A Typed Interface for Garbage Collection

Joseph C. Vanderwaart Karl Crary

Carnegie Mellon University

Abstract

An important consideration for certified code systems is the interaction of the untrusted program with the runtime system, most notably the garbage collector. Most certified code systems that treat the garbage collector as part of the trusted computing base dispense with this issue by using a collector whose interface with the program is simple enough that it does not pose any certification challenges. However, this approach rules out the use of many sophisticated high-performance garbage collectors. We present the language LGC, whose type system is capable of expressing the interface of a modern high-performance garbage collector. We use LGC to describe the interface to one such collector, which involves a substantial amount of programming at the type constructor level of the language.

Categories and Subject Descriptors

D.3.m [**Programming Languages**]: Miscellaneous; D.3.4 [**Programming Languages**]: Processors—*Compilers, Memory management*

General Terms

Languages, Security

Keywords

Type systems, Typed compilation, Certified code, Garbage collection

1 Introduction

In a certified code system, executable programs shipped from a producer to a client are accompanied by certificates that provide evidence of their safety. The validity of a certificate, which can be mechanically verified by the client, implies that the associated program is safe to execute. Examples of certified code frameworks include Typed Assembly Language [6] and Proof-Carrying Code [7, 8].

Most past research on certified code has focused on the safety of the untrusted mobile code itself. However, it is also important to consider the safety implications of the runtime system to which that code is linked. There are two options for dealing with this issue. One choice is to treat the runtime system as part of the untrusted code, and certify its safety. The other choice is to simply assume the runtime system is correct—*i.e.*, treat it as part of the *trusted computing base* that includes the certificate verifier. Of course, even if the runtime itself is assumed correct, the interaction of the program with the runtime must be certified to conform to the appropriate interface.

An important part of the runtime system for many modern languages is the garbage collector. Frameworks in which the runtime system must be certified must use certification technology capable of proving a garbage collector safe. Work on this approach includes that of Wang and Appel [10, 9] and of Monnier *et al.* [4]. Of the systems that take the second approach, many assume the existence of a trusted conservative garbage collector; the advantage of this is that the application interface of a conservative collector is so simple that it can almost be ignored. There are performance benefits to be gained by using a more precise collector; however, the interface of such a collector is more subtle, and the issue of certifying program conformance to this interface can no longer be ignored. In order to use a better garbage collector for certified code applications, the interface of such a collector must be described and expressed in a type system.

The topic of this paper is the specification of the interface for a particular, modern garbage collector, namely that of Cheng and Blelloch [1, 2], implemented in the TILT/ML runtime system. After informally describing the behavior of this collector and its interface to a running program, we present a language whose type system can express this interface. This language, called LGC, is built up from a simple stack based language we call LGC$^-$ by extension with the typing constructs necessary to express various elements of the collector's interface. As we present LGC, we describe the interface to Cheng's garbage collector, the precise definition of which involves a substantial amount of programming in the language of type constructors. Finally, we discuss the expressiveness of the LGC language.

This material is based on work supported in part by NSF grants CCR-9984812 and CCR-0121633, and by an NSF fellowship. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of this agency.

TLDI'03, January 18, 2003, New Orleans, Louisiana, USA.

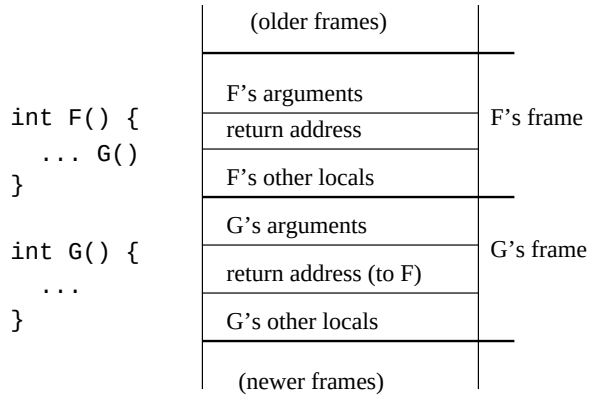

Figure 1: Frames on a Stack

1.1 The Garbage Collector's Interface

The first part of a garbage collector's job is to find the *root set*—those registers, globals and stack locations that contain pointers into the heap. This task is the part of garbage collection that requires compiler cooperation, and the part that makes assumptions about the behavior of the program. In this section we describe a simplified form of the root-finding algorithm used in TILT/ML. We will ignore complications such as an optimization for callee-save registers, and assume that all roots are stored on the stack. We can therefore ignore the additional work of finding roots among the registers or global variables.

The garbage collector assumes that the stack is laid out as a sequence of *frames*, each belonging to the particular function that created it. Each frame contains a number of data slots (including function arguments, local variables, and temporaries), as well as a *return address*. A section of a stack is illustrated in Figure 1. As usual, the stack is shown growing downwards. In the figure, the function `F` has called the function `G`; thus, the return address position in `G`'s frame will contain a location somewhere inside the code of `F`. In fact, the return address found in `G`'s frame uniquely identifies the point in the program from which `G` was called, and therefore also determines the layout of the frame above its own.

The garbage collector uses this property to "parse" the stack. When the program is compiled, the compiler emits type information that is collected by the runtime system into a *GC table*, which is a mapping from return addresses (identifying function call sites) to information about the stack frame of the function containing the call site. When the collector begins looking for roots, the newest frame on the stack is that of the collector itself; the return address in this frame can be looked up in the GC table to find a description of the next frame, which belongs to the untrusted program. The collector then moves through the stack, performing the following steps for each frame:

1. Using the return address from below the frame being examined, find the GC table entry that describes this frame.
2. Using this GC table entry, determine the following information:
   - The locations of pointers in the current frame. These are roots.
   - The location of the return address within the current frame.
   - The size of the current frame.
3. Using this information, find the start of the next frame and look up its GC table entry.

These steps are repeated until the base of the stack is reached.

Clearly, a correct GC table is essential for the operation of the garbage collector. An incorrect value in the table could lead to a variety of errors, from a single root pointer being ignored to derailment of the entire stack-parsing process. Put another way, it is crucial that the program structure its use of the stack consistently with the frame descriptions in the GC table. In this paper, we will present a language in which the stack's layout can be precisely controlled, giving us the ability to guarantee that the structure of the stack during collection will be consistent with the collector's expectations.

2 A Language With a GC Interface

The main goal of this paper is to describe a type system in which the shape of the program stack can be made to fit the pattern expected by the garbage collector; we must therefore have a language in which stack manipulation is explicit *and* which is expressive enough to describe the stack in very precise terms. In this section, we begin to describe our language, which we call LGC. We start with a simple core language we call LGC$^-$, which is a simple stack-based language that does not have the sophisticated type constructs to support garbage collection; we will then discuss the refinements necessary to enforce compliance with a GC table. The syntax and typing rules for full LGC are given in Appendix A.

2.1 The Core Language

The syntax of LGC$^-$ is given in Figure 2. The language is essentially a polymorphic λ-calculus with integers, booleans, tuples and sum types, plus a stack that is handled much the same way as in stack-based typed assembly language (STAL) [5]. The details of the language that do not directly relate to garbage collection are not particularly important for our purposes—indeed, there are many possible language designs that would work equally well—and so we will only discuss those aspects briefly here. Here and throughout the paper, we consider expressions that differ only in the names of bound variables to be identical, and we denote by $E[E_1, \ldots, E_n/X_1, \ldots, X_n]$ the result of the simultaneous capture-avoiding substitution of E_1 through E_n for the variables X_1 through X_n in E.

Programs An LGC$^-$ program consists of a sequence of mutually recursive code block definitions, followed by an expression. Each block has the form $\lambda(\Delta; \mathtt{sp}{:}\sigma).e$, indicating that it must be instantiated with some number and kind of type constructor arguments specified by Δ, and then may be invoked whenever the stack has type σ; invoking the block results in the evaluation of e. Notice that no data other than the stack itself is passed into a block; this means that

Kinds	k	::=	$\mathsf{T} \mid \mathsf{ST}$
Constructors	c, τ, σ	::=	$\alpha \mid \texttt{int} \mid \texttt{bool} \mid \texttt{code}(\Delta;\sigma) \to 0 \mid \tau_1 \times \cdots \times \tau_n \mid \tau_1 + \cdots + \tau_n \mid \exists \alpha{:}k.\tau$
		$\mid$	$\texttt{null} \mid \tau \otimes \sigma$
Values	v	::=	$x \mid n \mid b \mid \ell \mid v[c] \mid \texttt{pack}\,\langle c, v\rangle\,\texttt{as}\,\exists \alpha.\tau$
Expressions	e	::=	$\texttt{halt}\,v \mid \texttt{jump}\,v \mid \texttt{if}\,v\,\texttt{then}\,e_1\,\texttt{else}\,e_2 \mid \texttt{case}\,v\,\texttt{of}\,\texttt{inj}_1 x_1 \Rightarrow e_1 \mid \cdots \mid \texttt{inj}_n x_n \Rightarrow e_n \mid \texttt{let}\,d\,\texttt{in}\,e$
Declarations	d	::=	$x = v \mid x = aop(v_1, \ldots, v_k) \mid x = \texttt{inj}_\tau(i, v) \mid x = \langle v_1, \ldots, v_n\rangle \mid x = \pi_i\,v \mid \langle \alpha, x\rangle = \texttt{unpack}\,v$
		$\mid$	$x = \texttt{sp}(i) \mid \texttt{sp}(i) := v \mid \texttt{push}\,v \mid \texttt{pop}\,i$
Blocks	B	::=	$\lambda(\Delta; \texttt{sp}{:}\sigma).e$
Programs	P	::=	$\texttt{letcode}\,\ell_1 = B_1, \ldots, \ell_n = B_n\,\texttt{in}\,e$
Type Contexts	Δ	::=	$\cdot \mid \Delta, \alpha{:}k$
Contexts	Γ	::=	$\cdot \mid \Gamma, x{:}\tau$
Memory Types	Ψ	::=	$\cdot \mid \Psi, \ell : \tau$

Figure 2: Syntax of LGC$^-$

all function arguments and results must be passed on the stack. The return address in a function call must also be passed on the stack, leading to a continuation-passing style for programs. Also, because all the code blocks in a program appear at the top level, programs must undergo closure conversion before translation into LGC$^-$.

Expressions The body of each block is an expression. The expressions in LGC$^-$ include a `halt` instruction which stops the computation, a `jump` instruction which takes a code label and transfers control to the corresponding block, an `if-then-else` construct, case analysis on sums and a form of `let`-binding that performs one operation, possibly binding the result to a variable, and continues with another expression. The bindings that may occur in a `let` are a simple value binding, arithmetic operations ($x = aop(v_1, \ldots, v_k)$), injection into sum types ($x = \texttt{inj}_\tau(i, v)$), allocation of tuples ($x = \langle v_1, \ldots, v_k\rangle$), projection from tuples ($x = \pi_i\,v$), unpacking of values of existential type ($\langle \alpha, x\rangle = \texttt{unpack}\,v$, which binds a new constructor variable α in addition to the variable x) and stack operations. The stack operations are reading ($x = \texttt{sp}(i)$), writing ($\texttt{sp}(i) := v$), pushing of a value onto the stack ($\texttt{push}\,v$) and popping of i values off the stack ($\texttt{pop}\,i$).

