \tau$), so that expressions of type $!\tau$ cannot carry linear assumptions; this makes it safe to freely discard and duplicate expressions of type $!\tau$. Figure 1 extends the Γ_1, Γ_2 and $!\Gamma_1$ notations to cover all the environments in a context $C = \Delta; \Gamma; \Upsilon; \Psi$, where the type variable environment Δ and the code environment Υ contain only nonlinear assumptions, the location environment Ψ contains only linear assumptions, and, as described above, the variable environment Γ contains both linear and nonlinear assumptions.

The type system uses linearity to enable *strong updates* to registers and memory [2, 23]. By "strong update", we mean that a write to a register or memory location can change the type of the value stored in the register or memory location. Because of aliasing, such a type change would be unsafe without some linearity restriction.

The type system describes the state of registers and memory with a linear environment Ψ mapping locations ℓ to integer values n. The program manipulates Ψ via the linear *capability types* "Reg r n" and "Mem n_1 n_2", which indicate that register r (where r is a type of kind $\mathbb{R}$) currently holds the integer n (where n is a type of kind $\mathbb{N}$), and memory address n_1 holds the integer n_2 (where n_1 and n_2 are types of kind $\mathbb{N}$). A capability of type Reg r n, for example, gives the program the right to read value n from register r, and to change the contents of register r to any new value n', where changing n to n' consumes the linear capability Reg r n and produces a new linear capability Reg r n'. The linearity of the environment Ψ guarantees that only one capability for register r exists, so that it is safe to consume Reg r n and produce Reg r n'. Ψ's linearity does not mean that we restrict programs to linear data structures (trees). Instead, we use nonlinear function types $!(\tau_1 \multimap \tau_2)$ to encode aliasing and weak updates, as described in Section 6.

Conceptually, a program starts with an initial Ψ that describes the initial state of the registers and memory. As the program runs, Ψ evolves to track the updates to registers and memory. However, Ψ is merely a technical device used to establish the soundness of the language; our actual implementation of the type system omits Ψ. In practice, programs have a "main" block whose precondition specifies a variable x holding an array of register and memory capabilities, and the program uses variables to pass these capabilities from block to block. (The loader that loads and starts the program must ensure that the initial state of the memory and registers satisfies main's precondition.) The soundness of the language relies on

$$\tau \equiv \tau \qquad \frac{\tau_1 \equiv \tau_2}{\tau_2 \equiv \tau_1} \qquad \frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}$$

$$\begin{array}{c} (\lambda A{:}\kappa.\tau_b)\ \tau_a \equiv [A \leftarrow \tau_a]\tau_b \\ \text{elim } 0\ \tau_z\ \tau_s \equiv \tau_z \\ \text{elim } s(\tau_m)\ \tau_z\ \tau_s \equiv \tau_s\ \tau_m\ (\text{elim } \tau_m\ \tau_z\ \tau_s) \\ \text{rec } \tau \equiv \bigcirc(\tau\ (\text{rec } \tau)) \\ (\bigcirc\tau_1)\ \tau_2 \equiv \bigcirc(\tau_1\ \tau_2) \end{array}$$

Figure 3. Type equivalence rules (excluding congruence)

$$\vdash x : \tau \Longrightarrow \{\}; \{x : \tau\} \qquad \vdash !x : \tau \Longrightarrow \{\}; \{\, !x : \tau\}$$

$$\frac{\vdash p : \tau \Longrightarrow \Delta; \Gamma}{\vdash !(p) : !\tau \Longrightarrow \Delta; \Gamma} \qquad \frac{\vdash p : \tau \Longrightarrow \Delta; \Gamma}{\vdash A, p : \exists A{:}\kappa.\tau \Longrightarrow \Delta, A : \kappa; \Gamma}$$

$$\frac{\vdash p_1 : \tau_1 \Longrightarrow \Delta_1; \Gamma_1 \qquad \vdash p_2 : \tau_2 \Longrightarrow \Delta_2; \Gamma_2}{\vdash p_1, p_2 : \tau_1 \otimes \tau_2 \Longrightarrow \Delta_1\Delta_2; \Gamma_1\Gamma_2}$$

Figure 4. Proof pattern typing rules

$$!C,\ !x : \tau \vdash x : \tau \qquad !C, x : \tau \vdash x : \tau \qquad \frac{!C \vdash e : \tau}{!C \vdash !e : !\tau}$$

$$\frac{\begin{array}{c} C_1 \vdash e_1 : \tau_1 \qquad \vdash p : \tau_1 \Longrightarrow \Delta; \Gamma \\ (C_2)(\Delta; \Gamma) \vdash e_2 : \tau_2 \qquad C_1, C_2 \vdash \tau_2 : \mathbb{T} \end{array}}{C_1, C_2 \vdash \text{let}\, p = e_1\ \text{in}\, e_2 : \tau_2}$$

$$\frac{\vdash p : \tau_a \Longrightarrow \Delta; \Gamma \qquad (C)(\Delta; \Gamma) \vdash e : \tau_b \qquad C \vdash \tau_b : \mathbb{T}}{C \vdash \lambda p{:}\tau_a.e : \tau_a \multimap \tau_b}$$

$$\frac{\begin{array}{c} C_1 \vdash e_f : \tau_a \multimap \tau_b \\ C_2 \vdash e_a : \tau_a \end{array}}{C_1, C_2 \vdash e_f\ e_a : \tau_b} \qquad \frac{\begin{array}{c} C_1 \vdash e_1 : \tau_1 \\ C_2 \vdash e_2 : \tau_2 \end{array}}{C_1, C_2 \vdash e_1, e_2 : \tau_1 \otimes \tau_2}$$

$$\frac{\begin{array}{c} C_1,\ !C_2 \vdash \tau_n : \mathbb{N} \qquad C_1,\ !C_2 \vdash \tau_f : \mathbb{N} \rightarrow \mathbb{T} \\ C_1 \vdash e_z : \tau_f\ 0 \qquad !C_2 \vdash e_s : !\forall A{:}\mathbb{N}.\tau_f\ A \multimap \tau_f\ s(A) \end{array}}{C_1,\ !C_2 \vdash \text{elim } \tau_n\ \tau_f\ e_z\ e_s : \tau_f\ \tau_n}$$

$$\frac{C, A : \kappa \vdash e : \tau}{C \vdash \lambda A{:}\kappa.e : \forall A{:}\kappa.\tau} \qquad \frac{C \vdash e : \forall A{:}\kappa.\tau' \qquad C \vdash \tau : \kappa}{C \vdash e\ \tau : [A \leftarrow \tau]\tau'}$$

$$\frac{C \vdash \tau_1 : \kappa \qquad C \vdash e : [A \leftarrow \tau_1]\tau_2}{C \vdash \text{pack}[\tau_1, e]\,\text{as}\, \exists A{:}\kappa.\tau_2 : \exists A{:}\kappa.\tau_2} \qquad \frac{C \vdash e : \tau \qquad \tau \equiv \tau'}{C \vdash e : \tau'}$$

$$\frac{C \vdash e : \tau}{C \vdash \bigcirc e : \bigcirc\tau} \qquad \frac{C_1 \vdash e_1 : \bigcirc(\tau_a \multimap \tau_b) \qquad C_2 \vdash e_2 : \bigcirc\tau_a}{C_1, C_2 \vdash e_1 \bigcirc\!\!<\!\!<\, e_2 : \bigcirc\tau_b}$$

$$\Delta;\ !\Gamma; \Upsilon; \{r \mapsto n\} \vdash \text{fact} : \text{Reg } r\ n$$

$$\Delta;\ !\Gamma; \Upsilon; \{n \mapsto n'\} \vdash \text{fact} : \text{Mem } n\ n'$$

$$\frac{\begin{array}{c} \tau = \forall A_1{:}\kappa_1. \ldots .\forall A_n{:}\kappa_n.\text{Code } n\ \tau' \\ \Delta \vdash \tau_1 : \kappa_1 \ldots \Delta \vdash \tau_n : \kappa_n \end{array}}{\begin{array}{l} \Delta;\ !\Gamma; \Upsilon, n \mapsto \tau; \{\} \vdash \text{code}(n)[\tau_1 \ldots \tau_n] : \\ \qquad \text{Code } n\ ([A_n \leftarrow \tau_n] \ldots [A_1 \leftarrow \tau_1]\tau') \end{array}}$$

Figure 5. Proof term typing rules

one additional technical device: there must be some value that represents a capability. GTAL uses the special value "fact" to connect Ψ to the proper capabilities; like Ψ, the "fact" value appears only in the theory, not the implementation.

In contrast to the register and memory environment Ψ, the code environment Υ is nonlinear, which allows nonlinear code capabilities !(Code $n\ \tau$). Each code capability asserts that at memory location n, there is a block of code with precondition τ. Code blocks may be polymorphic over type variables $A_1 \ldots A_n$, so code values code$(n)[\tau_1 \ldots \tau_n]$ specify types τ_i to instantiate each A_i with. For example, the following type specifies a code block polymorphic over integers N, with a precondition that asks that register r contain N, the address of a continuation function specified by type Code N (Reg r N):

$$\forall N{:}\mathbb{N}.\text{Code } n\ (\text{Reg } r\ N \otimes !(\text{Code } N\ (\text{Reg } r\ N)))$$

The Reg, Mem, and Code capabilities refer to integers n, which we define to be 0 or the successor of an integer $s(n)$. Given zero and successor, we can define less-than-or-equal and array operators as Church encodings, which fold functions over the integers $A, A+1, A+2, \ldots, B-1$:

$$\begin{array}{l} \mathit{Le} = \lambda A{:}\mathbb{N}.\lambda B{:}\mathbb{N}.\forall F{:}\mathbb{N} \rightarrow \mathbb{T}. \\ \qquad !(\forall N{:}\mathbb{N}.F\ N \multimap F\ s(N)) \multimap F\ A \multimap F\ B \\ \mathit{Arr} = \lambda A{:}\mathbb{N}.\lambda B{:}\mathbb{N}.\lambda F{:}\mathbb{N} \rightarrow \mathbb{T}.\forall G{:}\mathbb{N} \rightarrow \mathbb{T}. \\ \qquad !(\forall N{:}\mathbb{N}.F\ N \multimap G\ N \multimap G\ s(N)) \multimap G\ A \multimap G\ B \end{array}$$

For instance, $\mathit{Arr}\ 5\ 8\ (\lambda N : \mathbb{N}.\exists M : \mathbb{N}.\text{Mem } N\ M)$ is an array of free memory in locations $5 \ldots 7$: for any type function G, it transforms $G\ 5$ into $G\ 8$ by applying the function $(\exists M : \mathbb{N}.\text{Mem } N\ M) \multimap G\ N \multimap G\ s(N)$ three times (once for $N = 5$, once for $N = 6$, once for $N = 7$). Given these definitions, it's straightforward to define lemmas about Le, such as transitivity, and lemmas about Arr, such as lemmas for splitting and combining adjacent arrays.

