Certifying Compilation for a Language with Stack Allocation

Limin Jia Frances Spalding David Walker Neal Glew
Princeton University Intel Corporation
{ljia, frances, dpw}@cs.princeton.edu aglew@acm.org

Princeton University Technical Report TR-724-05

Abstract

This paper describes an assembly-language type system capable of ensuring memory safety in the presence of both heap and stack allocation. The type system uses linear logic and a set of domain-specific predicates to specify invariants about the shape of the store. Part of the model for our logic is a tree of "stack tags" that tracks the evolution of the stack over time. To demonstrate the expressiveness of the type system, we define Micro-CLI, a simple imperative language that captures the essence of stack allocation in the Common Language Infrastructure. We show how to compile well-typed Micro-CLI into well-typed assembly.

1 Introduction

The grand challenge for the proof-carrying code paradigm is twofold: to develop an expressive logic for easily specifying and proving properties of low-level programs, and to develop certifying-compiler technology that automatically generates such proofs from the information embedded in high-level programs. Over the last eight years, tremendous progress has been made, but finding general-purpose logics for specifying memory-management properties and developing certifying-compiler technology targeting them continues to be problematic.

In this paper, we develop a low-level memory model that treats pointers to heap and stack locations uniformly in the presence of aliased data structures. Our type system extends previous work on the development of typed assembly languages [15, 14, 2] by using linear logic and domain-specific predicates to specify preconditions for assembly code. These preconditions describe the contents and shape of the stack and heap, and provide a safe but flexible way to allocate, deallocate, reference, and reuse data on the stack. The typing discipline is powerful enough to represent general stack pointers including pointers that might point into the heap or into the stack.

Defining an expressive type system and testing it on some ad hoc examples does not demonstrate that it is useful for certifying compilation. It is equally necessary to show that there is an algorithm for generating type-safe low-level code from type-safe high-level code. To demonstrate that our system does indeed form a foundation for certifying compilation in the presence of complex and important memory invariants, we define a high-level language called Micro-CLI, which captures the stack-allocation features found in the Common Language Infrastructure (CLI) [6, 9], and we give a translation of well-typed Micro-CLI into well-typed assembly code.

This is the first paper (1) to develop a unified low-level memory model for the heap and the stack in the presence of aliasing, and (2) to give an algorithm for a type-preserving compilation from a language with both heap and stack allocation to our typed assembly language. Previous work has either developed type (or proof) systems that are too restrictive to handle important, realistic invariants such as those present in the CLI, or it has not demonstrated any sort of compiler strategy for generating stack-based proof-carrying code.

1.1 Background and Related Work

Most earlier work on typed assembly language [14] and proof-carrying code [4] allows reasoning about data allocated in the current stack frame, but not much more. STAL [14] allows pointers deep into the stack, supporting a particular implementation of exceptions, but its type system is not very polymorphic: STAL distinguishes between stack and

heap pointers and tracks the exact ordering of the stack pointers into the stack. Consequently STAL does not support general stack allocation.

The logic we develop in this paper unifies a number of independent ideas in the literature. The first idea, which appears in O'Hearn, Reynolds, and Yang's separation logic [12, 16, 17, 18], is to use a substructural logic to describe memory as a collection of disjoint pieces. Guaranteed disjointness allows us to write "strong update" rules—in a type system, this means that the same piece of memory (say, a stack slot) can have different types at different times. In earlier work [2, 1], we attempted to use this idea alone to build a general system for assembly-level memory management. Unfortunately, while the logic is sound and quite flexible, we were unable to use it as a target for certifying compilation since we could not find a general translation for the types of aliased data structures into the logic.

Closely related to the work on separation logic is earlier work on alias types [19]. In alias types [19], the capabilities of locations can be either linear (unaliased) or non-linear (aliased any number of times); strong updates are allowed only on linear capabilities; invariant updates are allowed on non-linear capabilities. Alias types and separation logic are incomparable in power as alias types include formulas for aliased memory whereas separation logic contains implication, disjunction, and other connectives. This paper brings the power of both together in a single system.

In concurrent work, Morrisett et al. [13] have extended alias types in a different direction. They have studied methods for temporarily breaking the invariant associated with aliased data. Other research along the same line includes Foster et al.'s *restrict* primitive [8] and DeLine and Fähndrich's *adoption* and *focus* [7]. All of these efforts are carried out in a high-level language and do not concern themselves with presenting a uniform model of stack and heap memory.

A third area of related work includes region-based memory management [20, 3, 5, 11]. In region systems, types are tagged with the name of the region in which they are allocated. The type system tracks the regions that are currently live and disallows access to data structures that live in dead regions. Our assembly language employs a variant of the region idea by tagging descriptions of memory with *version numbers*, and keeping track of the valid versions. We do not allow programs to reason about memory using out-of-date descriptions. A crucial difference between our system and previous region-based systems is that we unify the idea of regions with a low-level model of stack and heap allocation and aliasing invariants. In order to do so, we use a more elaborate structure, a *tag tree*, in our memory model to keep track of the live "version numbers".

The richest source-level type-safe memory management system we are aware of is Grossman et al.'s Cyclone language [11, 10]. Cyclone allows data to be allocated on the stack. It uses a region-based type system including an outlives relation on regions. We can define this *outlives* relation within our logic, but we need to do more research to determine if we can compile all aspects of Cyclone's stack allocation discipline into our assembly language.

2 Informal Development

In this section, we summarize the major technical components that comprise our memory logic.

2.1 The Basic Setup

The first step in our development is to choose a simple way to describe the contents of individual memory locations, regardless of whether they are on the heap or on the stack. Since the type of the stack changes as the program progresses, we must use per-program point types for the stack. To unify stack and heap pointers into one framework, we need per-program point types for the heap as well. The general approach that we will use is to describe the whole state of the machine with formulas in a substructural logic.

One crucial operator in these logics is the multiplicative conjunction ("star" in the logic of bunched implications, also called tensor in this paper). The formula $F_1 \otimes F_2$ describes a state that can be partitioned into two disjoint parts, one described by F_1 and one described by F_2. The formula $(\ell \Rightarrow \tau)$ describes a store consisting of a single location ℓ that contains a value of type τ; it is also called a linear capability for ℓ. Before any dereference/assignment operation on location ℓ, $(\ell \Rightarrow \tau)$ must be proven, so that we know that the location exists, and for dereference, what its type is. If the state before an assignment to ℓ is described by $(\ell \Rightarrow \tau_1) \otimes F$, then the state after the assignment is described by $(\ell \Rightarrow \tau_2) \otimes F$ where τ_2 is the type of the value stored. We can use this as the basis for a typing rule for store

instructions, and this type of rule is usually called a strong-update rule, as the old type τ_1 is ignored and completely replaced by τ_2.

2.2 Aliasing of Heap Locations

Formulas involving $(\ell \Rightarrow \tau)$ and $\otimes$ are very precise, but inflexible. Consider a function that takes two integer pointers as arguments. It might be described with the type $\forall \ell_1, \ell_2.(\,(\ell_1 \Rightarrow \mathbf{int}) \otimes (\ell_2 \Rightarrow \mathbf{int})\,) \multimap \cdots$. Now if it is called with the same pointer, say ℓ, for both parameters, then it must be called in a state described by $(\ell \Rightarrow \mathbf{int}) \otimes (\ell \Rightarrow \mathbf{int})$. This formula is impossible to satisfy as it requires partitioning memory into two disjoint pieces that both have a location ℓ in their domain.

To solve this problem, we adapt the idea of non-linear capabilities from alias types [19] and L^3 [13]. We add a new predicate $(\mathsf{frzn}\ \ell\ \tau)$, called a frozen or unrestricted capability, which is similar to the linear capability that describes a location ℓ of type τ. We partition the state into linear and frozen memory. Linear capabilities describe locations in the linear memory; frozen capabilities describe locations in the frozen memory. The tensor operation partitions the linear memory between its subformulas, but shares the frozen memory to both subformulas. Thus, $(\mathsf{frzn}\ \ell\ \tau) \otimes (\mathsf{frzn}\ \ell\ \tau)$ holds of a state whose frozen memory contains a location ℓ of type τ. The function above can be given type $\forall \ell_1, \ell_2.((\mathsf{frzn}\ \ell_1\ \mathbf{int}) \otimes (\mathsf{frzn}\ \ell_2\ \mathbf{int})) \multimap \cdots$ and be called with the same pointer for both arguments. To make unrestricted capabilities safe, we must use an invariant update rule. This rule allows a store to location ℓ in a state $(\mathsf{frzn}\ \ell\ \tau) \otimes F$ if the value being stored has type τ, but it does not allow the type of ℓ to be changed. To get unrestricted capabilities, there is a *freezing* operation that transfers a location from the linear memory to the frozen memory, and it takes a state described by $(\ell \Rightarrow \tau) \otimes F$ to one described by $(\mathsf{frzn}\ \ell\ \tau) \otimes F$. Once frozen a location stays frozen and its type cannot change for the remainder of execution.

One technicality is worth noting. Unrestricted capabilities can be duplicated and dropped and thus act like the unrestricted formula $!F$ in linear logic. Since several of our domain-specific predicates have similar properties, we wrap all of these predicates in the unrestricted connective $(!)$. The duplication and weakening rules for unrestricted formulas in linear logic take care of the duplicating and dropping of these predicates. So the function above actually has type $\forall \ell_1, \ell_2.(!\,(\mathsf{frzn}\ \ell_1\ \mathbf{int}) \otimes !\,(\mathsf{frzn}\ \ell_2\ \mathbf{int})) \multimap \cdots$.

Using unrestricted capabilities and existential quantification, we can express pointer types if we arrange all pointed-to locations to be in the frozen memory. A pointer to τ is expressed as $\exists \ell.(\mathbf{S}(\ell) \otimes !\,(\mathsf{frzn}\ \ell\ \tau))$ where $\mathbf{S}(\cdot)$ is the singleton type constructor. This type states both that the value is some location and that the location contains a τ. Note that in our logic, we use more expressive formulas than just basic types to describe values. Only formulas that do not refer to linear memory can be used to describe values. We call them *pure formulas* and denote them by G. In Section 3, we will introduce the formal syntax of pure formulas.

2.3 Version Numbers for Stack Locations

Unfortunately the idea of unrestricted capabilities cannot be applied to both stacks and heaps in a uniform way. The problem is that freezing constrains the type of a location for the remainder of execution, but stack frames, in general, have shorter lifetimes than the rest of execution and are reused. The mechanisms described so far do not allow reuse of frozen memory.

Consider the operation of allocating an integer cell on the top of the stack. This operation results in memory described by $(r_{\mathrm{sp}} \Rightarrow \mathbf{S}(\ell)) \otimes (\ell \Rightarrow \mathbf{int}) \otimes \ldots$ where r_{sp} is a register holding the stack pointer. The top of the stack can be frozen and then passed to a function. In this case, the memory typing becomes $(r_{\mathrm{sp}} \Rightarrow \mathbf{S}(\ell)) \otimes !\,(\mathsf{frzn}\ \ell\ \mathbf{int}) \otimes \ldots$. Now imagine that the function terminates and returns to its caller. The caller would like to reuse the stack and put a different value into location ℓ. However, the logic described so far does not allow anything other than integers in ℓ for the remainder of execution. We need the flexibility to freeze a location for some amount of time, but later reuse it in any way we desire. To achieve this discipline, we introduce a sort of "version numbering" scheme for our locations that is somewhat reminiscent of region-based type systems.

First, we assume a countably-infinite set of *tags* ranged over by k. These tags name versions of locations. When a location is used for the first time or is reused, a new tag is chosen to name that use. The machine state includes a *tag tree* that keeps track of all these tags. This tag tree is only used for typing purposes and can be erased in forming an underlying untyped machine state.

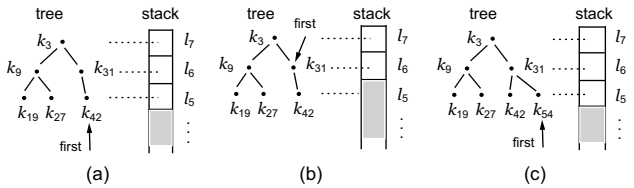

Figure 1: Example Memory Stack Tag Tree

These tags form a tree due to the LIFO nature of the stack. In the current work, heap cells are not versioned in the same way that stack cells are, and we assign the special tag H to all heap locations. The bottom (in the sense of the stacking discipline, often the highest address) of the stack goes through a sequence of versions that we can think of as the ordered children of H. The second to bottom location goes through a sequence of versions that exist entirely within the lifetime of the first version of the bottom of the stack, then a sequence of versions that exist entirely within the lifetime of the second version of the bottom of the stack, and so on. We can think of these versions as the ordered children of the respective version of the bottom of the stack. In this way, the versions of locations that make up the stack during program execution form a tree. The right-most spine of the tree contains the tags of the current versions of locations. In addition, one of these tags is marked as the current top of the stack. When the stack is popped, this marker is moved to its parent. When the stack is pushed, a fresh tag is selected and is added to the tag tree as the last child of the current top of the stack, and it becomes the current top of the stack. Figure 1 displays the transformation of the tag tree and stack during a sequence of push and pop operations. A stack pop occurs between Figure 1(a) and Figure 1(b). A stack push occurs between Figure 1(b) and Figure 1(c).