The syntactic values in our language are variables (x), numerals (n), boolean constants (b), code labels (ℓ), instantiations of polymorphic values with constructors ($v[c]$), and packages containing a constructor and a value ($\texttt{pack}\,\langle c, v\rangle\,\texttt{as}\,\exists \alpha.\tau$), which have existential type.

Types and Kinds Our type theory has two kinds, T and ST, which classify constructors. Constructors of kind T are called *types*, and describe values; constructors of kind ST are called *stack types* and describe the stack. The constructors themselves include constructor variables (α), the base types `int` and `bool`, code label types ($\texttt{code}(\Delta;\sigma) \to 0$), n-ary products ($\tau_1 \times \cdots \times \tau_n$) and sums ($\tau_1 + \cdots + \tau_n$), existential types ($\exists \alpha{:}k.\tau$), the empty stack type `null` and non-empty stack types of the form $\tau \otimes \sigma$. (In STAL one writes these as `nil` and $\tau :: \sigma$, respectively, but in LGC we prefer to use this ML-like list notation for actual lists of constructors.) The metavariable c will be used to range over all constructors; we will also use the names τ and σ when we intend that the constructor being named is a type or a stack type, respectively. We will extend the kind and constructor levels of the type system later in the paper in order to more precisely describe the shape of the stack.

Static Semantics The typing rules for this simple language are generally the expected ones. Due to space considerations, we will not present them all here; they are generally similar to those for the full LGC language, whose rules are in Appendix A. We will, however, discuss some of the typing rules before turning to examine how programs in this language interact with a garbage collector. One of the simplest typing rules for expressions in the language is the one for the unconditional jump instruction:

$$\frac{\Psi; \Delta; \Gamma \vdash v : \texttt{code}(\cdot;\sigma) \to 0}{\Psi; \Delta; \Gamma; \texttt{sp}{:}\sigma \vdash \texttt{jump}\,v}$$

This rule states that it is legal to `jump` to a fully-instantiated code pointer (that is, one that does not expect any more constructor arguments) provided the stack type σ expected by the code pointer is the same as the current stack type. To jump to a polymorphic code pointer, one must first instantiate it by applying it the appropriate number and kind of type arguments. The rule for doing this is the following:

$$\frac{\Psi; \Delta; \Gamma \vdash v : \texttt{code}(\alpha{:}k, \Delta;\sigma) \to 0 \quad \Delta \vdash c : k}{\Psi; \Delta; \Gamma \vdash v[c] : \texttt{code}(\Delta;\sigma[c/\alpha]) \to 0}$$

Another simple typing rule is the one for binding a value to a variable:

$$\frac{\Psi; \Delta; \Gamma \vdash v : \tau \quad \Psi; \Delta; (\Gamma, x{:}\tau); \texttt{sp}{:}\sigma \vdash e}{\Psi; \Delta; \Gamma; \texttt{sp}{:}\sigma \vdash \texttt{let}\,x = v\,\texttt{in}\,e}$$

Read algorithmically, this rule can be understood as follows: To check that the expression $\texttt{let}\,x = v\,\texttt{in}\,e$ is valid, first find the type τ of the value v; then check that e is valid under the assumption that x has type τ.

2.2 Requirements for Garbage Collection

For the goal of certifying interaction with a garbage collector, LGC$^-$ is unacceptably simplistic. In fact, the syntax and typing rules we have discussed so far appear to ignore the collector completely. In this section, we begin to identify the specific shortcomings of the language; once we have done this, the remainder of the paper will be devoted to adding the necessary refinements to the language, resulting in the full LGC type system.

As we have already explained, if a program in our language is to work properly with a garbage collector, the collector must be able to find the roots whenever it is invoked. In practice, the garbage collector is usually invoked when a program attempts to create a new object in the heap and there is insufficient space available. The expressions in our language that perform allocation are tuple formation and injection into sum types; this means that the garbage collector may need to be able to find the root set during evaluation of an expression of the form $\texttt{let}\, x = \langle v_1, \ldots, v_n \rangle\, \texttt{in}\, e$ or $\texttt{let}\, x = \texttt{inj}_\tau(i, v)\, \texttt{in}\, e$. (We will only discuss tuples, since the modifications necessary for sums are exactly analogous.) A naïve version of the typing rule for tuple allocation would be the following:

$$\frac{\Psi;\Delta;\Gamma \vdash v_i : \tau_i \ (1 \le i \le n) \quad \Psi;\Delta;(\Gamma, x{:}\tau_1 \times \cdots \times \tau_n), \texttt{sp}{:}\sigma \vdash e}{\Psi;\Delta;\Gamma, \texttt{sp}{:}\sigma \vdash \texttt{let}\, x = \langle v_1, \ldots, v_n \rangle\, \texttt{in}\, e}$$

There are two main changes that must be made to this rule. First, we must force all the roots to be on the stack where they can be found by the collector; and second, we must force the stack to have a structure the collector can parse.

The first problem stems from the fact that we have variables in our language that are not stack allocated, but we want to assume for the sake of simplicity that the garbage collector scans only the stack when looking for roots. A free occurrence of a variable y in the expression e above could denote a pointer; that pointer could be used in the evaluation of e, but if there is no copy of that pointer on the stack, the garbage collector may not identify it as live. The solution is to force the program to "dump" the contents of all its variables to the stack whenever a collection might occur. To accomplish this, we require the continuation e to be closed except for the result x of the allocation. The rule now looks like this:

$$\frac{\Psi;\Delta;\Gamma \vdash v_i : \tau_i \ (1 \le i \le n) \quad \Psi;\Delta;(x{:}\tau_1 \times \cdots \times \tau_n), \texttt{sp}{:}\sigma \vdash e}{\Psi;\Delta;\Gamma, \texttt{sp}{:}\sigma \vdash \texttt{let}\, x = \langle v_1, \ldots, v_n \rangle\, \texttt{in}\, e}$$

Note that a more realistic abstract machine would have registers instead of variables; in order to support GC table certification on such a machine we would have to apply the techniques we discuss in the remainder of this paper to the register file as well as the stack. This seems straightforward, but for the sake of simplicity we will limit our discussion in this paper to a collector that can only find roots in the stack.

The second modification that must be made to the allocation rule is significantly more difficult to formulate. In fact, the rest of this paper is devoted to adding a single additional premise to the rule, namely one that stipulates that the stack type σ is *parsable*. That is, we must describe the structure that the stack must have in order to be scanned by the collector, and express that structure in a way that can be enforced by the type system. The type system of LGC$^-$ is not up to this task, so before continuing we must endow it with the expressive power to meet our needs.

2.3 Enriching the Constructor Language

In order to be able to give a typing constraint in the allocation rule that precisely describes the required structure of the stack, we must enrich the constructor level of our language. For this purpose, we add a number of constructs from Crary and Weirich's LX type theory [3]. These additions to our language are shown in Figure 3. In addition to function spaces, products and sums over kinds ($k_1 \to k_2$, $k_1 \times k_2$, $k_1 + k_2$), LX provides *inductive kinds* $\mu j.k$, where j is a *kind variable* that may appear in positive positions within k. At the type constructor level, we change the syntax of product and sum types to $\times(c)$ and $+(c)$, where in each case c is a constructor of kind $\mu j.\mathsf{1} + \mathsf{T} \times j$ and represents a list of types. To keep the notation for LGC simple, we allow the syntax from LGC$^-$ to serve as shorthand, defined as follows:

$$\tau_1 \times \cdots \times \tau_n \triangleq \times(\texttt{fold}(\texttt{inj}_2\langle \tau_1, \ldots \texttt{fold}(\texttt{inj}_2\langle \tau_n, \texttt{fold}(\texttt{inj}_1 \star)\rangle)\ldots\rangle))$$

(The analogous notation is used for sums.) Finally, we have a kind $\mathsf{1}$ whose sole element is the constructor $\star$, and we add the types `unit` and `void` to the language. The type `unit` has the sole element $\langle\rangle$, while the type `void` contains no values.

The introduction forms and elimination forms for arrows, sums and products at the constructor level are the usual ones; inductive kinds are introduced with a `fold` construct and eliminated with *primitive recursion* constructors of the form $\texttt{pr}(j, \alpha{:}k, \beta{:}j \to k', c)$. If well-formed, this constructor will be a function of kind $\mu j.k \to k'[\mu j.k/j]$; c is the body of the function, in which α may appear as the parameter and β is the name of the function itself to be used for recursive calls.

For example, if we define the kind $N = \mu j.1 + j$ (representing the natural numbers), then we can define the function *iter* as follows:

$$\begin{array}{l} \mathit{iter} \ : \ (\mathsf{T} \ \to \mathsf{T}) \to \mathsf{T} \to \ \ N \to \mathsf{T} \ = \\ \quad \lambda \ \varphi{:}(\mathsf{T} \to \mathsf{T}).\lambda \ \alpha{:}\mathsf{T} \ . \\ \qquad \texttt{pr}(j, \beta{:}1 + j, \gamma{:}j \to \mathsf{T}, \\ \qquad\quad \texttt{case}(\beta, \delta_1.\alpha, \delta_2.\varphi(\gamma \ \ \delta_2))) \end{array}$$

The constructor *iter* is a function taking a function from types to types, a type and a natural number, and returning the result of iterating the function on the given type the specified number of times.

Clearly, the `pr` notation is somewhat unwieldy to read and write, so we will use an ML-like notation for working in the LX constructor language. We will, for many purposes, combine the notions of inductive and sum kind and define *datakinds*, akin to ML's datatypes. For example, we could write the definition of N above as follows:

$$\texttt{datakind}\ N\ \texttt{=}\ \mathit{Zero}\ \texttt{of}\ \mathsf{1}\ \texttt{|}\ \mathit{Succ}\ \texttt{of}\ N$$

The function *iter* would be more readably expressed in ML curried function notatin this way:

$$\begin{array}{l} \texttt{fun}\ \mathit{iter}\ \ \varphi\ \ \alpha\ \ \mathit{Zero}\ \texttt{=}\ \alpha \\ \quad \texttt{|}\ \ \mathit{iter}\ \ \varphi\ \ \alpha\ \ (\mathit{Succ}\,\beta)\ \texttt{=}\ \varphi(\mathit{iter}\,\varphi\,\alpha\,\beta) \end{array}$$

Kinds	k	::=	$\cdots \mid j \mid k_1 \to k_2 \mid k_1 \times k_2 \mid k_1 + k_2$
		$\mid$	$\mu j.k \mid \mathsf{1}$
Constructors	c	::=	$\cdots \mid \lambda\alpha{:}k.c \mid c_1\, c_2 \mid \langle c_1, c_2\rangle \mid \pi_i\, c$
		$\mid$	$\texttt{inj}_i c \mid \texttt{case}(c_1, \alpha_1.c_2, \alpha_2.c_3)$
		$\mid$	$\texttt{fold}_k c \mid \texttt{pr}(j, \alpha{:}k, \beta{:}j \to k', c)$
		$\mid$	$\star \mid \texttt{unit} \mid \texttt{void} \mid \times(c) \mid +(c)$
Values	v	::=	$\cdots \mid \langle\rangle$

Figure 3: Kinds and Constructors from LX

We will often write functions in this style, being careful only to write functions that can be expressed in the primitive recursion notation of LX. To further simplify the presentation, we will also use the familiar ML constructors `list` and `option` to stand for the analogous datakinds.

