Thesis Proposal
Reasoning about garbage collection in linear logic

Aleksey Kliger

April 24, 2008

Abstract

In a certified code system, a code producer wishes to convince a code consumer that her program is safe to execute. To that end, the producer provides to the consumer a certificate — a machine-checkable proof — of safety of her program in some formal language. The code consumer no longer has to trust the code producer, instead, the consumer only needs to trust the implementer of the certificate checker and the runtime facilities. Typically, a certified code system does not provide a way for the code producer to manually allocate and deallocate memory. Instead, a garbage collector is provided as part of the trusted computing base, without proof. Recently, formal proofs of correctness for the Cheney copying collector have been tackled in separation and linear logic. Expanding on this work, I propose to reason about garbage collection in a dependent, higher order, linear logic with few unusual connectives and in the setting of a machine model that is closer to real machine code and to produce machine-verifiable proofs of safety for garbage collector implementations. As a concrete example, I propose to prove the safety of an implementation of reference counting garbage collection.

Contents

1 Introduction

In a certified code system, a code producer wishes to convince a code consumer that her program is safe to execute. To that end, the producer provides to the consumer a certificate — a machine-checkable proof — of safety of her program in some formal language. Examples of certified code systems are Touchstone[Nec97, NL98], TAL[MWCG98], FPCC[AF00], and TALT[Cra03]. The ConCert project[CCD+02] uses proof-carrying code to distribute work units to users willing to donate their idle processor time.

In each certified code system, the code consumer no longer has to trust the code producer. Instead, the consumer only needs to trust the implementer of the certificate checker and the runtime facilities. One shortcoming of some systems is that the consumer has to trust that the formal system underlying the proofs generated by the code producer are sound. The FPCC and TALT systems try to overcome this problem by expressing the soundness of the policy itself in the well-studied formal system Twelf[PS99].

However even with FPCC and TALT, the trusted computing base must still include certain runtime facilities provided to the code producer's program. For example, calls OS kernel facilities (such as I/O) are asserted to be safe without verifying their implementation. Other runtime facilities may be difficult to implement within the type discipline imposed on the code producer.

Of the latter sort of trusted runtime facility, the most notable is the garbage collector. Typically a certified code system does not provide a way for the code producer to manually allocate and deallocate memory. Instead a garbage collector is treated as a black box (that is, it's safety is asserted in the proof system), and all memory management happens outside of the code consumer's control.

There are several shortcomings to this situation. First, a garbage collector is a large and sophisticated piece of code that interfaces with every program that runs on a certified code system. Bugs in a garbage collector may be difficult to spot and may undermine the safety of the whole system. If the purpose of certified code is to assure the code consumer that a program is safe to execute, it would make sense to provide such verification especially for the complex parts of a program. Second is the issue of flexibility. Programs have varying memory allocation needs, and it would make sense to offer programmers the opportunity to tailor memory management to their own situation. With a garbage collector as part of the trusted computing base, no such customization is possible. So there are reasons to move the collector out of the trusted computing base. Nonetheless, there are reasons why this is not easily done.

Automatic memory management is implemented as part of the trusted computing base because certified code systems are usually based on some variant of intuitionistic logic — a logic of truth, with underlying judgment A true. The truth judgment admits the principles of weakening and contraction: hypotheses A true may be used either not at all or multiple times in the course of a proof, and there is a general expectation that once some proposition is true, it does not "go away." This is precisely the cause of difficulties for reasoning about memory management: a hypothesis that a certain memory location contains a particular value is invalidated when the memory cell is deallocated.

What is needed is a logic without contraction and weakening; one whose judgments are more ephemeral than truth. Linear logic[Gir87, CCP03] is precisely such a logic. Its principal judgment is one of resources A res and their creation and consumption. Unlike truth, a resource must be used exactly once in the course of a proof: it cannot in general be discarded

or duplicated. As a result, the consumption of resources may be used to model state. If every byte of memory is a resource $\mathsf{at}\ m\, n$ (meaning that at address m is the value n), then writing a new value n' to that same address would consume the old resource and produce a new resource $\mathsf{at}\ m\, n'$. When a garbage collector deallocates memory, it consumes the resources corresponding to the freed memory, making them unavailable to user code for further access.

Garbage collection specifications, algorithms and implementations I propose to use linear logic to reason about a garbage collector implementation. To be precise about what I set out to do, it is useful to distinguish between three levels of abstraction for specifying memory management behavior: GC specifications, GC algorithms and GC implementations.

A high-level operational semantics of a programming language models the heap as a partial map from some set of locations to heap values. Operations such as allocation, memory reads and memory writes are specified in terms of this partial map in the small-step operational semantics of the programming language. Garbage in the heap can be specified abstractly[MFH95] as those locations that cannot influence the final result of a program. A garbage collection specification is an operational step that removes some of the garbage from the heap.

The abstract specification of a garbage collection is highly non-deterministic step of the operational semantics. It does not nail down details such as when garbage collection happens, or in what order garbage is collected. It is a tool for showing that a program with some of the garbage removed from the heap produces the same final result as one where there is no garbage collection at all. A GC algorithm describes a particular strategy for selecting garbage to collect. It picks out a particular subset of the possible executions of of the operational semantics of a language.

Finally a garbage collector implementation gives concrete code that implements the GC algorithm. An object language is compiled to machine instructions, and the small-step operational semantics of the high level object language correspond to several low-level instruction steps of the compiled code. The GC algorithm's steps, in particular, are compiled to calls into a concrete implementation of a garbage collector.

In support of my thesis, I propose to provide a machine-verifiable proof of safety for a particular implementation of a reference counting GC algorithm. The key theorem is to show that after a sequence of low-level instructions that correspond to a single high-level operation of the object language finish

executing, we can relate the concrete machine model to a state of the high-level operational semantics, even if the low-level implementation called the garbage collector. That is, I will show that the low-level implementation is safe with respect to the high-level semantics.

2 Reasoning about machine code in linear logic

In this section, I will introduce LSL informally, by showing how to specify various aspects of machine code: memory and registers, the decoding of instructions, and the specification of the execution behavior of the machine. Then I will present notation for partial correctness specifications for code sequences as a slightly higher-level abstraction over individual instructions. Finally, I will present call stacks and a C-like calling convention. I defer a formal presentation of LSL until Section 3 (and in particular, the connection between the logic and the operational semantics of the underlying machine model until Section 3.4), and a discussion of my object language, RCTAL, until Section 4.

2.1 Machine model

The machine model for LSL is a byte-addressed architecture with some fixed number of word-sized[1] general purpose registers, and a program counter register that points at an address in memory where the next instruction resides. At each step, the machine decodes some number of bytes starting at the program counter into an instruction, and then executes that instruction. The machine becomes stuck if the program counter points to some bytes that do not correspond to an instruction (that is, the decoding function is partial), or if the machine tries to read from or write to the byte at address zero.

Region resources Conceptually, we have the basic resource,

$$\texttt{mat} : \texttt{nat} \to \texttt{nat} \to \texttt{type} \stackrel{\text{def}}{=} \texttt{at}\,\texttt{memrgn}$$

A resource of type $\texttt{mat}\,m\,n$ says that memory address m holds the value n. It turns out however, that it is often useful to prove propositions that hold true about the contents of some piece of memory and about the contents of a register. As a result, we use *regions* to parametrize a more basic atomic resource $\texttt{at}\,r\,m\,n$, which says that address m of region r holds the byte n.

[1] For concreteness we take $\texttt{wordSize} = 4$, but this is somewhat arbitrary.

In the case of memory, there is a single region. And each register is in itself a region:

$$\begin{aligned}
\mathtt{reg} &\stackrel{\text{def}}{=} \mathsf{nat} \\
\mathtt{rgn} &\stackrel{\text{def}}{=} \mathtt{reg} \oplus 1 \\
\mathtt{regrgn}\, r &\stackrel{\text{def}}{=} \mathsf{inl}\ r \\
\mathtt{memrgn} &\stackrel{\text{def}}{=} \mathsf{inr}\ \star
\end{aligned}$$

Instruction encoding resources Next, we have a proposition that says that memory bytes m through $n-1$ hold instruction i:

$$\begin{aligned}
\mathtt{codeat} m\, n\, i = \Sigma a{:}\mathtt{array}.\ !(\mathtt{encodes}\, m\, i\, a) \\
\otimes\ (m + (\mathtt{alen}\, a) =_{\mathsf{nat}} n) \\
\otimes\ \mathtt{placeArray}\, a
\end{aligned}$$

An `array` is a list of pairs of natural numbers (i.e., an address and a byte value), and `encodes` is an unspecified disjoint sum that gives associates particular arrays with some instructions. For example, on the x86, one clause might be:

$$(a =_{\mathtt{array}} [\langle m, \mathsf{0x80}\rangle]) \otimes (i =_{\mathtt{instr}} \mathtt{mov}\,(\mathtt{rdest}\,\mathtt{r1}), (\mathtt{rco}\,\mathtt{r1}))$$

which says that if the array a is a singleton list of the address m and the byte `0x80`, then it encodes a `mov` instruction. The only thing we assume about `encodes` is that it requires the array a to be contiguous starting at address m. Although it is reasonable to also require it to be injective from instructions to bytes, this fact is actually not used anywhere. The resource `placeArray` is defined inductively as a multiplicative conjunction of `mat` resources, one for each byte of the array.

Operands and Destinations The arguments to an instruction are some number of operands, and at most one destination. Operands are the inputs and the destination is an output. The resources corresponding to operands must be present for the instruction to make sense, but they are not consumed, only consulted to get the values of the operands. The resources for the destination are consumed and then new resources are produced with an updated value.

The operands are either immediate values, the contents of a register, or the contents of some memory offset from a base address that is itself some operand:

$$\begin{aligned}
\mathtt{opnd} &\stackrel{\text{def}}{=} \mu\alpha{:}\mathsf{type}.\mathsf{nat} \oplus \mathtt{reg} \oplus \alpha \otimes \mathsf{nat} \\
\mathtt{imco}\, n &\stackrel{\text{def}}{=} \mathsf{roll}\ (\mathsf{inl}\ n) \\
\mathtt{rco}\, r &\stackrel{\text{def}}{=} \mathsf{roll}\ (\mathsf{inr}\ (\mathsf{inl}\ r)) \\
\mathtt{mco}\, o\, n &\stackrel{\text{def}}{=} \mathsf{roll}\ (\mathsf{inr}\ (\mathsf{inr}\ (o \otimes n)))
\end{aligned}$$

The destinations are either a register, or a memory address specified as an offset from an operand:

$$\begin{aligned}
\mathtt{dest} &\stackrel{\text{def}}{=} \mathtt{reg} \oplus \mathtt{opnd} \otimes \mathsf{nat} \\
\mathtt{rdest}\, r &\stackrel{\text{def}}{=} \mathsf{inl}\ r \\
\mathtt{mdest}\, o\, n &\stackrel{\text{def}}{=} \mathsf{inr}\ (o \otimes n)
\end{aligned}$$

The resources of an operand o that has value n are described by an inductively defined type family. For an immediate operand, there are no resources. For a register operand, the resources specify that the register contains the corresponding value. For a memory operand, the resources are specified inductively as those sufficient to establish that the base operand contains some address and those that specify the memory contents at the corresponding offset.

$$\begin{aligned}
\mathtt{specImco}\, n'\, n &\stackrel{\text{def}}{=} n' =_{\mathsf{nat}} n \\
\mathtt{specRco}\, r\, n &\stackrel{\text{def}}{=} \mathtt{rwat}\, r\, n \\
\mathtt{specMco}\, o\, n'\, n\, \alpha &\stackrel{\text{def}}{=} \Sigma \mathit{base}{:}\mathsf{nat}.\alpha\, o\, \mathit{base} \otimes \mathtt{mwat}\ (\mathit{base} +_{\mathsf{nat}} n')\, n \\
\mathtt{specOpnd}\, o\, n &\stackrel{\text{def}}{=} \mu\alpha{:}\mathtt{opnd} \to \mathsf{nat} \to \mathsf{type}. \\
&\qquad (\Sigma n'{:}\mathsf{nat}.o =_{\mathtt{opnd}} \mathtt{imco}\, n' \otimes \mathtt{specImco}\, n'\, n) \\
&\qquad \oplus (\Sigma r{:}\mathtt{reg}.o =_{\mathtt{opnd}} \mathtt{rco}\, r \otimes \mathtt{specRco}\, r\, n) \\
&\qquad \oplus (\Sigma o'{:}\mathtt{opnd}.\Sigma n'{:}\mathsf{nat}.o =_{\mathtt{opnd}} \mathtt{mco}\, o'\, n' \otimes \mathtt{specMco}\, o'\, n'\, n\, \alpha)
\end{aligned}$$

Destinations are specified in continuation passing style: the specification consumes the resources corresponding to the old contents of the destination and introduces new resources into the continuation.

$$
\begin{aligned}
\texttt{specRdest}\, r\, old\, new\, \kappa &= \texttt{rwat}\, r\, old \otimes (\texttt{rwat}\, r\, new \multimap \kappa) \\
\texttt{specMdest}'\, a\, old\, new\, \kappa &= \texttt{mwat}\, a\, old \otimes (\texttt{mwat}\, a\, new \multimap \kappa) \\
\texttt{specMdest}\, o\, n\, old\, new\, \kappa &= \Sigma base{:}\mathsf{nat}.(\texttt{specOpnd}\, o\, base \otimes \top) \\
&\qquad \& \texttt{specMdest}'\, (base + n)\, old\, new\, \kappa \\
\texttt{specDest}\, d\, old\, new\, \kappa &= (\Sigma r{:}\texttt{reg}.d =_{\texttt{dest}} \texttt{rdest}\, r \\
&\qquad \otimes \texttt{specRdest}\, r\, old\, new\, \kappa) \\
&\quad \oplus\ (\Sigma o{:}\texttt{opnd}.\Sigma n{:}\mathsf{nat}.d =_{\texttt{dest}} \texttt{mdest}\, o\, n \\
&\qquad \otimes \texttt{specMdest}\, o\, n\, old\, new\, \kappa)
\end{aligned}
$$

Of course in a typical instruction set, not every operation can take any combination of operands and destinations: some instruction sets only allow only one operand or destination to refer to memory; most modern architectures do not allow double memory references in a single operand. So the framework here is more general and does not rule out such instructions. We delegate responsibility to the `codeat` family for such architectures to rule out any inhabitants with such exotic operands.

Executable instructions To show that an address m has an executable instruction, we have a new atomic proposition, $\mathsf{executable}\, m$. There are two constants of this type, one for instructions where control flow simply continues to the next instruction, and one where control flow is unconditionally transferred elsewhere:

$$
\begin{aligned}
\mathsf{execStep}\, m\, n\, i :\ & (\texttt{codeat}\, i\, m\, n \otimes \top) \\
& \& \texttt{specStep}\, i\ (\Sigma k{:}\mathsf{nat}.\circ_k \mathsf{executable}\ n) \\
& \multimap \mathsf{executable}\, m. \\
\mathsf{execNoStep}\, m\, n\, i :\ & (\texttt{codeat}\, i\, m\, n \otimes \top)\ \& \texttt{specNoStep}\, i \\
& \multimap \mathsf{executable}\, m.
\end{aligned}
$$

Each of these constants are functions in continuation passing style with result $\mathsf{executable}\, m$, that consumes the resources for the precondition of the instruction i and introduces into the continuation the new resources available after the instruction executes. In the case of ordinary control flow, the continuation result must be an $\mathsf{executable}\, n$, if control flow is transferred elsewhere to address j then the continuation has the appropriate $\mathsf{executable}\, j$. The `specStep` and `specNoStep` families specify the resources consumed by each instruction and introduced into the continuation.

For example, an instruction $\texttt{mov}\, d, o$ has the following specification:

$$\begin{aligned}\texttt{specMov}\, i\, \kappa \stackrel{\text{def}}{=} & \Sigma d{:}\texttt{dest}.\Sigma o{:}\texttt{opnd}. \\ & (i =_{\texttt{instr}} \texttt{mov}\, d, o) \\ & \otimes \Sigma w_s{:}\mathsf{nat}.\, (\texttt{specOpnd}\, o\, w_s \otimes \top) \\ & \qquad \& \, \Sigma w_d{:}\mathsf{nat}.\texttt{specDest}\, d\, w_d\, w_s\, \kappa \\ \texttt{specStep}\, i\, \kappa \stackrel{\text{def}}{=} & \cdots \oplus \texttt{specMov}\, i\, \kappa \oplus \cdots\end{aligned}$$

This specification illustrates the use of additive conjunction and the additive unit ($\top$) to check that some resource is available, without consuming it. In general a proposition of the form $(A \otimes \top)\,\&\,(B \otimes (C \multimap \kappa))$ specifies: that resources A are available without consuming them; that resources B are to be consumed; and that new resources C are introduced into the continuation. An alternative would be to specify $A \otimes B \otimes (A \multimap C \multimap \kappa)$, but note that the latter specification has the disadvantage that A and B cannot refer to the same exact resources. For the `mov` operation, that would disallow moves from a register back to itself. (A less contrived situation would be an instruction like $\texttt{add}\, (\texttt{rdest}\, \texttt{r1}), (\texttt{rco}\, \texttt{r1}), (\texttt{rco}\, \texttt{r1})$ that uses the `add` instruction to double the value in a register.)