Second, unrestricted and linear capabilities include both a tag and a location. For example, $!\,(\mathsf{frzn}\ k.\ell\ \tau)$ says that version k of location ℓ is frozen at type τ; similarly, $(k.\ell \Rightarrow \tau)$ holds if location ℓ currently has version k and holds a value of type τ. A formula can include unrestricted capabilities with old tags; these formulas describe previous states of the location but not the current one.

Third, a frozen capability can be used to load from or store into a location only if the tag is the current version of the location. Even though unrestricted capabilities with old tags still exist in the formula, they cannot be used to access the locations.

Fourth, our logic includes formulas to reason about which tags are current. In particular, we have the two formulas $\mathsf{first}(k)$ and $!(k_2{=}k_1{+}n)$. The formula $\mathsf{first}(k)$ holds if the current top of the stack in the tag tree is k. The formula $!(k_2{=}k_1{+}n)$ holds if k_2 appears in the tree as the n-th ancestor of k_1. Together, $\mathsf{first}(k_1)$ and $!(k_2{=}k_1{+}n)$ mean that k_2 is on the right spine of the tag tree and therefore that it is a current tag.

Using these predicates and existential quantification, we are able to describe data structures that point deep into the stack. For example, if location $k.\ell$ contains location $k_1.\ell_1$ which is deeper in the stack, and $k_1.\ell_1$ contains an integer, then formula $(\mathsf{frzn}\ k.\ell\ (\exists xk \exists x\ell \exists n.\mathbf{S}(xk.x\ell) \otimes !(xk{=}k{+}n) \otimes !\,(\mathsf{frzn}\ xk.x\ell\ \mathbf{int})))$ describes location $k.\ell$, where $k_1.\ell_1$ and some natural number are the witnesses of the existential package.

To summarize, we have a logic that describes memories and tag trees. This logic includes linear logic to express separation and aliasing of memory parts, linear location types, frozen location types, and currency of tags.

Comments The tree appears to be a fairly heavyweight component in the memory model. We retain the complete tree structure in our model, even though parts of the tree are dead, because it facilitates the proof of certain monotonicity properties of our logic. More specifically, once a formula that relates tags such as $!(k_2{=}k_1{+}n)$ are satisfied by a particular model, we can prove that they are satisfied by all future models that may be generated as the program executes. Without the tree structure to describe the relationships between past tags, it would be more difficult to obtain this critical property.

The "tags" resemble the region names in the Capability Calculus [5]. Each stack location can be treated as a region containing only that location. We might be able to solve part of the problem by using some of the techniques in the Capability Calculus. However, our work focuses on developing a logic which provides a unified description of heap and stack locations in a low-level memory model. The Capability Calculus does not have such view of the memory

model.

3 A Formal Memory Model

In this section we formalize the logic that was informally described in Section 2. After introducing all the syntactic constructs, we will focus on the semantics of the logic. The semantic judgments give meanings to formulas in terms of memory layout and typing information, which is important to specify the memory safety policy of assembly programs. Section 4 will use this logic as the basis for a type system for assembly language. At the end of this section, we will give an example to show how various semantic judgments are used to decide if a memory satisfies a certain formula. The complete set of deduction rules is given in Figure 2.

3.1 Syntax

The syntactic constructs in our logic are given below; i ranges over integers, $i\omega$ over integer and infinity (written ∞), ℓ over locations, sk over tags for stack locations, and k over stack tags and the heap tag (written H). We use xi to range over integer variables, $xi\omega$ over infinite integer variables, $x\ell$ over location variables, xk over tag variables, xt over type variables, and xf over formula variables. We consider all syntactic constructs equivalent up to alpha conversion.

$$
\begin{array}{llll}
\textit{Int Expr} & ei & ::= & xi \mid i \mid ei_1 + ei_2 \mid -ei \\
\textit{Infinite Int Expr} & ei\omega & ::= & xi\omega \mid i\omega \mid ei\omega_1 + ei\omega_2 \mid -ei\omega \\
\textit{Location Expr} & e\ell & ::= & x\ell \mid \ell \mid e\ell + ei \\
\textit{Tags} & ek & ::= & xk \mid H \mid sk \\
\textit{General Loc} & g & ::= & ek.e\ell \mid r \\
\textit{Types} & \tau & ::= & xt \mid \mathbf{S}(ei) \mid \mathbf{S}(ek.e\ell) \mid (F) \to 0 \\
\textit{Bindings} & b & ::= & xi{:}\mathsf{I} \mid xi\omega{:}\mathsf{I}^{\omega} \mid x\ell{:}\mathsf{L} \mid xk{:}\mathsf{TG} \mid xt{:}\mathsf{T} \mid xf{:}\mathsf{F} \\
\textit{Predicates} & P & ::= & \tau \mid (g \Rightarrow G) \mid ek_2{=}ek_1{+}ei\omega \\
 & & & \mid \mathsf{first}(ek) \mid \mathsf{frzn}\ ek.e\ell\ G \mid ei\omega \geq 0 \\
 & & & \mid \mathsf{more}^{\leftarrow}(e\ell) \mid \mathsf{more}^{\rightarrow}(e\ell) \\
\textit{Pure Formula} & G & ::= & \tau \mid \mathbf{1} \mid G_1 \otimes G_2 \mid G_1 \multimap G_2 \mid G_1 \oplus G_2 \\
 & & & \mid G_1 \,\&\, G_2 \mid !\,F \mid \exists b.G \\
\textit{Formula} & F & ::= & xf \mid P \mid \mathbf{1} \mid F_1 \otimes F_2 \mid F_1 \multimap F_2 \mid \top \\
 & & & \mid F_1 \,\&\, F_2 \mid \mathbf{0} \mid F_1 \oplus F_2 \mid !\,F \mid \exists b.F
\end{array}
$$

We assume that locations come with an operation $\ell + i$ that intuitively returns the location i locations away from ℓ. Infinity is used in $ek_2{=}ek_1{+}ei\omega$ to express the difference in levels between a stack tag and the heap tag. In particular, $H{=}sk{+}\infty$ holds for any stack tag sk in the tag tree.

A general location is either a tagged location $ek.e\ell$ or a register r. Basic types τ consist of type variables, singleton types, and codes type $(F) \to 0$. As we explained briefly in the previous section, a syntactic category of *pure formulas* includes all the formulas that can be used to describe a value. A pure formula G can be a basic type, or $\mathbf{1}$, or an unrestricted formula $!\,F$, or it can be constructed using connectives: $\otimes$, $\multimap$, $\&$, $\oplus$, $\exists b$ from pure sub-formulas.

Models in our logic are pairs of a memory m and a tag tree t. The semantic judgments define which memory and tag tree pairs satisfy a formula. To precisely define the semantic judgments, we define a number of auxiliary judgments that are connected by contexts. The syntax for all these constructs is defined below.

A store value is an integer, a tagged location, or a code location c. A tag tree t is a quadruple (H, T, sk, fst) consisting of the heap tag H, the tree proper T, the stack tag of the top of the stack sk, and a witness *fst* for the predicate $\mathsf{first}(k)$ to indicate whether the top of the stack is known. A tree T consists of a root stack tag and an ordered list of subtrees. The code context Ψ maps each code label c to its type; the frozen memory type context Π maps a frozen location $k.\ell$ to the describing formula G of the value that location ℓ contains at version number k.

$\boxed{\Theta \parallel \Gamma; \Delta \vdash F}$

$$\frac{}{\Theta \parallel \Gamma; u{:}F \vdash F}\ \textit{Hyp (L)} \qquad \frac{}{\Theta \parallel \Gamma, u{:}F; \cdot \vdash F}\ \textit{Hyp (U)}$$

$$\frac{}{\Theta \parallel \Gamma; \cdot \vdash \mathbf{1}}\ 1I \qquad \frac{\Theta \parallel \Gamma; \Delta \vdash \mathbf{1} \quad \Theta \parallel \Gamma; \Delta' \vdash C}{\Theta \parallel \Gamma; \Delta, \Delta' \vdash C}\ 1E$$

$$\frac{\Theta \parallel \Gamma; \Delta_1 \vdash F_1 \quad \Theta \parallel \Gamma; \Delta_2 \vdash F_2}{\Theta \parallel \Gamma; \Delta_1, \Delta_2 \vdash F_1 \otimes F_2}\ \otimes I$$

$$\frac{\Theta \parallel \Gamma; \Delta \vdash F_1 \otimes F_2 \quad \Theta \parallel \Gamma; \Delta', u_1{:}F_1, u_2{:}F_2 \vdash C}{\Theta \parallel \Gamma; \Delta, \Delta' \vdash C}\ \otimes E$$

$$\frac{\Theta \parallel \Gamma; \Delta, u{:}F_1 \vdash F_2}{\Theta \parallel \Gamma; \Delta \vdash F_1 \multimap F_2}\ \multimap I \qquad \frac{\Theta \parallel \Gamma; \Delta \vdash F_1 \multimap F_2 \quad \Theta \parallel \Gamma; \Delta_1 \vdash F_1}{\Theta \parallel \Gamma; \Delta, \Delta_1 \vdash F_2}\ \multimap E$$

$$\frac{}{\Theta \parallel \Gamma; \Delta \vdash \top}\ \top I$$

$$\frac{\Theta \parallel \Gamma; \Delta \vdash F_1 \quad \Theta \parallel \Gamma; \Delta \vdash F_2}{\Theta \parallel \Gamma; \Delta \vdash F_1 \,\&\, F_2}\ \&I$$

$$\frac{\Theta \parallel \Gamma; \Delta \vdash F_1 \,\&\, F_2}{\Theta \parallel \Gamma; \Delta \vdash F_1}\ \&E1 \qquad \frac{\Theta \parallel \Gamma; \Delta \vdash F_1 \,\&\, F_2}{\Theta \parallel \Gamma; \Delta \vdash F_2}\ \&E2$$

$$\frac{\Theta \parallel \Gamma; \Delta \vdash F[a/x] \quad a \in K}{\Theta \parallel \Gamma; \Delta \vdash \exists x{:}K.F}\ \exists I$$

$$\frac{\Theta \parallel \Gamma; \Delta \vdash \exists x{:}K.F \quad \Theta, x{:}K \parallel \Gamma; u{:}F \vdash C}{\Theta \parallel \Gamma; \Delta \vdash C}\ \exists E$$

$$\frac{\Theta \parallel \Gamma; \cdot \vdash F}{\Theta \parallel \Gamma; \cdot \vdash\, !\, F}\ !I \qquad \frac{\Theta \parallel \Gamma; \Delta \vdash\, !\, F \quad \Theta \parallel \Gamma, u{:}F; \Delta' \vdash C}{\Theta \parallel \Gamma; \Delta, \Delta' \vdash C}\ !E$$

$$\frac{F_1 \equiv F_2}{\Theta \parallel \Gamma; F_1 \vdash F_2}\ \textit{Equiv} \qquad \frac{\cdot \parallel \Gamma; F_2 \vdash F_1}{\cdot \parallel \Gamma; (F_1) \to 0 \vdash (F_2) \to 0}\ \textit{Code}$$

$$\frac{\Theta \parallel \Gamma; F_1 \vdash F_2}{\Theta \parallel \Gamma;\ (r \Rightarrow F_1) \vdash\ (r \Rightarrow F_2)}\ \textit{Reg}$$

$$\frac{}{\Theta \parallel \Gamma; \cdot \vdash\, !\, (ek = ek - 0) \otimes \top}\ \textit{O-Reflex} \qquad \frac{}{\Theta \parallel \Gamma; \cdot \vdash\, !\, (ek = H - \infty) \otimes \top}\ \textit{O-Heap}$$

$$\frac{\Theta \parallel \Gamma; \Delta_1 \vdash\, !\, (ek_1 = ek_2 - ei\omega_1) \quad \Theta \parallel \Gamma; \Delta_2 \vdash\, !\, (ek_2 = ek_3 - ei\omega_2)}{\Theta \parallel \Gamma; \Delta_1, \Delta_2 \vdash\, !\, (ek_1 = ek_3 - (ei\omega_1 + ei\omega_2))}\ \textit{O-Trans}$$

Figure 2: Logical Deduction Rules

Store Value	sv	::=	$i \mid k.\ell \mid c$
Memory	m	$\in$	$Loc \cup Reg \rightharpoonup Sval$
Tree	T	::=	$(sk; T_1, T_2 \cdots, T_n)$
Hasfirst	*fst*	::=	*absent* | *present*
Tag Tree	t	::=	(H, T, sk, fst)
Code Contexts	Ψ	::=	$\cdot \mid \Psi, c : (F) \to 0$
Frozen Memory Typing	Π	::=	$\cdot \mid \Pi, k.\ell : F$

Operations on Memories To specify the semantics of our logic, we need some notation and operations on various components of our model.

- $\overline{g}$ denotes the untagged location: $\overline{r} = r$ and $\overline{ek.e\ell} = e\ell$.
- $m(\overline{g})$ denotes the store value stored at location $\overline{g}$.
- $m\,[\,\overline{g} := sv\,]$ denotes a memory m' in which $\overline{g}$ maps to sv but is otherwise the same as m.
- $m_1 \uplus m_2$ denotes the union of disjoint memories. It is undefined if the memories are not disjoint.

Operations on Trees A tag tree $t = (H, T, sk, \mathit{fst})$ is well formed ($\mathsf{wf}(t)$) if sk is on the right spine of T. The live tags of a tag tree $t = (H, T, sk, \mathit{fst})$ ($\mathsf{Live}\,(t)$) are the right spine of T up to sk, and also H.