Figure 3 shows the type equivalence rules. The type system includes a primitive recursion (fold) operation "elim $n\ \tau_z\ \tau_s$" that is equivalent to τ_z if $n = 0$ and is equivalent to $(\tau_s\ m\ (\text{elim } m\ \tau_z\ \tau_s))$ if $n = s(m)$. This lets the type system express addition, reasoning by cases, and predecessor:

$$\begin{array}{l} \mathit{Add} = \lambda A{:}\mathbb{N}.\lambda B{:}\mathbb{N}.\text{elim } B\ A\ (\lambda M{:}\mathbb{N}.\lambda \mathit{Accum}{:}\mathbb{N}.s(\mathit{Accum})) \\ \mathit{Case}_\kappa = \lambda A{:}\mathbb{N}.\lambda Z{:}\kappa.\lambda S{:}\mathbb{N} \rightarrow \kappa. \\ \qquad \text{elim } A\ Z\ (\lambda M{:}\mathbb{N}.\lambda \mathit{Accum}{:}\mathbb{N}.S\ M) \\ \mathit{Pred} = \lambda A{:}\mathbb{N}.\mathit{Case}_\mathbb{N}\ A\ 0\ (\lambda M{:}\mathbb{N}.M) \end{array}$$

The term language also contains an induction operation on integers, also called "elim". For simplicity, we ignore 32-bit and 64-bit arithmetic in this paper, and assume that registers and memory words can hold any natural number, but modifying the assembly language rules to use mod-2^n arithmetic would be straightforward.

3.2 Recursive syntax, recursive types, and modal operators

Supporting recursive specifications in a logic requires caution—adding unrestricted recursive types to the system can introduce unbounded recursion, which corresponds to circular reasoning. For example, suppose that a recursive type F_R is defined to be a function type (implication) taking F_R and returning False, so that the following type equivalence holds: $F_R \equiv (\,!F_R) \multimap \mathit{False}$. Then the term $(\lambda\, !x : !F_R.x\ !x)$ has type $(\,!F_R) \multimap \mathit{False}$, and so the (non-terminating) term $(\lambda\, !x{:}!F_R.x\ !x)\ !(\lambda\, !x{:}!F_R.x\ !x)$ has type False, and thus proves False.

Nevertheless, ATS and Jia *et al.*'s system already include a particular form of recursion between the proof language (which must disallow unbounded recursion) and the computation language (which allows nontermination): in these systems, the syntax for computation types and logical formulas is mutually recursive. For example, the logical formula $g \Rightarrow \tau$ in Jia *et al.*'s system asserts that memory location g has type τ, and the computation type $(F) \rightarrow 0$ specifies a typed assembly language code block with precondition F, as shown in this simplified subset of their syntax:

$$\begin{array}{rl} \textit{formula} & F = \ldots \mid g \Rightarrow \tau \\ \textit{type} & \tau = \ldots \mid (F) \rightarrow 0 \end{array}$$

The proof language can safely cooperate with the computation language because the proof language handles computation types τ opaquely; a proof with access to a formula $g_1 \Rightarrow ((F_1) \rightarrow 0)$ cannot actually invoke the typed assembly language block of type $(F_1) \rightarrow 0$ (which might have a side effect, or fail to terminate). Furthermore, even the formulas inside F_1 appear opaque to this proof, since the formula F_1 is buried inside the computation type $(F_1) \rightarrow 0$. In fact, the following computation recursive type equivalence is sound:

$$\tau_R \equiv ((g_2 \Rightarrow \tau_R) \rightarrow 0)$$

even though it introduces a logical recursive type equivalence, given the abbreviation $F_R = (g_2 \Rightarrow \tau_R)$:

$$F_R \equiv (g_2 \Rightarrow ((F_R) \rightarrow 0)))$$

We can generalize this idea in two steps. First, following Nakano [17, 1], we add an explicit opaqueness operator $\bigcirc$ to the logical formula syntax, and allow recursive type equivalences of the form $F_R = \bigcirc(\ldots F_R \ldots)$. For example, $F_R \equiv \bigcirc((\,!F_R) \multimap False)$ is a legal equivalence. Second, in the spirit of monadic IO [25], we distinguish between a pure language (the proof language) and an impure language (the computation language), and stipulate that only the impure language can extract formulas from underneath the $\bigcirc$ operator — proof terms can coerce F to $\bigcirc F$, but only computation terms can coerce $\bigcirc F$ to F. To see the intuition behind this restriction, consider again the non-terminating term $(\lambda\,!x :!F_R.x\ \ !x)\ \ !(\lambda\,!x :!F_R.x\ \ !x)$. The application "$x\ \ !x$" is now ill-typed, because $x's$ type $\bigcirc((\,!F_R) \multimap False)$ is not a function type, and the proof language cannot coerce x to the function type $((\,!F_R) \multimap False)$. By contrast, the equivalence $\tau_R \equiv (\bigcirc(g \Rightarrow \tau_R)) \rightarrow 0$ gives the computation language a way to express non-termination: a computation function f of type τ_R coerces its precondition $\bigcirc(g \Rightarrow \tau_R)$ to the formula $(g \Rightarrow \tau_R)$, uses this formula to load a function f' of type τ_R from address g, and then invokes f'. If f' and f are the same function, this computation diverges. Thus, it is still possible to encode non-terminating *computations* using recursive types, but Section 5 proves that all well-typed *proofs* terminate.

GTAL's recursive type "rec τ" uses the $\bigcirc$ operator to form recursive definitions, such as this recursive definition of a zero-terminated linked list:

$$\begin{array}{l} \text{rec } (\lambda \mathit{List}\!:\!\mathbb{N} \rightarrow \mathbb{T}.\lambda N\!:\!\mathbb{N}. \\ \quad !(\mathit{Le}\ 1\ N) \multimap (\exists N'\!:\!\mathbb{N}.\text{Mem}\ N\ N' \otimes \mathit{List}\ N')) \end{array}$$

Unlike many proof languages (such as TL's proof language), GTAL allows recursive type definitions that mention the recursively bound name in both positive and negative positions, and the $\bigcirc$ operator protects the proof terms from non-termination. (Note: the type equivalence rule rec $\tau \equiv \bigcirc(\tau\ (\text{rec}\ \tau))$ implements a form of *equi-recursive types*, which tend to be more challenging to type check than iso-recursive types. Our actual implementation of the proof language requires explicit term annotations to tell the type checker where to apply the rule rec $\tau \equiv \bigcirc(\tau\ (\text{rec}\ \tau))$.)

$$\begin{array}{rccl} \textit{coercion} & c & = & e \mid \sharp c \\ \textit{instruction} & i & = & c \\ & & \mid & [c]\text{movi}\, r \leftarrow n \\ & & \mid & [c_1|c_2]\text{mov}\, r_1 \leftarrow r_2 \\ & & \mid & [c_1|c_2]\text{addi}\, r_1 \leftarrow r_2 + n \\ & & \mid & [c_1|c_2|c_3]\text{add}\, r_1 \leftarrow r_2 + r_3 \\ & & \mid & [c_1|c_{2,mem}]\text{load}\, r_1 \leftarrow [r_2 + n] \\ & & \mid & [c_{mem}|c_1|c_2]\text{store}\, [r_1 + n] \leftarrow r_2 \\ & & \mid & [c_1|c_2|x.c_{jmp}]\text{ble}\, r_1 \leq r_2\ n \\ \textit{block} & \mathit{blk} & = & \text{let}\, p = i \text{ in}\, \mathit{blk} \\ & & \mid & [c_{jmp}]\text{jmp}\, n \\ & & \mid & [c|c_{jmp}]\text{jr}\, r \\ \textit{block w/ header} & b & = & \lambda A\!:\!\kappa.b \mid \lambda p\!:\!\tau.\mathit{blk} \\ \textit{code heap} & \Lambda & = & \{\} \mid \Lambda, n \mapsto b \\ \textit{program} & \mathit{prog} & = & \text{let}\, p = c \text{ in}\, \mathit{prog} \mid \Lambda\ \mathit{blk} \end{array}$$

Figure 6. Computation language syntax

It's often necessary to coerce a formula $\bigcirc A$ to a some related formula $\bigcirc B$. For example, GTAL encodes classes using recursive types, and a coercion from a subclass object to a superclass object coerces $\bigcirc A$ to $\bigcirc B$, where A describes the subclass and B describes the superclass. The garbage collector's heap extension lemma also coerces $\bigcirc A$ to $\bigcirc B$, where A describes an object in some heap H and B describes an object in an extended heap H'. It's possible to coerce $\bigcirc A$ to $\bigcirc B$ by first using the computation language to coerce $\bigcirc A$ to A, and then using the proof language to coerce A to B to $\bigcirc B$, but this often forces an awkward rendezvous between the computation and proof language. The heap extension lemma, for example, performs one $\bigcirc A$- to-$\bigcirc B$ coercion for each word in the heap, and it would be impractical to execute one computation step per heap word every time the heap grows. Luckily, there is a sound axiom, the modal "distribution axiom" (written here as "$\bigcirc <<$"), that lets the proof language perform a coercion inside a $\bigcirc$ operator:

$$\bigcirc << \quad : \quad \bigcirc(A \rightarrow B) \rightarrow (\bigcirc A \rightarrow \bigcirc B)$$

This axiom, when combined with the axiom $A \rightarrow \bigcirc A$, fits into a general framework of intuitionistic modal logics categorized by Simpson [22]. (Specifically, these two axioms are valid for Kripke models $(W, \leq, R, V)$ where R is a subset of $\leq$; see [22] for derivations of various intuitionistic modal logics based on various choices of R.) Nakano [17] describes additional axioms that may be appropriate for the $\bigcirc$ operator, though the two axioms above are sufficient for this paper. Note, though, the standard monadic *bind* axiom, "$>>=$", is inappropriate for GTAL, as it is strong enough to express non-terminating proofs when combined with GTAL's recursive types:

$$>>= \quad : \quad \bigcirc A \rightarrow (A \rightarrow \bigcirc B) \rightarrow \bigcirc B$$

Therefore, even though GTAL's use of $\bigcirc$ is analogous to monadic IO, it is fundamentally different. Nevertheless, Wadler [25] describes two slightly weaker operators ($\bigcirc A \rightarrow (A \rightarrow B) \rightarrow \bigcirc B$ and $\bigcirc A \rightarrow \bigcirc B \rightarrow \bigcirc(A \times B)$) that predate "$>>=$" and are safe for GTAL; these weaker operators can derive $\bigcirc <<$ and are derivable from $\bigcirc <<$.

4. A typed assembly language

GTAL's computation language, shown in Figure 6, consists of assembly language instructions for moving, adding, loading, storing, conditional branch, direct jumps, and jumps through registers. Like LTT [8], TL [21], and ATS [30], the computation language manipulates proof terms c explicitly, assigning proofs to variables x (occurring in patterns p) using "let" expressions. Each instruction requires one or more proof terms to provide evidence of the instruction's safety; for example, the move instruction "$[c_1|c_2]\text{mov}\, r_1 \leftarrow r_2$" requires proofs c_1 and c_2 that registers r_1 and r_2 are accessible. This approach results in verbose annotations, but makes type checking easy. Section 5's coercion termination theorem allows GTAL to erase the annotations after type-checking the code.

Unlike the proof language, the computation language supports the coercion $\bigcirc A \rightarrow A$. The operator "$\sharp$" erases a single $\bigcirc$ from a type:

$$\frac{C \vdash c : \bigcirc\tau}{C \vdash \sharp c : \tau}$$

We define a "coercion" c to be a proof term e preceded by zero or more $\sharp$ operators. Typically, a program uses the $\sharp$ operator to unroll a recursive type. If the variable x has type rec τ, then the computation term "let $y = \sharp x$ in blk" introduces an unrolled variable y of type $(\tau\ (\text{rec}\ \tau))$.