For a jump instruction, rather than continuing execution at the next instruction, the specification expects that the target of the jump is itself executable:

$$\begin{aligned}\texttt{specJmp}\, i = & \Sigma o{:}\texttt{opnd}.(i =_{\texttt{instr}} \texttt{jmp}\, o) \\ & \otimes\ \Sigma n{:}\mathsf{nat}.(\texttt{specOpnd}\, o\, n \otimes \top) \\ & \qquad \& \, (\Sigma k{:}\mathsf{nat}.\circ_k \mathsf{executable}\, n)\end{aligned}$$

Well-founded recursion and the monadic type One point worth noting is that the continuation does not have the expected type $\mathsf{executable}\ n$, but rather, the more complex $\circ_k \mathsf{executable}\ n$. The type $\circ_M A$ is an indexed family of monadic types. It is used to support reasoning about looping (and recursive) programs without having to introduce unbounded recursion into the proofs (indeed, a general recursion construct $\mathsf{fix}\ f.M$ would allow every type to be inhabited, making LSL unsound). Instead a restricted fixpoint construct introduces hypotheses $\circ_k A$ that are *weaker* than the conclusion A we're trying to form and $\mathsf{execStep}$ and $\mathsf{execNoStep}$ consume these weaker resources.

Intuitively, we want at least one instruction to be executed before we appeal to the fixpoint hypothesis. Furthermore, we use an index k on the

monad $\circ_k A$ to ensure that we cannot inhabit every monadic type without taking a step (it is the case for monads that $\circ \circ A \multimap \circ A$, however for our monad, the weaker $\circ_k \circ_l A \multimap \circ_{k+l} A$ holds). Our monadic type is closely related to the $\triangleright$ type constructor of [AMRV07] and $\bigcirc$ of [HHWC07], but rather than appealing to a semantic model to ensure that proofs are well-founded, we instead use a syntactic restriction by introducing an index.

As a piece of syntactic sugar, we have the abbreviation:

$$?A \stackrel{\text{def}}{=} \Sigma n{:}\mathsf{nat}.\circ_n A$$

At this point, we have described all of the "core" LSL features. The LSL safety theorem (Theorem 3.1) allows us to be sure that if various resources allow us to construct a proof of $\mathsf{executable}\, n$ then a corresponding machine execution will not "go wrong."

However, constructing proofs directly with the definitions given thus far is fairly tedious and is at too low a level of abstraction. In the remainder of this section we develop a collection of definitions to correct this shortcoming. Some of the choices we make in the sequel will restrict the kinds of programs we can reason about (for example we will rule out self-modifying code), but by dealing with more *typical* programs, we will be able to prove stronger theorems which will in turn reduce the amount of proof to be carried out within LSL to reason about our garbage collector.

2.2 Code sequences

There are several disadvantages to reasoning about instructions with just the executable proposition: every proof interleaves showing that the resources for the next instruction are available ($\texttt{codeat}\, i\, pc\, pc'$) with the proof about its behavior; the proofs have to deal with concrete code locations, rather than some more abstract notion of code label as in a typical assembly language; for each instruction the pre- and post- conditions are at the level of resources for individual machine words, rather than some higher-level data structures.

$cs \in$ Code Seqs	$::=$	$\cdot \mid c; cs$
$c \in$ Code	$::=$	$i \mid l: \mid l.c \mid cs$

Figure 1: Syntax for LSL Code Sequences

To address all these concerns, we build an additional abstraction on top of instructions. Code sequences cs abstract syntax is summarized in

Figure 1. Code sequences consist of zero or more pieces of code, which may be individual instructions i; or labels l: that give a name to a particular location within a code sequence. Labels are introduced by the $l.c$ construct, and are bound within c, and are typically used as immediate operands $\texttt{imco}\,l$. Finally, code may itself be a code sequence. In LSL code is defined using LSL variables of type nat, and lambda-abstraction for label abstraction:

$$\begin{aligned}
\texttt{code} &\stackrel{\text{def}}{=} \mu\alpha{:}\mathsf{type}.\texttt{instr} \oplus \mathsf{nat} \oplus (\mathsf{nat} \to \alpha) \oplus \texttt{list}\alpha \\
\texttt{cInstr}\,i &\stackrel{\text{def}}{=} \mathsf{roll}\ (\mathsf{inl}\ i) \\
\texttt{cHere}\,l &\stackrel{\text{def}}{=} \mathsf{roll}\ (\mathsf{inr}\ (\mathsf{inl}\ l)) \\
\texttt{cAbs}\,lc &\stackrel{\text{def}}{=} \mathsf{roll}\ (\mathsf{inr}\ (\mathsf{inr}\ (\mathsf{inl}\ lc))) \\
\texttt{cSeq}\,cs &\stackrel{\text{def}}{=} \mathsf{roll}\ (\mathsf{inr}\ (\mathsf{inr}\ (\mathsf{inr}\ cs)))
\end{aligned}$$

The machine language of LSL has only a few primitive instructions in order to keep the meta-theory small, but by making notational definition with code sequences, we can reimplement many useful operations as macros. The ability to do control transfers by using locally abstracted address labels is particularly useful in this regard.

Resolving code to instructions The process of compiling sequences of code to instructions is called resolution. A code sequence cs *resolves* to an array a at locations between pc and pc' if a consists of all the bytes for every piece of code in cs appended together. An instruction i at address pc resolves to an array a precisely if $\texttt{encodes}\,pc\,i\,a$. A label abstraction $l.c$ resolves to an array a at address pc if there is some choice of address x such that the code c resolves to array a at address pc with x in for all uses of l in c. And finally a label l : resolves to the empty array at location pc precisely if the choice of x for l was pc. That is, when resolving a label abstraction, we guess what address the label refers to, and when we actually come across that label in the body of the abstraction, we check that the guess was right.

The process of resolving a code sequence ultimately rules out certain undesirable situations such as using the same label to name two different locations in a single abstraction $l.cs$ (in that case, there would be no way to resolve one of the labels since they would be at unequal addresses and so there would be no single choice of x for l).

If a code sequence cs resolves to an array a we require that the array be placed in memory as a precondition for the whole code sequence. A

consequence of this is that we rule out self-modifying code, although we do not have any such code in our garbage collector.

$$\begin{aligned}
\texttt{resolve} &\stackrel{\text{def}}{=} \mu\alpha{:}\texttt{code} \to \texttt{array} \to \texttt{nat} \to \texttt{nat} \to \texttt{type}.\\
&\quad \lambda c{:}\texttt{code}.\lambda a{:}\texttt{array}.\lambda pc{:}\texttt{nat}.\lambda pc'{:}\texttt{nat}.\\
&\quad \cdots \oplus \texttt{resolveAbs}\, c\, a\, pc\, pc'\, \alpha\\
&\quad\quad \oplus \texttt{resolveHere}\, c\, a\, pc\, pc' \oplus \cdots\\
\texttt{resolveAbs}\, c\, a\, pc\, pc'\, \alpha &\stackrel{\text{def}}{=} \Sigma c'{:}\texttt{nat} \to \texttt{code}.(c =_{\texttt{code}} \texttt{cAbs}\, c')\\
&\quad\quad \otimes \Sigma l{:}\texttt{nat}.\alpha\, (c'\ l)\, a\, pc\, pc'\\
\texttt{resolveHere}\, c\, a\, pc\, pc' &\stackrel{\text{def}}{=} \Sigma l{:}\texttt{nat}.(c =_{\texttt{code}} \texttt{cHere}\, l)\\
&\quad\quad \otimes (a =_{\texttt{array}} \texttt{anil})\\
&\quad\quad \otimes (l =_{\texttt{nat}} pc) \otimes (pc =_{\texttt{nat}} pc')
\end{aligned}$$

2.3 Partial correctness specifications for code

In reasoning about sequences of code, we would like to be able to specify their pre- and post-conditions at a higher level of abstraction than individual machine words. To that end, we employ Hoare quads — partial correctness specifications similar to Hoare triples[Hoa69] — but with an additional fourth component.

The specification $\{\alpha\}c\{\beta\}$ says that if the precondition α holds, and the (resolved) code c is in memory between addresses pc and pc', then after the code executes, and control flow reaches address pc', then we can assume that the post condition β will hold when trying to show that pc' is executable.

Hoare triples have an entailment property, and Hoare triples in separation logic have a *frame rule* to pass unused resources from between the pre- and post-conditions. It is these two properties that ultimately make Hoare triples useful for us: we can specify code behavior in terms of high-level data structures which can be whittled away by entailment and the frame rule to just those pieces of the data structure that are ultimately accessed by a piece of code. So Hoare triples provide a convenient notation for stating theorems, but ultimately they give us no additional expressive power.

Since we're in a linear setting, we must also account for the disposition of the resources in case that control flow does not reach the next instruction at pc' after a piece of code. For example if the code c includes a conditional branch to some other address pc'', then the post-condition β will not necessarily hold, and indeed to prove that c is safe, it does not suffice to to

show that execution continues from pc' in that case. To account for this possibility, we add a fourth component — zero or more *escape conditions* γ — $\{\alpha\}c^{\gamma}_{\{\beta\}}$. An escape condition consumes the resources ρ that correspond the array a to which the code c resolves[2]:

$$\begin{aligned}\{\alpha\}c^{\gamma}_{\{\beta\}} \stackrel{\text{def}}{=} \Pi pc{:}\mathsf{nat}.\Pi pc'{:}\mathsf{nat}.\Pi a{:}\mathtt{array}.\forall\rho{:}\mathsf{type}.\\ \mathtt{resolve}\, c\, pc\, pc'\, a\\ \multimap (\rho \multimap \mathtt{placeArray}\, a \otimes \top)\\ \multimap (\alpha \otimes \rho \multimap (\beta \otimes \rho \multimap ?\mathsf{executable}\, pc') \,\&\, (\rho \multimap \gamma)\\ \multimap \mathsf{executable}\, pc)\end{aligned}$$

Essentially, Hoare quads are in continuation passing style: the post-condition is the type of the result that the default continuation expects and the escape conditions are alternate continuations available for a piece of code. The additive conjunction connective of linear logic allows us to specify that exactly one of the possible continuations will be used.

For an instruction such as `mov`, the escape condition is simply $\top$ since there is no non-local control transfer, so there is no need to consume the resources in an alternate manner, and $\top$ is the unit for additive conjunction:[3]

$$\{(\mathtt{specOpnd}\, o\, x \otimes \top) \,\&\, \Sigma y{:}\mathsf{nat}.\mathtt{specDest}\, d\, y\, x\, \alpha\}\mathtt{mov}\, d, o^{\top}_{\{\alpha\}}$$

On the other hand, an unconditional jump has an escape condition that requires that we can show that the precondition implies that the destination of the jump is executable:[4]

$$\{(\mathtt{specOpnd}\, o\, l \otimes \top) \,\&\, \alpha\}\mathtt{jmp}\, r^{(\alpha \multimap ?\mathsf{executable}\, l)}_{\{0\}}$$

Finally, a conditional jump may *either* continue normally if t is zero, or transfer control to a different address l otherwise:

$$\left\{\begin{array}{c}(\mathtt{specOpnd}\, o_t\, t \otimes \top) \,\&\, (\mathtt{specOpnd}\, o_l\, l \otimes \top)\\ \&\, (t =_{\mathsf{nat}} 0 \multimap \alpha) \,\&\, (0 < t \multimap \beta)\end{array}\right\}\mathtt{jnz}\, o_t, o_l{}^{(\beta \multimap ?\mathsf{executable}\, l)}_{\{\alpha\}}$$

[2] We chose to reason indirectly via ρ rather than dealing concretely with $\mathtt{placeArray}\, a$, so that we can reason compositionally about multiple pieces of code that implicitly pass around the whole set of resources ρ.

[3] Each Hoare quad that we present in this section is a proposition; the proof terms for each of them exist (in machine-checkable LSL), but are omitted from this document.

[4] The strongest post-condition is falsehood; since there is no way to transfer control to the point after the jump instruction, in any such state we can conclude anything we want.

A sequential composition of two pieces of code has pre- and post-conditions that compose as for Hoare triples. The escape conditions combine together with additive conjunction — we lose information in the conclusion about which piece of code can escape to which alternate continuation, only knowing that their composition can escape to one of the combination of alternatives:

$$\begin{aligned}&\{\alpha\}c_1{}^{\delta_1}_{\{\beta\}}\\ \to&\{\beta\}c_2{}^{\delta_2}_{\{\gamma\}}\\ \to&\{\alpha\}c_1; c_2{}^{\delta_1 \,\&\, \delta_2}_{\{\gamma\}}\end{aligned}$$

Hoare quads have the usual entailment behavior in their pre- and post-conditions. Since a piece of code decides which escape conditions to make use of, offering it more choices which it does not use is always possible. Therefore to weaken a Hoare quad, we strengthen the escape condition:

$$\begin{aligned}&\{\alpha\}c^{\delta}_{\{\beta\}}\\ \to&(\alpha' \multimap \alpha) \to (\beta \multimap \beta') \to (\delta' \multimap \delta)\\ \to&\{\alpha'\}c^{\delta'}_{\{\beta'\}}\end{aligned}$$

In addition, Hoare quads have a *frame rule*[Rey02] that allows for local reasoning: it suffices to only reason with the resources directly affected by a piece of code without the resources that pass unchanged:

$$\begin{aligned}&\{\alpha\}c^{\delta}_{\{\beta\}}\\ \to&\{\alpha \otimes \rho\}c^{(\rho \multimap \delta)}_{\{\beta \otimes \rho\}}\end{aligned}$$

A final example illustrates the interaction between the fixpoint construct and Hoare quads. In a loop, the loop invariant is the precondition: it entails that the label at the loop entry is executable. Code within the loop gains an additional escape condition back to the loop entry point which can only be used if the precondition is re-established. (Compare the escape condition below with the escape condition for a jump instruction):

$$\begin{aligned}&\{\alpha\}c^{(\alpha \multimap ?\mathsf{executable}\, l) \,\&\, \gamma}_{\{\beta\}}\\ \to&\{\alpha\}l\colon c^{\gamma}_{\{\beta\}}\end{aligned}$$

The proof term for the preceding proposition uses the fixpoint construct to form an unrestricted hypothesis $\circ_{\overline{1}}\{\alpha\}c^{\gamma}_{\{\beta\}}$. That entails ($\alpha \multimap \circ_{\overline{2}}\mathsf{executable}\, pc$), which may be consumed in the course of showing the safety of a jump back to the loop entry point.

Backward reasoning Although there may generally be several equivalent[5] Hoare quads for each instruction, of particular interest are ones where the postcondition and escape conditions are universally quantified type variables. Suppose we wish to show:

$$\{P\}c_1; c_2{}^{Es}_{\{Q\}}$$

if we have

$$\forall\beta.\forall\gamma.\{P_1(\beta,\gamma)\}c_1{}^{\gamma}_{\{\beta\}}$$

and

$$\forall\beta.\forall\gamma.\{P_2(\beta,\gamma)\}c_2{}^{\gamma}_{\{\beta\}}$$

We can reason backwards as follows:

1. Instantiate the second Hoare quad with Q and Es:
   $\{P_2(Q,Es)\}c_2{}^{Es}_{\{Q\}}$
2. Instantiate the first Hoare quad with $P_2(Q,Es)$ and Es:
   $\{P_1(P_2(Q,Es),Es)\}c_1{}^{Es}_{\{P_2(Q,Es)\}}$
3. Use the rule for sequence composition:
   $\{P_1(P_2(Q,Es),Es)\}c_1; c_2{}^{Es\,\&\,Es}_{\{Q\}}$
4. Use the entailment rule to conclude $\{P\}c_1; c_2{}^{Es}_{\{Q\}}$ given proofs of $P \multimap P_1(P_2(Q,Es),Es)$, $Q \multimap Q$ and $Es \multimap Es\,\&\,Es$

In fact, these steps can be carried out completely automatically leaving us with one non-trivial proof left provided that we have backward reasoning Hoare quads for every piece of code. As it happens, such Hoare quads are available for all non-looping code, and as a result a proof assistant can be used for reasoning about straight-line and forward-jumping code with minimal user intervention culminating in a single non-trivial subgoal as in the previous example.

[5] in the sense that they entail each other

2.4 Stacks and calling convention

From the point of view of LSL, a stack and the closely related notion of a procedure calling convention is just a particular convention in the use of memory resources and (a dedicated) stack register.