2.4 Approaching Garbage Collection

The language LX was originally designed for intensional type analysis. The basic methodology was to define a datakind of analyzable constructors, which we will call *TR* (for "type representation"), a function $interp : TR \to \mathsf{T}$ to turn a constructor representation (suitable for analysis) into an actual type (suitable for adorning a variable binding), and a function $R : TR \to \mathsf{T}$ to turn a constructor into the type of a value that represented it at run time. In addition to explaining the somewhat mysterious operation of run-time type analysis in more primitive terms, this had the effect of isolating a particular subset of types for analysis: only those types that appeared in the image of the *interp* mapping could be passed or analyzed at run time. For garbage collection, we want to do something similar: we want to isolate the set of stack types that are structured such that the collector can parse the stack using the algorithm outlined in Section 1.1. We can then add the appropriate stack structure condition to the allocation rule by asserting that the current stack type lies in that set.

To do this, the remainder of this paper will define the following LX objects:

1. A datakind *SD* (for "stack descriptor"), whose elements will be passed around in our programs in place of stack types.
2. A datakind *DD* (for "data descriptor"), whose elements will be static representations of GC tables. Every program in our language will designate one particular constructor to be its *static GC table*, or SGCT. This constructor *SGCT* will have kind *DD*.
3. A constructor $interpS : DD \to SD \to \mathsf{ST}$, that will turn a stack descriptor into a stack type *provided* that it only uses stack frames whose shapes are determined by a particular static GC table. We will be careful to write *interpS* such that for any constructors $s : SD$ and $SGCT : DD$, stacks of type $interpS\ SGCT\ s$ will always be parsable.

Once we have definitions for all of these, ensuring that the stack is parsable for garbage collection is simple: if the current stack type is σ, we need to require that there exist some constructor $s : SD$ such that $\sigma = interpS\ SGCT\ s$. Of course, we do not want the type-checker to have to guess the appropriate s, so we change the syntax slightly to make it the programmer's responsibility. The next version of our allocation rule (modulo the definitions of these new LX expressions) is:

$$\frac{\begin{array}{l}\Psi;\Delta;\Gamma \vdash v_i : \tau_i\ (1 \le i \le n) \\ \Psi;\Delta;(x{:}\tau_1 \times \cdots \times \tau_n), \mathtt{sp}{:}\sigma \vdash e \\ \Delta \vdash s : SD \quad \Delta \vdash \sigma = interpS\ SGCT\ s : \mathsf{ST}\end{array}}{\Psi;\Delta;\Gamma;\mathtt{sp}{:}\sigma \vdash \mathtt{let}\ x = \langle v_1, \ldots, v_n \rangle[s]\ \mathtt{in}\ e}$$

There will be one more development of this rule in Section 3.1.

In order for this expression of the interface to the garbage collector in terms of *SGCT* to guarantee correct programs, we must be sure that the actual data structure used as the GC table agrees with its static representation. While LX is capable of expressing a type for the GC table that guarantees this, we have chosen a simpler approach. Rather than forcing the program to provide its own GC table in both "static" constructor form and "dynamic" value form, we assume that the type-checker in our certified code system transforms the static GC table into a real GC table and provides the latter to the runtime system before the program starts. Thus, we consider the generation of the GC table itself from the static representation to be part of the trusted computing base.

The remaining sections of this paper will present the definitions of the kinds *SD* and *DD*, and will describe the behavior of *interpS* and the auxiliary functions needed to define it. The definition of *interpS* is nontrivial and involves an unusual amount of programming at the type constructor level of the language. The complete code for the special kinds and constructors used in our GC interface can be found in Appendix B.

3 Describing the Stack

Since the collector requires the stack to be structured as a sequence of frames, our LX representation of the stack type will be essentially a list of *frame descriptors*, which we will represent by constructors of another datakind, called *FD*. A frame descriptor must allow two major operations: (1) since lists of descriptors are passed around in the program instead of stack types, it must be possible to *interpret* a descriptor to get the partial stack type it represents, and (2) since we are structuring the stack this way so as to ensure agreement with a GC table, it must be possible to *check* a descriptor against an entry in the table. The individual entries in the static representation of the GC table will be constructors of a kind called *FT*, for *frame template*, that we will also define shortly.

3.1 Labels and Singletons

As we have mentioned before, a key property of the stack layout required by the garbage collector is that the return address of one frame determines, via the GC table, the expected shape of the next older frame. As a result, in order for our constraint on the stack's type before a collection to guarantee proper functioning of the collector, we must ensure that the *value* stored in the return address position in each frame corresponds, via the static GC table, to the *type* of the next frame.

To make this happen, we must be able to reason about labels—*i.e.,* pointers to code—at the constructor level of our language. We therefore lift label literals from the value level of the language to the constructor level, and add a new primitive kind, L, to classify them. In addition, we add a construct for forming *singleton types* from labels. Using this construct, we will be able to force the return address stored in a stack frame to have precisely the value it must in order to correctly predict the shape of the next frame on the stack.

The syntax and typing rules for labels and singletons are shown in Figure 4. If c is the label of a code block of type τ, then $\mathcal{S}(c : \tau)$ is the type that contains only instances of c. In order to make use of values of singleton type, we introduce

$$
\begin{array}{lrcl}
\textit{Kinds} & k & ::= & \cdots \mid \mathsf{L} \\
\textit{Constructors} & c & ::= & \cdots \mid \ell \mid \mathcal{S}(c:\tau) \\
\textit{Values} & v & ::= & \cdots \mid \ell \mid \mathtt{blur}\, v \mid v\{c\}
\end{array}
$$

$$
\frac{}{\Delta \vdash \ell : \mathsf{L}} \qquad \frac{\Delta \vdash c : \mathsf{L} \quad \Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \mathcal{S}(c:\tau) : \mathsf{T}}
$$

$$
\frac{\Psi(\ell) = \tau}{\Psi;\Delta;\Gamma \vdash \ell : \mathcal{S}(\ell:\tau)} \qquad \frac{\Psi;\Delta;\Gamma \vdash v : \mathcal{S}(c:\tau)}{\Psi;\Delta;\Gamma \vdash \mathtt{blur}\, v : \tau}
$$

$$
\frac{\Psi;\Delta;\Gamma \vdash v : \mathcal{S}(c' : \mathtt{code}(\alpha{:}k, \Delta; \sigma) \to 0) \quad \Delta \vdash c : k}{\Psi;\Delta;\Gamma \vdash v\{c\} : \mathcal{S}(c' : \mathtt{code}(\Delta; \sigma[c/\alpha]) \to 0)}
$$

Figure 4: Syntax and Typing for Labels

the coercion `blur`, which forgets the identity of a singleton value, yielding a value which is an appropriate operand to a `jump` instruction. Since values of singleton type are code labels, which are usually polymorphic, we have found it necessary to add a way to apply a label to a constructor argument while maintaining its singleton type; this is accomplished by writing $v\{c\}$.

The sensitivity of the garbage collector to labels found in the stack raises another issue that must be addressed in the typing rule for allocation. In order for the collector to begin the process of scanning the stack, it must be able to find the GC table entry for its caller's frame (*i.e.,* the newest program frame). It is therefore necessary to associate a label with each allocation site, and require that the first frame descriptor in the stack descriptor correspond to that label. (Since this label is intended to denote the return address of the call to the garbage collector, we must assume that all such labels in the program are distinct.) We also define a function *retlab* of kind $DD \to SD \to \mathsf{L}$ that extracts the label of the newest frame of the given stack descriptor; making one final change to the syntax of allocation to include a label, the final typing rule is as follows:

$$
\frac{\begin{array}{l}\Psi;\Delta;\Gamma \vdash v_i : \tau_i \ (1 \le i \le n) \\ \Psi;\Delta;(x{:}\tau_1 \times \cdots \times \tau_n), \mathtt{sp}{:}\sigma \vdash e \\ \Delta \vdash \ell = \mathit{retlab}\ \mathit{SGCT}\ s : \mathsf{L} \\ \Delta \vdash s : \mathit{SD} \quad \Delta \vdash \sigma = \mathit{interpS}\ \mathit{SGCT}\ s : \mathsf{ST}\end{array}}{\Psi;\Delta;\Gamma;\mathtt{sp}{:}\sigma \vdash \mathtt{let}\ x = \langle v_1, \ldots, v_n\rangle[s,\ell]\ \mathtt{in}\ e}
$$

3.2 Stack Descriptors

The general structure of the kind *SD* is given in Figure 5, along with an illustration of the interpretation of a constructor of this kind into a stack type by *interpS*. (The validity checking performed by *interpS* will be discussed in the next section.) As the kind definitions show, a stack descriptor is either "empty"—in which case it carries the label identifying the return address of the top frame—or it consists of a frame descriptor and a descriptor for the rest of the stack.

A frame descriptor consists of a label, which identifies the point in the program that "owns" the frame[1], the return type of the function whose frame it is, and two lists of *slots*. The kind *Slot* of slots is not defined here; we address its definition in the next section. A slot describes a single location on the stack; a constructor of kind *Slot* must support (1) interpretation into a type in the fashion indicated by the arrows in the illustration, and (2) examination to determine what the specification of this slot in the GC table ought to be.

[1]That is, the return address of the currently pending function call executed by the function instance that created the frame.