Each block b in the program specifies a precondition τ. For example, the following block's precondition requires that register r hold a number A, and that memory location A hold a number B that is the address of another block of code:

$$\begin{array}{l}
\lambda A\!:\!\mathbb{N}.\lambda B\!:\!\mathbb{N}.\lambda(xr, xm, !xc) : \\
\qquad\qquad (\text{Reg}\ r\ A) \otimes (\text{Mem}\ A\ B) \otimes !(\forall X\!:\!\mathbb{T}.\text{Code}\ B\ X). \\
\quad \text{let}\, xr' = [xr|xr, xm]\text{load}\, r \leftarrow [r+0]\ \text{in} \\
\quad [xr'|xc\ ((\text{Reg}\ r\ B) \otimes (\text{Mem}\ A\ B)), (xr', xm)]\text{jr}\, r
\end{array}$$

The block accepts its precondition in the variables xr, xm, and xc, and then executes a load and a jump. Each instruction requires evidence that the instruction is safe, and produces new evidence for subsequent instructions. For example, the load instruction requires two coercions as evidence: the coercion c_1 proves that the destination register is available, and $c_{2,mem}$ proves that the source register and memory location are available. In this example, r is both the source and destination register, so the same evidence xr satisfies both requirements. (Note that even though xr is linear, it is safe and useful for the load instruction's typing rule to share the context C's linear assumptions among the operands so that both operands can use xr, rather than splitting C disjointly between the operands.) The load instruction consumes the evidence for the destination register and produces new evidence saying that the register now contains the loaded value from memory:

$$\frac{\begin{array}{c} C = C_a, C_r \qquad C = C', C'' \\ C' \vdash c_{2,mem} : (\text{Reg}\ r_2\ \tau_2) \otimes (\text{Mem}\ \ \tau_2 + n\ \ \tau_m) \\ C_r \vdash c_1 : \text{Reg}\ r_1\ \tau_1 \\ C_a, x \mapsto (\text{Reg}\ r_1\ \tau_m) \vdash \mathit{blk} \end{array}}{C \vdash \text{let}\, x = [c_1|c_{2,mem}]\text{load}\, r_1 \leftarrow [r_2 + n]\ \text{in}\ \mathit{blk}}$$

(For simplicity, we show a special case of the general rule for load, accepting just variables x rather than patterns p.) In the example above, xr' is assigned type Reg r B. This prepares the block for a jump to the code at address B:

$$\frac{\begin{array}{c} C = C', C'' \\ C' \vdash c : \text{Reg}\ r\ \tau_r \\ C \vdash c_{jmp} : (\text{Code}\ \tau_r\ \tau_c) \otimes \tau_c \end{array}}{C \vdash [c|c_{jmp}]\text{jr}\, r}$$

The code evidence xc shown above is polymorphic over all preconditions X, so the example instantiates X with a particular precondition $\tau_c = (\text{Reg}\ r\ B) \otimes (\text{Mem}\ A\ B)$, and then provides evidence (xr', xm) of type τ_c.

Other instructions are similar. The store instruction consumes a memory assertion Mem τ_1 τ_2 to produce a new memory assertion Mem τ_1 τ_2'. The add instructions consume a Reg τ_1 τ_2 to produce a Reg τ_1 $s(s(\ldots s(\tau_2)\ldots))$. The conditional branch produces an assertion $!(\mathit{Le}\ s(\tau_2)\ \tau_2)$ for the instructions following a comparison of τ_1 to τ_2, and an assertion $!(\mathit{Le}\ \tau_1\ \tau_2)$ for the branch target (the variable x shown in the syntax holds the latter assertion, so that the coercion c can use x to satisfy the branch target's precondition).

A program prog consists of a currently executing block blk and a mapping Λ from code addresses to blocks, preceded by zero or more "let" declarations. The "let" declarations are used to establish libraries of types and proofs for use by the coercions and expressions inside the blocks; for example, the garbage collector in Section 7 establishes a large library of basic types (e.g., True, False, Le, Arr) and types for garbage collection invariants. The complete typing and evaluation rules are available online [10].

5. Formal properties

GTAL is safe in the standard sense of type preservation (subject reduction) and progress; the preservation and progress theorems for programs prog encompass preservation and progress for blocks blk, coercions c, and expressions e; proofs are by induction over judgments:

Theorem [type preservation]: If $C = \Delta; \Gamma; \Upsilon; \Psi$ and $\vdash C$ and $C \vdash \mathit{prog}$ and $\Psi; \mathit{prog} \longrightarrow \Psi'; \mathit{prog}'$ then $\vdash C'$ and $C' \vdash \mathit{prog}'$ where $C' = \Delta; \Gamma; \Upsilon; \Psi'$.

Theorem [progress]: If $C = \{\}; \{\}; \Upsilon; \Psi$ and $\vdash C$ and $C \vdash \mathit{prog}$ then there is some $\Psi'; \mathit{prog}'$ such that $\Psi; \mathit{prog} \longrightarrow \Psi'; \mathit{prog}'$.

The termination theorems make it safe to erase the annotations from a TAL program before running it, so that TAL execution is just untyped assembly language execution:

Theorem [expression termination]: If $C = \{\}; \{\}; \Upsilon; \Psi_1$ and $\vdash C$ and $C \vdash e_1 : \tau$ then all sequences of reduction steps $e_1 \longrightarrow e_2 \longrightarrow e_3 \longrightarrow \ldots$ terminate at some $e_n = v_n$. (Note that values v do not step; particularly, the value $\bigcirc e$ does not step.) Proof by preservation and progress, and by mapping e and τ onto calculus of inductive construction terms/types [21, 12], erasing any e inside a $\bigcirc e$ and any τ inside a $\bigcirc\tau$. The key observation is that in a proof term $\bigcirc e$, the term e plays no role in the reduction. More formally, we define an erasure that maps $\bigcirc e$ to the dummy value true (of type True) and maps $\bigcirc\tau$ and rec τ to dummy types (e.g., True, if $\tau : \mathbb{T}$), and prove that this erasure has no effect on the number of reduction steps. We then prove that the erased term is well-typed, so that proving termination of well-typed terms in the original language reduces to proving termination of well-typed terms in a language without the $\bigcirc$ operator and rec τ type. Note that if the proof language included a coercion $\bigcirc\tau \rightarrow \tau$, the corresponding evaluation rule $\bigcirc e \longrightarrow e$ would destroy the proof, because e could escape into the rest of the evaluation and affect the number of subsequent reduction steps. Intuitively, this is why the monadic operator $>\!\!>\!\!=$ is unsafe for the $\bigcirc$ operator: the rule $(\bigcirc e_a)\ >\!\!>\!\!=\ e_b \longrightarrow e_b\ e_a$ allows e_a to escape outside the $\bigcirc$ operator and affect the rest of the computation. By contrast, the rule $(\bigcirc e_b)\bigcirc\!<\!\!<\ (\bigcirc e_a) \longrightarrow \bigcirc(e_b\ e_a)$ produces an opaque value, with no escape.

Theorem [coercion termination]: If $C = \{\}; \{\}; \Upsilon; \Psi_1$ and $\vdash C$ and $C \vdash c_1 : \tau$ then all sequences of reduction steps $c_1 \longrightarrow c_2 \longrightarrow c_3 \longrightarrow \ldots$ terminate at some $c_n = v_n$. Proof by induction on the number of $\sharp$ operators in c (using expression termination in both base and induction cases).

Detailed proofs of these theorems are available online [10].

6. Heaps and heap object types

Sections 3-5 described GTAL's syntax completely; the rest of the paper adds no new syntax. As promised in Section 1, GTAL includes no types or expressions for heaps, heap objects, allocation, and deallocation. This section describes how to encode these concepts using the existing GTAL syntax.

GTAL's types are sufficient to implement a heap and nonlinear, arbitrarily aliased pointers into the heap. For example, suppose that the heap contains just two words A_0 and A_1 at memory locations N and $N+1$. Define the heap $M = M_0 \otimes M_1$ where, for an immutable heap, $M_k = \text{Mem } N+k\ A_k$ (using the abbreviations $N+0 = N$, $N+1 = s(N)$, $N+2 = s(s(N))$, etc.), or, for a mutable heap, $M_k = \exists A_k{:}\mathbb{N}.\ (\text{Mem } N{+}k\ A_k) \otimes (F_k\ A_k)$ where F_k is an invariant that A_k must satisfy. A nonlinear pointer P_k into the heap M is simply a nonlinear function $!(M \multimap M_k \otimes \ldots)$ that extracts M_k from M. The "…" indicates everything else in M that is not contained in M_k; for the typed assembly language presented below, it's helpful to describe "everything else" precisely, using the type $(M_k \multimap M)$ (which intuitively can be thought of as "M_k subtracted from M"):

$$P_k = !(M \multimap M_k \otimes (M_k \multimap M))$$

Given a nonlinear pointer P_k and a linear heap M, a simple proof term proves M_k and $(M_k \multimap M)$. The computation language can then use M_k to load A_k from memory location $N+k$ or store A_k to memory location $N+k$. After loading or storing, a simple proof uses M_k and $(M_k \multimap M)$ to reconstitute the heap M. A program passes the linear heap M from function to function explicity as the program executes; this allows every function in the program to use the pointers P_k for loads and stores at any time.

Unfortunately, if the program extends the heap with a third memory word M_2, this strategy breaks down. The new heap type $M' = M_0 \otimes M_1 \otimes M_2$ is different from the old heap type M, and this means that the old pointers P_k no longer apply to the current heap (since they refer to M, not M'). One strategy is to restrict pointers to a particular pattern, as Jia *et al.* do for stacks [13]; we can imitate their approach by observing that $!(M' \multimap M \otimes (M \multimap M'))$, so that pointers into M are still usable via a two-step process: extract M from M', then extract M_k from M. This is good for stacks, which have very restricted usage patterns for pointers, but insufficient for heaps, because pointers to M have different types than pointers to M', and this prevents a program from using these pointers interchangeably.

Another strategy is to find all old pointers in the program and update their types to refer to M'. Updating an old pointer is easy, but finding the old pointers is hard, since they may be hidden inside functions, inside recursive data types, their types may appear non-positively in other types, etc. The traditional meta-theoretic approach to heaps uses an induction over terms to prove a heap extension lemma [16]; this is a straightforward induction at the meta-level, but it appears difficult to encode an induction over terms efficiently from *inside* the language (i.e., to construct terms that perform structural induction over all other terms).