Stack layout A particular register is singled out as `stackReg` and the value in it, *sp*, is the stack pointer. A value is a stack pointer if it is a multiple of $wordWidth$ and it points within the stack: a contiguous set of memory resources (between addresses, `stackMin` and `stackMax`). The stack is divided into the *unused* stack space (between `stackMin` and $sp - wordWidth$) and the *used* stack (between *sp* and `stackMax`).

$$\texttt{stackPointer}\, sp \stackrel{\text{def}}{=} \texttt{pointer}\, sp \otimes (\texttt{stackMin} \leq sp) \otimes (sp \leq \texttt{stackMax})$$

Stack frames, pushes and pops Common programming practice is to only access a local part of the used stack, the *current stack frame*. With linear logic, we can restrict ourselves to reasoning about only well-behaved stack operations (in the sense of respecting the current stack frame) by only providing the linear resources for this prefix:

$$\begin{aligned}
\texttt{unusedStack}\, spmin\, sp\, a \stackrel{\text{def}}{=}\ & \Sigma l{:}\texttt{list nat}.!(\texttt{encodesWords}\, spmin\, a\, l) \\
& \otimes (spmin + (\texttt{alen}\, a) =_{\mathsf{nat}} sp) \\
\texttt{usedStack}\, l\, a\, sp\, spmax \stackrel{\text{def}}{=}\ & !(\texttt{encodesWords}\, sp\, a\, l) \\
& \otimes (sp + (\texttt{alen}\, a) =_{\mathsf{nat}} spmax) \\
& \otimes (spmax \leq \texttt{stackMax})
\end{aligned}$$

A `pop` instruction, simply increments the stack pointer by $wordWidth$. As a precondition P, however, we require that there be at least one element x on the current stack frame. In the post-condition, the resources are re-associated, and the location of x goes from the used stack space to the unused:

$$\texttt{pop} \stackrel{\text{def}}{=} \texttt{add}\, \texttt{stackReg}, \texttt{stackReg}, wordWidth$$

The Hoare quad is $\{P\}\texttt{pop}_Q^\top$ where

$$\begin{aligned} P = {} & !(\texttt{stackPointer}\, sp) \otimes !(\texttt{unusedStack}\, \texttt{stackMin}\, sp\, un) \\ & \otimes !(\texttt{usedStack}\, (x{::}xs)\, frame\, sp\, frameTop) \\ & \otimes \texttt{rwat}\, \texttt{stackReg}\, sp \otimes \texttt{placeArray}\, un \otimes \texttt{placeArray}\, frame \end{aligned}$$

and

$$\begin{aligned} Q = {} & \Sigma sp'{:}\mathsf{nat}.!(\texttt{stackPointer}\, sp') \otimes (sp' =_{\mathsf{nat}} wordWidth + sp) \\ & \otimes \texttt{rwat}\, \texttt{stackReg}\, sp' \\ & \otimes (\Sigma un'{:}\texttt{array}.!(\texttt{unusedStack}\, \texttt{stackMin}\, sp'\, un') \\ & \qquad \otimes \texttt{placeArray}\, un') \\ & \otimes \Sigma frame'{:}\texttt{array}.!(\texttt{usedStack}\, xs\, frame'\, sp'\, frameTop) \\ & \qquad \otimes \texttt{placeArray}\, frame' \end{aligned}$$

To pop an element from the stack, we need to show as a post-condition that the new stack pointer address sp' will be valid. Since the precondition specifies that there was at least one element on the stack frame, we can deduce that sp' is still bounded by `stackMax`. On the other hand, when doing a `push` operation, it is not clear that there is an unused stack slot available. Consequently, the code for a push must do a runtime check to ensure that sp does not equal `stackMin` (if it does, we halt the program[6]):

$$\begin{aligned} \texttt{push}\, o \stackrel{\text{def}}{=} l.\ \ & \texttt{add}\, \texttt{stackReg}, \texttt{stackReg}, \texttt{negStackMin} \\ & \texttt{jnz}\, \texttt{stackReg}, (\texttt{imco}\, l) \\ & \texttt{halt} \\ l{:}\ & \texttt{add}\, \texttt{stackReg}, \texttt{stackReg}, (\texttt{stackMin} - wordWidth) \\ & \texttt{mov}\, (\texttt{mdest}\, (\texttt{rco}\, \texttt{stackReg})\, \overline{0}), o \end{aligned}$$

The Hoare quad is $\{P(\alpha)\}\texttt{push}\, o_{\{\alpha\}}^\top$ where:

[6] A more realistic machine model would account for interaction with an operating system, in which case writes to certain memory pages may induce an OS trap. Most operating systems unmap the page immediately below the the bottom of the stack, so a runtime check would not be needed, since the postcondition for a memory write would allow us to deduce that the stack pointer had been within bounds.

$$
\begin{aligned}
P(\kappa) = {} & !(\texttt{stackPointer}\ sp) \otimes !(\texttt{usedStack}\ xs\ frame\ sp\ stackMax) \\
& \otimes !(\texttt{unusedStack}\ \texttt{stackMin}\ sp\ un) \\
& \otimes ((\texttt{specOpnd}\ o\ x \otimes \top) \\
& \quad \& (\texttt{rwat}\ \texttt{stackReg}\ sp \otimes \texttt{placeArray}\ un \otimes \texttt{placeArray}\ frame \\
& \qquad \otimes (\Pi sp'{:}\mathsf{nat}.\Pi un'{:}\texttt{array}.\Pi frame'{:}\texttt{array}. \\
& \qquad\quad (sp =_{\mathsf{nat}} wordWidth + sp') \\
& \qquad\quad \to \texttt{stackPointer}\ sp' \to \texttt{unusedStack}\ \texttt{stackMin}\ sp'\ un' \\
& \qquad\quad \to \texttt{usedStack}\ (x :: xs)\ frame'\ sp'\ stackMax \\
& \qquad\quad \to \texttt{rwat}\ \texttt{stackReg}\ sp' \\
& \qquad\quad\ \otimes \texttt{placeArray}\ frame' \otimes \texttt{placeArray}\ un' \\
& \qquad\quad \multimap \kappa)))
\end{aligned}
$$

Loads We can access an element of the current stack frame using an indirect move with a positive $wordWidth$-multiple. For example if the stack frame contains $x_0{::}x_1{::}xs$, we can load x_1 with the instruction:

$$\texttt{mov}\ d, (\texttt{mco}\ (\texttt{rco}\ \texttt{stackReg})\ (wordWidth * \overline{1}))$$

Although the general Hoare quad for moves from memory can be applied equally well to the stack, a more specific rule simplifies reasoning by allowing us to reuse proofs for breaking apart the `placeArray` resource into its constituent resources:

$$
\begin{aligned}
P(\kappa) = {} & (\Sigma sp{:}\mathsf{nat}.\Sigma frame{:}\texttt{array}.!(\texttt{usedStack}\ xs\ frame\ sp\ spmax) \\
& \qquad\qquad \otimes !(\texttt{elemNth}\ xs\ k\ x) \\
& \qquad\qquad \otimes \texttt{placeArray}\ frame \otimes \texttt{rwat}\ \texttt{stackReg}\ sp \\
& \qquad\qquad \otimes \top) \\
& \quad \& (\Sigma y{:}\mathsf{nat}.\texttt{specDest}\ d\ y\ x\ \kappa)
\end{aligned}
$$

in the Hoare quad

$$\{P(\alpha)\}\texttt{mov}\ d, (\texttt{mco}\ (\texttt{rco}\ \texttt{stackReg})\ (wordWidth * k))^{\top}_{\{\alpha\}}$$

Stack frame layouts and calling conventions Procedure calling conventions further subdivide the current stack frame into components such as the procedure arguments, a return address, and zero or more local variables. Operationally, a procedure call pushes the return address and transfers control to the caller. Logically, a procedure call is also responsible for shifting from the caller's view of the current stack frame, to the callee's view of their own new stack frame: some of the caller's pushed locals become the arguments of the callee. Correspondingly, a return instruction pops the return address from the stack and transfers control back to the caller and logically it shifts the view back to the caller.

We say that a stack frame is *valid* if it is laid out according to our calling convention and includes only the current function's local view of the stack (i.e., the largest used stack address is at the last function argument):

$$\begin{aligned}&\texttt{validFrame}\, sp\, locals\, ra\, args\, unused\, frame \stackrel{\text{def}}{=}\\ &\quad \texttt{unusedStack}\, \texttt{stackMin}\, sp\, unused\\ &\quad \otimes\, \texttt{usedStack}\, (locals +\!\!+ [ra] +\!\!+ args)\, frame\\ &\qquad\qquad sp\, (wordWidth * (1 + |args|) + sp)\end{aligned}$$

A return instruction simply pops an address from the top of the stack and jumps to it. The escape condition accounts for the callee's view of the stack after the return address is popped off:

$$\begin{aligned}&\texttt{normalFuncEscapeCond}\, \beta\, args\, sp\, ra \stackrel{\text{def}}{=}\\ &\quad \Pi unused{:}\texttt{array}.\Pi frame{:}\texttt{array}.\\ &\qquad \texttt{unusedStack}\, sp\, unused\\ &\quad \rightarrow \texttt{usedStack}\, args\, frame\, sp\, (wordWidth * |args| + sp)\\ &\quad \rightarrow \beta\ args \otimes \texttt{rwat}\, \texttt{stackReg}\, sp\\ &\qquad \otimes \texttt{placeArray}\, unused \otimes \texttt{placeArray}\, frame\\ &\quad \multimap ?\mathsf{executable}\, ra\end{aligned}$$

Here β::(`list` `nat`) $\rightarrow$ `type` is the function exit invariant and $args$ were the arguments with which the function was called. Note that the local stack maximum is $wordWidth * |args| + sp$, so the callee's local view of the stack only includes a short prefix portion of the caller's stack frame (namely the callee's arguments).

A return instruction has the Hoare quad:

$$\{P(\beta)\}\texttt{ret}_{\{0\}}^{\texttt{normalFuncEscapeCond}\,\beta\,args\,sp\,ra}$$

where

$$\begin{aligned} P(\sigma) = {} & !(\texttt{validFrame}\,(wordWidth + sp)\,[]\,ra\,args\,unused\,frame) \\ & \otimes \sigma\ args \otimes \texttt{rwat}\,\texttt{stackReg}\,(wordWidth + sp) \\ & \otimes \texttt{placeArray}\,unused \otimes \texttt{placeArray}\,frame \end{aligned}$$

The entry point to a function must similarly provide a valid stack frame, an an additional entry point invariant (that perhaps mentions the function arguments). So if a function f that takes arguments $args$ has entry point invariant $\alpha\ args$ and an exit invariant $\beta\ args$ and is implemented by code c, we are obligated to prove the Hoare quad:

$$\{P(\alpha)\}c_{\{0\}}^{\texttt{normalFuncEscapeCond}\,\beta\,args\,sp\,ra}$$

The post-condition 0 indicates that functions must exit via a return instruction, rather than by falling through at the bottom.

To call a function, the callee must push the address immediately after the call instruction onto the stack, and then jump to the function's entry point:

$$\begin{aligned} \texttt{call}\,o \stackrel{\text{def}}{=} {} & l.\texttt{push}\,(\texttt{imco}\,l) \\ & \texttt{jmp}\,o \\ & l: \end{aligned}$$

Given

$$\begin{aligned} & P\,\sigma\,sp\,args\,locals\,oldRA\,oldArgs = \\ & \quad \Sigma un{:}\texttt{array}.\Sigma frame{:}\texttt{array}. \\ & \qquad !(\texttt{validFrame}\,sp\,(args +\!\!+ locals)\,oldRA\,oldArgs\,un\,frame) \\ & \qquad \otimes \texttt{rwat}\,\texttt{stackReg}\,sp \otimes \texttt{placeArray}\,frame \\ & \qquad \otimes \texttt{placeArray}\,un \\ & \qquad \otimes \sigma\,args \end{aligned}$$

the precondition is

$$
\begin{array}{l}
Q\,o\,\alpha\,\beta\,sp\,args\,locals\,oldRA\,oldArgs = \\
\quad \Sigma fa{:}\mathsf{nat}.(\mathtt{specOpnd}\,o\,fa \otimes \top) \\
\qquad\qquad \& \,(\mathtt{funSpec}\,fa\,\alpha\,\beta \otimes P\,\alpha\,sp\,args\,locals\,oldRA\,oldArgs)
\end{array}
$$

for a specification

$$\{Q\,o\,\alpha\,\beta\,sp\,args\,locals\,oldRA\,oldArgs\}\mathtt{call}\,o^{\top}_{\{P\,\beta\,sp\,args\,locals\,oldRA\,oldArgs\}}$$

Where $\mathtt{funSpec}\,fa\,\alpha\,\beta$ is a specification for some function at address fa that has a precondition $\alpha\,args$ and post-condition $\beta\,args$ (compare with the definition of Hoare quads, p. 13):

$$
\begin{array}{l}
\mathtt{funSpec}\,fa\,\alpha\,\beta \stackrel{\text{def}}{=} \Sigma c{:}\mathtt{code}. \\
\qquad \Pi args{:}\mathtt{list}\,\mathsf{nat}.\Pi sp{:}\mathsf{nat}.\Pi ra{:}\mathsf{nat}.\Pi a{:}\mathtt{array}.\forall\rho{:}\mathsf{type}. \\
\qquad\quad (\Sigma pc'{:}\mathsf{nat}.\mathtt{resolve}\,c\,fa\,pc'\,a) \\
\qquad\quad \to (\rho \multimap \mathtt{placeArray}\,a \otimes \top) \\
\qquad\quad \to \mathtt{validFrame}\,sp\,[]\,ra\,unused\,frame \\
\qquad\quad \to\ \alpha\,args \otimes \mathtt{rwat}\,\mathtt{stackReg}\,sp \\
\qquad\qquad \otimes \mathtt{placeArray}\,unused \otimes \mathtt{placeArray}\,frame \\
\qquad\qquad \otimes \rho \\
\qquad\quad \multimap (\rho \multimap \mathtt{normalFuncEscapeCond}\,\beta\,args\,sp\,ra) \\
\qquad\quad \multimap ?\mathsf{executable}\,fa
\end{array}
$$

2.5 Object language and memory graph

With a mechanism for specifying functions, higher levels of abstraction are possible. We can, for example, build a library for manipulating queues of elements.

Such a library can, in turn, be used to implement a reference counting garbage collector.

For simplicity, we assume that all objects allocated by the reference counting allocator are of uniform size and layout: pairs of elements each of which can be either a non-pointer integer, or a pointer to another element. Each live element keeps a reference count of the number of other elements in the graph that point at it.

We chose to implement a lazy reclamation strategy: when the reference count of an element becomes zero, rather than immediately reclaiming it, we place it on a queue of other such elements. They each have the property that they have zero references to them, but may themselves contribute to the reference counts of live objects. We collect all such objects in a reclamation queue.

When user code calls the allocator to create a new object, we do one step of reclamation by removing an element from the reclamation queue and decrementing the reference counts of the objects that it points to, possibly placing them at the tail of the reclamation queue. Then we place the dequeued element onto a queue of freed objects.

This method of lazy reclamation has the advantage of using a bounded amount of time for each garbage collection operation while incurring a negligible additional space cost between reclamations[Boe04].

The interaction between the object language and the garbage collector is specified at three levels: the object language's type system, an abstract intuitionistic specification in terms of graph elements and queues, and a concrete linear specification in terms of bytes residing at particular memory addresses.

For each operation in the object language, we have an LSL definition that gives its behavior in terms of the abstract memory graph. (This is essentially an LSL encoding of the object language operational semantics specialized to our chosen GC implementation.) Additionally, we have the concrete implementation of the object language operation as a piece of LSL code.

For example, for the allocate instruction $\texttt{cons}\, x\, y_1\, y_2$ the operational semantics may be[7]:

$$\frac{a \notin \mathrm{dom}(H)}{(H, R; \texttt{cons}\, x\, y_1\, y_2) \rightarrow (H\{a \mapsto \langle R(y_1), R(y_2)\rangle^1\}, R\{x \mapsto a^1\})}$$

with typing rule:

$$\frac{\Gamma \vdash x : \mathsf{ns} \quad \Gamma \vdash y_i : \mathsf{nat}}{\Gamma \vdash \texttt{cons}\, x\, y_1, y_2 \Rightarrow \Gamma\{x : \mathsf{nat} \times_1 \mathsf{nat}\}}$$

Abstractly, the state consists only of the memory graph H and a root set R. And the operational semantics does not explain how we're to come up with the fresh address a.

[7] this is the operational semantics for an allocation operation where y_1 and y_2 contain non-pointer values

The judgment $\Gamma \vdash \iota \Rightarrow \Gamma'$ means that in a state where the rootset has type Γ and we execute instruction ι, the resulting state has a rootset with type Γ'. As usual, preservation and progress lemmas ensure that the abstract operational semantics do not go wrong.