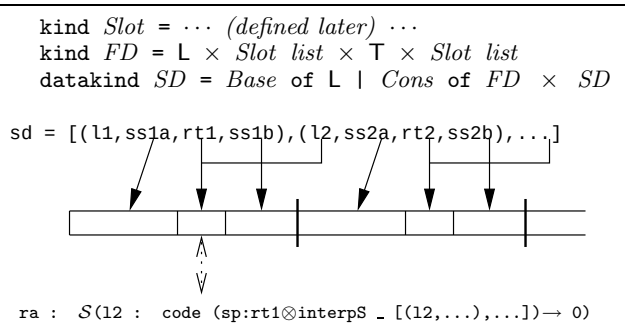

Figure 5: Structure and Interpretation of Stack Descriptors

The first list of slots in a frame descriptor corresponds to the slots that come before the return address, the second list describes the slots after the return address. As shown in the diagram, *interpS* builds each frame of the stack type by interpreting the slots into types, and constructs the return address using the function's return type as specified in the frame descriptor, forming a singleton with the label associated with the next frame. The code for *interpS* is in Appendix B.

In keeping with the usual LX methodology, it is our intention that LGC programs pass constructors of kind *SD* where programs in a GC-ignorant language would pass stack types. For example, the code of a function that takes two integers and returns a boolean (such as a comparison function) might have the type:

$$
\begin{array}{l}\mathtt{code}(\rho{:}\mathit{SD}; \mathtt{code}(\cdot; \mathtt{bool} \otimes \mathit{interpS}\ \mathit{SGCT}\ \rho) \to 0 \\ \qquad \otimes \mathtt{int} \otimes \mathtt{int} \otimes \mathit{interpS}\ \mathit{SGCT}\ \rho) \to 0\end{array}
$$

Unfortunately, this type does not quite capture the relationship between the return address (of type $\mathtt{code}(\cdot; \mathtt{bool} \otimes \mathit{interpS}\ \mathit{SGCT}\ \rho) \to 0$) and the caller's frame (which is hidden "inside" ρ). A code block with this type will be unable to perform any allocation, because its return address does not have a singleton type. In order to give the return address a singleton type, we must extract the label from the calling frame using the function *retlab* mentioned in Section 3.1. We then use the following more accurate type in place of the above:

$$
\begin{array}{l}\mathtt{code}(\rho{:}\mathit{SD}; \mathcal{S}(\mathit{retlab}\ \mathit{SGCT}\ \rho \\ \qquad : \mathtt{code}(\cdot; \mathtt{bool} \otimes \mathit{interpS}\ \mathit{SGCT}\ \rho) \to 0) \\ \qquad \otimes \mathtt{int} \otimes \mathtt{int} \otimes \mathit{interpS}\ \mathit{SGCT}\ \rho) \to 0\end{array}
$$

A more detailed example of the use of stack descriptors (but with no allocation) is shown in Figure 6. The most interesting part of the function shown in the figure is the recursive call. If we let $\sigma_1 = \mathit{interpS}\ \mathit{SGCT}\ (\mathit{Cons}(\mathit{factframe}, \rho))$, then the return address of the recursive call to `factcode`, $\mathtt{factreturn}\{\rho\}$, has type $\mathcal{S}(\mathtt{factreturn} : \mathtt{code}(\cdot; \mathtt{int} \otimes \sigma_1) \to 0$. Furthermore,

Suppose `intsl` : *Slot* specifies a slot of type `int`, and define $factframe = \langle \texttt{factreturn}, [\,], \texttt{int}, [\texttt{intsl}] \rangle$.

```
factcode = code( ρ:SD ; S( retlab SGCT ρ : code(·;int⊗(interpS SGCT ρ))->0)
                         ⊗ int ⊗ (interpS SGCT ρ) ) .
     let x = sp(1) in
     let b = =(x,0) in
     if b then
          let ra = sp(0) in                          ; get return address off stack
          pop 2 in                                   ; clear away our frame
          push 1 in                                  ; (0! = 1)
          call blur(ra)                              ; return
     else
          let x = -(x,1) in                          ; call fact(x-1)
          push x in                                  ;   push argument
          push factreturn{ρ} in                      ;   push return address
          call factcode[ Cons(factframe,ρ) ]         ;   jump
factreturn = code ( ρ:SD; int⊗(interpS SGCT (Cons(factframe,ρ))) ) .
     let x = sp(2) in
     let callres = sp(0) in
     let result = *(x,callres) in                    ; x * fact(x-1)
     let ra = sp(1) in
     pop 3 in                                        ; clear away our frame
     push result in
     call blur(ra)                                   ; return
```

Figure 6: Sample Code Using Stack and Frame Descriptors

if we observe that $retlab\ SGCT\ (Cons(factframe, \rho)) = \texttt{factreturn}$, then the type of the address in the call instruction is:

$$\texttt{code}(\mathcal{S}(\texttt{factreturn} : \texttt{code}(\texttt{int} \otimes \sigma_1) \to 0) \otimes \texttt{int} \otimes \sigma_1)$$

To see that the stack type in this code type matches the current stack at the call site, observe that the first value on the stack is the return address, whose type we have already seen to be equal to the one required for the call. The second value on the stack is the argument to the recursive call, which has type `int`. Finally, σ_1 describes the function's own frame and the pre-existing stack. In particular, $\sigma_1 = \tau_r \otimes \texttt{int} \otimes \sigma_0$, where τ_r stands for the type of the original return address and $\sigma_0 = interpS\ SGCT\ \rho$ is the unknown base portion of the stack.

4 Checking Frame Validity

In addition to enforcing the property that the stack is a sequence of frames, the condition $\sigma = interpS\ SGCT\ s$ must also guarantee that the frames themselves are correctly described by the GC table. To accomplish this, we ensure that the equality can only hold if the frame descriptors in s are consistent with the information about them contained in $SGCT$, the GC table's constructor-level representation. Since the actual GC table is a mapping from return addresses to frame layout information, it makes sense to structure $SGCT$ as a mapping from labels to frame layouts as well.

The basic structure of DD, the kind of $SGCT$, is given in Figure 7. The static GC table is structured as a list of pairs, each consisting of a label and a constructor of kind FT, which stands for *frame template*. A frame template is essentially an LX constructor representation of the information in a real GC table entry; it consists of two lists of *table slots* (constructors of kind *TSlot*), which correspond to the two lists of slots in a frame descriptor. Checking a frame descriptor for validity therefore consists of looking up the label from the frame descriptor in $SGCT$ and checking each of the slots in the FD against the table slots in the FT. We will give definitions for *Slot* and *TSlot* and discuss this consistency checking shortly.

```
kind TSlot = ··· (defined later) ···
kind FT = TSlot list × TSlot list
datakind DD = NilDD of 1
            | ConsDD of L × FT × DD

con lookupDD : DD → L → FT + 1 = ...
```

Figure 7: Structure of the Static GC Table

First, however, we must make one final addition to LGC. In order to be able to write the all-important *lookupDD* function that finds the frame template for a given label, our constructor language must be able to compare labels for equality. The syntax and semantics of label equality at the constructor level are given in Figure 8. The constructor $\texttt{ifeq}(c_1, c_2, c_3, c_4)$ is definitionally equal to c_3 if the labels c_1 and c_2 are the same, c_4 if they are not the same. Note that the reduction rules for `ifeq` only apply when c_1 and c_2 are label literals, so the equational theory remains well-behaved. With these constructs in place, *lookupDD* is easy to write using primitive recursion.

4.1 Monomorphic Programs

In this section we will give definitions of *Slot* and *TSlot* that allow "monomorphic" programs to be written in LGC. By

$$\textit{Constructors} \quad ::= \quad \cdots \mid \texttt{ifeq}(c_1, c_2, c_3, c_4)$$

$$\frac{\Delta \vdash c_1 : \mathsf{L} \quad \Delta \vdash c_2 : \mathsf{L} \quad \Delta \vdash c_3 : k \quad \Delta \vdash c_3 : k}{\Delta \vdash \texttt{ifeq}(c_1, c_2, c_3, c_4) : k} \qquad \frac{\Delta \vdash c_3 : k \quad \Delta \vdash c_4 : k}{\Delta \vdash \texttt{ifeq}(\ell, \ell, c_3, c_4) = c_3 : k}$$

$$\frac{\Delta \vdash c_3 : k \quad \Delta \vdash c_4 : k \quad (\ell_1 \neq \ell_2)}{\Delta \vdash \texttt{ifeq}(\ell_1, \ell_2, c_3, c_4) = c_4 : k}$$

Figure 8: Label Equality

```
datakind Slot = KnownSlot of TR
datakind TSlot = Trace of 1 | NoTrace of 1
datakind B = True of 1 | False of 1

con Slot2TSlot : Slot → TSlot = ...
con eqTSlot : TSlot → TSlot → B = ...
con checkFD : DD → FD → B = ...
```

Figure 9: Monomorphic Slots and Table Slots

"monomorphic" we here mean programs in which all the values a function places in its stack frame have types that are known at compile time.[2] If the type of every value in a function's stack frame has a non-variable type, then it can be determined statically whether each slot in the frame contains a pointer that must be traced. More importantly, the traceability of any slot will be the same for every instance of the function. Consequently, the GC table only needs one bit for each slot, and all that needs to be checked at each allocation site is whether the types of all the slots have the traceabilities specified in the table.

The definitions for *Slot* and *TSlot* are given in Figure 9 along with the kinds of two constructor functions we will use to check frames. In the case of monomorphic code, a slot is simply a type representation in the usual style of LX; a table slot is simply a flag indicating whether a location is traceable or not. We will not discuss the definition of *TR* further, as any representation of types that can be coded in LX will do for the purposes of this paper. We do, however, assume the existence of the usual interpretation and representation functions $\textit{interp} : \textit{TR} \to \mathsf{T}$ and $R : \textit{TR} \to \mathsf{T}$; as usual for LX, *interp* turns a type representation into the type it represents, and R turns a type representation into the type of the value representing that type. The stack interpretation function *interpS* must make use of *interp* to translate a slot (which is really a type representation) into a type; we will use R in the next section, when we cover polymorphic programs.

The function *checkFD* checks that a frame descriptor is valid with respect to the static GC table. First, it must look up the frame descriptor's label to get the corresponding frame template if there is one. If there is no frame template for that label, the descriptor is rejected as invalid. The

[2]Note that no nontrivial program in a stack-based language can really be totally monomorphic, since every function must be parametric in the stack type so that it can be called at any time.

```
con factFT = ⟨[ ], [NoTrace]⟩
con SGCT = [⟨factreturn, factFT⟩]
```

Figure 10: Static GC Table for Factorial Example

function *Slot2TSlot* simply decides whether a given type representation is traceable; given a frame template, *checkFD* applies the *Slot2TSlot* to each of the slots in the frame descriptor and uses *eqTSlot* to determine whether the resulting *TSlot* matches the corresponding one in the frame template.

To ensure that the stack can be parsed by the garbage collector, the interpretation function *interpS* calls *checkFD* on each of the frame descriptors it sees. This portion of the code of *interpS* is essentially the following:

```
fun interpS SGCT (Cons ⟨fd, rest⟩) =
    case checkFD SGCT fd of
        True => ...
      | False => void ⊗ null
```

In the case where the frame descriptor is not valid with respect to *SGCT*, the body of *interpS* reduces to `void⊗null`, which is an unsatisfiable stack type since the type `void` is uninhabited. If the stack type is *interpS SGCT s* at some reachable program point, then obviously *interpS SGCT s* must be inhabited. Therefore, reduction of this definition must not have taken the *False* branch, so it follows that all the frame descriptors in *s* must be valid in the sense of *checkFD*.