In earlier work [11], we overcame this problem by adding a meta-level extension lemma to the proof language. This caused some tension with our stated goal of not baking the heap into the language, though — if there's no heap built into the language, what does the extension lemma extend? Our answer to this was rather eccentric (the extension lemma extended a set of recursive type bindings), so for this paper we've chosen a cleaner approach, closer in spirit to work by Crary and Weirich [9] and Shao *et al.* [21]. Shao *et al.* define "source types" Ω as inductive type definitions within their proof language (CIC):

$$\begin{array}{rl}\text{Inductive}\,\Omega : \text{Kind} \ := & \mathit{snat} : \mathit{Nat} \rightarrow \Omega \\ \mid & \mathit{sbool} : \mathit{Bool} \rightarrow \Omega \\ \mid & \twoheadrightarrow : \Omega \rightarrow \Omega \rightarrow \Omega \\ \mid & \mathit{tup} : \mathit{Nat} \rightarrow (\mathit{Nat} \rightarrow \Omega) \rightarrow \Omega \\ \mid & \forall_s : \Pi k : s.(k \rightarrow \Omega) \rightarrow \Omega \\ \mid & \exists_s : \Pi k : s.(k \rightarrow \Omega) \rightarrow \Omega \end{array}$$

In this example (taken from [21]), the source types include singleton natural numbers (such as "$\mathit{snat}\ 7$"), singleton booleans, function types (such as "$\twoheadrightarrow\ (\mathit{snat}\ 7)\ (\mathit{sbool}\ \mathit{true})$"), tuple types (specified by a length and a mapping from indices to types), and polymorphic and existential types (both written in a higher-order abstract syntax style). The program then performs analysis and transformations on these types from within the language, without requiring any meta-level proofs about the source types. A particularly useful transformation is Crary and Weirich's "interp" function, which maps the inductively defined sources types to concrete representations.

Following these approaches, we first define an inductive syntax for what we'll call "heap types", and then define a function that maps heap types to concrete representations. Given these heap types, we can define a heap and prove heap extension inside the language, without requiring a meta-level heap extension lemma.

The ability to analyze inductively defined heap types within the language comes at a cost: the types of objects appearing in the heap are limited to whatever types can be expressed using inductive definitions. There are two reasons to believe that this cost is reasonable. First, the *representations* of the heap types may still incorporate any types in the type system (existential types, capabilities, etc.), not just heap types. Second, the Ω example above shows that a proof language like CIC is powerful enough to express common programming language features. Nevertheless, CIC's rules for well-formed inductive types and operations on inductive types are intricate and subtle; for this paper, we prefer to stick with a simpler type system (and leave the integration of GTAL with CIC as future work). Luckily, the proof system from Section 3 already defines one inductive type, natural numbers, and this one type alone is enough to encode classes, methods, and subtyping in the style of Chen and Tarditi's LIL$_C$ [6].

Just as Crary and Weirich define mappings from inductively defined types to concrete representations, Chen and Tarditi define mappings from *class names* to concrete representations. To encode this approach, we use natural numbers as class names, and then define a mapping from natural numbers to representations. Let the kind of class names $\mathbb{C}$ be an abbreviation for the natural number kind: $\mathbb{C} = \mathbb{N}$. Then the representation function Rep is a recursive type of the following form, where τ_{Rep} (defined in Section 6.1) has kind $\mathbb{T}$:

$$\begin{array}{l}\mathit{Rep} = \text{rec } (\lambda \mathit{Rep} : \mathbb{C} \rightarrow \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{C}) \rightarrow \mathbb{N} \rightarrow \mathbb{T}. \\ \qquad \lambda \mathit{ClassName} : \mathbb{C}. \\ \qquad \lambda \mathit{Offset} : \mathbb{N}. \\ \qquad \lambda \mathit{SpaceMap} : \mathbb{N} \rightarrow \mathbb{C}. \\ \qquad \lambda \mathit{Value} : \mathbb{N}.\tau_{\mathit{Rep}})\end{array}$$

Each class representation specifies a word representation for each word at offset Offset. For example, consider a class Point, with a header word and integer data word, and a class Link with a header word and two data words:

class Point {τ_{PointHdr}; int}
class Link {τ_{LinkHdr}; int; Point}

Point will define word representations for offsets 0 and 1, and Link will define word representations for offsets 0, 1, and 2. Each word representation takes a parameter $\mathit{SpaceMap}$, described below, and a value indicating the contents of memory at offset Offset from the beginning of an object. Given these parameters, the word representation produces an invariant that must be satisfied by the contents of memory at Offset. For example, a word of type "int" may contain any natural number, so the word representation is the trivial invariant "$\lambda \mathit{SpaceMap} : \mathbb{N} \rightarrow \mathbb{C}. \lambda \mathit{Value} : \mathbb{N}. \mathit{True}$". A type "pos" that holds non-zero natural numbers would have the word representation "$\lambda \mathit{SpaceMap} : \mathbb{N} \rightarrow \mathbb{C}. \lambda \mathit{Value} : \mathbb{N}. \mathit{Le}\ 1\ \mathit{Value}$".

The $\mathit{SpaceMap}$ parameter describes the memory layout at a given point in the program's execution. For the moment, suppose that a single $\mathit{SpaceMap}$ function defines the current state of the entire heap. (The next section actually defines two $\mathit{SpaceMap}$ functions, one for from-space and one for to-space.) Each object in the heap consists of a header word followed by zero or more interior words. $\mathit{SpaceMap}$ maps word addresses to descriptions of words. Specifically, $\mathit{SpaceMap}\ N = 0$ if address N holds a free word, $\mathit{SpaceMap}\ N = 1$ if address N holds an interior word, and $\mathit{SpaceMap}\ N = s(s(C))$ if address N holds the header word of an object of class C.

Word representations for class-pointer types define constraints on $\mathit{SpaceMap}$. The word representation of Link's last field, of type Point, is:

$$\begin{array}{l}
\lambda \mathit{SpaceMap}\!:\!\mathbb{N} \to \mathbb{C}.\lambda \mathit{Value}\!:\!\mathbb{N}. \\
\quad !(\mathit{Eq}\ (\mathit{SpaceMap}\ \ \mathit{Value})\ s(s(C_{\text{Point}}))) \\
\quad \otimes\, !(\mathit{Eq}\ (\mathit{SpaceMap}\ \ s(\mathit{Value}))\ 1)
\end{array}$$

where C_{Point} is the class name for Point. This representation requires that at addresses Value and $s(\mathit{Value})$, there lives an object of class Point. (Actually, we've fibbed here slightly; the real definition of the pointer-to-Point representation allows any subclass of Point to reside at address Value, but we defer subclasses to Appendix A. Also, for simplicity, we assume non-null pointers in this paper; the representation of a possibly null pointer would have the form "$\lambda \mathit{SpaceMap} : \mathbb{N} \to \mathbb{C}.\lambda \mathit{Value} : \mathbb{N}.!(\mathit{Le}\ 1\ \ \mathit{Value}) \multimap \tau$", stating that the invariant τ is only relevant if the address Value is not 0.)

The heap maintains two properties for each heap address n. First, a linear capability "Mem $n\ \ \mathit{Value}$" specifies that address n currently holds some value Value. Second, if n holds a field of an allocated object, then Value must satisfy the word representation of that field (e.g. Value must satisfy "$\mathit{Le}\ 1\ \ \mathit{Value}$" for a field of type "pos"). The mapping from n to n's word representation consists of two steps: first, use $(\mathit{SpaceMap}\ \ n)$ to find the class $\mathit{ClassName}$ and offset Offset stored at n; second, use Rep to find the word representation $(\mathit{Rep}\ \ \mathit{ClassName}\ \ \mathit{Offset})$. Suppose, for example, that $\mathit{SpaceMap}$ maps address n to class Point and offset 1. Then $(\mathit{Rep}\ C_{\text{Point}}\ 1)$ will be the word representation for "int": $\lambda \mathit{SpaceMap} : \mathbb{N} \to \mathbb{C}.\lambda \mathit{Value} : \mathbb{N}.\mathit{True}$. A program can use the capability Mem $n\ \ \mathit{Value}$ to load value Value from address n, and the word representation to discover that Value satisfies the (trivial) constraint "True". The program can also use the capability Mem $n\ \ \mathit{Value}$ to store a new value Value' into address n, provided that Value' also satisfies the constraint (in this case, all values satisfy the constraint "True"). Notice that as long as a new value Value' satisfies the constraint for the word representation at n, storing Value' into n does not change the global map $\mathit{SpaceMap}$. In other words, the heap can treat "weak updates" to address n (updates that do not change the invariant at n) locally, without considering other memory addresses.

6.1 Generalized representations

The Point and Link classes provide examples of how to define a particular definition of Rep. This rest of this section generalizes Rep's definition to cover all classes defined by the following grammar:

$$\begin{array}{l}
\mathit{class} = \tau_{methods}\ \mathit{field}_1\ \ \ldots\ \ \mathit{field}_n \\
\mathit{field} = n_{class}\ \mid\ \tau_{primitive}
\end{array}$$

Each class defines a layout consisting of a method table layout, specified by any type τ_{method} of kind $\mathbb{N} \to \mathbb{T}$, and zero or more fields. Each field is either a pointer to a class, specified by the integer name of the pointed-to class, or a primitive type specified by any type $\tau_{primitive}$, of kind $\mathbb{N} \to \mathbb{T}$. For example, $\tau_{primitive}$ could be $\lambda \mathit{Value}\!:\!\mathbb{N}.\mathit{Le}\ 1\ \ \mathit{Value}$ to represent a positive integer field. Since the τ_{method} and $\tau_{primitive}$ types do not depend on $\mathit{SpaceMap}$, the garbage collector will not care how they are defined; it will simply copy the value Value from one memory location to another.

Just as we used natural numbers to encode class names, we use natural numbers to encode class layouts. The encoding consists of three functions, where τ_{size} has kind $\mathbb{N}$ and $\tau_{PtrClassName}$ and τ_{Prim} have kind $\mathbb{T}$:

$$\begin{array}{l}
\mathit{SizeOf} = \lambda \mathit{ClassName}\!:\!\mathbb{C}.\tau_{size} \\
\mathit{Ptrs} = \lambda \mathit{ClassName}\!:\!\mathbb{C}. \\
\qquad\quad \lambda \mathit{Offset}\!:\!\mathbb{N}.\tau_{PtrClassName} \\
\mathit{Prims} = \lambda \mathit{ClassName}\!:\!\mathbb{C}. \\
\qquad\quad \lambda \mathit{Offset}\!:\!\mathbb{N}. \\
\qquad\quad \lambda \mathit{Value}\!:\!\mathbb{N}.\tau_{Prim}
\end{array}$$

For each $\mathit{ClassName}$, the type Ptrs maps each field of the class to either 0, to indicate a primitive type, or to $s(\mathit{ClassName}')$, to indicate a pointer type to class $\mathit{ClassName}'$. For each $\mathit{ClassName}$, the type Prims maps each field of the class to either an invariant τ_{Prim} for a primitive type field, or to the type False for a pointer type field.