Lastly, we define newFirst and delFirst operations. Operation $\mathsf{newFirst}(t, k)$ adds k as the last child of the current stack top and makes k the stack top. Operation $\mathsf{delFirst}(t)$ moves the stack top up one level.

We define $\mathsf{newFirst}(t, k)$ using an auxiliary function $newFirst'(\,T, sk, k)$ which which takes three arguments: the tree of stack tags T, the tag of the top of the stack sk, and the new tag k we would like to insert. It returns the new tree of stack tags with k inserted. If sk is the root of T, then we insert $(k, \cdot)$ as the rightmost child as of sk, otherwise, we recursively call $newFirst'$ on the rightmost child of T.

$$\frac{}{newFirst'((sk; T_1, \ldots, T_n), sk, k) = (sk; T_1, \ldots, T_n, (k; \cdot))}$$

$$\frac{T' = newFirst'(T_n, sk, k) \quad sk \neq sk'}{newFirst'((sk'; T_1, \ldots, T_n), sk, k) = (sk'; T_1, \ldots, T_{n-1}, T')}$$

$$\frac{T' = newFirst'(T, sk, k)}{\mathsf{newFirst}((H, T, sk, \mathit{fst}), k) = (H, T', k, \mathit{fst})}$$

We define $\mathsf{delFirst}(t)$ using an auxiliary function $delFirst'$ that takes as arguments: the parent tag of T sk', the current tag of the top of the stack sk, and T, and returns the new tag of the top of the stack. If sk is the root of T, we return sk', otherwise, we recursively call $delFirst'$ on the root node of T, sk, and the rightmost child of T.

$$\frac{}{delFirst'(sk', sk, (sk; T_1, \ldots, T_n)) = sk'}$$

$$\frac{sk_1 = delFirst'(sk'', sk, T_n) \quad sk' \neq sk''}{delFirst'(sk', sk, (sk''; T_1, \ldots, T_n)) = sk_1}$$

$$\frac{sk'' = delFirst'(sk', sk, T_n) \quad T = (sk'; T_1, \ldots, T_n)}{\mathsf{delFirst}(H, T, sk, \mathit{fst}) = (H, T, sk'', \mathit{fst})}$$

The semantics of tensor ($\otimes$) is to disjointly partition the linear parts of the model between the sub-formulas. The linear parts of the model are the linear memory and the stack top. To formalize this partitioning, we overload the merge operators($\uplus$) to merge two disjoint tag trees.

$$\mathit{fst} \uplus \mathit{absent} \stackrel{\text{def}}{=} \mathit{fst} \qquad \mathit{absent} \uplus \mathit{fst} \stackrel{\text{def}}{=} \mathit{fst}$$
$$(H, T, sk, \mathit{fst}_1) \uplus (H, T, sk, \mathit{fst}_2) \stackrel{\text{def}}{=} (H, T, sk, \mathit{fst}_1 \uplus \mathit{fst}_2)$$

Abbreviations We define the following commonly used abbreviations.

$$\begin{array}{lcl}
\mathbf{int} & \stackrel{\text{def}}{=} & \exists xi{:}\mathsf{I}.\mathbf{S}(xi) \\
\mathbf{ns} & \stackrel{\text{def}}{=} & \exists xf{:}\mathsf{F}.xf \\
ek_1 \,\mathsf{outlives}\, ek_2 & \stackrel{\text{def}}{=} & \exists xi\omega{:}\mathsf{I}^{\omega}.\, ek_1{=}ek_2{+}xi\omega \otimes !\,(xi\omega \geq 0) \\
\mathsf{live}\,(ek) & \stackrel{\text{def}}{=} & \exists xk{:}\mathsf{TG}.\,\mathsf{first}(xk) \otimes !\,(\,ek \,\mathsf{outlives}\, xk)
\end{array}$$

3.2 Semantics

There are six semantic judgments:

$m, t \vDash^{\Psi} F \,at\, p$ overall	model m, t satisfy F
$\Pi_{old}; t \vDash^{\Psi} m : \Pi$	frozen memory m has type Π
$\Pi_{old}; \Pi; m; t \vDash^{\Psi} F \,at\, p$	linear state satisfies F
$\vDash^{\Psi} sv : \tau$	store value sv has type τ
$F_1 \vDash^{\Psi} F_2$	F_2 is a semantic consequence F_1
$F_1 \equiv F_2$	F_1 and F_2 are equivalent

The place p at the end of the first and third judgment is either root $*$ or a location g—the latter is used to specify the semantics of $(g \Rightarrow G)$.

Context Π_{old} contains all the frozen type bindings of stack locations that were popped off. Each judgment and its rules are explained in the following sections.

Overall Judgment A model satisfies a formula if it can be split into a frozen part and a linear part such that the frozen part satisfies the frozen judgment, the linear part satisfies the linear judgment, and the two judgments are connected by the same contexts.

$(m, t) \vDash^{\Psi} F \,at\, p$ overall
iff exists m_1, m_2, t_1, t_2, Π, Π_{old}
such that $m = m_1 \uplus m_2$, $t = t_1 \uplus t_2$, $\Pi_{old}; \Pi; m_1; t_1 \vDash^{\Psi} F \,at\, p$,
$\Pi_{old}; t_2 \vDash^{\Psi} m_2 : \Pi$, and $\forall k.\ell \in dom(\Pi_{old} \cup \Pi) : k \in \mathsf{Live}\,(t)$ iff $k.\ell \in dom(\Pi)$

Context Π_{old} contains the typing information of all the frozen locations that were popped off the stack and Π contains typing information about all the live frozen locations. It is necessary that for any tagged location $k.\ell$ that belongs to the domain of Π, the version number k is current.

Frozen Judgment The second judgment checks that a frozen memory has a given frozen-memory typing.

$(\Pi_{old}; t) \vDash^{\Psi} m : \Pi$ iff for all $k.\ell \in dom(\Pi)$ $\Pi_{old}; \Pi; \ell \rightarrow m(\ell); t \vDash^{\Psi} \Pi(k.\ell) \,at\, k.\ell$

The two frozen-memory typing contexts are recursively used to check that a frozen memory has a given frozen-memory typing. For each location $k.\ell$ in the domain of Π, the piece of memory it points to should linearly satisfy the formula given by the frozen-memory typing.

- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \tau\, at\, p$ iff $p \neq *$ and $t.fst = \textit{absent}$, and $dom(m) = \overline{p}$ and $m(\overline{p}) = sv, \vDash^{\Psi} sv : \tau$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} (g \Rightarrow G)\ at\, p$, iff $dom(m) = \overline{g}$, if $g = k.\ell$ then $k \in \mathsf{Live}\,(t)$, and $\Pi_{old}; \Pi; m; t \vDash^{\Psi} G\, at\, g$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} k_2{=}k_1{+}i\omega\ at\, p$ iff $dom(m) = \emptyset$, $t.fst = \textit{absent}$, and one of the following holds: $i\omega = n \geq 0$ and k_2 is the nth ancestor of k_1 in tree t; $i\omega = -n < 0$ and k_1 is the nth ancestor of k_2 in tree t; $i\omega = +\infty$ and $k_2 = H$; or $i\omega = -\infty$ and $k_1 = H$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \mathsf{first}(k)\ at\, p$ iff $dom(m) = \emptyset$, and $t = (H, T, k, \textit{present})$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \mathsf{frzn}\ k.\ell\ G\, at\, p$ iff $dom(m) = \emptyset$, $t.fst = \textit{absent}$, $(\Pi_{old} \cup \Pi)(k.\ell) = G'$, and $G \equiv G'$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} ei\omega \geq 0\, at\, p$ iff $ei\omega \geq 0$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \mathsf{more}^{\leftarrow}(\ell)\, at\, p$ iff $t.fst = \textit{absent}$, and $dom(m) = \{\ell - n | n \in \mathsf{Nat}\}$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \mathsf{more}^{\rightarrow}(\ell)\, at\, p$ iff $t.fst = \textit{absent}$, and $dom(m) = \{\ell + n | n \in \mathsf{Nat}\}$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \mathbf{1}\, at\, p$ iff $t.fst = \textit{absent}$, and $dom(m) = \emptyset$
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} F_1 \otimes F_2\, at\, p$ iff $m = m_1 \uplus m_2$, $t = t_1 \uplus t_2$, $\Pi_{old}; \Pi; m_1; t_1 \vDash^{\Psi} F_1\, at\, p$, and $\Pi_{old}; \Pi; m_2; t_2 \vDash^{\Psi} F_2\, at\, p$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} F_1 \multimap F_2\, at\, p$ iff $t.fst = \textit{absent}$, for all memory m', tree t' and $t'.fst = \textit{absent}$, $\Pi_{old}; \Pi; m'; t' \vDash^{\Psi} F_1\, at\, p$ implies $\Pi_{old}; \Pi; m \uplus m'; t \uplus t' \vDash^{\Psi} F_2\, at\, p$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \top\, at\, p$, it is true for all memory models.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} F_1 \,\&\, F_2\, at\, p$ iff $\Pi_{old}; \Pi; m; t \vDash^{\Psi} F_1\, at\, p$, and $\Pi_{old}; \Pi; m; t \vDash^{\Psi} F_2\, at\, p$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \mathbf{0}\, at\, p$, no memory satisfies $\mathbf{0}$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} F_1 \oplus F_2\, at\, p$ iff $\Pi_{old}; \Pi; m; t \vDash^{\Psi} F_1\, at\, p$, or $\Pi_{old}; \Pi; m; t \vDash^{\Psi} F_2\, at\, p$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} !\, F\, at\, p$ iff $t.fst = \textit{absent}$, and $dom(m) = \emptyset$, and $\Pi_{old}; \Pi; m; t \vDash^{\Psi} F\, at\, p$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \exists x{:}K.F'\, at\, p$ iff there exists some $a \in K$ such that $\Pi_{old}; \Pi; m; t \vDash^{\Psi} F[a/x]\, at\, p$.

Figure 3: The Semantics of Formulas

Linear Judgment The main semantic judgment is the linear satisfaction judgment. It is defined in Figure 3. The set of pure formulas G is a subset of the set of formulas F, so the semantic judgments of G are also defined in Figure 3, if we substitute G for F.

The interesting clauses are the ones for our domain-specific predicates. Predicate τ holds at p if the memory contains only one location $\overline{p}$ and its content has type τ. Predicate $(g \Rightarrow G)$ holds if memory contains only location $\overline{g}$, g is live, and G holds at g. Predicate $k_2{=}k_1{+}i\omega$ holds if the two tags are related by $i\omega$ number of levels, positive means that k_2 is the ancestor, negative means that k_1 is the ancestor, and ∞ means the ancestor is H. Predicate $\mathsf{first}(k)$ holds if *fst* is *present* and the stack top is k in the tag tree; note that the other base predicates require *fst* to be *absent*. Predicate $(\mathsf{frzn}\ k.\ell\ G)$ holds if one of the two frozen-memory contexts maps $k.\ell$ to a formula equivalent to G. Predicates $\mathsf{more}^{\leftarrow}(\ell)$ and $\mathsf{more}^{\rightarrow}(\ell)$ describe the continuous unallocated space; they require the linear memory to contain the locations at and below (respectively above) ℓ. Note that the base predicates, which describe the properties of the tag tree, require the linear memory to be empty. The other connectives are those of linear logic, and the semantics are standard except that tensor uses the merge operators defined in the previous section on the linear parts of the model.

Semantics for Types Types have the expected semantics:

$$\frac{}{\vDash^{\Psi} i : \mathbf{S}(i)}\ \textit{int} \qquad \frac{}{\vDash^{\Psi} k.\ell : \mathbf{S}(k.\ell)}\ \textit{loc}$$

$$\frac{\Psi(c) = (F') \to 0 \quad F \vDash^{\Psi} F'}{\vDash^{\Psi} c : (F) \to 0}\ \textit{code}$$

Semantic Entailment $F_1 \vDash^{\Psi} F_2$ iff for all m and t, $(m, t) \vDash^{\Psi} F_1\, at\, p$ $\mathsf{overall}$ implies $(m, t) \vDash^{\Psi} F_2\, at\, p$ $\mathsf{overall}$.