The static GC table for the factorial example from Figure 6 is shown in Figure 10. Of course, this is a bit unrealistic since we have shown a "program" with only one function call site, so as a result there is only one entry in the GC table. If we were to add anything to the factorial program, such as a main program body that calls the function `factcode`, the GC table and its static representation would have to be augmented with descriptions of any new call sites we introduced.

4.2 Polymorphic Programs

It is a little more difficult to adapt LGC to certifying polymorphic programs, because in such programs a function may have arguments or local variables whose types are different each time the function is called. The TILT garbage collector handles such stack locations by requiring that, in any instance of a polymorphic frame, a value representing the type of each of these slots is available. The slot in the GC table corresponding to a location whose type is statically unknown, rather than directly giving traceability information, tells the collector where the representation can be found. TILT allows some flexibility in where the representations are stored: they can be on the stack, in a heap-allocated record with a pointer to the record on the stack, or in global storage. For our purposes, we will assume a simple, flat arrangment in which the type representations for a frame are all stored in that frame.

The new definitions of *Slot* and *TSlot* to account for polymorphism are shown in Figure 11. We also slightly modify the definition of *FD*, the kind of frame descriptors. Any frame in a polymorphic program will in general be parametric in some number of "unknown" types; since frame descriptors must be interpretable to give the type of the stack,

```
datakind N = Zero | Succ of N
datakind Slot = KnownSlot of TR
              | VarSlot of N | RepSlot of N
kind FD = L × Slot list × T × Slot list × TR list
datakind TSlot = Trace of 1 | NoTrace of 1
               | Var of N | Rep of N

con nth : TR list → N → TR + 1
```

Figure 11: Frames and Tables for Polymorphic Programs

a frame descriptor represents a single instance of a polymorphic frame. Therefore, the version of *FD* for polymorphic programs includes a list of type representations that "instantiate" the frame descriptor by providing representations of all the unknown types of values in the frame.

Each individual slot in a frame descriptor may now take one of three forms: it may be a slot whose type is known at compile time, as before; or it may be a slot whose type is one of the unknown types associated with the frame; or it may be the slot that holds the representation of one of those types. These three possibilities are reflected in the new definition of *Slot*; in the case of unknown-type and representation slots, the frame descriptor will carry a natural number indicating which of the type parameters gives the type of, or is represented by, the slot. Similarly, there are now four choices for a slot in the static GC table. A slot may be known to be traceable; it may be known to be untraceable; it may contain a value of variable type; or it may contain a representation. These four possibilites correspond to the arms of the new *TSlot* datakind.

The interpretation of slots into types is now a bit more complicated as well; for slots of known type the operation is unchanged, but for variable and representation slots *interpS* must look up the appropriate type representation in the list given by the frame descriptor. Once this representation is obtained, variable slots are turned into types using *interp* as before, while representation slots are turned into types using the R function described before. We therefore write the function *interpsl*, which interprets a single slot given the list of type representations from the frame descriptor.

```
fun interpsl trs (KnownSlot tr) = interp tr
  | interpsl trs (VarSlot n) =
    (case nth trs n of
         SOME tr => interp tr
       | NONE => void)
  | interpsl trs (RepSlot n) =
    (case nth trs n of
         SOME tr => R tr
       | NONE => void)
```

Notice that slots specifying invalid indices into the list of representations are given type `void`, to ensure that the frame described by the invalid descriptor cannot occur at run time.

In addition to the possibility of bad indices in variable and representation slots, there is another new way in which a frame descriptor may be invalid: the definition of *FD* allows a frame to contain a *VarSlot* for which it does not contain a corresponding *RepSlot*. Fortunately, the property that the set of indices given in *VarSlot*'s is contained in the set of indices given in *RepSlot*'s is easy to check primitive recursively. This responsibility falls to the polymorphic version of the function *checkFD*.

An example of a simple polymorphic function in LGC is shown in Figure 12. The code in this figure defines a function which, for any type representation α, takes a value of type *interp* α and boxes it—that is, allocates and returns a one-field tuple of type $\times[\textit{interp}\,\alpha]$ containing that value. The stack descriptor provided at the allocation site adds a descriptor for the current frame to the pre-existing stack descriptor ρ. This new frame descriptor contains two slots corresponding to the two values (other than the return address) that make up the function's stack frame: the first, *RepSlot Zero*, describes the run-time representation of α, which itself has type $R\,\alpha$; the second, *VarSlot Zero*, describes the argument to the function, which has type *interp* α.

5 Expressiveness

In order to experiment with the expressive power of LGC, we have implemented a type-checker for the language, including a "prelude" of constructor and kind definitions giving the meanings of *TR*, *SD*, *DD*, *interpS* and so on. We have also implemented a translation from a GC-ignorant source language into LGC, demonstrating that LGC is expressive enough to form the basis of a target of a general-purpose compiler.

The syntax of the source language is shown in Figure 13. Its design was driven solely by the goal of removing all explicit GC-related constructs while enabling a straightforward translation into LGC. We will briefly mention some of the issues that shaped the design of the source language, since they highlight the unusual properties of a language designed with a garbage collection interface in mind.

Implicit Stack Operations Since the garbage collector requires the stack to have a certain structure, it would be very inconvenient to allow the source program unrestricted use of stack manipulation operations. Therefore, we chose to remove the stack almost completely from the syntax of the source language. Source-level functions accept arguments and return results in the usual manner; the translation to LGC takes care of turning parameter-passing into stack manipulation. In addition, since return addresses play such a critical role in scanning the stack, we cannot allow source programs to manipulate those either. As a result, the source language abandons continuation-passing style for a more familiar `return` instruction (which we merge with the `halt` instruction since their semantics are similar).

Locals Since the source program cannot manipulate the stack, support for storing intermediate results there must be built into the language. A somewhat unfortunate consequence is that all decisions about what will and will not be stored on the stack must have been made before translation into LGC. If the design of LGC were to be applied to a compiler targeting typed assembly language, this would correspond to the fact that register allocation must be completed before generation of GC tables can begin. In order to use stack space for local variables and temporary storage, each code block in a source program begins with `lalloc` i, which indicates that the block wishes to allocate i mutable local variables on the stack. Special forms of declarations at the expression level provide access to these locals.

Define $\texttt{allocframe}(\alpha) = (\texttt{allocsite}, [\,], \times[\mathit{interp}\,\alpha], [\mathit{RepSlot\ Zero}, \mathit{VarSlot\ Zero}], [\alpha])$
and $\mathit{SGCT} = [(\texttt{allocsite}, [\mathit{Rep\ Zero}, \mathit{Var\ Zero}])]$.

```
boxcode = code (ρ:SD, α:TR; S(retlab SGCT ρ : code(· ; ×[interp α] ⊗ interpS SGCT ρ)→ 0)
                              ⊗ R α ⊗ interp α ⊗ interpS SGCT ρ) .
    let x = sp(2) in
    let cell = <x>[Cons(allocframe(α),ρ),allocsite] in
    let ra = sp(0) in
    pop 3
    push cell
    call blur(ra)
```

Figure 12: Polymorphic Allocation Example

Types	τ	::=	$\alpha \mid \texttt{int} \mid \texttt{bool} \mid \texttt{ns} \mid \texttt{code}(\tau_e; \Delta.(\tau_1, \ldots, \tau_n) \to \tau)$
		$\mid$	$\forall\Delta.(\tau_1, \ldots, \tau_n) \to \tau \mid \tau_1 \times \cdots \times \tau_n$
Values	v	::=	$x \mid n \mid b \mid \ell \mid v[\tau]$
Expressions	e	::=	$\texttt{return}\ v \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ d\ \texttt{in}\ e$
Declarations	d	::=	$x = v \mid x = \mathit{aop}(v_1, \ldots, v_k) \mid x = \texttt{closure}(v_1, v_2) \mid x = \langle v_1, \ldots, v_n\rangle \mid x = \pi_i\, v$
		$\mid$	$x = \texttt{local}(i) \mid \texttt{local}(i) := v \mid x = \texttt{arg}(i) \mid x = \texttt{call}(v_f, v_1, \ldots, v_n)$
Blocks	B	::=	$\lambda x_e{:}\tau_e.\Lambda(\Delta; \tau_1, \ldots, \tau_n) \to \tau.\texttt{lalloc}(i).e$
Programs	P	::=	$\texttt{letcode}\ \ell_1 = B_1, \ldots, \ell_n = B_n\ \texttt{in}\ e$
Type Contexts	Δ	::=	$\cdot \mid \Delta, \alpha$
Contexts	Γ	::=	$\cdot \mid \Gamma, x{:}\tau$
Memory Types	Ψ	::=	$\cdot \mid \Psi, \ell : \tau$
Local Storage Types	Λ	::=	$[\tau_1, \ldots, \tau_k]$

Figure 13: Syntax of the Source Language

Closures Since LGC requires all code blocks to be closed and hoisted to the top level, a translation from a higher-level language in which functions may be nested must perform closure conversion as part of translation into LGC. Since the interface of the garbage collector seems to have little impact on the closure conversion transformation itself, we chose to keep the source-to-LGC translation simple by assuming closure conversion had already been performed. Therefore, the source language also requires all code blocks to be at the top level. However, we do not include existential types in the source language, as providing representations for all the types hidden with existentials would add to the bloat associated with the translation. Since many of these representations turn out to be unnecessary, we find it more economical to introduce existential types as closures at the same time as the translation to LGC. We include in the source language special types of closures ($\forall\Delta.(\tau_1, \ldots, \tau_n) \to \tau$) and operations for creating them. Every code block expects a special argument, which is the environment of the closure; code pointers are made into functions using the `closure` operation, which packs a code value together with an environment.

6 Conclusion

We have presented a language in whose type system the interface to a modern high-performance garbage collector can be expressed. In so doing, we have demonstrated that code certification is indeed compatible with the use of sophisticated, accurate garbage collection technology. We have described the interface of one such collector in our language, and implemented a prototype type-preserving translation from a GC-ignorant source language into our target language.

The alert reader will have noticed the absence of an operational semantics or safety proof in this paper. An operational semantics is completely straightforward, except that the two rules that perform heap allocation must each have an additional side condition requiring that the stack be parsable. A type safety proof is boilerplate, based on the proof for LX by Crary and Weirich [3], except that in the cases of injection and allocation it must be shown that the typing conditions on the stack imply that it is parsable. However, it is not clear how to give a formal definition of parsability that is any simpler than our specification in Appendix B, so such a proof would be unenlightening.