For each rule of the operational semantics of the object language, we have an LSL encoding of a specification for the behavior of our particular allocator. For example, the cons operation above is specified by:

$$
\begin{array}{l}
\texttt{specCons}\, Q\, F\, H\, R\, x\, y_1\, y_2\, Q'\, F_2\, H_2\, R' \stackrel{\text{def}}{=} \\
\quad \Sigma F_1{:}\texttt{queue}.\Sigma H_1{:}\texttt{graph}. \\
\qquad \texttt{specReclaim}\, Q\, F\, H\, Q'\, F_1\, H_1 \\
\quad \otimes\, \Sigma v_1{:}\texttt{gvalue}.\Sigma v_2{:}\texttt{gvalue}. \\
\qquad \texttt{rootLookup}\, R\, y_1\, v_1 \\
\quad \otimes\, \texttt{rootLookup}\, R\, y_2\, v_2 \\
\quad \otimes\, \Sigma F_2{:}\texttt{queue}.\Sigma a{:}\texttt{address}.\texttt{specDequeue}\, F_1\, a\, F_2 \\
\qquad \otimes\, \texttt{graphInsert}\, H_1\, a\, v_1\, v_2\, H_2 \\
\qquad \otimes\, \texttt{rootUpdate}\, R\, x\, a\, R'
\end{array}
$$

At the concrete level, the instruction is implemented by:

$$
\begin{array}{lll}
\texttt{implCons}\, x\, y_1\, y_2 \stackrel{\text{def}}{=} & \texttt{call}\,(\texttt{imco}\, alloc) \\
& \texttt{mov}\,(\texttt{rootDest}\, x), (\texttt{rco}\, \texttt{retvalReg}) \\
& \texttt{mov}\,(\texttt{firstSlotDest}\,(\texttt{rco}\, \texttt{retvalReg})), (\texttt{rootOpnd}\, y_1) \\
& \texttt{mov}\,(\texttt{secondSlotDest}\,(\texttt{rco}\, \texttt{retvalReg})), (\texttt{rootOpnd}\, y_2)
\end{array}
$$

where *alloc* is implemented as

$$
\begin{array}{lll}
\texttt{implAlloc} \stackrel{\text{def}}{=} & alloc{:}\texttt{call}\,(\texttt{imco}\, reclaim) \\
& \qquad \texttt{push}\,(\texttt{imco}\, freelist) \\
& \qquad \texttt{call}\,(\texttt{imco}\, dequeueOrHalt) \\
& \qquad \texttt{pop} \\
& \qquad \texttt{ret}
\end{array}
$$

Where the call to the *alloc* function operates on resources that correspond to an abstract graph H but also the free queue F and the reclaimation queue Q.

To ensure that the concrete implementation is correct, we first relate the object language typing rules to the abstract state by reflecting the judgment $\vdash (H, R) : \Gamma$ — that shows the root set and memory graph are consistent with rootset type Γ — into LSL as $\texttt{consist}\, H\, R\, \Gamma$. Additionally, the state (F, Q, H, R) must be closed and consistent in the sense that all graph elements have the correct reference counts.

We are required to prove

$$\{P(H, R, \Gamma, x, y_1, y_2)\, \kappa\}\texttt{implCons}\, x\, y_1\, y_2 \,{}^{\top}_{\{\kappa\}}$$

where the precondition is

$$\begin{aligned}
P(H, R, \Gamma, x, y_1, y_2)\, \kappa = {} & \texttt{consist}\, H\, R\, \Gamma \\
& \otimes \Sigma Q{:}\texttt{queue}.\Sigma F{:}\texttt{queue.closed}\, Q\, F\, H\, R \\
& \otimes \texttt{placeState}\, Q\, F\, H \\
& \otimes \texttt{placeRoots}\, R \\
& \otimes (\Pi Q', F', H', R', \Gamma'. \\
& \quad \texttt{specCons}\, Q\, F\, H\, R\, x\, y_1\, y_2\, Q'\, F'\, H'\, R' \\
& \quad \to \texttt{consist}\, H'\, R'\, \Gamma' \\
& \quad \to \texttt{closed}\, Q'\, F'\, H'\, R' \\
& \quad \to \texttt{placeState}\, Q\, F\, H \\
& \quad \multimap \texttt{placeRoots}\, R' \\
& \quad \multimap \kappa)
\end{aligned}$$

The abstract graph behavior of cons is specified with unrestricted hypotheses, and the concrete behavior operates on the linear resources that correspond to the abstract state via `placeState` and `placeRoots`. The connection to linear resources allows us to reason locally about the behavior of helper functions that implement cons and *alloc*. For example, we can prove that an implementation of *dequeueOrHalt* obeys its specification once and for all for any queue. Additionally, linearity lets us reason simply about freshness: in the specification for cons we dequeue an address a from the free queue and we may immediately insert it into the graph H, fulfilling implicitly the side condition $a \notin \mathrm{dom}(H)$ of the object language evaluation rule for cons, on p. 22.

And similarly for all the remaining operations in the object language.

3 Linear Separation Logic

3.1 Syntax

$$
\begin{array}{rll}
K \in \text{Kinds} & ::= & \mathsf{type} \mid \Pi x{:}A.K \\
A \in \text{Type Families} & ::= & \alpha \mid \forall \alpha{:}K.A \\
& & \mid a \mid A\ M \mid \lambda x{:}A_1.A_2 \\
& & \mid 1 \mid A_1 \otimes A_2 \mid A_1 \multimap A_2 \\
& & \mid \top \mid A_1 \,\&\, A_2 \\
& & \mid 0 \mid A_1 \oplus A_2 \\
& & \mid\ !A \mid \Pi x{:}A_1.A_2 \mid \Sigma x{:}A_1.A_2 \\
& & \mid \mu\alpha{:}A_1{\to}\mathsf{type}.A_2 \\
& & \mid \mathsf{isl}_{A_1 \oplus A_2}\ M \\
& & \mid \circ_M A \\
& & \mid \mathsf{nat} \\
M \in \text{Terms} & ::= & x \mid u \mid c \\
& & \mid \Lambda\alpha.M \mid M\ [A] \\
& & \mid \star \mid \mathsf{let}\ \star = M_1\ \mathsf{in}\ M_2 \\
& & \mid M_1 \otimes M_2 \mid \mathsf{let}\ u_1 \otimes u_2 = M_1\ \mathsf{in}\ M_2 \\
& & \mid \hat{\lambda} u.M \mid M_1 \hat{}\, M_2 \\
& & \mid \langle\rangle \mid \langle M_1, M_2\rangle \mid \pi_{1,2} M \\
& & \mid \mathsf{any}\ M \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M \\
& & \mid \mathsf{case}\ M_1\ \mathsf{of}\ \mathsf{inl}\ u \Rightarrow M_2 \mid \mathsf{inr}\ u \Rightarrow M_3 \\
& & \mid\ !M \mid \mathsf{let}\ !x = M_1\ \mathsf{in}\ M_2 \\
& & \mid \lambda x.M \mid M_1\ M_2 \\
& & \mid \mathsf{pack}\ \langle M_1, M_2\rangle \mid \mathsf{let}\ \mathsf{pack}\ \langle x, u\rangle = M_1\ \mathsf{in}\ M_2 \\
& & \mid \mathsf{roll}\ M \mid \mathsf{pr}_{\alpha.A_3}^{\mu\alpha:A_1\to\mathsf{type}.A_2}(\alpha, f, x; u.M) \\
& & \mid \mathsf{isli}\ M \mid \mathsf{isle}\ M \\
& & \mid \mathsf{fix}\ x{:}A.M \mid \mathsf{circ}\ E \\
& & \mid \overline{n} \mid \mathsf{prnat}_A\ M_1\ (x, f.M_2) \\
& & \mid M{:}A \mid \mathsf{cut}\ u = M_1\ \mathsf{in}\ M_2 \mid \mathsf{cut!}\ x = M_1\ \mathsf{in}\ M_2 \\
E \in \text{Expressions} & ::= & \mathsf{tm}\ M \mid \mathsf{let}_{(M_1,M_2,M_3)} \circ u = M_4\ \mathsf{in}\ E \\
& & \mid \mathsf{let}\ u_1 \otimes u_2 = M\ \mathsf{in}\ E \\
& & \mid \mathsf{let}\ !x = M\ \mathsf{in}\ E \\
\Gamma \in \text{Contexts} & ::= & \cdot \mid \Gamma, x{:}A \mid \Gamma, \alpha{:}K \\
\Delta \in \text{Linear Ctxs} & ::= & \cdot \mid \Delta, u\hat{:}A
\end{array}
$$

Figure 2: Linear Separation Logic (Syntax)

The syntax of LSL is summarized in Figure 2.

LSL types — classified by kinds K — include type variables, and polymorphic types, term-indexed type family abstraction and application, multiplicative and additive linear logical connectives, dependent products and sums, certain inductive types, the type of natural numbers, and two types: $\mathsf{isl}_{A_1 \oplus A_2}\ M$ and $\circ_M A$ discussed below.

The terms of LSL include intuitionistic and linear variables, introduction and elimination forms for all the types, as well as a term annotated with a type (necessary for bidirectional type checking), as well as a term $\mathsf{cut}\ u = M\ \mathsf{in}\ N$ binding u to the value of M in N (in the bidirectional setting, this is not just an abbreviation for $(\hat{\lambda} u.N){:}(A \multimap B)\hat{}M$ because in the latter M is checked against type A but in the former it synthesizes a type).

3.2 Unusual types

The two types $\mathsf{isl}_{A_1 \oplus A_2}\ M$ and $\circ_M A$ are specific to LSL. The former is used to reason about disjoint sum index objects, while the latter is used to reason about well-founded recursion.

Reasoning about disjoint sums The proposition $\mathsf{isl}_{A_1 \oplus A_2}\ M$ says that M has type $A_1 \oplus A_2$, and moreover that it is a left injection $\mathsf{inl}\ M'$ for some M'. This proposition is used to derive contradictions in the case that in fact M is a right injection $\mathsf{inr}\ M''$.

The introduction form $\mathsf{isli}\ M'$ has type $\mathsf{isl}_{A_1 \oplus A_2}\ (\mathsf{inl}\ M')$ provided that M' has type A_1.

The elimination form $\mathsf{isle}\ M$ has type 0, provided that M is actually a right injection $\mathsf{inr}\ M'$.

For example, we can prove the proposition $(\mathsf{inl}\ \star) =_{1 \oplus 1} (\mathsf{inr}\ \star) \multimap 0$ by:

$$\lambda e.\mathsf{isle}\ (e\ [x.\mathsf{isl}_{1 \oplus 1}\ x]\hat{}(\mathsf{isli}\ \star))$$

Here e is a proof of the equality $(\mathsf{inl}\ \star) =_{1 \oplus 1} (\mathsf{inr}\ \star)$, which is defined as the Leibnitz equality, $\forall \phi{:}(1 \oplus 1) \to \mathsf{type}.\phi\ (\mathsf{inl}\ \star) \multimap \phi\ (\mathsf{inr}\ \star)$, which we apply to the type family $\lambda x{:}1 \oplus 1.\mathsf{isl}_{1 \oplus 1}\ x$, to get a function that converts $\mathsf{isl}_{1 \oplus 1}\ (\mathsf{inl}\ \star)$ to $\mathsf{isl}_{1 \oplus 1}\ (\mathsf{inr}\ \star)$. The latter two types are precisely what is needed by the intro and elim forms for isl .

Reasoning about well-founded recursion When reasoning about looping code, we have to assume that the code is executable under a certain precondition, in order to show that same fact.

To accommodate such reasoning, but still retain a sound logic, we allow showing A not under the same assumption, but under a weaker assumption $\circ A$. Proofs of such weaker assumptions are then consumed by the atomic proofs for executable instructions. Conceptually, if we can show A while assuming that A will be true at some point in the future after at least one machine instruction executes, then we can show A.

We could then have the typing rules[8]:

$$\frac{\Gamma, x{:}\circ A \vdash M : A}{\Gamma \vdash \mathsf{fixpt}\ x.M : A}\ (\textsc{wrong})$$

and

$$\frac{\Gamma \vdash E \div A}{\Gamma \vdash \mathsf{circ}\ E : \circ A}\ (\textsc{wrong})$$

Where the expressions E are either $\mathsf{tm}\ M$ which just returns the term M, or the monad elimination form $\mathsf{let} \circ y = M\ \mathsf{in}\ E$. The fix point construct then stands for $M[N/x]$ where N is $\mathsf{circ}(\mathsf{let} \circ z = (\mathsf{fixpt}\ x.M)\ \mathsf{in}\ \mathsf{tm}z)$.

The key metatheoretical property that we need is that closed terms of type $\circ A$ are all of the form $\mathsf{circ}\ (\mathsf{tm}\ M)$. That is, we can extract a proof of A from any such closed proof.

Unfortunately, the monad law $\circ \circ A \multimap \circ A$ means that we can, in fact inhabit any $\circ A$ with a closed term using the fix point construct:

$$\mathsf{fixpt}\ x.\mathsf{circ}(\mathsf{let} \circ y = x\ \mathsf{in}\ \mathsf{let} \circ z = y\ \mathsf{in}\ \mathsf{tm}z)$$

As a result, in metatheoretic reasoning, we cannot extract a proof of A from an arbitrary closed proof of $\circ A$. Doing so we could be stuck forever unrolling the above fixpoint term.

The solution we adopt is to index our monad with a natural number index and a non-standard elimination form that allows us to eliminate monads of smaller index. We no longer have the law $\circ \circ A \multimap \circ A$, but rather the weaker $\circ_m \circ_n A \multimap \circ_{(m+n)} A$. Metatheoretically, we can show that closed terms of $\circ_{\overline{0}} A$ are of the form $\mathsf{circ}(\mathsf{tm}\ M)$, while expressions of higher index must be composed of sub-expressions where the index strictly decreases.

Finally, the fixpoint construct introduces new hypotheses of type $\circ_{\overline{1}} A$. We change the specifications for instructions to consume proofs of

$$\Sigma k{:}\mathsf{nat}.\circ_k \mathsf{executable}\, pc$$

[8]These rules are reminiscent of the rules for the $\triangleright$ modality of [AMRV07] and the $\bigcirc$ modality of [HHWC07]

whose closed terms are meta-theoretically known to be pairs $\mathsf{pack}\ \langle \overline{k}, N \rangle$ where N has type $\circ_{\overline{k}} \mathsf{executable}\, pc$. Since the monad-elimination form requires subterms to have strictly lower index, the problematic term from above is now no longer well-typed, and we can decompose closed proofs of $\circ_{\overline{k}} A$ into proofs of A in finitely many steps.

3.3 Semantics

To type check LSL we use a bidirectional system for terms and expressions. The principal judgments are: $\vdash M \Leftarrow A$ to check a term M against a type A where both are inputs, and $\vdash M \Rightarrow A$ to infer the type A of M, with M as input and A as output. The other judgments are summarized in Table 1. The rules are presented in a form that lends itself to a typechecker implementation immediately. The full set of rules is in Appendix A, while we highlight some of the rules below.

Judgment	Meaning
$\vdash \Gamma\ \mathsf{ctx}$	Γ is a well-formed context
$\Gamma \vdash K\ \mathsf{kind}$	K is a well-formed kind
$\Gamma \vdash A : K$	A is a well-formed type family, and synthesizes kind K
$\vdash \alpha.A\mathsf{occ}\pm$	Type variable α occurs only positively (resp., negatively) in A
$\Gamma \vdash \Delta\ \mathsf{lctx}$	Δ is a well-formed linear context
$\Gamma; \Delta \vdash M \Rightarrow A$	M is well-formed and synthesizes type A
$\Gamma; \Delta \vdash M \overset{\text{wh}}{\Rightarrow} A$	M is well formed and synthesizes weak head-normal type A
$\Gamma; \Delta \vdash M \Leftarrow A$	M is well-formed and checks against type A
$\Gamma; \Delta \vdash M \overset{\text{wh}}{\Leftarrow} A$	M is well-formed and checks against weak head-normal type A
$\Gamma; \Delta \vdash E \div_M A$	E is a well-typed expression that checks against the type $\circ_M A$

Table 1: Typing Judgments

Inductive type formation In LSL, inductive types $\mu\alpha{:}A \to \mathsf{type}.B$ are formed at kind $A \to \mathsf{type}$. The type A is an index parameter. This allows for inductive definition of types that describe properties of the index object.

Judgment	Meaning
$\Gamma \vdash K_1 = K_2$	Kind equivalence
$\Gamma \vdash A_1 = A_2$	Type equivalence
$\Gamma, \Delta \vdash M_1 = M_2$	Term equivalence

Table 2: Equality Judgments

For example:

$$\texttt{divides}n \stackrel{\text{def}}{=} \mu\alpha{:}\mathsf{nat} \to \mathsf{type}.\\ \lambda m{:}\mathsf{nat}.(m =_{\mathsf{nat}} \overline{0}) \oplus (\Sigma m'{:}\mathsf{nat}.m =_{\mathsf{nat}} (n + m') \otimes \alpha\ m')$$

The type $\texttt{divides}\,3\,x$ is inhabited by terms witnessing that x is a multiple of 3.

We restrict the inductive type formation to exactly one index object of type A. If the index is not useful, the unit type 1 can be used as the index. If two index objects are needed, they can be passed via a single composite index object of type $\Sigma x{:}A_1 A_2$ where A_2 may depend on A_1.

Another restriction is that the types actually have to be inductive, that is the type variable α must not appear on the left of any function types. This is checked by the judgment $\vdash \alpha.A\ \mathsf{occ}+$.