Equivalence Equivalence of formulas, $F_1 \equiv F_2$, is reflexive, symmetric, transitive, congruent, and includes some rules for our domain-specific predicates. The most important domain specific rule involves the relationship between the heap tag H and the stack tags. Intuitively, if we know that xk outlives H, then xk must be H itself. Therefore if tag xk is a free variable in F_1 and F_2, and $F_1[H/xk] \equiv F_2[H/xk]$ then $F_1[H/xk]$ is equivalent to $\exists xk{:}\mathsf{TG}.F_2 \otimes !(xk \text{ outlives } H)$.

$$\frac{}{F \equiv F} \qquad \frac{F_2 \equiv F_1}{F_1 \equiv F_2} \qquad \frac{F_1 \equiv F_2 \quad F_2 \equiv F_3}{F_1 \equiv F_3}$$

$$\frac{F_1 \equiv F_1' \quad F_2 \equiv F_2'}{F_1 \,\&\, F_2 \equiv F_1' \,\&\, F_2'} \qquad \frac{F_1 \equiv F_1' \quad F_2 \equiv F_2'}{F_1 \otimes F_2 \equiv F_1' \otimes F_2'}$$

$$\frac{F_1 \equiv F_1' \quad F_2 \equiv F_2'}{F_1 \oplus F_2 \equiv F_1' \oplus F_2'} \qquad \frac{F_1 \equiv F_2}{\exists x{:}K.F_1 \equiv \exists x{:}K.F_2}$$

$$\frac{F_1 \equiv F_2}{(ek.e\ell \Rightarrow F_1) \;\equiv\; (ek.e\ell \Rightarrow F_2)}$$

$$\frac{F_1 \equiv F_2}{\mathsf{frzn}\ ek.e\ell\ F_1 \equiv \mathsf{frzn}\ ek.e\ell\ F_2}$$

$$\frac{F_1[H/xk] \equiv F_2[H/xk]}{F_1[H/xk] \equiv \exists xk{:}\mathsf{TG}.F_2 \otimes !(xk \text{ outlives } H)}$$

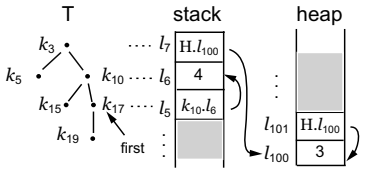

Figure 4: Example Memory

Example For this example, we consider just memory and not the registers. The memory m in Figure 4 consists of a linear part m_1 which contains all the unallocated locations; a frozen part m_2 which contains the five locations shown in the figure ($dom(m_2) = \{\ell_7, \ell_6, \ell_5, \ell_{100}, \ell_{101}\}$). It satisfies the following formula:

$$\begin{aligned} F \;=\;& \mathsf{more}^{\leftarrow}(\ell_4) \otimes \mathsf{more}^{\rightarrow}(\ell_{102}) \otimes \mathsf{first}(k_{17}) \\ & \otimes !(k_3{=}k_5{+}1) \otimes !(k_{10}{=}k_{15}{+}1) \otimes !(k_3{=}k_{10}{+}1) \\ & \otimes !(k_{10}{=}k_{17}{+}1) \otimes !(k_{17}{=}k_{19}{+}1) \\ & \otimes !\,(\mathsf{frzn}\ k_3.\ell_7\ F_4) \otimes !\,(\mathsf{frzn}\ k_{10}.\ell_6\ \mathbf{int}) \otimes !\,(\mathsf{frzn}\ k_{17}.\ell_5\ F_5) \\ & \otimes !\,(\mathsf{frzn}\ H.\ell_{100}\ \mathbf{int}) \otimes !\,(\mathsf{frzn}\ H.\ell_{101}\ F_4) \end{aligned}$$

where

$$\begin{aligned} F_4 \;=\;& \exists x\ell{:}\mathsf{L}.\mathbf{S}(H.x\ell) \otimes !\,(\mathsf{frzn}\ H.x\ell\ \mathbf{int}) \\ F_5 \;=\;& \exists xk{:}\mathsf{TG}.\exists x\ell{:}\mathsf{L}.\mathbf{S}(xk.x\ell) \otimes !\,(\mathsf{frzn}\ xk.x\ell\ \mathbf{int}) \\ & \otimes !\,(\,xk \text{ outlives } k_{17}) \end{aligned}$$

The overall semantic judgment is: $m, (H, T, k_{17}, \textit{present}) \vDash^{\Psi} F\ at\ *$ overall. To check this judgment is valid, we must show:

1. $\Pi_{old}; \Pi; m_1; (H, T, k_{17}, \textit{present}) \vDash^{\Psi} F\ at *$
2. $\Pi_{old}; (H, T, k_{17}, \textit{absent}) \vDash^{\Psi} m_2 : \Pi$

where

$$\begin{array}{lcl} \Pi_{old} & = & (k_5.\ell_6{:}F_1), (k_{15}.\ell_5{:}F_2), (k_{19}.\ell_4{:}F_3) \\ \Pi & = & (k_3.\ell_7{:}F_4), (k_{10}.\ell_6{:}\mathbf{int}), (k_{17}.\ell_5{:}F_5), \\ & & (H.\ell_{100}{:}\mathbf{int}), (H.\ell_{101}{:}F_4) \end{array}$$

The first judgment is the linear semantic judgment. The tensor operators distribute the unallocated stack space to $\mathsf{more}^{\leftarrow}(\ell_4)$, the unallocated heap space to $\mathsf{more}^{\rightarrow}(\ell_{102})$, the stack top to $\mathsf{first}(k_{17})$, and empty linear state to the remaining formulas. Each of the predicates is easily verified.

The second judgment type checks the frozen memory. We need to verify that for each location ℓ in Π, $\Pi(\ell)$ describes the piece of memory that ℓ points to. For example, for location $k_3.\ell_7$ we need to verify the following judgment: $\Pi_{old}; \Pi; \ell_7 \mapsto H.\ell_{100}; (H, T, k_{17}, \mathit{absent}) \vDash^{\Psi} F_4\, at\, k_3.\ell_7$, which is true with ℓ_{100} as the witness of the existential.

3.3 Semantics of Contexts and Soundness

Before we define the semantics of logical contexts, we introduce the definition of $!\Gamma$. It converts the unrestricted context Γ into a formula by wrapping a $!$ around each formula in Γ and tensor them together. We define $\bigotimes(\Delta)$ as the result of tensoring all the formulas in Δ together.

$$\begin{array}{rr} !(\cdot) = \mathbf{1} & !(F, \Gamma) = !\,F \otimes !\Gamma. \\ \bigotimes(\cdot) = \mathbf{1} & \bigotimes(F, \Delta) = F \otimes \bigotimes \Delta. \end{array}$$

We define the semantics of contexts as follows:

$$(m, t) \vDash^{\Psi} (\Gamma; \Delta)\, at\, p \text{ iff } (m, t) \vDash^{\Psi} (!(\Gamma) \otimes (\bigotimes \Delta))\, at\, p$$

We proved that the logical deduction is sound with regard to our semantics.

Lemma 1 (Soundness of logical deduction)

1. *If* $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \sigma(\Gamma; \Delta)\, at\, p$, *where* σ *is a substitution for* Θ, *and* $\Theta \parallel \Gamma; \Delta \vdash F$, *then* $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \sigma F\, at\, p$.
2. *If* $m; t \vDash^{\Psi} \sigma(\Gamma; \Delta)\, at\, p$ *and* $\Theta \parallel \Gamma; \Delta \vdash F$ *then* $m; t \vDash^{\Psi} \sigma(F)\, at\, p$.

Proof 1
By induction on the structure of the logical derivation $\Theta \parallel \Gamma; \Delta \vdash F$.

4 Assembly Language

In this section, we use the memory logic that we formalized in Section 3 as the basis for the type system of a simple assembly language. The type system is powerful enough to specify sound strong and invariant updates of both stack and heap locations. The expressiveness of our typed assembly language exceeds the previous TAL systems because they cannot deal with general stack allocation. In Section 5, we demonstrate a type-preserving translation from a source language that has flexible data allocation on both the stack and the heap down to our assembly language.

4.1 Syntax

The language is very similar to previous TALs. The novel parts are the inclusion of tag trees in machine states and the two instructions `stackgrow` and `stackcut`. These two instructions only update the tag tree and are erased in forming an underlying untyped machine code. They tell the type system to increase or decrease the stack by one location. Any stack pointer register that tracks the top of the stack must be adjusted by a separate instruction. The syntax of our assembly language is:

Operands	v	::=	$sv \mid r$
Instructions	ι	::=	$\texttt{add}\, r_d, r_s, v \mid \texttt{sub}\, r_d, r_s, v \mid \texttt{mov}\, r_d, v$
			$\mid \texttt{bz}\, r, v \mid \texttt{ld}\, r_d, r_s \mid \texttt{st}\, r_d, r_s$
			$\mid \texttt{stackgrow} \mid \texttt{stackcut}$
Blocks	B	::=	$\texttt{halt} \mid \texttt{jmp}\, v \mid \iota; B$
Code Regions	C	::=	$\cdot \mid C, c \mapsto B$
Machine States	Σ	::=	(C, m, t, B)

4.2 Operational Semantics

The operational semantics is given in Figure 5. We use $\Sigma \longmapsto \Sigma'$ to denote the small step relation between two machine states Σ and Σ'. We use $\widehat{m}(v)$ to convert an operand to a store value. In particular, $\widehat{m}(r)$ is the value contained in the register r.

$$\widehat{m}(i) = i \qquad \widehat{m}(c) = c \qquad \widehat{m}(e\ell) = e\ell \qquad \widehat{m}(r) = m(r)$$

$(C, m, t, B) \longmapsto \Sigma$ where	
If $B =$	then $\Sigma =$
$\texttt{add}\, r_d, r_s, v; B'$	$(C, m\,[\, r_d := (m(r_s) + \widehat{m}(v))\,], t, B')$
$\texttt{sub}\, r_d, r_s, v; B'$	$(C, m\,[\, r_d := (m(r_s) - \widehat{m}(v))\,], t, B')$
$\texttt{bz}\, r, v; B'$	(C, m, t, B') and $m(r) \neq 0$
$\texttt{bz}\, r, v; B'$	(C, m, t, B'') $\;m(r) = 0$ where $C(\widehat{m}(v)) = B''$
$\texttt{mov}\, r_d, v; B'$	$(C, m\,[\, r_d := \widehat{m}(v)\,], t, B')$
$\texttt{ld}\, r_d, r_s; B'$	$(C, m\,[\, r_d := m(m(r_s))\,], t, B')$
$\texttt{st}\, r_d, r_s; B'$	$(C, m\,[\, m(r_d) := m(r_s)\,], B')$
$\texttt{jmp}\, v$	(C, m, t, B'') where $C(\widehat{m}(v)) = B''$
$\texttt{stackgrow}; B'$	(C, m, t', B'), k is fresh where $t' = \mathsf{newFirst}(t, k)$
$\texttt{stackcut}; B'$	(C, m, t', B') where $t' = \mathsf{delFirst}(t)$

Figure 5: Operational Semantics

We overload the `add` and `sub` instructions to perform both arithmetic operations on integers and tagged locations. We formally define the arithmetic of tagged locations as follows:

$$\begin{array}{rrcll}
0 \leq i < \infty & sk.e\ell + i & = & sk'.(e\ell + i) & \text{where } sk' \text{ is an ancestor of } sk \text{ in } t \\
 & & & & \text{and the distance between } sk \text{ and } sk' \text{ is } i \\
 & H.e\ell + i & = & H.(e\ell + i) & \\
 & sk.e\ell - i & = & sk'.(e\ell - i) & \text{where } sk' \text{ is the rightmost child of } sk \text{ in } t \\
 & & & & \text{and the distance between } sk \text{ and } sk' \text{ is } i \\
 & H.e\ell - i & = & H.(e\ell - i) &
\end{array}$$

Due to the tree structure of the tags the choices for the result of $ek - i$ is not unique, however only the live one is of interest to us.

Most operational semantics rules are straight forward. Only the last two rules for instruction `stackgrow` and `stackcut` worth some explanation. The instruction `stackgrow` always precedes an addr-sub operation on r_{sp} to

complete a stack allocation operation. Operationally, it generates a fresh new stack tag sk, and adds sk to the tag tree t. Similar to `stackgrow`, instruction `stackcut` succeeds a addr-add operation on r_{sp} to complete a stack deallocation operation. It moves the first marker in tag tree t up to its parent thus disposes the tag associated with the location that is just deallocated. Both of these instructions do not have any effect on the memory m, so our assembly language could be easily mapped to another one that doesn't have these two instructions.

4.3 Typing Rules

The type system consists of the following judgments:

$\Theta \parallel \Gamma; \Delta \vdash^{\Psi} F$	Logical rules
$\Theta \parallel F \vdash^{\Psi} v : \tau$	Operand v has type τ
$\Theta \parallel F \vdash^{\Psi} \iota : F'$	The precondition of instruction ι is F, and the postcondition is F'
$\Theta \parallel F \vdash^{\Psi} B$ ok	Block B is type checked with the context $\Theta \parallel F$
$\vdash C : \Psi$	Coderegion C has type Ψ
$\vdash^{\Psi} \Sigma$ ok	The abstract machine state Σ is well-formed

We use Θ to contain the free variables, Γ as the unrestricted context, and Δ as the linear context. For simplicity, we omit the code context Ψ from the typing judgments.

Operations on Formulas

The notation $(F\,[\,g := G\,])_{\Theta}$ to denotes the result of "updating" formula F at location g by G. More precisely, $(F\,[\,g := G\,])_{\Theta} = F_1 \otimes (g \Rightarrow G)$ iff $\Theta \parallel \cdot ; F \vdash F_1 \otimes (g \Rightarrow G')$ where all the free variables are in Θ.

The following lemmas state that formula $update$ operations are sound with respect to the semantics.

Lemma 2 (Formula Copy)

1. *If* $\Pi_{old}; \Pi; m; t \vDash^{\Psi} G\,at\,g$ *then* $dom(m) = \overline{g}$ *or* $dom(m) = \emptyset$
2. *If* $\Pi_{old}; \Pi; \emptyset; t \vDash^{\Psi} G\,at\,g$ *then* $\Pi_{old}; \Pi; \emptyset; t \vDash^{\Psi} G\,at\,g'$
3. *If* $\Pi_{old}; \Pi; \overline{g} \mapsto sv; t \vDash^{\Psi} G\,at\,g$ *then* $\Pi_{old}; \Pi; \overline{g'} \mapsto sv; t \vDash^{\Psi} G\,at\,g'$

Proof 2

By induction on the structure of G.