SizeOf specifies the size, in words, of each class's layout. Given this size information, we can define a pointer to an object of class $\mathit{ClassName}$ (generalizing the earlier definition of "pointer-to-Point"):

$$\begin{array}{l}
\mathit{ExactPtr} = \\
\quad \lambda \mathit{ClassName}\!:\!\mathbb{C}. \\
\quad \lambda \mathit{SpaceMap}\!:\!\mathbb{N} \to \mathbb{N}. \\
\quad \lambda \mathit{Value}\!:\!\mathbb{N}. \\
\quad !(\mathit{Eq}\ (\mathit{SpaceMap}\ \ \mathit{Value})\ s(s(\mathit{ClassName}))) \\
\quad \otimes !(\mathit{Arr}\ 1\ (\mathit{SizeOf}\ \ \mathit{ClassName})\ (\lambda N\!:\!\mathbb{N}. \\
\qquad\quad !(\mathit{Eq}\ (\mathit{SpaceMap}\ (\mathit{Add}\ \ \mathit{Value}\ \ N))\ 1)))
\end{array}$$

The type $\mathit{ExactPtr}$ specifies that an object of class $\mathit{ClassName}$ resides at addresses $\mathit{Value} \ldots \mathit{Value}{+}(\mathit{SizeOf}\ \ \mathit{ClassName}){-}1$ — specifically, it specifies that $\mathit{SpaceMap}\ \ \mathit{Value} = \mathit{ClassName}{+}2$, and that $\mathit{SpaceMap}\ (\mathit{Value} + 1) = 1 ... \mathit{SpaceMap}\ (\mathit{Value} + (\mathit{SizeOf}\ \ \mathit{ClassName}){-}1) = 1$. (The prefix "Exact" indicates that the pointer points to an object whose class is exactly $\mathit{ClassName}$, and not a subclass of $\mathit{ClassName}$; we defer subclass pointers to Appendix A.)

A single definition of Rep, written in terms of SizeOf, Ptrs, and Prims, now suffices for all sets of class layouts:

$$\begin{array}{l}
\mathit{Rep} = \text{rec}\ (\lambda \mathit{Rep}\!:\!\mathbb{C} \to \mathbb{N} \to (\mathbb{N} \to \mathbb{C}) \to \mathbb{N} \to \mathbb{T}. \\
\quad \lambda \mathit{ClassName}\!:\!\mathbb{C}. \\
\quad \lambda \mathit{Offset}\!:\!\mathbb{N}. \\
\quad \lambda \mathit{SpaceMap}\!:\!\mathbb{N} \to \mathbb{C}. \\
\quad \lambda \mathit{Value}\!:\!\mathbb{N}. \\
\quad\ \mathit{Case}\ \ \mathit{Offset}\ (\mathit{Prims}\ \ \mathit{ClassName}\ 0\ \ \mathit{Value} \otimes \\
\qquad\qquad \mathit{GcHeader}\ \ \mathit{ClassName}\ \ \mathit{Value})\ (\lambda M\!:\!\mathbb{N}. \\
\quad\ \ \mathit{Case}\ (\mathit{Ptrs}\ \ \mathit{ClassName}\ \ \mathit{Offset}) \\
\qquad (\mathit{Prims}\ \ \mathit{ClassName}\ \ \mathit{Offset}\ \ \mathit{Value}) \\
\qquad (\lambda \mathit{ClassName}'\!:\!\mathbb{C}. \\
\qquad\qquad \mathit{ExactPtr}\ \ \mathit{ClassName}'\ \ \mathit{SpaceMap}\ \ \mathit{Value}))))
\end{array}$$

The definition of Rep is by cases, using the definition of Case from Section 3 ($\mathit{Case}\ 0\ \tau_z\ \tau_s\ \equiv\ \tau_z$ and $\mathit{Case}\ s(M)\ \tau_z\ \tau_s\ \equiv\ \tau_s\ \ M$). The most important case is inner case, which defines primitive type fields in terms of Prims and defines pointer type fields to be an $\mathit{ExactPtr}$ to the pointed-to class $\mathit{ClassName}'$. The outer case differentiates the header word from the fields; the header word is defined in terms of Prims and then augmented with some extra information for the garbage collector, as defined in the next section.

The definition above expands Rep into one big recursive datatype. Our actual implementation parameterizes Rep's defin-

```
FwdPtr =
 load Rtmp2 <- [Rptr + 0]  // load header (fromspace)
 ble(Rdl<=Rtmp2) FwdNoCopy // already forwarded?
 load Rtmp1 <- [Rtmp2 + 0] // load (SizeOf C)
 add Rtmp1 <- Rtk + Rtmp1  // allocate (SizeOf C) words
 ble(Rtmp1<=Rtl) FwdCopy   // enough space?
 jmp Fail                  // not enough space
FwdNoCopy =
 jr Rretc                  // return forwarding pointer
FwdCopy =
 store [Rtk + 0] <- Rtmp2  // copy word 0
 mov Rtmp4<-Rptr           // initialize fromspace ptr
 mov Rtmp2<-Rtk            // save tospace address
 jmp FwdCopyLoop
FwdCopyLoop =
 addi Rtmp4 <- Rtmp4 + 1   // increment fromspace ptr
 addi Rtk <- Rtk + 1       // increment tospace ptr
 ble(Rtmp1 <= Rtk) FwdDone // reached end of object?
 load Rtmp3 <- [Rtmp4 + 0] // \
 store [Rtk + 0] <- Rtmp3  //  - copy one word
 jmp FwdCopyLoop
FwdDone =
 store [Rptr + 0] <- Rtmp2 // set forwarding pointer
 jr Rretc                  // return copied object

GcStart =
 mov Rfk <- Rtk            // \
 mov Rtj <- Rfi            //   \
 mov Rtk <- Rfi            //     \
 mov Rfi <- Rti            //         swap fromspace,
 mov Rti <- Rtj            //            tospace
 mov Rtmp1 <- Rfl          //     /
 mov Rfl <- Rtl            //   /
 mov Rtl <- Rtmp1          // /
 jr Rretg

GcLoop =
 ble(Rtk<=Rtj) GcDone      // queue empty?
 load Rtmp1 <- [Rtj + 0]   // load header (tospace)
 load Rtmp1 <- [Rtmp1 + 1] // load scan function
 jr Rtmp1                  // jump to scan function
GcDone =
 jr Rretg                  // finished scanning
```

Figure 7. Unannotated garbage collector code

```
ScanPoint =
 addi Rtj <- Rtj + 2       // TJ←next object in queue
 jmp GcLoop                // finished scanning object

ScanLink =
 load Rptr <- [Rtj + 2]    // load ptr to Point object
 movi Rretc <- ScanLink2   // set return address
 jmp FwdPtr                // forward the Point ptr
ScanLink2 =
 store [Rtj + 2] <- Rtmp2  // store forwarded pointer
 addi Rtj <- Rtj + 3       // TJ←next object in queue
 jmp GcLoop                // finished scanning object
```

Figure 8. Unannotated scan code for Point and Link classes

ition over all possible $Ptrs$, $Prims$, and $SizeOf$:

$$Rep = \lambda Ptrs: \ldots \lambda Prims: \ldots \lambda SizeOf: \ldots \text{rec } (\ldots)$$

This allows the lemmas about Rep, such as the heap extension lemma described in the next section, to be polymorphic over all $Ptrs$, $Prims$, and $SizeOf$, so that they need not be reproved for each choice of class layouts. Similarly, the implementation parameterizes $ExactPtr$ over all possible $SizeOf$. Finally, the implementation breaks Rep into smaller, mutually recursive pieces (encoding mutual recursion by parameterizing the pieces over Rep). For simplicity, this paper omits these parameterizations, and treats Rep as a monolithic recursive datatype.

7. A garbage collector

This section describes a simple garbage collector written in GTAL. Figures 7 and 8 show the unannotated code for the collector. For clarity, the figure uses textual labels for blocks (e.g., "GcLoop") in place of integer code addresses. We have mechanically typechecked the annotated version of the garbage collector using a type checker written in OCaml; after introducing the garbage collection algorithm, this section describes the main invariants that we used to annotate the collector so that it could be type checked. Unfortunately, the proof annotations are much larger than the code itself: about 1000 lines of proofs to establish lemmas for arithmetic, arrays, and equality, 1500 lines of lemmas for the object and heap invariants (e.g. the heap extension lemma), and 500 lines of annotations on the garbage collector instructions. A garbage collector is atypical TAL code, though, because it manipulates unusually complex invariants. Simpler TAL code requires much less annotation; for example, the mov instructions in the GcStart block of Figure 7 require only one line of annotation per instruction.

The garbage collector implements the well-known Cheney-queue algorithm [27], which copies live data from "from-space" to "to-space", using to-space as a work-queue to save space. To-space and from-space will each be contiguous ranges of memory, the former occupying addresses $TI \ldots TL - 1$ and the latter occupying addresses $FI \ldots FL - 1$. The program allocates objects in to-space, starting at TI and continuing until the allocation reaches the limit of to-space (TL), at which point the program starts a collection.

The garbage collection algorithm proceeds as follows:

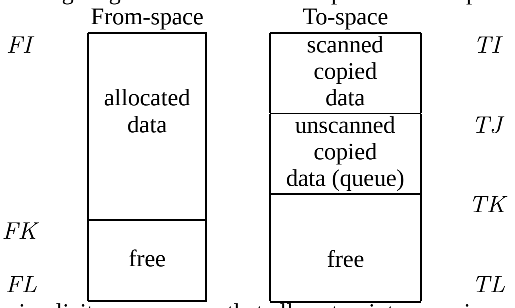
1. Assume that to-space contains objects in addresses $TI \ldots TK - 1$ (if $TK < TL$, then $TK \ldots TL - 1$ contains free memory). The algorithm will garbage collect these objects. Assume that register *Rti* holds TI and register *Rtk* holds TK. Assume that from-space contains free space in addresses $FI \ldots FL - 1$. Assume that register *Rfi* holds FI and register *Rfl* holds FL.
2. First, the algorithm swaps to-space and from-space: the old to-space, holding the objects, is now called from-space, and the old from-space is now called to-space. The program calls GcStart, which swaps the registers that describe from-space and to-space. GcStart also sets the *scan pointer* TJ and the *allocation pointer* TK to the beginning of the new to-space, and then returns to the program by jumping to a return address ("jr Rretg"). Each time the collector copies an object into to-space, it adds the size of the object to TK. In the remaining steps, "from-space" refers to the new from-space, and "to-space" refers to the new to-space.
3. For each register holding a pointer (each "root"), the program calls FwdPtr to copy the pointed-to object to to-space. FwdPtr overwrites the old from-space object's header with a *forwarding pointer* that points to the new copy; any subsequent calls to FwdPtr for this object return a pointer to the existing copy

rather than making additional copies. The newly copied object in to-space is a word-for-word replica of the original from-space object, which means that the pointers inside the copied object still point back to from-space.

4. The program calls GcLoop, which scans each object in the Cheney queue (i.e., it scans the objects at addresses $TJ \ldots TK - 1$). GcLoop repeats the following operation until the queue is empty (i.e., until $TJ = TK$):

   (a) For the object at address TJ, call FwdPtr on each pointer in the object. Overwrite the old (from-space) pointer values in the object with the forwarded pointers so that the object points to to-space.

The following diagram illustrates the heap's state in steps 3 and 4:



For simplicity, we assume that all root pointers are in registers. To support a stack, the collector would have to traverse the roots on the stack as well as roots in the registers; we expect stack traversal to be no harder than heap object traversal (with the possible exception of scanning saved callee-save registers). Stack scanning is outside the scope of this paper.

The type system imposes one inefficiency on the collector code: each allocation in to-space must check that there is enough free space for the new object. In theory, this check is unnecessary because to-space is the same size as from-space, and it therefore contains enough room for one copy of each from-space object. Proving this statically would require extending the invariants considerably, and we wanted to keep the invariants as simple as possible.