The complete formation rule for inductive types is:

$$\frac{\Gamma \vdash A_1 : \mathsf{type} \quad \Gamma, \alpha{:}A_1 \to \mathsf{type} \vdash A_2 : A_1 \to \mathsf{type} \quad \vdash \alpha.A_2\ \mathsf{occ}+}{\Gamma \vdash \mu\alpha{:}A_1 \to \mathsf{type}.A_2 : A_1 \to \mathsf{type}}\ \mathrm{F}\mu$$

Inductive type introduction and elimination Members of inductive types are introduced by the term $\mathsf{roll}\ M$. Because of the indexed inductive types, the term is checked against the type $(\mu\alpha{:}B \to \mathsf{type}.C)\ M'$ where M' is the index object.

$$\frac{\Gamma; \Delta \vdash M \Leftarrow (C[A/\alpha])\ M'}{\Gamma; \Delta \vdash \mathsf{roll}\ M \stackrel{\text{wh}}{\Leftarrow} A\ M'}\ \mu\mathrm{I} \qquad \text{where } A = \mu\alpha{:}B \to \mathsf{type}.C$$

Ignoring the index object M', this is just the usual checking rule for inductive types, where the term M is checked against an unrolling of the inductive type with the whole type substituting for α in its body.

The elimination form for inductive types is the primitive recursion operator $\mathsf{pr}^{(\mu\alpha:B\to\mathsf{type}.C)}_{\alpha.D}(\alpha, f, x; u.M)$. This term ultimately stands for a function that takes in terms of type $A\ x$ and returns terms of type $D[A/\alpha]\ x$ where A is the inductive type $\mu\alpha{:}B \to \mathsf{type}.C$. The term M is the body of this

function, and f stands for the function itself, x is the index object. The variable u will have the term N substituted for it when the inductive function is applied to some $\mathsf{roll}\ N$ of the type $A\ x$. That is, u putatively has the type $C[A/\alpha]\ x$.

Of course nothing so far prevents the body M of the inductive function from uselessly rolling u back up and calling f and looping forever. So in order to keep LSL sound, we borrow the technique of [CW99] and restrict M to only calling f on sub-terms of u. We do so by checking the body M in a context where we do not substitute A back in for α in the type of u, but by keeping it abstract. The complete elimination rule is:

$$\frac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \mathcal{D}_3 \qquad \begin{array}{r}\Gamma, \alpha{:}B \to \mathsf{type},\\ f{:}\Pi y{:}B.\alpha\ y \multimap D\ y,\\ x{:}B;\\ \Delta, u\hat{:}C\ x \vdash M \Leftarrow D\ x\end{array}}{\Gamma; \cdot \vdash \mathsf{pr}^A_{\alpha.D}(\alpha, f, x; u.M) \Rightarrow \Pi x{:}B.A\ x \multimap D[A/\alpha]\ x}\ \mu\mathrm{E}$$

where the subderivations are:

$$\mathcal{D}_1 :: \Gamma \vdash \mu\alpha{:}B \to \mathsf{type}.C : B \to \mathsf{type} \qquad \mathcal{D}_2 :: \vdash \alpha.D\ \mathsf{occ}+$$
$$\mathcal{D}_3 :: \Gamma, \alpha{:}B \to \mathsf{type} \vdash D : B \to \mathsf{type}$$

Fixpoint induction One subtlety of the fixed point term $\mathsf{fix} f{:}A.M$ is that the hypothesis f is unrestricted, not linear: in the course of reasoning about a looping piece of code, the hypothesis may be used multiple times or not at all. But if f is to be unrestricted, since it stands for the whole fixpoint term, the whole term must make sense without using any resources.

$$\frac{\Gamma \vdash A : \mathsf{type} \quad \Gamma, x{:}\circ_{\overline{1}} A; \cdot \vdash M \Leftarrow A}{\Gamma; \cdot \vdash \mathsf{fix}\ x{:}A.M \Rightarrow A}\ \textsc{fix}$$

Suppose we wish to show $\mathtt{executable}\, m$ when we have the resource $\mathtt{codeat}\,(\mathtt{jmp}\,(\mathtt{imco}\, m))\, m\,(m+2)$ (recall that this is an abbreviation for $\mathsf{at}\,\mathtt{memrgn}\, m\, I_1 \otimes \mathsf{at}\,\mathtt{memrgn}\,(m+1)\, I_2$ where I_1, I_2 are the bytes that make up this particular jump instruction). We wish to use well-founded recursion, but we cannot do so at the executable type with the resources in the context. Instead we must abstract away the resources and apply the fixpoint at the type $\mathtt{codeat}\,(\mathtt{jmp}\,(\mathtt{imco}\, m)\, m\,(m+2)) \multimap \mathtt{executable}\, m$.

Expression checking The expression checking judgment $\Gamma; \Delta \vdash E \div_N C$ checks that the expression E has type C and its index is N. That is, $\mathsf{circ}\ E$ has type $\circ_N C$.

For expressions $\mathsf{tm}\ M$ that just return M, the index N must be exactly 0:

$$\frac{N\ \rightarrow^{\mathrm{wh}}\ \overline{0} \quad \Gamma;\Delta \vdash M \Leftarrow C}{\Gamma;\Delta \vdash \mathsf{tm}\ M \div_N C}\ \Leftarrow\mathrm{E}$$

To typecheck the monad elimination expression $\mathsf{let}\ \circ u = M\ \mathsf{in}\ E$, we need to establish that M has some monadic type $\circ_{N_1} A$ and then check that E has type C under the additional hypothesis that u has type A with some index N_2. But in addition we have to check that the indices N_1 and N_2 are both smaller than N. The index N_1 can be discovered from the type of M, however the index N_2, as well as the proofs that $N_1 < N$ and $N_2 < N$ cannot be inferred. Therefore, the entire elimination expression carries them as subscripts $\mathsf{let}_{(N_2,M_1,M_2)}\ \circ u = M\ \mathsf{in}\ E$:

$$\frac{\begin{array}{cc}\Gamma;\cdot \vdash N_2 \Leftarrow \mathsf{nat} & \Gamma;\Delta_1 \vdash M \stackrel{\mathrm{wh}}{\Rightarrow} \circ_{N_1} A \\ \Gamma;\cdot \vdash M_1 \Leftarrow (N_2 < N) & \Gamma;\cdot \vdash M_2 \Leftarrow (N_1 < N) \quad \Gamma;\Delta_2, u{:}A \vdash E \div_{N_1} C\end{array}}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{let}_{(N_2,M_1,M_2)}\ \circ u = M\ \mathsf{in}\ E \div_N C}\ \circ\mathrm{E}$$

3.4 Operational Semantics and Safety

We next wish to establish the soundness of LSL with respect to a particular machine model. Intuitively we want to correlate proofs of $\mathsf{executable}pc$ with a guarantee that a certain operational semantics for the machine model does not get stuck.

3.4.1 LSL Machine Model

The LSL machine has a byte-addressed memory and a fixed number of general-purpose registers, in addition to a program counter register. At each step, the machine decodes an instruction at the address in the program counter (or gets stuck if no decoding is possible) and takes a step corresponding to that instruction. The machine model is summarized in Figure 3 and the operational semantics for each decoded instruction are given in Table 3.

The operational semantics makes use of the following auxiliary judgments:

$\boxed{(H, R, o) \Downarrow n}$

$$\frac{}{(H,R,\mathsf{imco}(\overline{n})) \Downarrow \overline{n}} \qquad \frac{}{(H,R,\mathsf{rco}(r)) \Downarrow R[r]} \qquad \frac{(H,R,o) \Downarrow \overline{n}}{(H,R,\mathsf{mco}(o,\overline{m})) \Downarrow H[\overline{n}\overline{+}\overline{m}]}$$

$o \in$ Operands	$::= \texttt{imco}(\overline{n}) \mid \texttt{rco}(r) \mid \texttt{mco}(o, \overline{n})$
$d \in$ Destinations	$::= \texttt{rdest}(n) \mid \texttt{mdest}(o, \overline{n})$
$i \in$ Instructions	$::= \texttt{mov}\, d, o \mid \texttt{add}\, d, o_1, o_2 \mid \texttt{jnz}\, o_1, o_2 \mid \texttt{jmp}\, o \mid \texttt{halt}$
$H \in$ Heap	$::= \langle b_1, \ldots, b_{2^{wordWidth}-1} \rangle$
$R \in$ Registers	$::= \langle n_1, \ldots, n_k \rangle$
$pc \in$ Prog. Counter	$::= \overline{n}$
$S \in$ States	$::= (H, R, pc)$

Figure 3: The LSL machine, syntax

if $decode(H, pc) = (i, pc')$ and $i =$	then $(H, R, pc) \to S'$ where $S' =$
$\texttt{mov}\, d, o$	(H', R', pc') where $(H, R, o) \Downarrow n$, $(H, R)[d \leftarrow n] = (H', R')$
$\texttt{add}\, d, o_1, o_2$	(H', R', pc') where $(H, R, o_i) \Downarrow n_i$, $(H, R)[d \leftarrow (n_1 \overline{+} n_2)] = (H', R')$
$\texttt{jnz}\, o_1, o_2$	(H, R, pc'') where $(H, R, o_1) \Downarrow n$, $(H, R, o_2) \Downarrow a$, $pc'' = \begin{cases} pc' & \text{if} n = 0 \\ a & \text{otherwise} \end{cases}$
$\texttt{jmp}\, o$	(H, R, pc'') where $(H, R, o) \Downarrow pc''$

Table 3: The LSL machine operational semantics

$\boxed{(H, R)[d \leftarrow \overline{n}] = (H', R')}$

$$\frac{}{(H, R)[\mathsf{rdest}(r) \leftarrow \overline{n}] = (H, R[r \leftarrow \overline{n}])}$$

$$\frac{(H, R, o) \Downarrow \overline{n'}}{(H, R)[\mathsf{mdest}(o, \overline{m}) \leftarrow \overline{n}] = (H[\overline{n' \mathbin{+} m} \leftarrow \overline{m}], R)}$$

Where the operation $\overline{+}$ is addition modulo $2^{wordWidth}$.

We also assume that there is a (partial) function $decode(H, pc) = (i, pc')$ that decodes the bytes in the heap H between pc and $pc'-1$ to the instruction i.

A machine state (H, R, pc) is *terminal* if $decode(H, pc) = (\texttt{halt}, pc')$.

3.4.2 Good states and witnesses

We wish to use LSL to provide evidence that a good machine state does not get stuck. A state is good if it provides linear resources that allow us to show that the current program counter is pc and that at pc there is an instruction that we can execute:

$$\begin{aligned} \mathsf{good} &: \mathsf{type} \\ \mathsf{pc} &: \mathsf{nat} \to \mathsf{type} \\ \mathsf{goodi} &: \Pi pc{:}\mathsf{nat}.\mathsf{pc}\, pc \otimes \mathsf{executable}\, pc \multimap \mathsf{good} \end{aligned}$$

A new judgment $\vdash S \ \mathsf{wb}\ \Omega$ means that the machine state S is *witnessed by* a set or resources Ω. The elements of Ω are witnesses w_A. We also extend the typing rules with witness sets: $\Gamma; \Delta \vdash_\Omega M : A$, and a new typing rule for witnesses:

$$\Gamma; \cdot \vdash_{\{w_A\}} w_A : A$$

Witness sets behave similar to the linear context Δ — they are split by multiplicative connectives and are not subject to weakening or contraction.

$\boxed{\vdash S \ \mathsf{wb}\ \Omega}$

$$\frac{\begin{array}{l} \Omega_H = \{w_{\mathsf{at}\,\mathtt{memrgn}\,\overline{i}\,\overline{b_i}} | 1 \le i \le 2^{wordWidth} - 1\} \\ \Omega_R = \bigcup_{1 \le i \le k} \{w_{\mathsf{at}\,(\mathtt{regrgn}\ \overline{i})\,\overline{j}\,\overline{n_{ij}}} | 0 \le j < wordWidth\} \end{array}}{\vdash (\langle b_1, \ldots, b_{2^{wordWidth}-1}\rangle, \langle n_1, \ldots, n_k\rangle, pc)\ \mathsf{wb}\ (\Omega_H \cup \Omega_R \cup \{w_{\mathsf{pc}\,pc}\})}$$

Theorem 3.1 (Safety)**.** *If* $\vdash S \ \mathsf{wb}\ \Omega$ *and* $\Omega' \subseteq \Omega$ *and* $\cdot; \cdot \vdash_{\Omega'} M : \mathsf{good}$ *and if* $S \to^* S'$*, then either* S' *is terminal or it can take a step.*

Proof sketch The proof is by induction on the number of evaluation steps, using progress and preservation lemmas.

An additional lemma (a corollary of cut-elimination for LSL) shows that in an empty context (but with a witness set Ω), certain types (*e.g.*, $\mathsf{at}\ M\ M'\ M''\ M'''$) are only inhabited by witnesses. Once that is established, the proofs of both progress and preservation proceed by induction on the typing derivation that shows that good is inhabited, with inversion on $\vdash S\ \mathsf{wb}\ \Omega$ to establish the preconditions for the operational semantics.

4 An Object Language

In this section, I present RCTAL: the reference counting typed assembly language. The operational semantics of RCTAL manipulates heap values that are tagged with a reference count. The garbage collector for RCTAL removes heap values with a reference count of zero from the heap. The static semantics ensures that well-typed RCTAL programs do not go wrong by dereferencing dangling pointers.

RCTAL distinguishes between word values which may be either flat natural numbers or boxed pointers to heap values (which for simplicity are always pairs of word values). RCTAL is a RISC-like language — explicit load and store instructions are used to move word values from the first or second component of a heap value into the register file. This is a design choice due to reference counting: if the values being loaded or stored are pointers, the pointed-at object's reference count must be updated to reflect the new reference count.

In order to keep the amount of reference count manipulation down, we allow registers to conceptually point multiple times at a heap value. If a register has type $\tau_1 \times_n \tau_2$, we say that the register has *static reference count* n, and guarantee that the *dynamic reference count* is at least n. When a boxed value is copied from one register to another, the static reference count is split between the two copies, and the dynamic reference count need not be updated at all.

Since we do not track pointer aliasing statically, it would be incorrect to use a similar technique for pointers from within the heap itself. If we had two registers r1 and r2 each with type $\tau_1 \times (\tau_2 \times \tau_3)$, and we updated the second component of the heap value that r1 points to, we have no way of knowing if the second component of r2 is also updated (if the registers aliased one another) or not. As a result, if there were a static reference count on $\tau_2 \times \tau_3$, it would be possible to violate type safety. Consider, r2 with a reference count of 2 for its second component, and r1 aliased r2. If we overwrote the second component of r1 with a pointer to a freshly allocated cons cell with a dynamic reference count of 1, the static reference count via the second component of r2 would be greater than the dynamic reference count. If the dynamic reference count ever went to zero, the cons cell would be garbage collected, and the second component of r2 would be a dangling pointer.

Therefore, at the time of the load, we must increment the dynamic reference count of the loaded pointer. Dually, if we overwrite a component

of a heap value that happens to be a pointer, we must first decrement the reference count of the heap value that it used to point to.

We maintain the invariant that as long as the sum of the static reference counts for each address is positive, the corresponding heap value is not garbage and has a dynamic reference count that is at least as large as the sum. Of course the dynamic reference count may be larger if other values in the heap point to that heap value.

Conceptually, the static reference count exists only during typechecking, and RCTAL could be compiled to a lower-level language that erases the dynamic reference counts. Indeed, this is just what is done when we encode RCTAL in LSL. The linear resources corresponding to the RCTAL register file only record the actual values in each register, and not the static reference counts.

The operational semantics that we give in this section underspecify RCTAL in the sense that, for example, the allocation instruction only has the side condition that the newly allocated cons cell be fresh, and does not specify how we're to come up with such a value. The encoding of RCTAL in LSL implements the allocation instruction concretely via a lazy-reclamation dual-queue reference counting collector, thus removing the ambiguity.

4.1 Syntax

RCTAL has a kind system that classifies its types into those that may be stored in registers tr or in memory tm. Each type has a most specific kind that determines if the values of the type are flat (tflat) or boxed. Boxed types with a static reference count are classified by tbox_ω while those without are classified by tbox_1. A subkinding judgment (discussed below) divides the more specific kinds into register and memory kinds.

The types of values of RCTAL may be the nonsense type (inhabited by unspecified values), natural numbers, or pair types $\tau_1 \times_n \tau_2$, where n is a static reference count, or 1 if the pair has kind tbox_1, and τ_1 and τ_2 are each types of memory kind.

RCTAL has k registers, classified by a register file type Γ of k components each of which is a type of register kind.

RCTAL instruction take zero or more operands which may be either natural number literals or registers, they may also have a destination register.