Lemma 3 (Formula Update)

If σ *is a substitution for* Θ *and* $\Pi_{old}; \Pi; \sigma(\overline{g} \mapsto sv); t \vDash^{\Psi} \sigma\,(g \Rightarrow G)\ at\,p$ *and* $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \sigma(F)\,at\,p$ *then* $\Pi_{old}; \Pi; m[\sigma(\overline{g} := sv)]; t \vDash^{\Psi} \sigma(F\,[\,g := G\,]_{\Theta})\,at\,p$

Proof 3

By the Soundness of Logical deduction lemma and Lemma 2.

Operand Typing: $\Theta \parallel F \vdash^{\Psi} v : \tau$

The typing rules for operands extend the typing rules for store values to include typing rules for registers. The type of register r is obtained by looking up the its type in the formula F that currently describes the memory.

$$\frac{}{\Theta \parallel F \vdash^{\Psi} i : \mathbf{S}(i)}\ \textit{int} \qquad \frac{}{\Theta \parallel F \vdash^{\Psi} k.\ell : \mathbf{S}(k.\ell)}\ \textit{loc}$$

$$\frac{\Psi(c) = (F') \to 0 \quad \cdot \parallel \cdot ; u : F \vdash F'}{\Theta \parallel F \vdash^{\Psi} c : (F) \to 0}\ \textit{code} \qquad \frac{\Theta \parallel \cdot ; F \vdash\ (r \Rightarrow \tau) \otimes \top}{\Theta \parallel F \vdash^{\Psi} r : \tau}\ \textit{reg}$$

Figure 6: Operand Typing

$$\frac{\Theta \parallel F \vdash r_s : \mathbf{int} \qquad \Theta \parallel F \vdash v : \mathbf{int}}{\Theta \parallel F \vdash \mathtt{add}\, r_d, r_s, v : F\,[\,r_d := \mathbf{int}\,]}\ \textsf{(add)} \qquad \frac{\Theta \parallel F \vdash r_s : \mathbf{int} \qquad \Theta \parallel F \vdash v : \mathbf{int}}{\Theta \parallel F \vdash \mathtt{sub}\, r_d, r_s, v : F\,[\,r_d := \mathbf{int}\,]}\ \textsf{(sub)}$$

$$\frac{\Theta \parallel F \vdash v : \tau}{\Theta \parallel F \vdash \mathtt{mov}\, r_d, v : F\,[\,r_d := \tau\,]}\ \textsf{(mov)} \qquad \frac{\Theta \parallel F \vdash r : \mathbf{int} \qquad \Theta \parallel F \vdash v : (F') \to 0 \qquad \Theta \parallel u : F \vdash F'}{\Theta \parallel F \vdash \mathtt{bz}\, r, v : F}\ \textsf{(bz)}$$

$$\frac{\Theta \parallel F \vdash r_s : \mathbf{S}(ek.e\ell) \quad \Theta \parallel \cdot ; F \vdash\ (ek.e\ell \Rightarrow G) \otimes \top}{\Theta \parallel F \vdash \mathtt{ld}\, r_d, r_s : F\,[\,r_d := G\,]}\ \textsf{(ld)} \qquad \frac{\Theta \parallel F \vdash r_d : \mathbf{S}(ek.e\ell) \quad \Theta \parallel \cdot ; F \vdash\ (r_s \Rightarrow G) \otimes \top}{\Theta \parallel F \vdash \mathtt{st}\, r_d, r_s : F\,[\,ek.e\ell := G\,]}\ \textsf{(st)}$$

$$\frac{\Theta \parallel F \vdash r_s : \mathbf{S}(ek.e\ell) \quad \Theta \parallel \cdot ; F \vdash \mathsf{frzn}\ ek.e\ell\ G \otimes \top \quad \Theta \parallel \cdot ; F \vdash \mathsf{live}\,(ek) \otimes \top}{\Theta \parallel F \vdash \mathtt{ld}\, r_d, r_s : F\,[\,r_d := G\,]}\ \textsf{(ld-inv)}$$

$$\frac{\Theta \parallel F \vdash r_d : \mathbf{S}(ek.e\ell) \quad \Theta \parallel \cdot ; F \vdash\ (r_s \Rightarrow G) \otimes \top \quad \Theta \parallel \cdot ; F \vdash \mathsf{frzn}\ ek.e\ell\ G \otimes \top \quad \Theta \parallel \cdot ; F \vdash \mathsf{live}\,(ek) \otimes \top}{\Theta \parallel F \vdash \mathtt{st}\, r_d, r_s : F}\ \textsf{(st-inv)}$$

$$\frac{\Theta \parallel F \vdash r_s : \mathbf{S}(ek.e\ell) \qquad \Theta \parallel F \vdash v : \mathbf{S}(i) \qquad \Theta \parallel \cdot ; F \vdash\ !(ek'{=}ek{+}i) \otimes \top}{\Theta \parallel F \vdash \mathtt{add}\, r_d, r_s, v : F\,[\,r_d := \mathbf{S}(ek'.(e\ell + i))\,]}\ \textsf{(addr-add)}$$

$$\frac{\Theta \parallel F \vdash r_s : \mathbf{S}(ek.e\ell) \qquad \Theta \parallel F \vdash v : \mathbf{S}(i) \qquad \Theta \parallel F \vdash\ !(ek{=}ek'{+}i) \otimes \top}{\Theta \parallel F \vdash \mathtt{add}\, r_d, r_s, v : F\,[\,r_d := \mathbf{S}(ek'.(e\ell - i))\,]}\ \textsf{(addr-sub)}$$

$$\frac{\Theta \parallel \cdot ; F \vdash F' \otimes \mathsf{more}^{\leftarrow}(e\ell) \otimes \mathsf{first}(ek)}{\Theta \parallel F \vdash \mathtt{stackgrow} : F' \otimes \mathsf{more}^{\leftarrow}(e\ell - 1) \otimes (\exists xk{:}\mathsf{TG}.\, \mathsf{first}(xk) \otimes\, !(ek{=}xk{+}1) \otimes\ (xk.e\ell \Rightarrow \mathbf{ns})\,)}\ \textsf{(stackgrow)}$$

$$\frac{\Theta \parallel \cdot ; F \vdash \mathsf{more}^{\leftarrow}(e\ell - 1) \otimes\ \mathsf{first}(ek) \otimes\, !(ek'{=}ek{+}1) \otimes (\,(ek.e\ell \Rightarrow G) \oplus \mathsf{frzn}\ ek.e\ell\ G) \otimes F'}{\Theta \parallel F \vdash \mathtt{stackcut} : \mathsf{more}^{\leftarrow}(e\ell) \otimes\ \mathsf{first}(ek') \otimes\, !(ek'{=}ek{+}1) \otimes F'}\ \textsf{(stackcut)}$$

Figure 7: Instruction Typing

Instruction Typing: $\Theta \parallel F \vdash^{\Psi} \iota : F'$

The typing judgments for instructions resemble Hoare logic. The formula on the left hand side of the turnstile describes the precondition of the instruction, and the formula on the right hand side describes the memory state after the execution of the instruction.

The typing judgments for instruction `add`, `sub`, and `mov` are straight forward. The typing rule for branch instruction `bz` requires that register r is of type int, so that it can be compared with 0; and the second operand is a code value where the instruction might branch to; and the current state satisfies the precondition of the code that the instruction might branch to so that the branch is safe.

There are two sets of typing rules of load and store instructions. The first set of rules requires linear capabilities for the locations that we load from and store into. The rule for `ld` checks that the source register holds a location, looks up the describing formula for this location, and updates the destination register to be described by this formula. The `st` instruction checks that the destination register holds a location, gets the describing formula of the source register, and updates the location's describing formula. Note that the update formulas use linear typing of the register or location

and just replace the old type with the new one—a strong update rule.

The second set of typing rules requires unrestricted capabilities for the locations we operate on. The rule for ld-inv checks that the tag ek is live to make sure that the unrestricted capability is valid. The rule for st-inv checks that the destination register holds a location $ek.e\ell$, and the describing formula of the source register is the same as the describing formula of location $ek.e\ell$, given by the unrestricted capability (frzn $ek.e\ell\ G$), and the tag associated with the unrestricted capability is live. The postcondition is the same as the precondition, since we update the location with a value that has the same describing formula —an invariant update.

We give separate typing rules for address add/sub operations because they involve determining the correct tags for the addresses.

The two stack instructions `stackgrow` and `stackcut` change the shape of the formula to reflect the changes to the tag tree due to stack allocation and deallocation respectively. The `stackgrow` instruction takes one location out of $\mathsf{more}^{\leftarrow}(e\ell)$ and results in $\mathsf{more}^{\leftarrow}(e\ell-1)\otimes(xk.e\ell \Rightarrow \mathbf{ns})$. A fresh tag xk is chosen (the existential achieves the freshness), and is declared the new stack top $\mathsf{first}(xk)$ and a direct child of the old stack top $!(ek{=}xk{+}1)$. The `stackcut` instruction determines the stack top's tag $\mathsf{first}(ek)$ and parent's tag $!(ek'{=}ek{+}1)$, and returns the stack top to the unallocated stack taking $\mathsf{more}^{\leftarrow}(e\ell-1)\otimes((ek.e\ell \Rightarrow G)\oplus!(\mathsf{frzn}\ ek.e\ell\ G))$ to $\mathsf{more}^{\leftarrow}(e\ell)$. It also makes the parent tag the new stack top $\mathsf{first}(ek')$.

Block Typing: $\Theta \parallel F \vdash^{\Psi} B\ \mathsf{ok}$

The block typing rules check the well-formedness of a code block with the context $\Theta \parallel F$.

$$\frac{\Theta \parallel F \vdash r_{sp}{:}\mathbf{S}(ek.e\ell) \qquad \Theta \parallel \cdot; F \vdash (ek.e\ell \Rightarrow \mathbf{int}) \otimes \top}{\Theta \parallel F \vdash \mathtt{halt}\ \mathsf{ok}}\ \text{(b-halt)}$$

$$\frac{\Theta \parallel F \vdash v{:}(F') \to 0 \quad \Theta \parallel \cdot; u{:}F \vdash F'}{\Theta \parallel F \vdash \mathtt{jmp}\ v\ \mathsf{ok}}\ \text{(b-jmp)} \qquad \frac{\Theta \parallel F \vdash \iota{:}F' \quad \Theta \parallel F' \vdash B\ \mathsf{ok}}{\Theta \parallel F \vdash \iota; B\ \mathsf{ok}}\ \text{(b-instr)}$$

$$\frac{\Theta \parallel \mathsf{more}^{\rightarrow}(e\ell+1) \otimes (H.e\ell \Rightarrow \mathbf{ns}) \otimes F \vdash B\ \mathsf{ok}}{\Theta \parallel \mathsf{more}^{\rightarrow}(e\ell) \otimes F \vdash B\ \mathsf{ok}}\ \text{(b-heapgrow)} \qquad \frac{\Theta \parallel !(\mathsf{frzn}\ ek.e\ell\ G) \otimes F \vdash B\ \mathsf{ok}}{\Theta \parallel (ek.e\ell \Rightarrow G) \otimes F \vdash B\ \mathsf{ok}}\ \text{(b-freeze)}$$

$$\frac{\Theta, b \parallel F' \vdash B\ \mathsf{ok}}{\Theta \parallel \exists b.F' \vdash B\ \mathsf{ok}}\ \text{(b-unpack1)} \qquad \frac{\Theta, b \parallel (g \Rightarrow G) \otimes F \vdash B\ \mathsf{ok}}{\Theta \parallel (g \Rightarrow \exists b.G) \otimes F \vdash B\ \mathsf{ok}}\ \text{(b-unpack2)}$$

$$\frac{\Theta, b \parallel (g \Rightarrow \exists b.G) \otimes F \vdash B\ \mathsf{ok}}{\Theta, b \parallel (g \Rightarrow G) \otimes F \vdash B\ \mathsf{ok}}\ \text{(b-pack)} \qquad \frac{\Theta \parallel (g \Rightarrow G_1) \otimes !\,G_2 \otimes F \vdash B\ \mathsf{ok}}{\Theta \parallel (g \Rightarrow G_1 \otimes !\,G_2) \otimes F \vdash B\ \mathsf{ok}}\ \text{(b-bang1)}$$

$$\frac{\Theta \parallel (g \Rightarrow G_1 \otimes !\,G_2)) \otimes F \vdash B\ \mathsf{ok}}{\Theta \parallel (g \Rightarrow G_1) \otimes !\,G_2 \otimes F \vdash B\ \mathsf{ok}}\ \text{(b-bang2)} \qquad \frac{\Theta \parallel \cdot; u{:}F \vdash F' \quad \Theta \parallel F' \vdash B\ \mathsf{ok}}{\Theta \parallel F \vdash B\ \mathsf{ok}}\ \text{(b-weaken)}$$

Figure 8: Block Typing

There are standard typing rules for instructions, `halt`, and `jmp`, as well as type manipulation rules including growing the heap (for heap allocation) and freezing locations.

The rule b-freeze deals with the process of converting a linear capability, to an unrestricted capability.