The interface of our garbage collector is subtle, and expressing this interface in a type system requires a fair amount of programming at the level of type constructors. Type-checking programs in this language, in turn, involves deciding equivalences of a lot of large constructors that are many reduction steps away from normal form. Our prototype type-checker for LGC decides equivalence using a straightforward, recursive weak-head-normalize and compare algorithm, and while our implementation is not yet serious enough to reach any conclusions about efficiency, preliminary results indicate the amount of work involved is not unreasonably large.

This paper has examined a garbage collector interface based on the one used by the TILT/ML compiler, but considerably simpler. However, we believe that what we have described is sufficient to handle most of the issues that arise in a real collector. For instance, it does not appear diffi-

cult to account for registers (which TILT treats essentially the same as stack slots) or global variables (whose types are fixed).

Our proof-of-concept implementation does not address the possibility of translating higher-order polymorphism into LGC. Higher-order polymorphism arises in the setting of compiling the full ML language, in which abstract parameterized types can occur. TILT is able to use a similar GC table format to the one we have described, even for higher-order polymorphic programs; it performs a program transformation called *reification* to introduce variable bindings for any types of registers or stack locations that are unknown at compile time. We believe that by performing something similar to reification we can translate programs with higher-order polymorphism into LGC, but this remains a topic for future work.

References

[1] Perry Cheng. *Scalable Real-Time Parallel Garbage Collection for Symmetric Multiprocessors*. PhD thesis, Carnegie Mellon University, 2001. Available as Technical Report CMU-CS-01-174.

[2] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.

[3] Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 233–248, September 1999.

[4] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 81–91, June 2001.

[5] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, January 2002.

[6] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

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

[8] George Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, October 1997.

[9] Daniel C. Wang. *Managing Memory With Types*. PhD thesis, Princeton University, 2002.

[10] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *Proceedings of the Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 166–178, London, January 2001.

A Syntax and Typing Rules of LGC

A.1 Syntax

The syntax of LGC is given in Figure 14. As in the body of the paper, we use the shorthand

$$\tau_1 \times \cdots \times \tau_n \triangleq \times(\mathtt{fold}(\mathtt{inj}_2\langle\tau_1, \ldots \mathtt{fold}(\mathtt{inj}_2\langle\tau_n, \mathtt{fold}(\mathtt{inj}_1\star)\rangle)\ldots\rangle))$$

where possible to write product types, and similarly for sum types.

Static Semantics

$\boxed{\vdash \Delta \text{ context}}$

$$\frac{}{\vdash \cdot \text{ context}} \qquad \frac{\vdash \Delta \text{ context}}{\vdash \Delta, j \text{ context}}\ (j \notin \mathrm{Dom}(\Delta))$$

$$\frac{\vdash \Delta \text{ context} \quad \Delta \vdash k \text{ kind}}{\vdash \Delta, \alpha{:}k \text{ context}}\ (\alpha \notin \mathrm{Dom}(\Delta))$$

$\boxed{\Delta \vdash k \text{ kind}}$

$$\frac{(j \in \Delta)}{\Delta \vdash j \text{ kind}} \qquad \frac{}{\Delta \vdash \mathsf{T} \text{ kind}} \qquad \frac{}{\Delta \vdash \mathsf{ST} \text{ kind}} \qquad \frac{}{\Delta \vdash \mathsf{L} \text{ kind}}$$

$$\frac{}{\Delta \vdash 1 \text{ kind}} \qquad \frac{\Delta \vdash k_1 \text{ kind} \quad \Delta \vdash k_2 \text{ kind}}{\Delta \vdash k_1 \to k_2 \text{ kind}}$$

$$\frac{\Delta \vdash k_1 \text{ kind} \quad \Delta \vdash k_2 \text{ kind}}{\Delta \vdash k_1 \times k_2 \text{ kind}} \qquad \frac{\Delta \vdash k_1 \text{ kind} \quad \Delta \vdash k_2 \text{ kind}}{\Delta \vdash k_1 + k_2 \text{ kind}}$$

$$\frac{\Delta, j \vdash k \text{ kind}}{\Delta \vdash \mu j.k \text{ kind}}\ (j \text{ positive in } k)$$

$\boxed{\Delta \vdash c : k}$

$$\frac{(\alpha{:}k \in \Delta)}{\Delta \vdash \alpha : k} \qquad \frac{}{\Delta \vdash \mathtt{int} : \mathsf{T}} \qquad \frac{}{\Delta \vdash \mathtt{bool} : \mathsf{T}}$$

$$\frac{}{\Delta \vdash \star : 1} \qquad \frac{}{\Delta \vdash \mathtt{unit} : \mathsf{T}} \qquad \frac{}{\Delta \vdash \mathtt{void} : \mathsf{T}}$$

$$\frac{}{\Delta \vdash \mathtt{null} : \mathsf{ST}} \qquad \frac{\Delta \vdash c : \mu j.1 + \mathsf{T} \times j}{\Delta \vdash \times(c) : \mathsf{T}}$$

$$\frac{\Delta \vdash c : \mu j.1 + \mathsf{T} \times j}{\Delta \vdash +(c) : \mathsf{T}} \qquad \frac{\Delta \vdash k \text{ kind} \quad \Delta, \alpha{:}k \vdash \tau : \mathsf{T}}{\Delta \vdash \exists \alpha{:}k.\tau : \mathsf{T}}$$

$$\frac{\vdash \Delta, \Delta' \text{ context} \quad \Delta, \Delta' \vdash \sigma : \mathsf{ST}}{\Delta \vdash \mathtt{code}(\Delta'; \sigma) \to 0 : \mathsf{T}} \qquad \frac{\Delta \vdash \tau : \mathsf{T} \quad \Delta \vdash \sigma : \mathsf{ST}}{\Delta \vdash \tau \otimes \sigma : \mathsf{ST}}$$

$$\frac{\Delta \vdash c_1 : \mathsf{L} \quad \Delta \vdash c_2 : \mathsf{L} \quad \Delta \vdash c_3 : k \quad \Delta \vdash c_3 : k}{\Delta \vdash \mathtt{ifeq}(c_1, c_2, c_3, c_4) : k} \qquad \frac{}{\Delta \vdash \ell : \mathsf{L}}$$

$$
\begin{array}{lrcl}
\textit{Kinds} & k & ::= & j \mid \mathsf{T} \mid \mathsf{ST} \mid \mathsf{L} \mid k_1 \to k_2 \mid k_1 \times k_2 \mid k_1 + k_2 \mid \mu j.k \mid \mathtt{1} \\
\textit{Constructors} & c, \tau, \sigma & ::= & \alpha \mid \mathtt{unit} \mid \mathtt{void} \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{code}(\Delta; \sigma \to 0) \mid \times(c) \mid +(c) \mid \exists \alpha{:}k.\tau \mid \mathtt{null} \mid \tau \otimes \sigma \\
 & & \mid & \lambda \alpha{:}k.c \mid c_1\, c_2 \mid \langle c_1, c_2 \rangle \mid \pi_i\, c \mid \mathtt{inj}_i c \mid \mathtt{case}(c_1, \alpha_1.c_2, \alpha_2.c_3) \mid \mathtt{fold}_k c \mid \mathtt{pr}(j, \alpha{:}k, \beta{:}j \to k', c) \\
 & & \mid & \star \mid \ell \mid \mathcal{S}(c : \tau) \mid \mathtt{ifeq}(c_1, c_2, c_3, c_4) \\
\textit{Values} & v & ::= & x \mid \langle\rangle \mid n \mid b \mid \ell \mid v[c] \mid \mathtt{pack}\, \langle c, v \rangle\, \mathtt{as}\, \exists \alpha.\tau \mid \mathtt{blur}(v) \mid v\{c\} \\
\textit{Expressions} & e & ::= & \mathtt{halt}\ v \mid \mathtt{jump}\ v \mid \mathtt{if}\, v\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 \mid \mathtt{case}\, v\, \mathtt{of}\, \mathtt{inj}_1 x_1 \Rightarrow e_1 \mid \cdots \mid \mathtt{inj}_n x_n \Rightarrow e_n \mid \mathtt{let}\, d\, \mathtt{in}\, e \\
\textit{Declarations} & d & ::= & x = v \mid x = aop(v_1, \ldots, v_k) \mid x = \mathtt{inj}_\tau(i, v)[c, \ell] \mid x = \langle v_1, \ldots, v_n \rangle [c, \ell] \mid x = \pi_i\, v \\
 & & \mid & \langle \alpha, x \rangle = \mathtt{unpack}\, v x = \mathtt{sp}(i) \mid \mathtt{sp}(i) := v \mid \mathtt{push}\ v \mid \mathtt{pop}\ i \\
\textit{Blocks} & B & ::= & \lambda(\Delta; \mathtt{sp}{:}\sigma).e \\
\textit{Programs} & P & ::= & \mathtt{letcode}\ \ell_1 = B_1, \ldots, \ell_n = B_n\ \mathtt{in}\ e \\
\textit{Type Contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}k \mid \Delta, j \\
\textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\textit{Memory Types} & \Psi & ::= & \cdot \mid \Psi, \ell : \tau
\end{array}
$$

Figure 14: Syntax of LGC

$$\frac{\Delta \vdash c : \mathsf{L} \quad \Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \mathcal{S}(c : \tau) : \mathsf{T}} \qquad \frac{\Delta \vdash k\ \mathsf{kind} \quad \Delta, \alpha{:}k \vdash c : k'}{\Delta \vdash \lambda \alpha{:}k.c : k \to k'}$$

$$\frac{\Delta \vdash c_1 : k_2 \to k \quad \Delta \vdash c_2 : k_2}{\Delta \vdash c_1\, c_2 : k} \qquad \frac{\Delta \vdash c_1 : k_1 \quad \Delta \vdash c_2 : k_2}{\Delta \vdash \langle c_1, c_2 \rangle : k_1 \times k_2}$$

$$\frac{\Delta \vdash c : k_1 \times k_2}{\Delta \vdash \pi_i c : k_i} \qquad \frac{\Delta \vdash c : k_i \quad \Delta \vdash k_1 + k_2\ \mathsf{kind}}{\Delta \vdash \mathtt{inj}_i c : k_1 + k_2}$$

$$\frac{\Delta \vdash c : k_1 + k_2 \quad \Delta, \alpha_1 : k_1 \vdash c_1 : k \quad \Delta, \alpha_2 : k_2 \vdash c_2 : k}{\Delta \vdash \mathtt{case}(c, \alpha_1.c_1, \alpha_2.c_2) : k}$$

$$\frac{\Delta \vdash \mu j.k\ \mathsf{kind} \quad \Delta \vdash c : k[\mu j.k/j]}{\Delta \vdash \mathtt{fold}_{\mu j.k} c : \mu j.k}$$