RCTAL program states S are triples of a heap H, a register file R and a sequence I of instructions ι. The heap maps addresses a to heap values h, while the register file is a k-tuple of register values rv for some fixed k. Heap values are pairs of word-values w tagged with the dynamic reference

count c. Register values are either integer literals, addresses a tagged with a static reference count, or the nonsense value ns. Word values are either integer literals, or addresses (without a reference count tag). Instructions are discussed with their typing rules, below.

$$
\begin{array}{rll}
\kappa \in \text{Kinds} & ::= & \mathsf{tm} \mid \mathsf{tr} \mid \mathsf{tflat} \mid \mathsf{tbox}_1 \mid \mathsf{tbox}_\omega \\
\tau \in \text{Types} & ::= & \mathsf{ns} \mid \mathsf{nat} \mid \tau_1 \times_n \tau_2 \\
\Gamma \in \text{Register File Types} & ::= & \{\tau_1, \ldots, \tau_k\} \\
\Psi \in \text{Heap Types} & ::= & \{a_1 : \tau_1, \ldots, a_j : \tau_j\} \\
 & & \\
o \in \text{Operands} & ::= & \overline{n} \mid \mathsf{r}i \\
\iota \in \text{Instructions} & ::= & \mathsf{mov}\ \mathsf{r}d, o \\
 & \mid & \mathsf{mov}_c\ \mathsf{r}d, o \\
 & \mid & \mathsf{ld}_{\{1,2\}}\ \mathsf{r}d, \mathsf{r}s \\
 & \mid & \mathsf{st}_{\{1,2\}}\ \mathsf{r}d, o \\
 & \mid & \mathsf{add}\ \mathsf{r}d, rs_1, rs_2 \\
 & \mid & \mathsf{cons}\ \mathsf{r}d, o_1, o_2 \\
 & \mid & \mathsf{inc}\ \mathsf{r}i \\
 & \mid & \mathsf{dec}\ \mathsf{r}i \\
 & \mid & \mathsf{jnz}\ \mathsf{r}i, \ell \\
I \in \text{Instr. Seqs.} & ::= & \mathsf{halt} \mid \iota; I \\
w \in \text{Word Values} & ::= & \overline{n} \mid a \\
h \in \text{Cons Cells} & ::= & \langle w_1, w_2 \rangle^c \\
H \in \text{Heaps} & ::= & \{a_1 \mapsto h_1, \ldots, a_j \mapsto h_j\} \\
rv \in \text{Register Values} & ::= & \overline{n} \mid a^c \mid \mathsf{ns} \\
R \in \text{Register Files} & ::= & \{rv_1, \ldots, rv_k\} \\
S \in \text{States} & ::= & (H, R, I)
\end{array}
$$

Figure 4: Reference Counting TAL (Syntax)

4.2 Semantics

4.2.1 Static Semantics

The static semantics of RCTAL are given by several judgments that are summarized in Table 4.

Subkinding $\boxed{\kappa_1 \leq \kappa_2}$

The subkinding judgment primarily classifies the kinds of flat and boxed

Judgment	Meaning
$\kappa_1 \leq \kappa_2$	Kind κ_1 is a subkind of κ_2
$\vdash \tau : \kappa$	Type τ has kind κ
$\vdash \Gamma\ \mathsf{rft}$	Register file type Γ is well-formed
$\Gamma \vdash o : \tau$	Operand o has type τ
$\Gamma \vdash \iota \Rightarrow \Gamma'$	Insruction ι is well typed
	in rft Γ and produces a new rft Γ'
$\Gamma \vdash I$	Instruction sequence I is well-typed
$\vdash \Psi$	Heap type Ψ is well-formed
$\Psi \vdash w : \tau$	Heap word w has type τ
$\vdash h : \tau$	Cons-cell h has type τ
$\vdash H : \Psi$	Heap H has heap type Ψ
$\Psi \vdash rv : \tau$	Register value rv has type τ
$\Psi \vdash R : \Gamma$	Register file type R has type Γ
$\vdash S$	State S is well-typed

Table 4: Reference Counting TAL typing judgments

types into those that may be stored in registers and those that may be stored in memory. Flat types may belong in either. Boxed types with a static reference count (classified by tbox_ω) get stored in registers, and those without in memory.

$$\frac{}{\kappa \leq \kappa} \qquad \frac{}{\mathsf{tbox}_\omega \leq \mathsf{tr}} \qquad \frac{}{\mathsf{tflat} \leq \mathsf{tr}} \qquad \frac{}{\mathsf{tbox}_1 \leq \mathsf{tm}} \qquad \frac{}{\mathsf{tflat} \leq \mathsf{tm}}$$

Type formation $\boxed{\vdash \tau : \kappa}$

The rules for the type formation judgment are fairly standard. A kind subsumption theorem is admissible.

There are two rules for forming product types $\tau_1 \times_n \tau_2$. In both, the component types $\tau_{1,2}$ must have memory kind since cons cell belong in the heap. If n is 1, then the whole product type can be given kind tbox_1, that is, the cons cell does not have a static reference count. For any positive n, the type may be given kind tbox_ω, that is, the cons cell *does* have a static reference count.

$$\frac{\mathsf{tflat}\ \leq \kappa}{\vdash \mathsf{ns} : \kappa} \qquad \frac{\mathsf{tflat}\ \leq \kappa}{\vdash \mathsf{int} : \kappa} \qquad \frac{\vdash \tau_1 : \mathsf{tm} \quad \vdash \tau_2 : \mathsf{tm} \quad \mathsf{tbox}_1\ \leq \kappa}{\vdash \tau_1 \times_1 \tau_2 : \kappa}$$

$$\frac{\vdash \tau_1 : \mathsf{tm} \quad \vdash \tau_2 : \mathsf{tm} \quad \mathsf{tbox}_\omega\ \leq \kappa \quad 1 \leq n}{\vdash \tau_1 \times_n \tau_2 : \kappa}$$

Register File Type formation $\boxed{\vdash \Gamma\ \mathsf{rft}}$

All registers must have register kind.

$$\frac{\vdash \tau_i : \mathsf{tr} \quad (\text{for } 1 \le i \le k)}{\vdash \{\tau_1, \ldots, \tau_k\}\ \mathsf{rft}}$$

Henceforth we assume that all register file types are well-formed on the left of the turnstile, and preserve that property throughout the judgment.

Operand typing $\boxed{\Gamma \vdash o : \tau}$

The operand typing rules are unsurprising.

$$\frac{}{\Gamma \vdash \overline{n} : \mathsf{int}} \qquad \frac{1 \le i \le k}{\{\mathsf{r}1{:}\tau_1, \ldots, \mathsf{r}k{:}\tau_k\} \vdash \mathsf{r}i : \tau_i}$$

Instruction typing $\boxed{\Gamma \vdash \iota \Rightarrow \Gamma'}$

In the instruction typing judgment, Γ' is an output. We take care to ensure that it is well formed.

$$\frac{\Gamma \vdash o : \mathsf{int} \quad (\mathsf{r}d{:}\tau) \in \Gamma \quad \vdash \tau : \mathsf{tflat}}{\Gamma \vdash \mathsf{mov}\ \mathsf{r}d, o \Rightarrow \Gamma\{\mathsf{r}d{:}\mathsf{int}\}}$$

An unannotated move instruction is used for operands that are integers. In this, and all remaining rules, we check that the destination register holds a value that is flat. If it were a pointer, the pointed-at object would lose a reference when the move instruction overwrote it.

$$\frac{(\mathsf{r}s{:}\tau_1 \times_a \tau_2) \in \Gamma \quad (\mathsf{r}d{:}\tau) \in \Gamma \quad \vdash \tau : \mathsf{tflat} \quad a = b + c \quad 1 \le b \quad 1 \le c}{\Gamma \vdash \mathsf{mov}_c\ \mathsf{r}d, \mathsf{r}s \Rightarrow \Gamma\{\mathsf{r}d{:}\tau_1 \times_b \tau_2\}\{\mathsf{r}s{:}\tau_1 \times_c \tau_2\}}$$

If the object being moved is a pointer, we split its static reference count between the source operand (necessarily a register) and the destination register. Operationally, the dynamic reference count is not changed, since the total number of references remains the same.

$$\frac{(\mathsf{r}s{:}\tau_1 \times_a \tau_2) \in \Gamma \quad (\mathsf{r}d{:}\tau) \in \Gamma \quad \vdash \tau : \mathsf{tflat}}{\Gamma \vdash \mathsf{ld}_i\ \mathsf{r}d, \mathsf{r}s \Rightarrow \Gamma\{\mathsf{r}d{:}\tau_i\}}$$

The typing rule for a load instruction is unsurprising. Operationally, if the loaded value is a pointer, the pointed-at object will have its reference count incremented.

$$\frac{(\mathsf{r}s{:}\sigma_1 \times_{a+2} \sigma_2) \in \Gamma \quad (\mathsf{r}d{:}\tau_1 \times_b \tau_2) \in \Gamma}{\Gamma \vdash \mathsf{st}_i\ \mathsf{r}d, \mathsf{r}s\ \Rightarrow \Gamma\{\mathsf{r}s{:}\sigma_1 \times_{a+1} \sigma_2\}} \quad \text{where } \tau_i = \sigma_1 \times_1 \sigma_2$$

$$\frac{(\mathsf{r}s{:}\mathsf{int}) \in \Gamma \quad (\mathsf{r}d{:}\tau_1 \times_a \tau_2) \in \Gamma}{\Gamma \vdash \mathsf{st}_i\ \mathsf{r}d, \mathsf{r}s \Rightarrow \Gamma} \text{ where } \tau_i = \mathsf{int}$$

There are two typing rules for store instructions. If the value being stored is a pointer, then we must decrement its static reference count by one (and leave its dynamic reference count unchanged). Operationally, the value being overwritten must have its reference count decremented.

$$\frac{(\mathsf{r}i{:}\tau_1 \times_{a+2} \tau_2) \in \Gamma}{\Gamma \vdash \mathsf{dec}\ \mathsf{r}i \Rightarrow \Gamma\{\mathsf{r}i{:}\tau_1 \times_{a+1} \tau_2\}} \qquad \frac{(\mathsf{r}i{:}\tau_1 \times_1 \tau_2) \in \Gamma}{\Gamma \vdash \mathsf{dec}\ \mathsf{r}i \Rightarrow \Gamma\{\mathsf{r}i{:}\mathsf{ns}\}}$$

The decrement instruction is used to decrease the static and dynamic reference count of a heap value. There are two rules, depending on whether decrementing the static reference count would become zero or not after the decrement. If the static reference count was at least two, then we simply decrement the counts by one. If the static reference count was exactly one, then we make the heap value inaccessible by changing the type of the register to the nonsense type. Note that operationally only this second case needs to invoke the garbage collector. If the register had the last pointer to the cons cell, the dynamic reference count may have been one, and we could deallocate the cons cell.

$$\frac{\Gamma \vdash o_1 : \mathsf{int} \quad \Gamma \vdash o_2 : \mathsf{int} \quad (\mathsf{r}d{:}\tau) \in \Gamma \quad \vdash \tau : \mathsf{tflat}}{\Gamma \vdash \mathsf{add}\ \mathsf{r}d, o_1, o_2 \Rightarrow \Gamma\{\mathsf{r}d{:}\mathsf{int}\}}$$

$$\frac{(\mathsf{r}i{:}\tau_1 \times_a \tau_2) \in \Gamma}{\Gamma \vdash \mathsf{inc}\ \mathsf{r}i \Rightarrow \Gamma\{\mathsf{r}i{:}\tau_1 \times_{a+1} \tau_2\}} \qquad \frac{(\mathsf{r}i{:}\mathsf{int}) \in \Gamma \quad \ell : \Gamma \to 0}{\Gamma \vdash \mathsf{jnz}\ \mathsf{r}i, \ell \Rightarrow \Gamma}$$

These rules are unsurprising.

$$\frac{(\mathsf{r}d{:}\tau) \in \Gamma \quad \vdash \tau : \mathsf{tflat} \quad \Gamma \vdash o_1 : \tau_1 \quad \Gamma \vdash o_2 : \tau_2}{\Gamma \vdash \mathsf{cons}\ \mathsf{r}d, o_1, o_2 \Rightarrow (dec(o_1, \tau_1, dec(\Gamma, o_2, \tau_2)))\{\mathsf{r}d{:}tomem(\tau_1) \times_1 tomem(\tau_2)\}}$$

where $dec(o, \tau, \Gamma)$ and $tomem(\tau)$ are defined by:

$$dec(o, \tau, \Gamma) = \begin{cases} \Gamma\{\mathsf{r}i : \tau_1 \times_{1+c} \tau_2\} & \text{if } \tau = \tau_1 \times_{2+c} \tau_2 \text{ and } o = \mathsf{r}i \\ \Gamma & \text{otherwise} \end{cases}$$

$$tomem(\tau) = \begin{cases} \tau_1 \times_1 \tau_2 & \text{if } \tau = \tau_1 \times_c \tau_2 \\ \tau & \text{otherwise} \end{cases}$$

If the head or tail of a new cons cell are pointers, the cons cell that they point at gets an additional reference. So to keep the dynamic reference count unchanged, we decrease the static reference count of the corresponding operand by one.

Instruction Sequence Typing $\boxed{\Gamma \vdash I}$

$$\frac{}{\Gamma \vdash \mathsf{halt}} \qquad \frac{\Gamma \vdash \iota \Rightarrow \Gamma' \quad \Gamma' \vdash I}{\Gamma \vdash \iota; I}$$

We thread the register file type through the instruction sequence.

The preceeding judgments suffice for typechecking an RCTAL program. The remaining judgments are needed to prove type safety for RCTAL.

Heap Type Formation $\boxed{\vdash \Psi}$

$$\frac{\vdash \tau_i : \mathsf{tm} \quad (\text{for } 1 \leq i \leq j)}{\vdash \{a_1{:}\tau_1, \ldots, a_j{:}\tau_j\}}$$

Heap types are well-formed if every address has a type of memory kind.

Heap Word Typing $\boxed{\Psi \vdash w : \tau}$

$$\frac{}{\Psi \vdash \overline{n} : \mathsf{int}} \qquad \frac{}{\Psi \vdash a : \Psi(a)} \qquad \frac{}{\Psi \vdash \mathsf{ns} : \mathsf{ns}}$$

Unsurprising.

Cons cell typing $\boxed{\Psi \vdash h : \tau}$

$$\frac{\Psi \vdash w_i : \tau_i \quad (\text{for } i \in \{1,2\}) \quad c \geq 1}{\Psi \vdash \langle w_1, w_2 \rangle^c : \tau_1 \times_1 \tau_2}$$

The type of a cons cell does not mention its dynamic reference count c.

Heap Typing $\boxed{\vdash H : \Psi}$

$$\frac{\Psi \vdash H(a_i) : \tau_i \quad (\text{for } 1 \leq i \leq j)}{\vdash H : \Psi} \qquad \text{where } \Psi = \{a_1{:}\tau_1, \ldots, a_j{:}\tau_j\}$$

We typecheck a heap by checking that every address in the heap type has a value in the heap, and that that value is well-typed under the assumption that all the other addresses in the heap type are well-typed. This rule allows for cycles in the heap, and for there to be additional (possibly not well-formed) values in the heap that are not mentioned by Ψ.

Register Value Typing $\boxed{\Psi \vdash rv : \tau}$

$$\frac{}{\Psi \vdash \overline{n} : \mathsf{int}} \qquad \frac{\Psi(a) = \tau_1 \times_1 \tau_2}{\Psi \vdash a^c : \tau_1 \times_c \tau_2} \qquad \frac{}{\vdash \mathsf{ns} : \mathsf{ns}}$$

Register values that are pointers have an associated static reference count c that appears in its type.

Register File Typing $\boxed{\Psi \vdash R : \Gamma}$

$$\frac{\Psi \vdash rv_i : \tau_i \quad (\text{for } 1 \leq i \leq k)}{\Psi \vdash \{rv_1, \ldots, rv_k\} : \{\tau_1, \ldots, \tau_k\}}$$

State Typing $\boxed{\vdash S}$

$$\frac{\vdash \Psi \quad \vdash H : \Psi \quad \vdash \Gamma\ \mathsf{rft} \quad \Psi \vdash R : \Gamma \quad \Gamma \vdash I}{\vdash (H, R, I)}$$

4.2.2 Operational Semantics

The RCTAL operational semantics are given by the judgment $S \rightarrow S'$, specified in Appendix B.

4.3 Safety

We prove the usual preservation and progress theorems for RCTAL. One caveat is that preservation is only true for states S which respect a closure condition that for each address: the sum of the static reference counts to that address plus the number of references from the heap, must equal the dynamic reference count.

$$\begin{aligned} \mathit{Closed}(H, R, I) \iff\ & \forall (a \mapsto \langle -, - \rangle^c) \in H. \\ & \mathit{SCnt}(a, R) + \mathit{HCnt}(a, H) = c \end{aligned}$$

where

$$SCnt(a, \{rv_1, \ldots, rv_k\}) = \sum_{i=1}^{k} RVCnt(a, rv_i)$$

$$HCnt(a, H) = \sum_{a' \in \mathrm{dom}(H)} HVCnt(a, H(a'))$$

$$RVCnt(a, rv) = \begin{cases} c & \text{if } rv = a^c \\ 0 & \text{otherwise} \end{cases}$$

$$HVCnt(a, \langle w_1, w_2 \rangle^c) = WCnt(a, w_1) + WCnt(a, w_2)$$

$$WCnt(a, w) = \begin{cases} 1 & \text{if } w = a \\ 0 & \text{otherwise} \end{cases}$$

Theorem 4.1 (Preservation)**.** *If $Closed(S)$ and $\vdash S$ and $S \rightarrow S'$ then $\vdash S'$ and $Closed(S')$.*

Theorem 4.2 (Progress)**.** *If $\vdash S$ then either $S = (H, R, \mathsf{halt})$ or there exists an S' such that $S \rightarrow S'$.*

5 Conclusion

I've shown that LSL is an expressive linear logic suitable for reasoning about many aspects of machine code. I've also presented a reference-counted object language. It is a goal of my dissertation to implement a reference counting garbage collector along with a proof of its safety in LSL. Below I summarize some closely related work and give a summary of work I've already completed in support of my thesis.