Lemma 4 (Soundness of the Block Typing Rules)

1. *If there exists a substitution σ for Θ,*
   and $(m,t) \vDash^{\Psi} \sigma(\mathsf{more}^{\rightarrow}(e\ell) \otimes F)\ at\ *$ *overall*
   then $(m,t) \vDash^{\Psi} \sigma(\mathsf{more}^{\rightarrow}(e\ell+1) \otimes (H.e\ell \Rightarrow \mathbf{ns}) \otimes F)\ at\ *$ *overall.*

2. *If there exists a substitution σ for Θ,*
   and $(m,t) \vDash^{\Psi} \sigma(F \otimes (ek.e\ell \Rightarrow F_1))\ at\ *$ *overall,*
   then $(m,t) \vDash^{\Psi} \sigma(F \otimes !\,(\mathsf{frzn}\ ek.e\ell\ F_1))\ at\ *$ *overall.*

3. *If there exists a substitution* σ *for* Θ*, and* $(m,t) \vDash^{\Psi} \sigma(\exists x{:}K.F')\, at\ *$ *overall*
   then there exists a substitution $a \in K$ *for* x *such that* $(m,t) \vDash^{\Psi} (\sigma, a/x)(F')\, at\ *$ *overall.*

4. *If there exists a substitution* σ *for* Θ*, and* $(m,t) \vDash^{\Psi} \sigma(\,(g \Rightarrow \exists b.G) \otimes F)\, at\ *$ *overall*
   then there exists a substitution $a \in K$ *for* x
   such that $(m,t) \vDash^{\Psi} (\sigma, a/x)(\,(g \Rightarrow G) \otimes F)\, at\ *$ *overall.*

5. *If there exists a substitution* σ *for* Θ*,*
   and $(m,t) \vDash^{\Psi} \sigma(\,(g \Rightarrow G) \otimes F)\, at\ *$ *overall,*
   then$(m,t) \vDash^{\Psi} \sigma(\,(g \Rightarrow \exists b.G \otimes F)\,)\, at\ *$ *overall.*

6. *If there exists a substitution* σ *for* Θ*,*
   and $(m,t) \vDash^{\Psi} \sigma(\,(g \Rightarrow G_1 \otimes !\, G_2) \otimes F)\, at\ *$ *overall,*
   then$(m,t) \vDash^{\Psi} \sigma(\,(g \Rightarrow G_1) \otimes !\, G_2 \otimes F)\, at\ *$ *overall.*

7. *If there exists a substitution* σ *for* Θ*,*
   and $(m,t) \vDash^{\Psi} \sigma(\,(g \Rightarrow G_1) \otimes !\, G_2 \otimes F)\, at\ *$ *overall,*
   then$(m,t) \vDash^{\Psi} \sigma(\,(g \Rightarrow G_1 \otimes !\, G_2)) \otimes F)\, at\ *$ *overall.*

Proof 4
By inspection of the semantics of formulas.

Code Typing: $\vdash C : \Psi$

The code region C has type Ψ if their domains are equal and for each label c in the domain, the code of c is well formed in the context assigned to c by Ψ.

$$\frac{\begin{array}{c} dom(C) = dom(\Psi) \\ \forall c \in dom(C).\ \ \Psi(c) = (F) \rightarrow 0\ \ \textit{implies}\ \ \cdot\ \|\ F \vdash^{\Psi} C(c)\ \mathsf{ok} \end{array}}{\vdash C : \Psi}\ \text{(codergn)}$$

Figure 9: Code Typing

State Typing: $\vdash \Sigma\ \mathsf{ok}$

A machine state (C, m, t, B) is well formed if there is a Ψ and F such that C has type Ψ, m and t satisfy F, B is well formed under F, and the tag tree is well formed.

$$\frac{\begin{array}{cc} \vdash C : \Psi & (m,t) \vDash^{\Psi} F\ \mathsf{at}\ *\ \mathsf{overall} \\ \mathsf{wf}(t) & \cdot\ \|\ F \vdash^{\Psi} B\ \mathsf{ok} \end{array}}{\vdash (C, m, t, B)\ \mathsf{ok}}\ \text{(state)}$$

Figure 10: State Typing

4.4 Type Safety

Our type system is provably safe. The execution of the program is defined in terms of the small step relation between abstract machine states which is denoted by $\Sigma \longmapsto \Sigma'$. We proved the standard progress and preservation theorems for our assembly language. The theorem states that a well-formed machine state is either a terminal state, or it can make a step into another well-formed state.

Theorem 5 (Type Safety)
If $\vdash^{\Psi} (C, m, t, B)$ ok *then:*

1. *Either* $B = \texttt{halt}$ *and* $m(r_{sp}) = k.\ell$ *and* $m(\ell) = sv$ *and* $\models^{\Psi} sv : \mathbf{int}$ *or exists* Σ *such that* $(C, m, t, B) \longmapsto \Sigma$.
2. *If* $(C, m, t, B) \longmapsto \Sigma$ *then* $\vdash \Sigma$ ok.

Proof 5
By induction the tying derivation, inspection of the small step operational semantic rules, and Lemma 4.

5 Translation

To show how our logic can be used in a certifying compilation framework, in this section we sketch the translation from a simple language with stack and heap allocation to our assembly language.

5.1 Micro-CLI

The key features we want to capture are the stack-allocation operations of CLI [6, 9]. CLI includes the concepts of references and managed pointers. Managed pointers can point to local variables on the stack and also to fields of objects in the heap, but they cannot (in verifiable code) be returned from methods nor stored into fields of objects. References always point to objects in the heap and their use is unrestricted. We abstract CLI into Micro-CLI, a simple imperative language with integers and pointers. The type $\tau *_S$ contains pointers to stack or heap cells of type τ, and has the above restrictions on its use; we also disallow updating a stack pointer if it points to another stack pointer. It has a subtype $\tau *_H$ that contains just pointers into the heap, and this type is unrestricted.

Figure 11 gives the grammar for Micro-CLI. While the grammar is slightly more complicated than necessary to express the structure of the language, it is structured to facilitate the translation.

The declaration $\tau *_S x \ = \ \mathbf{new}_S \, v$ allocates a new cell on the stack with initial value v of type τ and places a pointer to the cell into x; $\tau *_H x \ = \ \mathbf{new}_H \, v$ is similar but allocates on the heap. The statement $x \ = \ v$ updates the variable x with the value v. The statement $x \ = \ ! \, v$ dereferences the pointer v and updates x with the contents, and the statement $v_1 \ := \ v_2$ updates the contents of the pointer v_1 with the value v_2. The other statements are straightforward.

The typing rules for Micro-CLI are shown in Figures 12 and 13. The main rules of interest are the rules which involve restricting pointers: the judgment for return blocks $\Gamma \vdash^{\tau} rb$ ok ensures that stack pointers are not returned from functions, and the judgment for local declarations $\Gamma \vdash ld : \Gamma'$ allows stack locations to point to older stack locations (that are already in scope) and any location to point to a heap location or an integer, but does not allow heap locations to point to stack locations. We use the notation $\Gamma[x : \tau]$ to mean Γ extended so that x maps to τ.

qualifiers	q	::=	$S \mid H$
types	τ	::=	$\mathbf{int} \mid \tau *_q$
values	v	::=	$n \mid x$
program	p	::=	$fds\ \ rb$
function decls	fds	::=	$\cdot \mid fd\ fds$
function decl	fd	::=	$\tau\, f(ads)\, rb$
argument decls	ads	::=	$ad \mid ad,\ ads$
argument decl	ad	::=	$\tau\, x$
return block	rb	::=	$\{lds; ss; \mathbf{return}\, v\}$
local decls	lds	::=	$\cdot \mid ld;\ lds$
local decl	ld	::=	$\tau\, x\, =\, v \mid \tau\, x\, =\, \mathbf{new}_q\, v$
statement list	ss	::=	$\cdot \mid s;\ ss$
statement	s	::=	$\mathbf{if}\, v\, \mathbf{then}\, ss\, \mathbf{else}\, ss \mid x\, =\, v \mid x\, =\, v_1 + v_2 \mid x\, =\, v_1 - v_2$ $\mid x\, =\, f(avs) \mid x\, =\, !\, v \mid v_1\, :=\, v_2$
argument values	avs	::=	$av \mid av\ avs$
argument value	av	::=	v

Figure 11: The syntax of Micro-CLI used in the translation.

$\boxed{\vdash p \text{ ok}}$

$$\frac{\cdot \vdash fds : \Gamma \quad \Gamma \vdash^{int} rb \text{ ok}}{\vdash fds \;\; rb \text{ ok}} \text{ (p-ok)}$$

$\boxed{\Gamma \vdash fds : \Gamma'}$

$$\frac{}{\Gamma \vdash \cdot : \Gamma} \text{ (fds-}\cdot\text{)} \qquad \frac{\Gamma \vdash fd : \Gamma'' \quad \Gamma'' \vdash fds : \Gamma'}{\Gamma \vdash fd\, fds : \Gamma'} \text{ (fds)}$$

$\boxed{\Gamma \vdash fd : \Gamma'}$

$$\frac{\Gamma \vdash ads : (\Gamma', \tau list) \quad \Gamma'[f : \tau list \to \tau] \vdash^{\tau} rb \text{ ok}}{\Gamma \vdash \tau\, f(ads)\, rb : \Gamma[f : \tau list \to \tau]} \text{ (fd)}$$

$\boxed{\Gamma \vdash ads : (\Gamma, \tau list)}$

$$\frac{}{\Gamma \vdash \cdot : (\Gamma, [\,])} \text{ (ads-}\cdot\text{)} \qquad \frac{\Gamma \vdash ad : (\Gamma'', \tau) \qquad \Gamma'' \vdash ads : (\Gamma', \tau list)}{\Gamma \vdash ad\, ads : (\Gamma', \tau :: \tau list)} \text{ (ads)}$$

$\boxed{\Gamma \vdash ad : (\Gamma, \tau)}$

$$\frac{}{\Gamma \vdash \tau\, x : (\Gamma[x : \tau], \tau)} \text{ (ad)}$$

$\boxed{\Gamma \vdash^{\tau} rb \text{ ok}}$

$$\frac{\Gamma \vdash lds : \Gamma' \quad \Gamma' \vdash ss \text{ ok} \quad \Gamma' \vdash v : \tau \quad (\tau \neq \tau' *_S)}{\Gamma \vdash^{\tau} \{lds; ss; \mathbf{return}\, v\} \text{ ok}} \text{ (rb-ok)}$$

$\boxed{\Gamma \vdash lds : \Gamma'}$

$$\frac{}{\Gamma \vdash \cdot : \Gamma} \text{ (lds-}\cdot\text{)} \qquad \frac{\Gamma \vdash ld : \Gamma'' \quad \Gamma'' \vdash lds : \Gamma'}{\Gamma \vdash ld\, lds : \Gamma'} \text{ (lds)}$$

$\boxed{\Gamma \vdash ld : \Gamma'}$

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \tau\, x \;=\; v : \Gamma[x : \tau]} \text{ (ld-val)} \qquad \frac{\Gamma \vdash v : \tau' *_H \quad (\tau = \tau' *_H *_q)}{\Gamma \vdash \tau\, x \;=\; \mathbf{new}_q\, v : \Gamma[x : \tau]} \text{ (ld-q-to-h)}$$

$$\frac{\Gamma \vdash v : \tau' *_S \quad (\tau = \tau' *_S *_S)}{\Gamma \vdash \tau\, x \;=\; \mathbf{new}_S\, v : \Gamma[x : \tau]} \text{ (ld-s-to-s)} \qquad \frac{\Gamma \vdash v : \mathbf{int} \quad (\tau = \mathbf{int} *_q)}{\Gamma \vdash \tau\, x \;=\; \mathbf{new}_q\, v : \Gamma[x : \tau]} \text{ (ld-int-ref)}$$

Figure 12: The typing rules for Micro-CLI (1 of 2).