The garbage collector's two central operations, copying an object and scanning an object, depend on an object's layout, and the garbage collector code shown in Figure 7 must be polymorphic over all classes. Therefore, the garbage collector needs some runtime information about each object's layout. In general, this information may come in two forms. First, the information may consist of meta-data in memory, such as an integer specifying the size of an object, and a bit field specifying which fields of the objects are pointers. Second, the information may implicitly reside in per-class code that actually implements the copy and scan operations. To demonstrate that GTAL handles both these approaches, we choose the former approach for the copy operation and the latter for the scan operation.

The garbage collector reads information about each object from the object's method table, pointed to by the header word. The method tables reside in a static data area at addresses $DI \ldots DL - 1$, where DI and DL are constants. Each class C has its own method table, and each method table contains a size word, a scan function pointer, and zero or more method pointers:

SizeOf C
scan function
method 0
method 1
...

The Rep function from Section 6 uses $\mathit{GcHeader}$ to add the size word and scan function pointer to each method table:

$$
\begin{aligned}
&\mathit{GcHeader} = \lambda \mathit{ClassName}\!:\!\mathbb{C}.\lambda \mathit{Value}\!:\!\mathbb{N}.\\
&\qquad !(\mathit{StaticPtr}\ \mathit{Value}\ (\mathit{SInt}\ (\mathit{SizeOf}\ \mathit{ClassName})))\\
&\quad\ \otimes !(\mathit{StaticPtr}\ s(\mathit{Value})\ (\mathit{ScanCode}\ C))\\
&\mathit{SInt} = \lambda N\!:\!\mathbb{N}.\lambda \mathit{Value}\!:\!\mathbb{N}.\mathit{Eq}\ \mathit{Value}\ N\\
&\mathit{StaticPtr} = \lambda N\!:\!\mathbb{N}.\lambda \mathit{Inv}\!:\!\mathbb{N} \to \mathbb{T}.\\
&\qquad !(\mathit{Le}\ s(N)\ \mathit{DL})\\
&\quad\ \otimes !(D \multimap \mathit{MemF}\ N\ \mathit{Inv} \otimes (\mathit{MemF}\ N\ \mathit{Inv} \multimap D))\\
&\mathit{MemF} = \lambda N\!:\!\mathbb{N}.\lambda \mathit{Inv}\!:\!\mathbb{N} \to \mathbb{T}.\exists V\!:\!\mathbb{N}.\mathrm{Mem}\ N\ V \otimes !(\mathit{Inv}\ V)\\
&\mathit{RegF} = \lambda R\!:\!\mathbb{R}.\lambda \mathit{Inv}\!:\!\mathbb{N} \to \mathbb{T}.\exists V\!:\!\mathbb{N}.\mathrm{Reg}\ R\ V \otimes !(\mathit{Inv}\ V)\\
&D = \mathrm{rec}\ (\lambda D\!:\!\mathbb{T}.\mathrm{Mem}\ \mathit{DI}\ M_{\mathit{DI}} \otimes \ldots \otimes \mathrm{Mem}\ \mathit{DL} - 1\ M_{\mathit{DL}-1})
\end{aligned}
$$

The type is D a recursive, linear type that specifies the layout of the static data area (we omit the detailed definition of D here). Each $\mathit{StaticPtr}$ implements a nonlinear pointer into D, following the strategy in Section 6. We require that the static data area reside in a lower area of memory than from-space and to-space (i.e., $\mathit{DL} \le \mathit{FI}$ and $\mathit{DL} \le \mathit{TI}$). This enables the second instruction of FwdPtr to distinguish between a header word that points to a method table in D, and a header word that is a forwarding pointer to an object in to-space. The following definition captures the overall state of the heap, including from-space and to-space, D, and registers holding various heap addresses:

$$
\begin{aligned}
&\mathit{Gc} = \lambda \mathit{Sf}, \mathit{St}, \mathit{FI}, \mathit{FK}, \mathit{FL}, \mathit{TI}, \mathit{TJ}, \mathit{TK}, \mathit{TL}.\\
&\qquad \mathrm{Reg}\ \mathit{Rdi}\ \mathit{DI} \otimes \mathrm{Reg}\ \mathit{Rdl}\ \mathit{DL}\\
&\quad\ \otimes \mathrm{Reg}\ \mathit{Rfi}\ \mathit{FI} \otimes \mathrm{Reg}\ \mathit{Rfk}\ \mathit{FK} \otimes \mathrm{Reg}\ \mathit{Rfl}\ \mathit{FL}\\
&\quad\ \otimes \mathrm{Reg}\ \mathit{Rti}\ \mathit{TI} \otimes \mathrm{Reg}\ \mathit{Rtj}\ \mathit{TJ} \otimes \mathrm{Reg}\ \mathit{Rtk}\ \mathit{TK} \otimes \mathrm{Reg}\ \mathit{Rtl}\ \mathit{TL}\\
&\quad\ \otimes !(\mathit{Le}\ \mathit{DL}\ \mathit{FI}) \otimes !(\mathit{Le}\ \mathit{DL}\ \mathit{TI})\\
&\quad\ \otimes D\\
&\quad\ \otimes \mathit{FromSpace}\ \mathit{Sf}\ \mathit{St}\ \mathit{FI}\ \mathit{FK}\ \mathit{FL}\\
&\quad\ \otimes \mathit{ToSpace}\ \mathit{Sf}\ \mathit{St}\ \mathit{TI}\ \mathit{TJ}\ \mathit{TK}\ \mathit{TL}
\end{aligned}
$$

For brevity, we omit the "$\mathbb{N} \to \mathbb{T}$" kind annotations on the from- and to-space maps Sf and St, and the "$\mathbb{N}$" kind annotations on the heap addresses. The following type expresses the state of to-space:

$$
\begin{aligned}
&\mathit{ToSpace} = \lambda \mathit{Sf}, \mathit{St}, \mathit{TI}, \mathit{TJ}, \mathit{TK}, \mathit{TL}.\\
&\qquad !(\mathit{Le}\ \mathit{TI}\ \mathit{TJ}) \otimes !(\mathit{Le}\ \mathit{TJ}\ \mathit{TK}) \otimes !(\mathit{Le}\ \mathit{TK}\ \mathit{TL})\\
&\quad\ \otimes !(\forall N\!:\!\mathbb{N}.\mathit{Iff}\ \ !(\mathit{Le}\ 1\ (\mathit{St}\ N))\\
&\qquad\qquad\qquad\quad !(\,!(\mathit{Le}\ \mathit{TI}\ N) \otimes !(\mathit{Le}\ s(N)\ \mathit{TK})))\\
&\quad\ \otimes !(\mathit{Not}\ \ !(\mathit{Eq}\ (\mathit{St}\ \mathit{TI})\ 1))\\
&\quad\ \otimes \mathit{Arr}\ \mathit{TI}\ \mathit{TJ}\ (\lambda N\!:\!\mathbb{N}.\mathit{MemF}\ N\ (\mathit{RepS}\ \mathit{St}\ \mathit{St}\ N))\\
&\quad\ \otimes \mathit{Arr}\ \mathit{TJ}\ \mathit{TK}\ (\lambda N\!:\!\mathbb{N}.\mathit{MemF}\ N\ (\mathit{RepS}\ \mathit{St}\ \mathit{Sf}\ N))\\
&\quad\ \otimes \mathit{Arr}\ \mathit{TK}\ \mathit{TL}\ (\lambda N\!:\!\mathbb{N}.\exists V\!:\!\mathbb{N}.\mathit{Mem}\ N\ V)
\end{aligned}
$$

The $\mathit{ToSpace}$ type enforces these invariants:

- The to-space addresses are in order: $\mathit{TI} \le \mathit{TJ} \le \mathit{TK} \le \mathit{TL}$
- An address N holds allocated data ($\mathit{St}\ N > 0$) if and only if $\mathit{TI} \le N < \mathit{TK}$. Otherwise, N holds a free word ($\mathit{St}\ N = 0$).
- The first word of to-space may be free ($\mathit{St}\ \mathit{TI} = 0$) or contain the first word of a class object ($\mathit{St}\ \mathit{TI} = s(s(C))$), but it never contains an interior word of an object ($\mathit{St}\ \mathit{TI} \ne 1$).
- Each address $\mathit{TI} \le N < \mathit{TJ}$ holds data that points from to-space to to-space. This data conforms to the invariants specified in Rep and the to-space map St.
- Each address $\mathit{TJ} \le N < \mathit{TK}$ holds data that points from to-space to from-space. This data conforms to the invariants specified in Rep and the to-space map St.
- Each address $\mathit{TK} \le N < \mathit{TL}$ holds free memory.

The function $\mathit{RepS}\ \mathit{St}\ S\ N$ specifies the invariant that each word of to-space must satisfy:

$$\mathit{RepS} = \lambda S, S', N.\mathit{Rep}\ (\mathit{SType}\ S\ N)\ (\mathit{SOffset}\ S\ N)\ S'$$

RepS relies on auxiliary primitive recursive functions SType and $\mathit{SOffset}$, defined so that $(\mathit{SType}\ S\ N) = C$ if address N holds some word of some object of class C, and $(\mathit{SOffset}\ S\ N)$ equals the offset from the start of the word (where the header word is at offset 0). If address N is free, then $(\mathit{SType}\ S\ N) = 0$ and $(\mathit{SOffset}\ S\ N) = 0$.

The from-space definition is similar, except that each header word may contain a forwarding pointer $\mathit{ExactPtr}\ C\ \mathit{St}\ V$, rather than conforming to the invariants in Rep:

$$\begin{array}{l}
\mathit{FromSpace} = \lambda \mathit{Sf}, \mathit{St}, \mathit{FI}, \mathit{FK}, \mathit{FL}. \\
\qquad\quad !(\mathit{Le}\ \mathit{FI}\ \mathit{FK}) \otimes !(\mathit{Le}\ \mathit{FK}\ \mathit{FL}) \\
\qquad \otimes !(\forall N{:}\mathbb{N}.\mathit{Iff}\ \ !(\mathit{Le}\ 1\ (\mathit{St}\ N)) \\
\qquad\qquad\qquad\qquad\qquad !(\,!(\mathit{Le}\ \mathit{FI}\ N) \otimes !(\mathit{Le}\ s(N)\ \mathit{FK}))) \\
\qquad \otimes !(\mathit{Not}\ \ !(\mathit{Eq}\ (\mathit{Sf}\ \mathit{FI})\ 1)) \\
\qquad \otimes \mathit{Arr}\ \mathit{FI}\ \mathit{FK}\ (\lambda N{:}\mathbb{N}.\mathit{Union} \\
\qquad\qquad\qquad (\mathit{MemF}\ N\ (\mathit{RepS}\ \mathit{Sf}\ \mathit{Sf}\ N)) \\
\qquad\qquad\qquad (\mathit{MemF}\ N\ (\mathit{RepFwdS}\ \mathit{Sf}\ \mathit{St}\ N))) \\
\qquad \otimes \mathit{Arr}\ \mathit{FK}\ \mathit{FL}\ (\lambda N{:}\mathbb{N}.\exists V{:}\mathbb{N}.\text{Mem}\ N\ V) \\
\mathit{RepFwdS} = \lambda \mathit{Sf}, \mathit{St}, N. \\
\qquad \mathit{RepFwd}\ (\mathit{SType}\ \mathit{Sf}\ N)\ (\mathit{SOffset}\ \mathit{Sf}\ N)\ \mathit{Sf}\ \mathit{St} \\
\mathit{RepFwd} = \lambda C{:}\mathbb{C}.\lambda \mathit{Offset}{:}\mathbb{N}.\lambda \mathit{Sf}, \mathit{St}.\lambda V{:}\mathbb{N}. \\
\qquad \mathit{Case}\ \mathit{Offset} \\
\qquad\qquad (\mathit{ExactPtr}\ C\ \mathit{St}\ V) \\
\qquad\qquad (\lambda M{:}\mathbb{N}.\mathit{Rep}\ C\ \mathit{Offset}\ \mathit{Sf}\ V)
\end{array}$$