5.1 Related work

Formal reasoning about garbage collection has been explored via several different avenues. Concise specifications of GCs as steps in the operational semantics of a programming language have been explored in [MH97]. With GC as an explicit step in the operational semantics, it can then be shown that GC does not change the results of a program. But the specification of a GC algorithm may be far removed from the low level implementation details of GC on a real machine. Consequently, we are concerned with relation the abstract specification of GC with its concrete realization on a particular architecture. The proof that we wish to carry out within LSL

is concerned precisely with the details that prior work sought to abstract away: value layout in memory, concrete representation of addresses, tagging bits, intermediate GC data structures, etc.

Such prior work has been carried out in the context of separation logic for a copying collector in [BTSR04]. Indeed much of our presentation of Hoare quads builds on the local reasoning ideas in that work. However rather than using linear logic as simply the logic of assertions for partial correctness specifications, LSL actually encodes Hoare quads as notational definitions. That means that we justify Hoare quads by exhibiting proofs of them as propositions within LSL, rather than appealing to the semantics of the logic. Further we intend to carry out our GC proof in a machine checkable manner.

A copying collector is implemented by [HHWC07] in their typed assembly language. In many ways the difference between their system and ours is similar to the difference between TAL and TALT: certain aspects of a machine such as the encoding of procedures on the heap are explicitly encoded in our system and are primitive notions in theirs. Furthermore, the two systems diverge in the fragments of linear logic that they utilize. Other technical differences include the presence of inductive types in our system and monad-moderated general recursive types in theirs. It is not clear whether the advantages of expressiveness of general recursive types are mitigated by having to thread reasoning through the monadic type when dealing with inductively defined structures.

Our choice of working with a reference counting collector, rather than a copying collector as in much of the existing work was initially motivated by pragmatic concerns: algorithms for reference counting collection are apparently simpler to implement. However there is reason to believe that the proof of correctness would actually be more involved. Part of the reason is that the reference counting invariant is in some sense stronger than the invariant for a copying collector: whereas in a copying collector every live object has *at least one* reference to it from the rootset or another live object, in a reference counting collector it must be the case that every object has a reference count that is *exactly equal* to the total number of references to it from another live object or from the rootset. So reasoning about reference counting may in fact be a more involved workout for a formal logic than a copying collector. Moreover, we're not aware of any other formal proofs to address this class of collectors.

5.2 Dissertation goals

My thesis is that LSL is suitable for formal reasoning about machine code implementations of garbage collectors. To that end, I propose to prove the safety of a reference counting garbage collector in LSL. To accomplish that task, I will produce the following artifacts:

1. A (paper) proof of soundness for LSL
2. A Twelf type checker for LSL terms
3. A proof assistant for developing proofs in LSL
4. A machine-checkable proof of correctness for a reference counting garbage collector

The proof assistant is needed because safety of machine code in LSL depends on a lot of defined notions such as arrays of bytes, lists of instructions, etc. All these data structures will likely require a large library of theorems. As a result, developing formal proofs entirely by hand is infeasible.

Dissertation timeline An approximate timeline of the immediate dissertation goals is presented below.

Task	Time	Completed
LSL metatheory	6 weeks	25%
RCTAL metatheory	2 weeks	50%
LSL checker implementation	1 month	90%
LSL proof assistant implementation	3 months	99%
RCTAL implementation proof in LSL	21 months	99%
Dissertation writing	2 months	1%

References

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 243–253, Boston, January 2000. 1

[AMRV07] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *Thirty-Fourth ACM Symposium on Principles of Programming Languages*, pages 109–122, Nice, France, January 2007. 2.1, 8

[Boe04] Hans-J. Boehm. The space cost of lazy reference counting. In *Thirty-First ACM Symposium on Principles of Programming Languages*, pages 210–219, Venice, Italy, January 2004. 2.5

[BTSR04] Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. In *Thirty-First ACM Symposium on Principles of Programming Languages*, pages 220–231, Venice, Italy, January 2004. 5.1

[CCD+02] Bor-Yuh Evan Chang, Karl Crary, Margaret DeLap, Robert Harper, Jason Liszka, Tom Murphy VII, and Frank Pfenning. Trustless grid computing in ConCert. In *Third International Workshop on Grid Computing*, volume 2536 of *Lecture Notes in Computer Science*, pages 112–125, Baltimore, Maryland, November 2002. 1

[CCP03] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131R, Carnegie Mellon University, School of Computer Science, December 2003. 1

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *Thirtieth ACM Symposium on Principles of Programming Languages*, pages 198–212, New Orleans, Louisiana, January 2003. 1

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In *1999 ACM International Conference on Functional Programming*, pages 233–248, Paris, September 1999. 3.3

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. 1

[HHWC07] Chris Hawblitzel, Heng Huang, Lea Wittie, and Juan Chen. A garbage-collecting typed assembly language. In *2007 ACM Workshop on Types in Language Design and Implementation*, pages 41–52, Nice, France, January 2007. 2.1, 8, 5.1

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. 2.3

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Conference on*

Functional Programming Languages and Computer Architecture, pages 66–77, La Jolla, California, June 1995. 1

[MH97] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1997. 5.1

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998. Extended version published as Cornell University technical report TR97-1651. 1

[Nec97] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, January 1997. 1

[NL98] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *1998 SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, June 1998. 1

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logic framework for deductive systems. In *Sixteenth International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag. 1

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Seventeenth IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, July 2002. 2.3

A LSL Typing rules

The complete LSL typing rules appear here. Selected rules are discussed in Section 3.3

Context formation $\boxed{\vdash \Gamma\ \mathsf{ctx}}$

$$\frac{}{\vdash \cdot\ \mathsf{ctx}}\ \textsc{CtxNil} \qquad \frac{\vdash \Gamma\ \mathsf{lctx} \quad \Gamma \vdash A : \mathsf{type}}{\vdash \Gamma, x{:}A\ \mathsf{ctx}}\ \textsc{CtxBind}$$

$$\frac{\vdash \Gamma\ \mathsf{ctx} \quad \Gamma \vdash K\ \mathsf{kind}}{\vdash \Gamma, \alpha{:}K\ \mathsf{ctx}}\ \textsc{CtxTBind}$$

In all subsequent judgments we assume all contexts Γ are well-formed and maintain that invariant when we add hypotheses.

Kind formation $\boxed{\Gamma \vdash K\ \mathsf{kind}}$

$$\frac{}{\Gamma \vdash \mathsf{type}\ \mathsf{kind}}\ \textsc{KFtype} \qquad \frac{\Gamma \vdash A : \mathsf{type} \quad \Gamma, x{:}A \vdash K\ \mathsf{kind}}{\Gamma \vdash \Pi x{:}A.K\ \mathsf{kind}}\ \textsc{KF}\Pi$$

Type family formation $\boxed{\Gamma \vdash A : K}$

$$\frac{\alpha{:}K \in \Gamma}{\Gamma \vdash \alpha : K}\ \textsc{Fvar} \qquad \frac{\Gamma \vdash K\ \mathsf{kind} \quad \Gamma, \alpha{:}K \vdash A : \mathsf{type}}{\Gamma \vdash \forall \alpha{:}K.A : \mathsf{type}}\ \mathrm{F}\forall$$

$$\frac{\Gamma \vdash A : \Pi x{:}A'.K \quad \Gamma; \cdot \vdash M \Leftarrow A'}{\Gamma \vdash A\ M : K[M/x]}\ \textsc{Fapp} \qquad \frac{\Gamma \vdash A_1 : \mathsf{type} \quad \Gamma, x{:}A_1 \vdash A_2 : K}{\Gamma \vdash \lambda x{:}A_1.A_2 : \Pi x{:}A_1.K}\ \mathrm{F}\lambda$$

$$\frac{}{\Gamma \vdash \mathsf{nat} : \mathsf{type}}\ \textsc{Fnat} \qquad \frac{}{\Gamma \vdash 1 : \mathsf{type}}\ \mathrm{F1} \qquad \frac{\Gamma \vdash A_1 : \mathsf{type} \quad \Gamma \vdash A_2 : \mathsf{type}}{\Gamma \vdash A_1 \otimes A_2 : \mathsf{type}}\ \mathrm{F}\otimes$$

The rules $\mathrm{F}{\multimap}, \mathrm{F}\&, \mathrm{F}\oplus$ are the same as $\mathrm{F}\otimes$. The rules $\mathrm{F}\top, \mathrm{F}0$ are the same as F1.

$$\frac{\Gamma \vdash A : \mathsf{type}}{\Gamma \vdash\ !A : \mathsf{type}}\ \mathrm{F}! \qquad \frac{\Gamma \vdash A_1 : \mathsf{type} \quad \Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}}{\Gamma \vdash \Pi x{:}A_1.A_2 : \mathsf{type}}\ \mathrm{F}\Pi$$

The rule $\mathrm{F}\Sigma$ is the same as $\mathrm{F}\Pi$.

$$\frac{\Gamma \vdash A_1 : \mathsf{type} \quad \Gamma, \alpha{:}A_1 \to \mathsf{type} \vdash A_2 : A_1 \to \mathsf{type} \quad \vdash \alpha.A_2\ \mathsf{occ}+}{\Gamma \vdash \mu\alpha{:}A_1 \to \mathsf{type}.A_2 : A_1 \to \mathsf{type}}\ \mathrm{F}\mu$$

$$\frac{\Gamma \vdash A_1 \oplus A_2 : \mathsf{type} \quad \Gamma; \cdot \vdash M \overset{\mathrm{wh}}{\Leftarrow} A_1 \oplus A_2}{\Gamma \vdash \mathsf{isl}_{A_1 \oplus A_2}\ M : \mathsf{type}}\ \textsc{Fisl}$$

$$\frac{\Gamma; \cdot \vdash M \overset{\mathrm{wh}}{\Leftarrow} \mathsf{nat} \quad \Gamma \vdash A : \mathsf{type}}{\Gamma \vdash \circ_M A : \mathsf{type}}\ \mathrm{F}\circ$$

Type variable occurrence $\boxed{\vdash \alpha.A\ \mathsf{occ}\pm}$

In the rules for this judgment, $\pm$ in the premises stands consistently for either $+$ or $-$, whichever appears in the conclusion; $\mp$ stands consistently for the opposite polarity of the one in the conclusion.

$$\frac{}{\vdash \alpha.\alpha\ \mathsf{occ}+}\ \textsc{OccVar} \qquad \frac{}{\vdash \alpha.1\ \mathsf{occ}\pm}\ \textsc{Occ}1 \qquad \frac{\vdash \alpha.A_1\ \mathsf{occ}\pm \quad \vdash \alpha.A_2\ \mathsf{occ}\pm}{\vdash \alpha.A_1 \otimes A_2\ \mathsf{occ}\pm}\ \textsc{Occ}\otimes$$

$$\frac{\vdash \alpha.A_1\ \mathsf{occ}\mp \quad \vdash \alpha.A_2\ \mathsf{occ}\pm}{\vdash \alpha.A_1 \multimap A_2\ \mathsf{occ}\pm}\ \textsc{Occ}{\multimap} \qquad \frac{\vdash \alpha.A\ \mathsf{occ}\pm}{\vdash \alpha.!A\ \mathsf{occ}\pm}\ \textsc{Occ}!$$

Note that there is no variable rule for negative occurrences, and that the polarity changes in the domain of $\multimap$. The rules $\textsc{Occ}\top, \textsc{Occ}0$ are the same as $\textsc{Occ}1$.The rules $\textsc{Occ}\,\&\,, \textsc{Occ}\oplus, \textsc{Occ}\Sigma$ are the same as $\textsc{Occ}\otimes$. The rule $\textsc{Occ}\Pi$ is the same as $\textsc{Occ}\ \multimap$. The rule $\textsc{Occ}\circ$ is the same as $\textsc{Occ}!$.

Linear context formation $\boxed{\Gamma \vdash \Delta\ \mathsf{lctx}}$

$$\frac{}{\Gamma \vdash \cdot\ \mathsf{lctx}}\ \textsc{LctxNil} \qquad \frac{\Gamma \vdash \Delta\ \mathsf{lctx} \quad \Gamma \vdash A : \mathsf{type}}{\Gamma \vdash \Delta, u\hat{:}A\ \mathsf{lctx}}\ \textsc{LctxBind}$$

In the judgments that follow, we assume that linear contexts are well-formed and check any additional hypotheses when we add them.

Term checking $\boxed{\Gamma; \Delta \vdash M \Leftarrow A}$

$$\frac{A \rightarrow^{\mathrm{wh}} A' \quad \Gamma; \Delta \vdash M \overset{\mathrm{wh}}{\Leftarrow} A'}{\Gamma; \Delta \vdash M \Leftarrow A}\ \overset{\mathrm{wh}}{\Leftarrow}\Leftarrow$$

$\boxed{\Gamma; \Delta \vdash M \overset{\mathrm{wh}}{\Leftarrow} A}$

$$\frac{}{\Gamma; \cdot \vdash \star \overset{\mathrm{wh}}{\Leftarrow} 1}\ 1\mathrm{I} \qquad \frac{\Gamma; \Delta_1 \vdash M_1 \overset{\mathrm{wh}}{\Rightarrow} 1 \quad \Gamma; \Delta_2 \vdash M_2 \overset{\mathrm{wh}}{\Leftarrow} A_2}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{let}\ \star = M_1\ \mathsf{in}\ M_2 \overset{\mathrm{wh}}{\Leftarrow} A_2}\ 1\mathrm{E}$$

$$\frac{\Gamma; \Delta \vdash M_1 \Leftarrow A_1 \quad \Gamma; \Delta \vdash M_2 \Leftarrow A_2}{\Gamma; \Delta_1, \Delta_2 \vdash M_1 \otimes M_2 \overset{\mathrm{wh}}{\Leftarrow} A_1 \otimes A_2}\ \otimes\mathrm{I}$$

$$\frac{\Gamma; \Delta_1 \vdash M_1 \overset{\mathrm{wh}}{\Rightarrow} A_1 \otimes A_2 \quad \Gamma; \Delta_2, u_1\hat{:}A_1, u_2\hat{:}A_2 \vdash M_2 \overset{\mathrm{wh}}{\Leftarrow} C}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{let}\ u_1 \otimes u_2 = M_1\ \mathsf{in}\ M_2 \overset{\mathrm{wh}}{\Leftarrow} C}\ \otimes\mathrm{E}$$

$$\frac{\Gamma, \alpha{:}K; \Delta \vdash M \Leftarrow A}{\Gamma; \Delta \vdash \Lambda\alpha.M \overset{\mathrm{wh}}{\Leftarrow} \forall\alpha{:}K.A}\ \forall\mathrm{I} \qquad \frac{\Gamma; \Delta, u\hat{:}A_1 \vdash M \Leftarrow A_2}{\Gamma; \Delta \vdash \hat{\lambda}u.M \overset{\mathrm{wh}}{\Leftarrow} A_1 \multimap A_2}\ \multimap\mathrm{I}$$

$$\frac{\Gamma, x{:}A_1; \Delta \vdash M \Leftarrow A_2}{\Gamma; \Delta \vdash \lambda x.M \overset{\mathrm{wh}}{\Leftarrow} \Pi x{:}A_1.A_2}\ \Pi\mathrm{I}$$

$$\frac{}{\Gamma; \Delta \vdash \langle\rangle \overset{\mathrm{wh}}{\Leftarrow} \top}\ \top\mathrm{I} \qquad \frac{\Gamma; \Delta \vdash M_1 \Leftarrow A_1 \quad \Gamma; \Delta \vdash M_2 \Leftarrow A_2}{\Gamma; \Delta \vdash \langle M_1, M_2\rangle \overset{\mathrm{wh}}{\Leftarrow} A_1 \,\&\, A_2}\ \&\mathrm{I}$$

$$\frac{\Gamma; \Delta \vdash M \overset{\mathrm{wh}}{\Rightarrow} 0}{\Gamma; \Delta \vdash \mathsf{any}\ M \overset{\mathrm{wh}}{\Leftarrow} A}\ 0\mathrm{E} \qquad \frac{\Gamma; \Delta \vdash M \Leftarrow A_1}{\Gamma; \Delta \vdash \mathsf{inl}\ M \overset{\mathrm{wh}}{\Leftarrow} A_1 \oplus A_2}\ \oplus\mathrm{I}_1$$