$\boxed{\Gamma; \tau list \vdash avs \text{ ok}}$

$$\frac{}{\Gamma;\ [\,] \vdash \cdot \text{ ok}}\ \text{(avs-·-ok)} \qquad \frac{\Gamma \vdash av : \tau \quad \Gamma; \tau list \vdash avs \text{ ok}}{\Gamma; \tau :: \tau list \vdash av\ avs \text{ ok}}\ \text{(avs-ok)}$$

$\boxed{\Gamma \vdash av : \tau}$

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash v : \tau}\ \text{(av)}$$

$\boxed{\Gamma \vdash ss \text{ ok}}$

$$\frac{}{\Gamma \vdash \cdot \text{ ok}}\ \text{(ss-·-ok)} \qquad \frac{\Gamma \vdash s \text{ ok} \quad \Gamma \vdash ss \text{ ok}}{\Gamma \vdash ss \text{ ok}}\ \text{(ss-ok)}$$

$\boxed{\Gamma \vdash s \text{ ok}}$

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash v : \tau \quad (\tau \neq \tau' *_S)}{\Gamma \vdash x\ =\ v \text{ ok}}\ \text{(s-val)} \qquad \frac{\Gamma \vdash x : \mathbf{int} \quad \Gamma \vdash v_1 : \mathbf{int} \quad \Gamma \vdash v_2 : \mathbf{int}}{\Gamma \vdash x\ =\ v_1 + v_2 \text{ ok}}\ \text{(s-add)}$$

$$\frac{\Gamma \vdash x : \mathbf{int} \quad \Gamma \vdash v_1 : \mathbf{int} \quad \Gamma \vdash v_2 : \mathbf{int}}{\Gamma \vdash x\ =\ v_1 - v_2 \text{ ok}}\ \text{(s-sub)} \qquad \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash v : \tau *_q \quad (\tau \neq \tau' *_S)}{\Gamma \vdash x\ =\ !\, v \text{ ok}}\ \text{(s-deref)}$$

$$\frac{\Gamma \vdash v_1 : \tau *_q \quad \Gamma \vdash v_2 : \tau \quad (\tau \neq \tau' *_S)}{\Gamma \vdash v_1\ :=\ v_2 \text{ ok}}\ \text{(s-update)} \qquad \frac{\Gamma \vdash v : \mathbf{int} \quad \Gamma \vdash ss_1 \text{ ok} \quad \Gamma \vdash ss_2 \text{ ok}}{\Gamma \vdash \mathbf{if}\, v\, \mathbf{then}\, ss_1\, \mathbf{else}\, ss_2 \text{ ok}}\ \text{(s-if)}$$

$$\frac{\Gamma \vdash f : \tau list \to \tau \qquad \Gamma \vdash x : \tau \qquad \Gamma; \tau list \vdash avs \text{ ok}}{\Gamma \vdash x\ =\ f(avs) \text{ ok}}\ \text{(s-funcall)}$$

$\boxed{\Gamma \vdash v : \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{(v-var)} \qquad \frac{}{\Gamma \vdash n : \mathbf{int}}\ \text{(v-int)}$$

$$\frac{\Gamma \vdash v : \tau' \quad \tau' \leq \tau}{\Gamma \vdash v : \tau}\ \text{(v-subtype)}$$

$\boxed{\tau \leq \tau'}$

$$\frac{}{\tau \leq \tau}\ \text{(sub-reflex)} \qquad \frac{\tau' \leq \tau}{\tau' *_H \leq \tau *_S}\ \text{(sub-h-s)}$$

Figure 13: The typing rules for Micro-CLI (2 of 2).

5.2 Type Translation

The type translation, shown in Figure 14, has the form $[\![\tau]\!]_\tau\ ek$ where τ is the type to translate and ek is a tag expression for the current top of stack. The latter is used in the translation of stack pointers to ensure that the pointer outlives its target.

The translation of pointer types combines a singleton location type with an unrestricted capability to indicate the type of the contents of the location. Heap pointers use H to tag the location, stack pointers use an existentially quantified tag and an additional predicate to state the validity of the tag. Note that the rule we described in the formula equivalence section makes $\mathbf{S}(H.x\ell)\otimes !(\mathsf{frzn}\ H.x\ell\ F)$ equivalent to $\exists xk{:}\mathsf{TG}.\mathbf{S}(xk.x\ell)\otimes !\,(\ xk\ \mathsf{outlives}\ H)\otimes !(\mathsf{frzn}\ xk.x\ell\ F)$ witnessing the subtyping of heap pointers as stack pointers.

The translation of function types is more complicated. Note that polymorphism is expressed as existentials on the left of $\rightarrow\ 0$. All functions are polymorphic in the locations of the top of the stack and allocation frontier, the tags of various stack locations, and the caller's store, which includes its stack, heap, and tag information. The return address is polymorphic in newly allocated heap space and the tag of the stack location used to pass the return value. The precondition has a part for memory including the stack and heap, a part for the registers, a statement of the tag for the top of the stack, an empty thaw set, and the relationships between the various stack tags of relevance. The postcondition is similar but reflects the state at return rather than call. The calling convention is: arguments are pushed onto the stack left to right, the callee pops arguments, the result pushed onto the stack, the stack pointer is in r_{sp}, the return address is in r_{ra}, and frontier pointer is in r_{alloc}.

$$\frac{}{[\![\mathbf{int}]\!]_\tau\ \_\ =\ \mathbf{int}}\ \text{(trans-int)}\qquad \frac{[\![\tau]\!]_\tau\ H\ =\ G}{[\![\tau\ *_H]\!]_\tau\ \_\ =\ \exists x\ell{:}\mathsf{L}.(\mathbf{S}(H.x\ell)\otimes !(\mathsf{frzn}\ H.x\ell\ G))}\ \text{(trans-}\tau\ *_H\text{)}$$

$$\frac{[\![\tau]\!]_\tau\ xk\ =\ G}{[\![\tau\ *_S]\!]_\tau\ ek_c\ =\ \exists xk{:}\mathsf{TG}, x\ell{:}\mathsf{L}.\ (\mathbf{S}(xk.x\ell)\otimes !(xk\ \mathsf{outlives}\ ek_c)\otimes !(\mathsf{frzn}\ xk.x\ell\ G))}\ \text{(trans-}\tau\ *_S\text{)}$$

$$\frac{[\![\tau_1]\!]_\tau\ xk_{\text{a1}}\ =\ G_{\text{1}}\quad\ldots\quad[\![\tau_n]\!]_\tau\ xk_{\text{an}}\ =\ G_{\text{n}}\quad[\![\tau]\!]_\tau\ xk_{\text{ret}}\ =\ G_{\text{ret}}}{\begin{array}{l}[\![\tau_1 * \ldots * \tau_n \rightarrow \tau]\!]_\tau\ \_\ = \\ \quad(\exists\ x\ell{:}\mathsf{L}, x\ell'{:}\mathsf{L}, xk{:}\mathsf{TG}, xk_{\text{a1}}{:}\mathsf{TG}, \ldots, xk_{\text{an}}{:}\mathsf{TG}, \mathit{store}{:}\mathsf{F}. \\ \quad\ \mathsf{more}^{\leftarrow}(x\ell)\otimes\ (xk_{\text{an}}.x\ell{+}1\Rightarrow G_{\text{n}})\ \otimes\ldots\otimes\ (xk_{\text{a1}}.x\ell{+}n\Rightarrow G_{\text{1}}) \\ \quad\ \otimes\ \mathsf{more}^{\rightarrow}(x\ell')\otimes\ (r_{\text{alloc}}\Rightarrow\mathbf{S}(H.x\ell'{-}1))\otimes \mathit{store} \\ \quad\ \otimes\ (r_1\Rightarrow\mathbf{ns})\otimes(r_2\Rightarrow\mathbf{ns})\otimes(r_{\text{fp}}\Rightarrow\mathbf{ns}) \\ \quad\ \otimes\ (r_{\text{sp}}\Rightarrow\mathbf{S}(xk_{an}.x\ell{+}1)) \\ \quad\ \otimes\ (r_{\text{ra}}\Rightarrow\quad(\exists\ x\ell''{:}\mathsf{L}, F_{\text{H}}{:}\mathsf{F}, xk_{\text{ret}}{:}\mathsf{TG}. \\ \qquad\qquad\mathsf{more}^{\leftarrow}(x\ell{+}n{-}1)\otimes\ (xk_{\text{ret}}.x\ell{+}n\Rightarrow F_{\text{ret}}) \\ \qquad\qquad\otimes\ \mathsf{more}^{\rightarrow}(x\ell'')\otimes\ (r_{\text{alloc}}\Rightarrow\mathbf{S}(H.x\ell''{-}1))\otimes F_{\text{H}}\otimes\mathit{store} \\ \qquad\qquad\otimes\ (r_1\Rightarrow\mathbf{ns})\otimes(r_2\Rightarrow\mathbf{ns})\otimes(r_{\text{fp}}\Rightarrow\mathbf{ns})\otimes(r_{\text{ra}}\Rightarrow\mathbf{ns}) \\ \qquad\qquad\otimes\ (r_{\text{sp}}\Rightarrow\mathbf{S}(xk_{\text{ret}}.x\ell{+}n)) \\ \qquad\qquad\otimes\ \mathsf{first}(xk_{\text{ret}})\otimes !(xk{=}xk_{\text{ret}}{+}1)\ )\rightarrow 0\ ) \\ \quad\ \otimes\ \mathsf{first}(xk_{\text{an}})\otimes !(xk_{\text{an-1}}{=}xk_{\text{an}}{+}1)\ \otimes\ldots\otimes !(xk_{\text{a1}}{=}xk_{\text{a2}}{+}1)\otimes !(xk{=}xk_{\text{a1}}{+}1)\ )\rightarrow 0\end{array}}\ \text{(trans-fun)}$$

Figure 14: Type Translation

5.3 Environment Translations

Between each step in the translation, the shape of the memory can be reconstructed from: the variable context Γ, the variable map V that maps variables in Γ to the offset from the frame pointer at which they appear on the stack, the size sz that the local declarations use on the stack (which is equivalent to the number of local variables plus the amount of storage space used for stack allocated data), and the current function label f.

We assume that Γ is ordered so that we can use it to determine the order in which the arguments and the local variables were pushed onto the stack. Γ contains all the functions in scope, then all the function arguments, and then all the local variables. We also use Γ to determine the type of the current function f.

The variable map V needs to be consistent with Γ. In other words, it should map each of the n arguments a_i to $n - i$ (a positive offset) since the arguments are pushed in order (a_1 first, and a_n last) and the frame pointer r_{fp} points to a_n. The m local variables are also allocated on the stack, but they are intermixed with space used for allocating stack data. They will be given negative offsets from the frame pointer.

During the translation of lds, if the local variable is of type $\tau' *_S$ then space is pushed onto the stack for the translation of τ'. (Since only values can appear in local declarations, we are guaranteed to only need one space for the data.) And then in all cases space is pushed onto the stack for the local variable. So if a local variable is not a pointer to a stack location, its offset is one below the offset of the previous local variable. If the variable is a pointer to a stack location, its data will be stored at an offset one below the previous local variable and its offset will be two below.

Once we have determined that Γ and V are consistent, we can use this to determine how the local declarations shape the stack. The local declarations use sz space on the stack. For each stack slot j from 1 to sz, we use Γ and V to determine if the slot contains a local variable (j is in the range of V) or stack allocated data (j is not in the range of V, but $j - 1$ is) and what the type of the slot is.

The translation of the environment $[\![\Gamma,\ V,\ sz,\ f]\!]$ is shown in Figure 15. We know that the top sz slots will be used for the local declarations and the next n slots will contain the arguments. There is some abstract store *store* that is unknown. The stack pointer r_{sp} will point to the top of the stack, and the frame pointer r_{fp} will point to the last argument. The return address in r_{ra} will be a function that expects the local declarations and arguments to be popped off and the return value of the function τ to be pushed on the stack. The temporary registers r_1 and r_2 contain nonsense.

$[\![\Gamma, V, sz, f, \tau done]\!]$ is similar, except it represents the shape of memory just before a function call when the first t of the parameters to the call (with the list of types $\tau done$) have already been pushed on the stack. So the state of memory is as in $[\![\Gamma,\ V,\ sz,\ f]\!]$ except that there are now t new values on top of the stack (with the stack pointer and adjacency information updated appropriately).

$\boxed{[\![\Gamma,\ V,\ sz,\ f]\!] = (\Theta,\ F)}$ and $\boxed{[\![\Gamma, V, sz, f, \tau done]\!] = (\Theta,\ F)}$

$$
\begin{array}{rcl}
j & \in & 1 \ldots sz \\
F_j & = & \begin{cases} (xk_j.x\ell - j \Rightarrow [\![\tau_x]\!]_\tau\ xk_j) & \text{if } \exists x.\ V(x_i) = -j \text{ and } \Gamma(x) = \tau_x \neq \tau' *_S \\ !(\mathsf{frzn}\,(xk_j.x\ell - j)\,([\![\tau]\!]_\tau\ xk_{j-1})) & \text{if } \exists x.\ V(x_i) = -j-1 \text{ and } \Gamma(x) = \tau *_S \end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
F_{ra} & = & (\exists\, x\ell'' {:}\mathsf{L}, F_{\mathrm{H}} {:}\mathsf{F}, xk_{\mathrm{ret}} {:}\mathsf{TG}. \\
 & & \mathsf{more}^{\leftarrow}(x\ell{+}an) \otimes (xk_{\mathrm{ret}}.x\ell{+}an{-}1 \Rightarrow [\![\tau]\!]_\tau\ xk_{\mathrm{ret}}) \\
 & & \otimes\, \mathsf{more}^{\rightarrow}(x\ell'') \otimes (r_{\mathrm{alloc}} \Rightarrow \mathbf{S}(H.x\ell''{-}1)) \otimes F_{\mathrm{H}} \otimes \mathit{store} \\
 & & \otimes\, (r_1 \Rightarrow \mathbf{ns}) \otimes (r_2 \Rightarrow \mathbf{ns}) \otimes (r_{\mathrm{fp}} \Rightarrow \mathbf{ns}) \otimes (r_{\mathrm{ra}} \Rightarrow \mathbf{ns}) \\
 & & \otimes\, (r_{\mathrm{sp}} \Rightarrow \mathbf{S}(xk_{\mathrm{ret}}.x\ell{+}an{-}1)) \\
 & & \otimes\, \mathsf{first}(xk_{\mathrm{ret}}) \otimes\, !(xk{=}xk_{\mathrm{ret}}{+}1)\ ) \rightarrow 0\ )
\end{array}
$$