Using these definitions, the annotated garbage collector code defines preconditions for each block. The preconditions are long, so this section just highlights one of them: the precondition for the scan functions, parameterized over all possible $\mathit{ClassNames}$:

$$\begin{array}{l}
\mathit{ScanCode} = \lambda \mathit{ClassName}{:}\mathbb{C}.\lambda \mathit{Value}{:}\mathbb{N}. \\
\qquad \forall \mathit{Sf}, \mathit{St}, \mathit{St}', \mathit{FI}, \mathit{FK}, \mathit{FL}, \mathit{TI}, \mathit{TJ}, \mathit{TK}, \mathit{TL}.\text{Code}\ \mathit{Value}\ (\\
\qquad\qquad \mathit{Gc}\ \mathit{Sf}\ \mathit{St}\ \mathit{FI}\ \mathit{FK}\ \mathit{FL}\ \mathit{TI}\ \mathit{TJ}\ \mathit{TK}\ \mathit{TL} \\
\qquad\quad \otimes !(\mathit{ExactPtr}\ \mathit{ClassName}\ \mathit{St}'\ \mathit{TJ}) \\
\qquad\quad \otimes !(\mathit{ScanQueue}\ \mathit{St} \\
\qquad\qquad\qquad (\mathit{Add}\ \mathit{TJ}\ (\mathit{SizeOf}\ \mathit{ClassName}))\ \mathit{TK}) \\
\qquad\quad \otimes !(\mathit{SpaceExtend}\ \mathit{St}\ \mathit{St}') \\
\qquad\quad \otimes \mathit{RegJunk}(\mathit{Rptr}, \mathit{Rtmp1}, \mathit{Rtmp2}, \mathit{Rtmp3}, \mathit{Rtmp4}, \\
\qquad\qquad\qquad\quad \mathit{Rretc}) \\
\qquad\quad \otimes \mathit{RetG}\ \mathit{Sf}\ \mathit{St}) \\
\mathit{RegJunk} = \lambda R{:}\mathbb{R}.\exists V{:}\mathbb{N}.\text{Reg}\ R\ V \\
\mathit{RegJunk}(R_1 \ldots R_n) = \mathit{RegJunk}\ R_1 \otimes \ldots \otimes \mathit{RegJunk}\ R_n
\end{array}$$

Each scan function expects an exact pointer to class $\mathit{ClassName}$ (for example, Figure 8's ScanLink expects a pointer to Link). This allows the scan function to load and store fields of the class. When the scan function finishes, it jumps back to GcLoop. The scan function only scans the first object currently in the queue; an invariant "$\mathit{ScanQueue}$" (omitted here) provides an exact pointer for each remaining object in the queue.

As the collector allocates new objects in to-space, it extends the to-space space map. By the time the collector finishes, the final space map St' differs from the initial St. Any pointers that still refer to St would be useless unless they could be updated to refer to St'. This is the "heap extension lemma" described in Section 6, and the garbage collector invariants maintain a relation between the original St and the current St':

$$\begin{array}{l}
\mathit{SpaceExtend} = \lambda S{:}\mathbb{N} \to \mathbb{C}.\lambda S'{:}\mathbb{N} \to \mathbb{C}. \\
\qquad \forall N{:}\mathbb{N}.\,!(\mathit{Le}\ 1\ (S\ N)) \multimap !(\mathit{Eq}\ (S\ N)\ (S\ N'))
\end{array}$$

This relation ensures that St' extends St by changing free ($\mathit{St}\ N = 0$) entries of the map to allocated (non-zero) entries, but St' does not alter any entries in St that were already allocated. From this, the annotated garbage collector proves that data representation valid under St remains valid under St':

$$\begin{array}{l}
\forall S, S', C, \mathit{Offset}, V.\,!(\mathit{SpaceExtend}\ S\ S') \multimap \\
\qquad !(\mathit{Rep}\ C\ \mathit{Offset}\ S\ V) \multimap !(\mathit{Rep}\ C\ \mathit{Offset}\ S'\ V)
\end{array}$$

The proof uses the $\bigcirc\!<<$ operator to step inside the recursive type $\mathit{Rep} = \text{rec}\ (\ldots \tau_{\mathit{Rep}})$, and coerce τ_{Rep}'s data from S to S'. Based on the extension lemma for Rep, the annotated garbage collector proves extension lemmas for from-space and to-space.

When GcLoop finishes, it jumps to a return address specified by RetG (omitted here). At this point $\mathit{TJ} = \mathit{TK}$, so all allocated data in to-space points to to-space, not to from-space, and the data in from-space is recycled into free space. (This collector is a stop-the-world collector; if the collector were incremental, the program would need a run-time check, or *barrier* [27] to distinguish between the to-space-to-to-space pointers in $\mathit{TI} \le N < \mathit{TJ}$ and the to-space-to-from-space pointers in $\mathit{TJ} \le N < \mathit{TK}$.)

8. Conclusions

We've presented a type system that gives programmers powerful, primitive abstractions with which to implement type-safe memory management. GTAL differs from traditional TAL systems by encoding objects and classes, rather than including them directly in the type system. This requires some initial effort to build up the appropriate abstractions (as the long series of definitions in Sections 6 and 7 illustrate), but offers complete flexibility over memory layout and memory management. Yet, it stops short of requiring a complete mechanical formalization of a typed assembly language's soundness, as in foundational PCC approaches. Furthermore, there's no need for a separate mechanical proof that the garbage collector and GTAL interact safely — the garbage collector *is* GTAL code, and is verified safe by the GTAL type checker.

In contrast to program-logic-based assembly languages like XCAP [19], GTAL does not require extra effort to support embedded code pointers, such as the pointers to the scan functions in the method tables; "Code" types are first-class types. To deal with genuinely "circular specifications" [19], such as the circular contract between the heap type (which specifies that header words point to scan functions) and scan functions (which take the heap as an argument), GTAL supports recursive types, and provides lightweight operators "$\bigcirc$" and "$\bigcirc <<$" to manipulate recursive types within GTAL's logic.

Compared to provably correct GC [2], our collector's invariants are weaker in one dimension (proving safety, not correctness), but stronger in another dimension (type-checking the program and garbage collector together ensures their safe interaction). For protection in a language-based operating system, safety is the first concern. Nevertheless, it would be nice to get the best of both approaches; since GTAL contains a powerful logic, it may be possible to extend the garbage collector's invariants to capture the correctness criteria (graph isomorphism) and thereby simultaneously prove partial correctness and type-safe program interaction.

For simplicity, GTAL's logic contains only a single inductive type, natural numbers. We plan to explore general inductive types, as used in [21], to make our encodings more elegant and to enable more advanced heap types, such as parameterized classes and existentially quantified closures; this would enable interaction with polymorphic functional languages like ML and System F, in addition to the class-based languages currently supported.

References

[1] Andrew W. Appel, Paul-Andre Mellies, Christopher D. Richards, and Jerome Vouillon. A very modal model of a modern, major, general type system. In *Proceedings of the 34th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 2007.

[2] L. Birkedal, N. Torp-Smith, and J. Reynolds. Local reasoning about a copying garbage collector. In *Symposium on Principles of programming languages*, 2004.

[3] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 197–245. The MIT Press, Cambridge, MA, 1994.

[4] Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *Conference on Verification, Model Checking, and Abstract Interpretation*, 2006.

[5] J. Chen, D. Wu, A. Appel, and H. Fang. A provably sound TAL for back-end optimization, 2003.

[6] Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–49, New York, NY, USA, 2005. ACM Press.

[7] Karl Crary. Typed compilation of inclusive subtyping. In *2000 ACM SIGPLAN International Conference on Functional Programming*, pages 68–81, 2000.

[8] 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.

[9] Karl Crary and Stephanie Weirich. Flexible type analysis. In *International Conference on Functional Programming*, pages 233–248, 1999.

[10] Chris Hawblitzel. http://research.microsoft.com/˜ chrishaw/.

[11] Chris Hawblitzel, Edward Wei, Heng Huang, Eric Krupski, and Lea Wittie. Low-level linear memory management. In *Workshop on Semantics, Program Analysis, and Computing Environments For Memory Management*, 2004.

[12] G. Huet, G. Kahn, and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial, July 1999.

[13] Limin Jia, Frances Spalding, David Walker, and Neal Glew. Certifying compilation for a language with stack allocation. In *Logic in Computer Science, 2005*, 2005.

[14] 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.

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

[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 21, pages 527–568. ACM Press, 1999.

[17] Hiroshi Nakano. A modality for recursion. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, 2000.

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

[19] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 320–333, New York, NY, USA, 2006. ACM Press.

[20] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

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

[22] Alex K. Simpson. The proof theory and semantics of intuitionistic modal logic, phd thesis, department of philosophy, university of edinburgh, 1994.

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

[24] P. L. Wadler. A taste of linear logic. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science, Gdánsk*, New York, NY, 1993. Springer-Verlag.

[25] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

[26] 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.

[27] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[28] 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.

[29] Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proc. 2003 European Symposium on Programming (ESOP'03)*, April 2003.

[30] Dengping Zhu and Hongwei Xi. Safe programming with pointers through stateful views. In *Practical Aspects of Declarative Languages*, 2005.

A. Polymorphism, subclasses, and methods

Sections 6 and 7 demonstrated the type system's support for classes, and described garbage collector code that is polymorphic over classes. Section 7's garbage collector also supports objects with subclassing and inheritance; we have deferred the discussion of subclassing and inheritance to this appendix because they are almost entirely orthogonal to the garbage collector code and the garbage collector invariants. This appendix demonstrates that given the invariants from Section 7, accommodating object-oriented data representations in the garbage collector is straightforward.

Following LIL$_C$ [6], the object orientation centers around the idea of an *approximate representation*, defined by a recursive type function $\mathit{ApproxRep}\ C\ A$:

$$\begin{array}{l}
\mathit{ApproxRep} = \\
\ \text{rec}\ (\lambda \mathit{ApproxRep}:\mathbb{C} \to \mathbb{C} \to \mathbb{N} \to (\mathbb{N} \to \mathbb{C}) \to \mathbb{N} \to \mathbb{T}. \\
\ \ \lambda \mathit{ClassName}:\mathbb{C}. \\
\ \ \lambda A:\mathbb{C}. \\
\ \ \lambda \mathit{Offset}:\mathbb{N}. \\
\ \ \lambda \mathit{SpaceMap}:\mathbb{N} \to \mathbb{C}. \\
\ \ \lambda \mathit{Value}:\mathbb{N}. \\
\ \ \mathit{Case}\ \mathit{Offset}\ (\mathit{Prims}\ \mathit{ClassName}\ 0\ A\ \mathit{Value} \otimes \\
\qquad\qquad \mathit{GcHeader}\ A\ \mathit{Value})\ (\lambda M:\mathbb{N}. \\
\ \ \ \mathit{Case}\ (\mathit{Ptrs}\ \mathit{ClassName}\ \mathit{Offset}) \\
\ \ \ (\mathit{Prims}\ \mathit{ClassName}\ \mathit{Offset}\ A\ \mathit{Value}) \\
\ \ \ (\lambda \mathit{ClassName}':\mathbb{C}.\mathit{Ptr}\ \mathit{ClassName}'\ \mathit{SpaceMap}\ \mathit{Value}))))
\end{array}$$

$\mathit{ApproxRep}$ differs from Rep in several ways: it takes an extra parameter A, and passes A to Prims and $\mathit{GcHeader}$, and it replaces $\mathit{ExactPtr}$ with Ptr (defined below). The original "exact" representation function is now just an abbreviation for $\mathit{ApproxRep}$:

$$\mathit{Rep}\ C = \mathit{ApproxRep}\ C\ C$$

Other definitions are unchanged; for example, the $\mathit{ToSpace}$ and $\mathit{FromSpace}$ definitions continue to refer to Rep as before.