$$\frac{\Gamma; \Delta \vdash M \Leftarrow A_2}{\Gamma; \Delta \vdash \mathsf{inr}\ M \overset{\mathrm{wh}}{\Leftarrow} A_1 \oplus A_2}\ \oplus\mathrm{I}_2$$

$$\frac{\Gamma; \Delta_1 \vdash M \overset{\mathrm{wh}}{\Rightarrow} A_1 \oplus A_2 \quad \Gamma; \Delta_2, u\hat{:}A_i \vdash M_i \overset{\mathrm{wh}}{\Leftarrow} C}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\ u \Rightarrow M_1 \mid \mathsf{inr}\ u \Rightarrow M_2 \overset{\mathrm{wh}}{\Leftarrow} C}\ \oplus\mathrm{E}$$

$$\frac{\Gamma; \cdot \vdash M \Leftarrow A}{\Gamma; \cdot \vdash\ !M \overset{\mathrm{wh}}{\Leftarrow} !A}\ !\mathrm{I} \qquad \frac{\Gamma; \Delta_1 \vdash M_1 \overset{\mathrm{wh}}{\Rightarrow} !A \quad \Gamma, x{:}A; \Delta_2 \vdash M_2 \overset{\mathrm{wh}}{\Leftarrow} C}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{let}\ !x = M_1\ \mathsf{in}\ M_2 \overset{\mathrm{wh}}{\Leftarrow} C}\ !\mathrm{E}$$

$$\frac{\Gamma;\cdot \vdash M_1 \Leftarrow A_1 \quad \Gamma;\Delta \vdash M_2 \Leftarrow A_2[M_1/x]}{\Gamma;\Delta \vdash \mathsf{pack}\ \langle M_1, M_2\rangle \overset{\text{wh}}{\Leftarrow} \Sigma x{:}A_1.A_2}\ \Sigma\text{I}$$

$$\frac{\Gamma;\Delta_1 \vdash M_1 \overset{\text{wh}}{\Rightarrow} \Sigma x{:}A_1.A_2 \quad \Gamma, x{:}A_1;\Delta_2, u\hat{:}A_2 \vdash M_2 \overset{\text{wh}}{\Leftarrow} C}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{let\ pack}\ \langle x,u\rangle = M_1\ \mathsf{in}\ M_2 \overset{\text{wh}}{\Leftarrow} C}\ \Sigma\text{E}$$

$$\frac{\Gamma;\Delta \vdash M \Leftarrow (A_2[A/\alpha])\ M'}{\Gamma;\Delta \vdash \mathsf{roll}\ M \overset{\text{wh}}{\Leftarrow} A\ M'}\ \mu\text{I} \qquad \text{where } A = \mu\alpha{:}A_1 \to \mathsf{type}.A_2$$

$$\frac{\Gamma;\Delta \vdash M \Leftarrow A_1 \quad \Gamma,\Delta \vdash \mathsf{inl}\ M = M'}{\Gamma;\Delta \vdash \mathsf{isli}\ M \overset{\text{wh}}{\Leftarrow} \mathsf{isl}_{A_1\oplus A_2}\ M'}\ \text{ISL}\text{I} \qquad \frac{\Gamma;\Delta \vdash E \div_M A}{\Gamma;\Delta \vdash \mathsf{circ}\ E \overset{\text{wh}}{\Leftarrow} \circ_M A}\ \circ\text{I}$$

$$\frac{}{\Gamma;\Delta \vdash \overline{n} \overset{\text{wh}}{\Leftarrow} \mathsf{nat}}\ \text{NAT}\text{I} \qquad \frac{\Gamma;\Delta_1 \vdash M_1 \Rightarrow A \quad \Gamma;\Delta_2, u\hat{:}A \vdash M_2 \overset{\text{wh}}{\Leftarrow} C}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{cut}\ u = M_1\ \mathsf{in}\ M_2 \overset{\text{wh}}{\Leftarrow} C}\ \text{CUT}$$

$$\frac{\Gamma;\cdot \vdash M_1 \Rightarrow A \quad \Gamma, x{:}A;\Delta \vdash M_2 \overset{\text{wh}}{\Leftarrow} C}{\Gamma;\Delta \vdash \mathsf{cut!}\ x = M_1\ \mathsf{in}\ M_2 \overset{\text{wh}}{\Leftarrow} C}\ \text{CUT!}$$

$$\frac{\Gamma;\Delta \vdash M \overset{\text{wh}}{\Rightarrow} A' \quad \Gamma,\Delta \vdash A' = A}{\Gamma;\Delta \vdash M \overset{\text{wh}}{\Leftarrow} A}\ \overset{\text{wh}}{\Rightarrow}\overset{\text{wh}}{\Leftarrow}$$

Term inference $\boxed{\Gamma;\Delta \vdash M \overset{\text{wh}}{\Rightarrow} A}$

$$\frac{\vdash M \Rightarrow A' \quad A' \ \to^{\text{wh}}\ A}{\Gamma;\Delta \vdash M \overset{\text{wh}}{\Rightarrow} A}\ \Rightarrow\overset{\text{wh}}{\Rightarrow}$$

$\boxed{\Gamma;\Delta \vdash M \Rightarrow A}$

$$\frac{x{:}A \in \Gamma}{\Gamma;\cdot \vdash x \Rightarrow A}\ \text{VAR} \qquad \frac{}{\Gamma; u{:}A \vdash u \Rightarrow A}\ \text{LVAR}$$

$$\frac{\Gamma;\Delta_1 \vdash M_1 \overset{\text{wh}}{\Rightarrow} A_1 \multimap A_2 \quad \Gamma;\Delta_2 \vdash M_2 \Leftarrow A_1}{\Gamma;\Delta_1,\Delta_2 \vdash M_1\hat{}M_2 \Rightarrow A_2}\ \multimap\text{E}$$

$$\frac{\Gamma;\Delta \vdash M_1 \overset{\text{wh}}{\Rightarrow} \Pi x{:}A_1.A_2 \quad \Gamma;\cdot \vdash M_2 \Leftarrow A_1}{\Gamma;\Delta \vdash M_1\ M_2 \Rightarrow A_2[M_2/x]}\ \Pi\text{E} \qquad \frac{\Gamma;\Delta \vdash M \overset{\text{wh}}{\Rightarrow} A_1 \,\&\, A_2}{\Gamma;\Delta \vdash \pi_i M \Rightarrow A_i}\ \&\,\text{E}_i$$

$$\frac{\Gamma;\Delta \vdash M \overset{\text{wh}}{\Rightarrow} \forall\alpha{:}K.A_2 \quad \Gamma \vdash A_1 : K' \quad \Gamma \vdash K = K'}{\Gamma;\Delta \vdash M\ [A_1] \Rightarrow A_2[A_1/\alpha]}\ \forall\text{E}$$

$$\frac{\Gamma;\Delta \vdash M \overset{\text{wh}}{\Rightarrow} \mathsf{isl}_A\ M' \quad M' \ \to^{\text{wh}}\ \mathsf{inr}\ M''}{\Gamma;\Delta \vdash \mathsf{isle}\ M \Rightarrow 0}\ \text{ISL}\text{E}$$

In the following rule, let $A = \mu\alpha{:}B \to \mathsf{type}.C$ and $\Gamma' = \Gamma, \alpha{:}B \to \mathsf{type}, f{:}\Pi x{:}B.\alpha\ x \multimap D\ x$

$$\Gamma \vdash A : B \to \mathsf{type}$$
$$\vdash \alpha.D\ \mathsf{occ}+$$
$$\frac{\Gamma, \alpha{:}B \to \mathsf{type} \vdash D : B \to \mathsf{type} \quad \Gamma', x{:}B;\Delta, u\hat{:}C\ x \vdash M \Leftarrow D\ x}{\Gamma;\cdot \vdash \mathsf{pr}^A_{\alpha.D}(\alpha, f, x; u.M) \Rightarrow \Pi x{:}B.C\ x \multimap D[A/\alpha]\ x}\ \mu\text{E}$$

$$\frac{\Gamma \vdash A : \mathsf{type} \quad \Gamma;\cdot \vdash M_z \Leftarrow A \quad \Gamma, x{:}\mathsf{nat}, f{:}A;\cdot \vdash M_s \Leftarrow A}{\Gamma;\cdot \vdash \mathsf{prnat}_A\ M_z\ (x, f.M_s)\ \Rightarrow \mathsf{nat} \to A}\ \textsc{nat}\mathrm{E}$$

$$\frac{\Gamma \vdash A : \mathsf{type} \quad \Gamma, x{:}\circ_{\mathrm{I}} A;\cdot \vdash M \Leftarrow A}{\Gamma;\cdot \vdash \mathsf{fix}\ x{:}A.M \Rightarrow A}\ \textsc{fix} \qquad \frac{\Gamma \vdash A : \mathsf{type} \quad \Gamma;\Delta \vdash M \Leftarrow A}{\Gamma;\Delta \vdash M{:}A \Rightarrow A}\ \textsc{ann}$$

Expression checking $\boxed{\Gamma;\Delta \vdash M \div_{M'} A}$

$$\frac{M' \to^{\mathrm{wh}} \overline{0} \quad \Gamma;\Delta \vdash M \Leftarrow A}{\Gamma;\Delta \vdash \mathsf{tm}\ M \div_{M'} A}\ \Leftarrow\mathrm{E}$$

$$\frac{\Gamma;\Delta_1 \vdash M \overset{\mathrm{wh}}{\Rightarrow} \circ_{N_1} A \qquad \Gamma;\cdot \vdash N_2 \Leftarrow \mathsf{nat} \quad \Gamma;\cdot \vdash M_1 \Leftarrow (N_1 < N) \quad \Gamma;\cdot \vdash M_2 \Leftarrow (N_2 < N) \quad \Gamma;\Delta_2, u\hat{:}A \vdash E \div_{N_2} C}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{let}_{(N_2,M_1,M_2)}\ \circ u = M\ \mathsf{in}\ E \div_N C}\ \circ\mathrm{E}$$

$$\frac{\Gamma;\Delta_1 \vdash M \overset{\mathrm{wh}}{\Rightarrow} !A \quad \Gamma, x{:}A;\Delta_2 \vdash E \div_{M'} C}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{let}\ !x = M\ \mathsf{in}\ E \div_{M'} C}\ !\mathrm{E}\mathrm{E}$$

$$\frac{\Gamma;\Delta_1 \vdash M \overset{\mathrm{wh}}{\Rightarrow} A_1 \otimes A_2 \quad \Gamma;\Delta_2, u_1{:}A_1, u_2{:}A_2 \vdash E \div_{M'} C}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{let}\ u_1 \otimes u_2 = M\ \mathsf{in}\ E \div_{M'} C}\ \otimes\mathrm{E}\mathrm{E}$$

And analogous ΣEE and $\oplus$EE.

B RCTAL Operational Semantics

If $R = \{rv_1, \ldots, rv_k\}$ then

$$R[\mathsf{r}i] = rv_i$$
$$R[\mathsf{r}i \leftarrow rv'] = \{rv_1, \ldots, rv_{i-1}, rv', rv_{i+1}, \ldots, rv_k\}$$

We extend $R[\mathsf{r}i]$ to operands o as:

$$R[o] = \begin{cases} R[\mathsf{r}i] & \text{if } o = \mathsf{r}i, \\ \overline{n} & \text{if } o = \overline{n}. \end{cases}$$

To convert word-values to register values, we define:

$$torv(w) = \begin{cases} a^1 & \text{if } w = a, \\ \overline{n} & \text{if } w = \overline{n}. \end{cases}$$

Conversely,

$$fromrv(rv) = \begin{cases} a & \text{if } rv = a^c, \\ \overline{n} & \text{if } rv = \overline{n}. \end{cases}$$

For an address a, if $H(a) = \langle w_1, w_2 \rangle^c$ then

$$inc(a, H) = H\{a \mapsto \langle w_1, w_2 \rangle^{c+1}\}$$

Furthermore, we extend inc to word-values as:

$$inc(w, H) = \begin{cases} inc(a, H) & \text{if } w = a, \\ H & \text{if } w = \overline{n}. \end{cases}$$

Define $update_i(h, w')$ by

$$update_i(\langle w_1, w_2 \rangle^c, w') = \begin{cases} \langle w', w_2 \rangle^c & \text{if } i = 1, \\ \langle w_1, w' \rangle^c & \text{if } i = 2. \end{cases}$$

If $(a \mapsto hv) \in H$, write $H \backslash a$ for the heap without that address and heap value.

If $H(a) = \langle w_1, w_2 \rangle^{1+c}$ then $dec(a, H) = H\{a \mapsto \langle w_1, w_2 \rangle^c\}$. Also extend to other memory words by:

$$dec(w, H) = \begin{cases} dec(a', H) & \text{if } w = a', \\ H & \text{otherwise.} \end{cases}$$

Additionally, for register values rv, define the unary operation $dec(rv)$ by:

$$dec(rv) = \begin{cases} a^{1+c} & \text{if } rv = a^{2+c} \\ \overline{n} & \text{if } rv = \overline{n} \end{cases}$$

With these definitions, the operational semantics of RCTAL are given by the following judgment.

$\boxed{S \ \rightarrow \ S'}$

$$\frac{}{(H, R, \mathsf{mov}\ \mathsf{r}d, o; I) \ \rightarrow \ (H, R[\mathsf{r}d \leftarrow R[o]], I)}$$

$$\frac{R[\mathsf{r}s] = a^b \quad c + d = b}{(H, R, \mathsf{mov}_c\ \mathsf{r}d, \mathsf{r}s; I) \ \rightarrow \ (H, R[\mathsf{r}s \leftarrow a^c][\mathsf{r}d \leftarrow a^d], I)}$$

$$\frac{R[\mathsf{r}s] = a \quad H(a) = \langle w_1, w_2 \rangle^c}{(H, R, \mathsf{ld}_i\mathsf{r}d, \mathsf{r}s; I) \ \rightarrow \ (inc(w_i, H), R[\mathsf{r}d \leftarrow torv(w_i)], I)}$$

$$\frac{R[\mathsf{r}s] = \overline{n} \quad R[\mathsf{r}d] = a}{(H, R, \mathsf{st}_i\mathsf{r}d, \mathsf{r}s; I) \ \rightarrow \ (H\{a \mapsto update_i(H(a), \overline{n})\}, R, I)}$$

$$\frac{R[\mathsf{r}s] = a^{2+c} \quad R[\mathsf{r}d] = a' \quad H(a') = \langle w_1, w_2 \rangle^-}{(H, R, \mathsf{st}_i\mathsf{r}d, \mathsf{r}s; I) \ \rightarrow \ ((dec(w_i, H))\{a' \mapsto update_i(H(a'), a)\}, R[\mathsf{r}s \leftarrow a^{1+c}], I)}$$

$$\frac{}{(H, R, \mathsf{add}\ \mathsf{r}d, \mathsf{r}s_1, \mathsf{r}s_2; I) \ \rightarrow \ (H, R[\mathsf{r}d \leftarrow R[\mathsf{r}s_1] + R[\mathsf{r}s_2]], I)}$$

$$\frac{a \notin \mathrm{dom}(H)}{(H, R, \mathsf{cons}\ \mathsf{r}d, o_1, o_2; I) \ \rightarrow \ (H', R', I)}$$

where $H' = H\{a \mapsto \langle fromrv(R[o_1]), fromrv(R[o_2]) \rangle^1\}$,
and $R' = R[\mathsf{r}d \leftarrow a^1][o_1 \leftarrow dec(o_1)][o_2 \leftarrow dec(o_2)]$.

$$\frac{R[\mathsf{r}i] = a^c}{(H, R, \mathsf{inc}\ \mathsf{r}i; I)\ \rightarrow\ (inc(a, H), R[\mathsf{r}i \leftarrow a^{c+1}], I)}$$

$$\frac{R[\mathsf{r}i] = a^{c+1}}{(H, R, \mathsf{dec}\ \mathsf{r}i; I)\ \rightarrow\ (dec(a, H), R[\mathsf{r}i \leftarrow a^{c}], I)}$$

$$\frac{R[\mathsf{r}i] = a^{1}}{(H, R, \mathsf{dec}\ \mathsf{r}i; I)\ \rightarrow\ (dec(a, H), R[\mathsf{r}i \leftarrow \mathsf{ns}], I)}$$

$$\frac{R[\mathsf{r}i] = \overline{0}}{(H, R, \mathsf{jnz}\ \mathsf{r}i, \ell; I)\ \rightarrow\ (H, R, I)} \qquad \frac{R[\mathsf{r}i] = \overline{n+1} \quad (\ell) = I'}{(H, R, \mathsf{jnz}\ \mathsf{r}i, \ell; I)\ \rightarrow\ (H, R, I')}$$

The garbage collection specification is:

$$\frac{H(a) = \langle w_1, w_2 \rangle^0}{(H, R, I)\ \rightarrow^{GC}\ (dec(w_2, dec(w_1, H \backslash a)), R, I)}$$