$$\frac{\begin{array}{c}(j \text{ only positive in } k') \\ \Delta, j, \alpha{:}k, \beta{:}j \to k' \vdash c : k' \\ \Delta \vdash \mu j.k\ \mathsf{kind} \quad \Delta, j \vdash k'\ \mathsf{kind}\end{array}}{\Delta \vdash \mathtt{pr}(j, \alpha{:}k, \beta{:}j \to k', c) : \mu j.k \to k'[\mu j.k/j]}$$

$$\boxed{\Delta \vdash c_1 = c_2 : k}$$

$$\frac{(\alpha{:}k \in \Delta)}{\Delta \vdash \alpha = \alpha : k} \qquad \frac{}{\Delta \vdash \mathtt{int} = \mathtt{int} : \mathsf{T}} \qquad \frac{}{\Delta \vdash \mathtt{bool} = \mathtt{bool} : \mathsf{T}}$$

$$\frac{}{\Delta \vdash \star = \star : \mathtt{1}} \qquad \frac{}{\Delta \vdash \mathtt{unit} = \mathtt{unit} : \mathsf{T}} \qquad \frac{}{\Delta \vdash \ell = \ell : \mathsf{L}}$$

$$\frac{}{\Delta \vdash \mathtt{void} = \mathtt{void} : \mathsf{T}} \qquad \frac{}{\Delta \vdash \mathtt{null} = \mathtt{null} : \mathsf{ST}}$$

$$\frac{\Delta \vdash c = c' : \mu j.\mathtt{1} + \mathsf{T} \times j}{\Delta \vdash \times(c) = \times(c') : \mathsf{T}}$$

$$\frac{\Delta \vdash c = c' : \mu j.\mathtt{1} + \mathsf{T} \times j}{\Delta \vdash +(c) = +(c') : \mathsf{T}}$$

$$\frac{\Delta \vdash k\ \mathsf{kind} \quad \Delta, \alpha{:}k \vdash \tau = \tau' : \mathsf{T}}{\Delta \vdash \exists \alpha{:}k.\tau = \exists \alpha{:}k.\tau' : \mathsf{T}}$$

$$\frac{\vdash \Delta, \Delta'\ \mathsf{context} \quad \Delta, \Delta' \vdash \sigma = \sigma' : \mathsf{ST}}{\Delta \vdash \mathtt{code}(\Delta'; \sigma) \to 0 = \mathtt{code}(\Delta'; \sigma') \to 0 : \mathsf{T}}$$

$$\frac{\Delta \vdash \tau = \tau' : \mathsf{T} \quad \Delta \vdash \sigma = \sigma' : \mathsf{ST}}{\Delta \vdash \tau \otimes \sigma = \tau' \otimes \sigma' : \mathsf{ST}}$$

$$\frac{\begin{array}{c}\Delta \vdash c_1 = c_1' : \mathsf{L} \quad \Delta \vdash c_2 = c_2' : \mathsf{L} \\ \Delta \vdash c_3 = c_3' : k \quad \Delta \vdash c_4 = c_4' : k\end{array}}{\Delta \vdash \mathtt{ifeq}(c_1, c_2, c_3, c_4) = \mathtt{ifeq}(c_1', c_2', c_3', c_4') : k}$$

$$\frac{\Delta \vdash c = c' : \mathsf{L} \quad \Delta \vdash \tau = tau' : \mathsf{T}}{\Delta \vdash \mathcal{S}(c : \tau) = \mathcal{S}(c' : \tau') : \mathsf{T}}$$

$$\frac{\Delta \vdash k\ \mathsf{kind} \quad \Delta, \alpha{:}k \vdash c = c' : k'}{\Delta \vdash \lambda \alpha{:}k.c = \lambda \alpha{:}k.c' : k \to k'}$$

$$\frac{\Delta \vdash c_1' : k_2 \to k \quad \Delta \vdash c_2' : k_2}{\Delta \vdash c_1\, c_2 = c_1'\, c_2' : k} \qquad \frac{\Delta \vdash c = c' : k_1 \times k_2}{\Delta \vdash \pi_i c = \pi_i c' : k_i}$$

$$\frac{\Delta \vdash c_1 = c_1' : k_1 \quad \Delta \vdash c_2 = c_2' : k_2}{\Delta \vdash \langle c_1, c_2 \rangle = \langle c_1', c_2' \rangle : k_1 \times k_2}$$

$$\frac{\Delta \vdash c = c' : k_i \quad \Delta \vdash k_1 + k_2\ \mathsf{kind}}{\Delta \vdash \mathtt{inj}_i c = \mathtt{inj}_i c' : k_1 + k_2}$$

$$\frac{\begin{array}{c}\Delta \vdash c = c' : k_1 + k_2 \\ \Delta, \alpha_1 : k_1 \vdash c_1 = c_1' : k \quad \Delta, \alpha_2 : k_2 \vdash c_2 = c_2' : k\end{array}}{\Delta \vdash \mathtt{case}(c, \alpha_1.c_1, \alpha_2.c_2) = \mathtt{case}(c', \alpha_1.c_1', \alpha_2.c_2') : k}$$

$$\frac{\Delta \vdash \mu j.k\ \mathsf{kind} \quad \Delta \vdash c = c' : k[\mu j.k/j]}{\Delta \vdash \mathtt{fold}_{\mu j.k} c = \mathtt{fold}_{\mu j.k} c' : \mu j.k}$$

$$\frac{\begin{array}{c}(j \text{ only positive in } k')\\ \Delta, j, \alpha{:}k, \beta{:}j \to k' \vdash c = c' : k' \\ \Delta \vdash \mu j.k \text{ kind} \quad \Delta, j \vdash k' \text{ kind}\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{pr}(j, \alpha{:}k, \beta{:}j \to k', c) = \mathtt{pr}(j, \alpha{:}k, \beta{:}j \to k', c') \\ \quad : \mu j.k \to k'[\mu j.k/j]\end{array}}$$

$$\frac{\Delta \vdash k \text{ kind} \quad \Delta, \alpha{:}k \vdash c_1 : k' \quad \Delta \vdash c_2 : k}{\Delta \vdash (\lambda \alpha{:}k.c_1)c_2 = c_1[c_2/\alpha] : k'}$$

$$\frac{\Delta \vdash c_1 : k_1 \quad \Delta \vdash c_2 : k_2}{\Delta \vdash \pi_i \langle c_1, c_2 \rangle = c_i : k_i}$$

$$\frac{\begin{array}{c}\Delta \vdash c : k_i \quad \Delta \vdash k_1 + k_2 \text{ kind} \\ \Delta, \alpha_1{:}k_1 \vdash c_1 : k \quad \Delta, \alpha_2{:}k_2 \vdash c_2 : k\end{array}}{\Delta \vdash \mathtt{case}(\mathtt{inj}_i\, c, \alpha_1.c_1, \alpha_2.c_2) = c_i[c/\alpha] : k}$$

$$\frac{\begin{array}{c}(j \text{ only positive in } k') \\ \Delta \vdash c_2 : k[\mu j.k/j] \quad \Delta, j, \alpha{:}k, \beta{:}j \to k' \vdash c_1 : k' \\ \Delta \vdash \mu j.k \text{ kind} \quad \Delta, j \vdash k' \text{ kind}\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{pr}(j, \alpha{:}k, \beta{:}j \to k', c_1)(\mathtt{fold}_{\mu j.k} c_2) \\ \quad = c_1[\mu j.k, c_2, \mathtt{pr}(j, \alpha{:}k, \beta{:}j \to k', c_1)/j, \alpha, \beta] : k\end{array}}$$

$$\frac{\Delta \vdash c_3 : k \quad \Delta \vdash c_4 : k}{\Delta \vdash \mathtt{ifeq}(\ell, \ell, c_3, c_4) = c_3 : k}$$

$$\frac{\Delta \vdash c_3 : k \quad \Delta \vdash c_4 : k \quad (\ell_1 \neq \ell_2)}{\Delta \vdash \mathtt{ifeq}(\ell_1, \ell_2, c_3, c_4) = c_4 : k}$$

$\boxed{\Psi; \Delta; \Gamma \vdash v : \tau}$

$$\frac{(x{:}\tau) \in \Gamma}{\Psi; \Delta; \Gamma \vdash x : \tau} \qquad \frac{}{\Psi; \Delta; \Gamma \vdash n : \mathtt{int}} \qquad \frac{}{\Psi; \Delta; \Gamma \vdash b : \mathtt{bool}}$$

$$\frac{}{\Psi; \Delta; \Gamma \vdash \langle\rangle : \mathtt{unit}} \qquad \frac{\Psi; \Delta; \Gamma \vdash v : \tau' \quad \Delta \vdash \tau = \tau' : \mathsf{T}}{\Psi; \Delta; \Gamma \vdash v : \tau}$$

$$\frac{\Psi(\ell) = \tau}{\Psi; \Delta; \Gamma \vdash \ell : \mathcal{S}(\ell : \tau)} \qquad \frac{\Psi; \Delta; \Gamma \vdash v : \mathcal{S}(c : \tau)}{\Psi; \Delta; \Gamma \vdash \mathtt{blur}\, v : \tau}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \mathtt{code}(\alpha{:}k, \Delta; \sigma) \to 0 \quad \Delta \vdash c : k}{\Psi; \Delta; \Gamma \vdash v[c] : \mathtt{code}(\Delta; \sigma[c/\alpha]) \to 0}$$

$$\frac{\Delta \vdash c : k \quad \Delta \vdash v : \tau[c/\alpha]}{\Psi; \Delta; \Gamma \vdash \mathtt{pack}\, \langle c, v \rangle \,\mathtt{as}\, \exists \alpha.\tau : \exists \alpha{:}k.\tau}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \mathcal{S}(c' : \mathtt{code}(\alpha{:}k, \Delta; \sigma) \to 0) \quad \Delta \vdash c : k}{\Psi; \Delta; \Gamma \vdash v\{c\} : \mathcal{S}(c' : \mathtt{code}(\Delta; \sigma[c/\alpha]) \to 0)}$$

$\boxed{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash e}$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \mathtt{code}(\cdot; \sigma) \to 0}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{jump}\, v} \qquad \frac{\Psi; \Delta; \Gamma \vdash v : \mathtt{int}}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{halt}\ v}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \mathtt{bool} \quad \Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash e_1 \quad \Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash e_2}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{if}\, v\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2}$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash v : \tau_1 + \cdots + \tau_n \\ \Psi; \Delta; (\Gamma, x_i{:}\tau_i); \mathtt{sp}{:}\sigma \vdash e_i \text{ for } 1 \leq i \leq n\end{array}}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{case}\, v\, \mathtt{of}\, \mathtt{inj}_1 x_1 \Rightarrow e_1 \mid \cdots \mid \mathtt{inj}_n x_n \Rightarrow e_n}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \tau \quad \Psi; \Delta; \Gamma, x{:}\tau; \mathtt{sp}{:}\sigma' \vdash e}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{let}\ x = v\, \mathtt{in}\, e}$$

$$\frac{\begin{array}{c}(aop \in \{\mathtt{+}, \mathtt{-}, \mathtt{*}\}) \\ \Psi; \Delta; \Gamma \vdash v_i : \mathtt{int} \quad \Psi; \Delta; (\Gamma, x{:}\mathtt{int}); \mathtt{sp}{:}\sigma \vdash e\end{array}}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{let}\ x = aop(v_1, v_2)\, \mathtt{in}\, e}$$