The notation $\mathit{Rep}\ C$ and $\mathit{ApproxRep}\ C\ A$ correspond to LIL$_C$'s $R(C)$ and $\mathit{ApproxR}(A, C)$. Intuitively, $\mathit{Rep}\ C$ describes exactly class C (not a subclass of C), while $ApproxRep\ C\ A$ describes some portion of some subclass A of C. More precisely, $ApproxRep\ C\ A$ describes the fields and methods of A that both A and C share. A pointer to C may actually point to any subclass A of C:

$$\begin{aligned}&Ptr = \lambda C\!:\!\mathbb{C}.\lambda S\!:\!\mathbb{N}\rightarrow\mathbb{C}.\lambda V\!:\!\mathbb{N}.\exists A\!:\!\mathbb{C}.\\ &\qquad !(Subclass\ A\ C)\otimes !(ExactPtr\ A\ S\ V)\end{aligned}$$

The program can use a "$Ptr\ C\ S\ V$" value to access a field or method of the pointed-to object, but these accesses require knowing something about the subclass A's fields and methods — A needs to support all the fields and methods supported by C (although A may have additional fields and methods, and have overridden implementations of C's methods). This knowledge comes from the $Subclass\ C_1\ C_2$ type, which specifies that if C_1 is a subclass of C_2 ("$C_1 \ll C_2$" in LIL$_C$'s notation), then C_1 supports C_2's methods and fields:

$$\begin{aligned}&Subclass = \lambda C_1\!:\!\mathbb{C}.\lambda C_2\!:\!\mathbb{C}.\forall A\!:\!\mathbb{C}.\forall S\!:\!\mathbb{N}\rightarrow\mathbb{C}.\forall V\!:\!\mathbb{N}.\\ &\qquad\quad !(Le\ (SizeOf\ C_2)\ (SizeOf\ C_1))\\ &\qquad \otimes !(ApproxRep\ C_1\ A\ 0\ S\ V \multimap ApproxRep\ C_2\ A\ 0\ S\ V)\\ &\qquad \otimes !(Arr\ 1\ (SizeOf\ C_2)\ (\lambda Offset\!:\!\mathbb{N}.\\ &\qquad\qquad !(Eq\ (ApproxRep\ C_1\ A\ Offset\ S\ V)\\ &\qquad\qquad\qquad (ApproxRep\ C_2\ A\ Offset\ S\ V))))\end{aligned}$$

Since the fields are immutable, C_1 and C_2 must agree exactly on the types of the fields shared between them (hence the "Eq" in the $Subclass$'s definition). On the other hand, the header word is immutable, and uses covariant subtyping: C_1's header word must be a subtype of C_2's header word, where following [3] and [7], the function type $\multimap$ encodes subtyping.

We can now extend Section 6.1's definition of a class to include explicit method declarations:

$$\begin{aligned}&class = method_1\ \ldots method_m\ field_1\ \ldots\ field_n\\ &field = n_{class}\mid\tau_{primitive}\\ &method = \forall X_1\ldots X_k.(type)\rightarrow type\\ &type = n_{class}\mid\tau_{primitive}\mid X\end{aligned}$$

For simplicity, each method takes one argument (plus an implicit "this" pointer) and returns one result. Each class C defines its method table (specified by $Prims\ C\ 0$, as discussed in Section 6.1) to hold a pointer to $method_1$ at offset 2 in the method table, a pointer to $method_2$ at offset 3, and so on (offsets 0 and 1 are still used for C's size and scan function):

$$\begin{aligned}&\llbracket\forall\overrightarrow{X}.(type_1)\rightarrow type_2\rrbracket = \lambda A\!:\!\mathbb{C}.\lambda V\!:\!\mathbb{N}.\overrightarrow{\forall X\!:\!\mathbb{C}}.\\ &\quad \forall K\!:\!\mathbb{N}.\forall Z\!:\!\mathbb{T}.\forall Sf, St, FI, FK, FL, TI, TK, TL.!(Code'\ V\\ &\qquad\quad Gc\ Sf\ St\ FI\ FK\ FL\ TI\ TK\ TK\ TL\\ &\qquad \otimes RegJunk(Rtmp1\ \ldots Rtmp4)\\ &\qquad \otimes RegF\ Rthis\ (Ptr\ A\ St)\\ &\qquad \otimes RegF\ Rarg\ (\llbracket type_1\rrbracket\ St)\\ &\qquad \otimes \text{Reg}\ Rret\ K\\ &\qquad \otimes Z\\ &\qquad \otimes\forall Sf, St, FI, FK, FL, TI, TK, TL.!(Code'\ K\\ &\qquad\qquad\quad Gc\ Sf\ St\ FI\ FK\ FL\ TI\ TK\ TK\ TL\\ &\qquad\qquad \otimes RegJunk(Rtmp1 ... Rtmp4, Rthis, Rret)\\ &\qquad\qquad \otimes RegF\ Rarg\ (\llbracket type_2\rrbracket\ St)))\end{aligned}$$

$$\begin{aligned}&Code' = \lambda N\!:\!\mathbb{N}.\lambda P\!:\!\mathbb{T}.\exists P'\!:\!\mathbb{T}.!(P\multimap P')\multimap \text{Code}\ N\ P'\\ &\llbracket n_C\rrbracket = Ptr\ n_C\\ &\llbracket X\rrbracket = Ptr\ X\\ &\llbracket\tau_{primitive}\rrbracket = \lambda S\!:\!\mathbb{N}\rightarrow\mathbb{C}.\tau_{primitive}\\ &Prims\ C\ 0\ A\ V =\\ &\qquad\quad !(StaticPtr\ s(s(V))\ (\llbracket method_1\rrbracket\ A))\\ &\qquad \otimes !(StaticPtr\ s(s(s(V)))\ (\llbracket method_2\rrbracket\ A))\\ &\qquad\qquad \ldots\end{aligned}$$

Notice that the definition of $\llbracket method\rrbracket$ is independent of the class C that contains the method. In particular, as in LIL$_C$, the "this" pointer has type $Ptr\ A\ St$, where A is a parameter to $\llbracket method\rrbracket$, not type $Ptr\ C\ St$. This makes subclassing straightforward. Suppose that C_2 has methods $method_1 \ldots method_m$ and C_1 declares the same methods, followed by additional methods: $method_1, \ldots, method_m, method_{m+1}, \ldots$. Because the translation of $method_k$ for $1 \leq k \leq m$ is the same for C_1 and C_2, the definition of $Prims\ C_1\ 0$ and $Prims\ C_2\ 0$ proves $Subclass$'s criteria relating C_1's header word to C_2's header word (the proof uses $\bigcirc\!\!<<$ to step inside the $ApproxRep$ recursive type):

$$!(ApproxRep\ C_1\ A\ 0\ S\ V \multimap ApproxRep\ C_2\ A\ 0\ S\ V)$$

Even though C_1 and C_2 declare the same method types, they may implement their methods differently. In other words, C_1 may override C_2's method implementations. Suppose the program wants to allocate a C_1 object. To do this, the program constructs values of types $Rep\ C_1\ 0\ St\ V_0$ ... $Rep\ C_1\ (n-1)\ St\ V_{n-1}$, where $n = (SizeOf\ C_1)$. To construct the $Rep\ C_1\ 0\ St\ V_0$ value, it chooses V_0 to be the address of C_1's method table. Notice that since $Rep\ C_1 = ApproxRep\ C_1\ C_1$, the program instantiates the "A" parameter to the method table with C_1, which means that each method in C_1's method table accepts a "this" pointer of type $Ptr\ C_1\ St$, and therefore has access to C_1's fields and methods.

Rather than override all of C_2's method implementations, C_1 may choose to inherit one or more of C_2's method implementations. (C_1 could make its own copies of C_2's implementations, but it's more efficient to inherit the actual code from C_2 than to duplicate the code.) Each of C_2's method implementations has type $\llbracket\forall\overrightarrow{X}.(type_1)\rightarrow type_2\rrbracket\ C_2$, which is not the same as $\llbracket\forall\overrightarrow{X}.(type_1)\rightarrow type_2\rrbracket\ C_1$. Luckily, the type system can encode the subtyping relation between these two types, and this relation allows the method table to use a $\llbracket\forall\overrightarrow{X}.(type_1)\rightarrow type_2\rrbracket\ C_2\ V$ value where a $\llbracket\forall\overrightarrow{X}.(type_1)\rightarrow type_2\rrbracket\ C_1\ V$ is expected:

$$\begin{aligned}&\forall V\!:\!\mathbb{N}.!(\llbracket\forall\overrightarrow{X}.(type_1)\rightarrow type_2\rrbracket\ C_2\ V)\multimap\\ &\qquad !(\llbracket\forall\overrightarrow{X}.(type_1)\rightarrow type_2\rrbracket\ C_1\ V)\end{aligned}$$

This relation expresses the intuition that if a method implementation expects a $Ptr\ C_2$ as an argument, then the same implementation is happy to receive a subclass $Ptr\ C_1$ instead. Deriving this relation relies on $Code'\ N\ \tau$ being contravariant in τ, which is why $\llbracket\forall\overrightarrow{X}.(type_1)\rightarrow type_2\rrbracket$ uses "$Code'$" rather than "Code":

$$\begin{aligned}&\forall P\!:\!\mathbb{T}.\forall Q\!:\!\mathbb{T}.\forall N\!:\!\mathbb{N}.!(Q\multimap P)\multimap\\ &\qquad\qquad !(Code'\ N\ P)\multimap !(Code'\ N\ Q)\end{aligned}$$

Crary [7] points out the difficulty of encoding a general subtyping relation rec $\tau\ \leq$ rec τ' for recursive types. In GTAL's logic, a naive attempt to prove rec $\tau \multimap$ rec τ' might try to unroll rec τ infinitely many times, but the termination lemma shows that is not possible. Luckily, LIL$_C$ encodes a fully-featured object-oriented language without relying on a general rec $\tau\ \leq$ rec τ' relation. The closest that we come to rec $\tau \multimap$ rec τ' is the coercion from $ApproxRep\ C_1$ to $ApproxRep\ C_2$. In this coercion, though, the recursive type $ApproxRep$ doesn't change — only the argument to the recursive type changes (from C_1 to C_2), and implementing this change only requires a shallow traversal of the $ApproxRep$ type, not an infinite unrolling.