(Similar rules apply to other *aops*)

$$\frac{\Psi; \Delta; \Gamma \vdash v : \exists \alpha{:}k.\tau \quad \Psi; (\Delta, \alpha{:}k); (\Gamma, x{:}\tau); \mathtt{sp}{:}\sigma \vdash e}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{let}\ \langle \alpha, x \rangle = \mathtt{unpack}\, v\, \mathtt{in}\, e}$$

$$\frac{\begin{array}{l}\Delta \vdash \ell = \mathit{retlab}\ \mathit{SGCT}\ s : \mathsf{L} \\ \Delta \vdash s : \mathit{SD} \quad \Delta \vdash \sigma = \mathit{interpS}\ \mathit{SGCT}\ s : \mathsf{ST} \\ \Delta \vdash \tau = \tau_1 + \cdots + \tau_n : \mathsf{T} \\ \Psi; \Delta; \Gamma \vdash v : \tau_i \quad \Psi; \Delta; (x{:}\tau); \mathtt{sp}{:}\sigma \vdash e\end{array}}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{let}\ x = \mathtt{inj}_\tau(i, v)[s, \ell]\, \mathtt{in}\, e}$$

$$\frac{\begin{array}{l}\Delta \vdash \ell = \mathit{retlab}\ \mathit{SGCT}\ s : \mathsf{L} \\ \Delta \vdash s : \mathit{SD} \quad \Delta \vdash \sigma = \mathit{interpS}\ \mathit{SGCT}\ s : \mathsf{ST} \\ \Psi; \Delta; \Gamma \vdash v_i : \tau_i \ (1 \leq i \leq n) \\ \Psi; \Delta; (x{:}\tau_1 \times \cdots \times \tau_n), \mathtt{sp}{:}\sigma \vdash e\end{array}}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{let}\ x = \langle v_1, \ldots, v_n \rangle[s, \ell]\, \mathtt{in}\, e}$$

(where *SD*, *interpS*, *retlab* and *SGCT* are as specified in Appendix B)

$$\frac{\Psi; \Delta; \Gamma \vdash v : \tau_1 \times \cdots \times \tau_n \quad \Psi; \Delta; (\Gamma, x{:}\tau_i); \mathtt{sp}{:}\sigma \vdash e}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{let}\ x = \pi_i\, v\, \mathtt{in}\, e}$$

$$\frac{\Delta \vdash \sigma = \tau_0 \otimes \cdots \otimes \tau_i \otimes \sigma_2 : \mathsf{ST} \quad \Psi; \Delta; (\Gamma, x{:}\tau_i); \mathtt{sp}{:}\sigma \vdash e}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{let}\ x = \mathtt{sp}(i)\, \mathtt{in}\, e}$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma; \mathtt{sp}{:}(\tau_1 \otimes \cdots \otimes \tau_{i-1} \otimes \tau' \otimes \sigma_2) \vdash e \\ \Delta \vdash \sigma = \tau_0 \otimes \cdots \otimes \tau_i \otimes \sigma_2 : \mathsf{ST} \quad \Psi; \Delta; \Gamma \vdash v : \tau'\end{array}}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{sp}(i) := v\, \mathtt{in}\, e}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \tau \quad \Psi; \Delta; \Gamma; \mathtt{sp}{:}(\tau \otimes \sigma) \vdash e}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{push}\ v\, \mathtt{in}\, e}$$

$$\frac{\Delta \vdash \sigma = \tau_1 \otimes \cdots \otimes \tau_i \otimes \sigma_2 : \mathsf{ST} \quad \Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma_2 \vdash e}{\Psi; \Delta; \Gamma; \mathtt{sp}{:}\sigma \vdash \mathtt{pop}\ i\, \mathtt{in}\, e}$$

$\boxed{\vdash P}$

$$\frac{\Psi; \Delta_i; \cdot; \mathtt{sp}{:}\sigma_i \vdash e_i \quad \Psi; \cdot; \cdot; \mathtt{sp}{:}\mathtt{null} \vdash e}{\vdash \mathtt{letcode}\ \ell_1 = B_1, \ldots, \ell_n = B_n\ \mathtt{in}\ e}$$

where: $B_i = \lambda(\Delta_i; \mathtt{sp}{:}\sigma_i).e_i$
$\tau_{ic} = \mathtt{code}(\Delta_i; \sigma_i) \to 0$
$\Psi = \ell_1{:}\tau_{1c}, \ldots, \ell_n{:}\tau_{nc}$

B Implementation of Special Constructors

This appendix contains the definitions and implementations in LX of all the special kinds and constructors used in this paper to specify the interface of the garbage collector. For the sake of readability, the definitions here are written as datakinds and ML-style clausal function definitions rather than the pure μ-kinds and primitive recursion of LGC. To cut down on repetitiveness, we also use some notational shorthand for list kinds. In particular, for any kind k we define

$$k\,\texttt{list} \triangleq \mu j.1 + k \times j$$

and use familiar ML-style notation (including square brackets, `::` for cons, `@` for append, and `map`) when dealing with constructors of these kinds. Also, for any kind k we assume the definition

`datakind` k `option = SOME of` k `| NONE`

Finally, in our clausal function definitions we use ML's pattern-matching syntax, including tuple patterns, `as`, and the wildcard pattern "`_`".

```
datakind B = True | False
fun andalso True b2 = b2
  | andalso False b2 = False
fun orelse True b2 = True
  | orelse False b2 = b2

datakind N = Zero | Succ of N
fun eqN Zero Zero = True
  | eqN Zero _ = False
  | eqN (Succ _) Zero = False
  | eqN (Succ n1) (Succ n2) = eqN n1 n2

fun memberN (n:N) ([]:N list) = False
  | memberN (n:N) ((m::ms):N list) =
    case eqN n m of True => True
                  | False => memberN n ms
fun containN ([]:N list) (s2:N list) = True
  | containN (n::ns) s2 =
    andalso (memberN n s2) (containN ns s2)

(* Type representations, Sec. 2.4. *)
datakind TR = Int | Prod of T list | ...
fun nthTR ([]:TR list) (n:N) = NONE
  | nthTR (t::ts) Zero = SOME t
  | nthTR (t::ts) (Succ n) = nthTR ts n

fun interp Int = int
  | interp (Prod ts) = ×  ts
  | ...

(* A value of type R(c) will be a first inj if
   c is notrace, a second inj if c is trace. *)
fun R Int = unit + void
  | R (Prod _) = void + unit
  | ...

fun istrace Int = False
  | istrace (Prod _) = True
  | ...

(* Table slots, Sec. 4.2. *)
datakind TSlot = Trace | Notrace
               | Var of N | Rep of N

fun eqTSlot Trace Trace = True
  | eqTSlot Notrace Notrace = True
  | eqTSlot (Var n) (Var m) = eqN n m
  | eqTSlot (Rep n) (Rep m) = eqN n m
  | eqTSlot _ _ = False

fun TSlotl_eq [] [] = True
  | TSlotl_eq (t1::ts1) (t2::ts2) =
    andalso (eqTSlot t1 t2) (TSlotl_eq ts1 ts2)
  | TSlotl_eq _ _ = False

(* Slots, Sec. 4.2. *)
datakind Slot = KnownSlot of TR
              | VarSlot of N
              | RepSlot of N

fun interpsl (trl:TR list) (KnownSlot t) = interp t
  | interpsl trl (VarSlot n) =
    (case nthTR trl n of SOME t => interp t
                       | NONE => void)
  | interpsl trl (RepSlot n) =
    (case nthTR trl n of SOME t => R(t)
                       | NONE => void)
fun interpslL trl (sl:Slot list) = map (interpsl trl) sl

fun Slot2TSlot (KnownSlot t) =
    case istrace t of True => Trace | False => Notrace
  | Slot2TSlot (VarSlot n) = Var n
  | Slot2TSlot (RepSlot n) = Rep n
fun SlotL2TSlotL (ss:Slot list) = map Slot2TSlot ss

fun getVars ([]:Slot list) = []
  | getVars ((Var n)::rest) = n::(getVars rest)
  | getVars (_::rest) = getVars rest
fun getReps ([]:Slot list) = []
  | getReps ((Rep n)::rest) = n::(getReps rest)
  | getReps (_::rest) = getReps rest

(* Frame templates and the table, Sec. 4. *)
kind FT = TSlotl * TSlotl
datakind DD = NilDD
            | ConsDD of L * FT * DD
fun lookupDD (l:L) NilDD = NONE
  | lookupDD l (ConsDD (l',ft,dd)) =
    ifeq (l,l',SOME ft,
             lookupDD l dd)

(* Frame descriptors, Sec. 3.2 *)
kind FD = L * SlotL * T * SlotL * TRlist

(* Checking frame descriptors, Sec. 3.2. *)
fun checkVarsReps (l,ss1,rt,ss2,trl) =
    containN ((getVars ss1) @ (getVars ss2))
             ((getReps ss1) @ (getReps ss2))

fun checkFD dd (fd as (l,s1,r,s2,t):FD) =
    andalso (checkVarsReps fd)
     (case lookupDD l dd of
        SOME (ts1,ts2) => andalso
                  (TSlotl_eq ts1 (SlotL2TSlotL s1))
                  (TSlotl_eq ts2 (SlotL2TSlotL s2))
      | NONE => False)

(* Stack descriptors, Sec. 3.2. *)
datakind SD = Base of L
            | Cons of FD * SD

fun addtost ([]:T list) (st:ST) = st
  | addtost (t::ts) st = t ⊗ (addtost ts st)
fun addslotstost (trl:TR list) (sl:Slot list) (st:ST) =
    addtost (interpslL trl sl) st

(* Interpreting stack descriptors, Sec. 3.2. *)
fun interpS_ dd (Base l) = (null,l)
  | interpS_ dd (Cons (fd as (l,s1,r,s2,t),sd)) =
    (case checkFD dd fd of
      True =>
        let (l',rest) = interpS_ dd sd
        in addslotstost t s1
              (S(l' : code(r::rest)))⊗
              (addslotstost t s2 rest)
    | False => (void⊗null,l))

fun interpS (dd:DD) (sd:SD) = π1 (interpS_ dd sd)
fun retlab  (dd:DD) (sd:SD) = π2 (interpS_ dd sd)
```