$$
\begin{array}{rcl}
[\![\Gamma,\ V,\ sz,\ f]\!] & \triangleq & (\Theta, F) \\
\text{where } \Theta & = & xk_{\mathrm{l}}, \ldots, xk_{\mathrm{sz}}, xk_{\mathrm{a1}}, \ldots, xk_{\mathrm{an}}, xk, \mathit{store} \\
F & = & \mathsf{more}^{\leftarrow}(x\ell{-}sz{-}1) \otimes F_{\mathrm{sz}} \otimes \ldots \otimes F_{\mathrm{l}} \\
 & & \otimes\, (xk_{\mathrm{an}}.x\ell \Rightarrow [\![\tau_{an}]\!]_\tau\ xk_{\mathrm{an}}) \otimes \ldots \otimes (xk_{\mathrm{a1}}.x\ell{+}an{-}1 \Rightarrow [\![\tau_{a1}]\!]_\tau\ xk_{\mathrm{a1}}) \\
 & & \otimes\, \mathsf{more}^{\rightarrow}(x\ell') \otimes (r_{\mathrm{alloc}} \Rightarrow \mathbf{S}(H.x\ell'{-}1)) \otimes \mathit{store} \\
 & & \otimes\, (r_1 \Rightarrow \mathbf{ns}) \otimes (r_2 \Rightarrow \mathbf{ns}) \\
 & & \otimes\, (r_{\mathrm{fp}} \Rightarrow xk_{\mathrm{an}}.x\ell) \otimes (r_{\mathrm{sp}} \Rightarrow \mathbf{S}(xk_{\mathrm{sz}}.x\ell{-}sz)) \otimes (r_{\mathrm{ra}} \Rightarrow F_{ra}) \\
 & & \otimes\, \mathsf{first}(xk_{\mathrm{sz}}) \otimes\, !(xk_{\mathrm{sz\text{-}1}}{=}xk_{\mathrm{sz}}{+}1) \otimes \ldots \otimes\, !(xk_{\mathrm{l}}{=}xk_2{+}1) \otimes\, !(xk_{\mathrm{an}}{=}xk_{\mathrm{l}}{+}1) \\
 & & \otimes\, !(xk_{\mathrm{an\text{-}1}}{=}xk_{\mathrm{an}}{+}1) \otimes \ldots \otimes\, !(xk_{\mathrm{a1}}{=}xk_{\mathrm{a2}}{+}1) \otimes\, !(xk{=}xk_{\mathrm{a1}}{+}1)
\end{array}
$$

$$
\begin{array}{rcl}
[\![\Gamma, V, sz, f, [\tau_{p1}, \ldots \tau_{pt}]]\!] & \triangleq & (\Theta, F) \\
\text{where } \Theta & = & xk_{\mathrm{p1}}, \ldots, xk_{\mathrm{pt}}, xk_{\mathrm{rfp}}, xk_{\mathrm{ret}}, xk_{\mathrm{l}}, \ldots, xk_{\mathrm{sz}}, xk_{\mathrm{a1}}, \ldots, xk_{\mathrm{an}}, xk, \mathit{store} \\
F & = & \mathsf{more}^{\leftarrow}(x\ell{-}sz{-}t{-}3) \\
 & & \otimes\, (xk_{\mathrm{pt}}.x\ell{-}sz{-}t{-}2 \Rightarrow [\![\tau_{pt}]\!]_\tau\ xk_{\mathrm{pt}}) \otimes \ldots \otimes (xk_{\mathrm{p1}}.x\ell{-}sz{-}3 \Rightarrow [\![\tau_{p1}]\!]_\tau\ xk_{\mathrm{p1}}) \\
 & & \otimes\, (xk_{\mathrm{ra}}.x\ell{-}sz{-}2 \Rightarrow F_{ra}) \otimes (xk_{\mathrm{rfp}}.x\ell{-}sz{-}1 \Rightarrow \mathbf{S}(xk_{\mathrm{an}}.x\ell)) \otimes F_{\mathrm{sz}} \otimes \ldots \otimes F_{\mathrm{l}} \\
 & & \otimes\, (xk_{\mathrm{an}}.x\ell \Rightarrow [\![\tau_{an}]\!]_\tau\ xk_{\mathrm{an}}) \otimes \ldots \otimes (xk_{\mathrm{a1}}.x\ell{+}an{-}1 \Rightarrow [\![\tau_{a1}]\!]_\tau\ xk_{\mathrm{a1}}) \\
 & & \otimes\, \mathsf{more}^{\rightarrow}(x\ell') \otimes (r_{\mathrm{alloc}} \Rightarrow \mathbf{S}(H.x\ell'{-}1)) \otimes \mathit{store} \\
 & & \otimes\, (r_1 \Rightarrow \mathbf{ns}) \otimes (r_2 \Rightarrow \mathbf{ns}) \\
 & & \otimes\, (r_{\mathrm{fp}} \Rightarrow xk_{\mathrm{an}}.x\ell) \otimes (r_{\mathrm{sp}} \Rightarrow \mathbf{S}(xk_{\mathrm{pt}}.x\ell{-}sz{-}t)) \otimes (r_{\mathrm{ra}} \Rightarrow F_{ra}) \\
 & & \otimes\, \mathsf{first}(xk_{\mathrm{sz}}) \otimes\, !(xk_{\mathrm{p\text{-}1}}{=}xk_{\mathrm{pt}}{+}1) \otimes \ldots \otimes\, !(xk_{\mathrm{p1}}{=}xk_{\mathrm{p2}}{+}1) \otimes\, !(xk_{\mathrm{sz}}{=}xk_{\mathrm{p1}}{+}1) \\
 & & \otimes\, !(xk_{\mathrm{sz\text{-}1}}{=}xk_{\mathrm{sz}}{+}1) \otimes \ldots \otimes\, !(xk_{\mathrm{l}}{=}xk_2{+}1) \otimes\, !(xk_{\mathrm{an}}{=}xk_{\mathrm{l}}{+}1) \\
 & & \otimes\, !(xk_{\mathrm{an\text{-}1}}{=}xk_{\mathrm{an}}{+}1) \otimes \ldots \otimes\, !(xk_{\mathrm{a1}}{=}xk_{\mathrm{a2}}{+}1) \otimes\, !(xk{=}xk_{\mathrm{a1}}{+}1)
\end{array}
$$

Figure 15: Determining Memory Shape

5.4 Translating the Micro-CLI Grammar

Each nonterminal in the grammar has its own translation function. We will leave out the cases for accumulators such as function declarations fds, local declarations lds, statements ss, and argument values avs since this should be clear from the corresponding single cases.

We define seven translation functions:

$[\![\vdash p]\!]_p = (C',\ \Psi',\ B')$	Program Translation
$[\![\Gamma \vdash fd : \Gamma']\!]_{fd}\ C\ \Psi\ =\ (C',\ \Psi')$	Function Declaration Translation
$[\![\Gamma \vdash^{\tau} rb\ \mathsf{ok}]\!]_{rb}\ f\ C_0\ \Psi_1\ =\ (C',\ \Psi')$	Return Block Translation
$[\![\Gamma \vdash ld : \Gamma']\!]_{ld}\ V\ sz\ = (V',\ sz',\ I')$	Local Declaration Translation
$[\![\Gamma \vdash s\ \mathsf{ok}]\!]_s\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I\ = (C'_0,\ \Psi'_1,\ cb',\ I')$	Statement Translation
$[\![\Gamma \vdash v : \tau]\!]_v\ V\ sz\ f\ F_1\ r\ = I'$	Value Translation
$[\![\Gamma; v \vdash \tau]\!]_{av}\ V\ sz\ f\ \tau done = I'$	Argument Translation

The arguments to the translation functions are summarized below.

C	Code Region	Function mapping code block labels to blocks of code
Ψ	Code Context	Function mapping code block labels to code types
B	Code Block	Sequence of instructions ending with a jump
Γ	Variable Context	Function mapping variables to types
V	Variable Map	Function mapping local variables to their offset from the frame pointer
sz	Local Declaration Size	Number of spaces on the stack needed to store the frame of the current function
cb	Code Label	Label for the code block that is currently being translated
I	Instruction List	Sequence of instruction that does not end with a jump
$\tau list$	Type List	List of types for arguments that have already been pushed on the stack

We use the following abbreviations for commonly used instruction sequences. Note that `update`x requires the value to update x with to be in r_1 and will clobber any values in r_2.

$$\begin{array}{lcl} \texttt{update}\,x & \equiv & \texttt{mov}\,r_2, r_{\text{fp}};\ \texttt{add}\,r_2, r_2, V(x);\ \texttt{st}\,r_2, r_1 \\ \texttt{push}\,r & \equiv & \texttt{stackgrow};\ \texttt{sub}\,r_{\text{sp}}, r_{\text{sp}}, 1;\ \texttt{st}\,r_{\text{sp}}, r \\ \texttt{pop}\,r & \equiv & \texttt{ld}\,r, r_{\text{sp}};\ \texttt{add}\,r_{\text{sp}}, r_{\text{sp}}, 1;\ \texttt{stackcut} \end{array}$$

5.4.1 Program Translation: $[\![\vdash p]\!]_p = (C',\ \Psi',\ B)$

The translation of a program p returns the final code region C' with type Ψ' that contains all the blocks of code from the whole program and the block B' that jumps to main.

To translate a program $p = fds\ rb$, we start with the code region containing a block halt that simply halts the machine. To halt, memory must be in a basic state where the stack has only the return value, and the contents of all registers except for r_{sp} are arbitrary. At the start of the assembly program, we first push an integer (so that we are in the base memory state), set the return location to be halt, and jump to main.

The function declarations fds are translated beginning with this initial code region, accumulating code blocks and resulting in the code region C with type Ψ. The provided "main" function to the program is rb and we will map this using the label main. It begins with the state necessary to jump to halt and will also finish in that state. Using the code region C containing the function declarations, the main block rb is translated as function "main".

Lemma 6 proves that a well typed source program translates into a well typed assembly program.

Lemma 6 (Trans-p)
If $[\![\vdash p]\!]_p = (C,\ \Psi,\ B)$ *and* $t = (H, (sk;), sk, fst)$ *and* $m = \{r_{sp} \mapsto sk.\ell, r_{alloc} \mapsto H.\ell'\}$ *then* $\vdash^{\Psi} (C, m, t, B)\ \mathsf{ok}$.

Proof 6
By Lemma Trans-fds (similar to Lemma 7) and Lemma 8.

$$\frac{[\![\cdot \vdash fds : \Gamma]\!]_{fds}\ \{\mathsf{halt} \mapsto \mathtt{halt}\}\ \{\mathsf{halt} \mapsto F_{halt}\}\ =\ (C,\ \Psi) \quad [\![\Gamma \vdash^{\tau} rb\ \mathsf{ok}]\!]_{rb}\ \mathsf{main}\ C\ \Psi[\mathsf{main} \mapsto F_{halt}]\ =\ (C',\ \Psi')}{[\![\vdash fds\ rb]\!]_p = (C',\ \Psi',\ (\mathtt{push}\,1;\ \mathtt{mov}\,r_{\mathrm{ra}}, \mathsf{halt};\ \mathtt{jmp}\,\mathsf{main}))}\ \text{(trans-p)}$$

where $F_{halt} =\ \exists x\ell{:}\mathsf{L}, x\ell'{:}\mathsf{L}, xk{:}\mathsf{TG}.$
$\mathsf{more}^{\leftarrow}(x\ell\text{-}1) \otimes (xk.x\ell \Rightarrow \mathbf{int}) \otimes (r_{\mathrm{sp}} \Rightarrow \mathbf{S}(xk.x\ell))$
$\otimes\ \mathsf{more}^{\rightarrow}(x\ell') \otimes (r_{\mathrm{alloc}} \Rightarrow \mathbf{S}(H.x\ell'\text{-}1)) \otimes F_{\mathrm{H}}$
$\otimes\ (r_1 \Rightarrow \mathbf{ns}) \otimes (r_2 \Rightarrow \mathbf{ns}) \otimes (r_{\mathrm{ip}} \Rightarrow \mathbf{ns}) \otimes (r_{\mathrm{ra}} \Rightarrow \mathbf{ns})$
$\otimes\ \mathsf{first}(xk)$

Figure 16: Program Translation

5.4.2 Function Declaration Translation: $[\![\Gamma \vdash fd : \Gamma']\!]_{fd}\ C\ \Psi\ =\ (C',\ \Psi')$

The translation of a function declaration adds an entry for the translation of the function contents in the code region C under the label f. The return block rb in the function is translated in a context that already contains the type of the function f added to the code context Ψ. This is needed to allow the function to be recursive.

$$\frac{\Gamma \vdash ads : (\Gamma', \tau list) \quad [\![\Gamma'[f : \tau list \to \tau] \vdash^{\tau} rb\ \mathsf{ok}]\!]_{rb}\ f\ C\ \Psi[f : [\![\tau list \to \tau]\!]_{\tau}\ \_]\ =\ (C',\ \Psi')}{[\![\Gamma \vdash \tau\ f(ads)\ rb : \Gamma[f : \tau list \to \tau]]\!]_{fd}\ C\ \Psi\ =\ (C',\ \Psi')}\ \text{(trans-fd)}$$

Figure 17: Function Declaration Translation

Lemma 7 checks that the given code region C and code context Ψ are consistent and that the values it returns are also consistent.

Lemma 7 (Trans-fd)
If $[\![\Gamma \vdash fd : \Gamma']\!]_{fd}\ C\ \Psi\ =\ (C',\ \Psi')$ *and* $\vdash C : \Psi$ *then* $\vdash C' : \Psi'$.

Proof 7
By Lemma 8.

5.4.3 Return Block Translation: $[\![\Gamma \vdash^{\tau} rb\ \mathsf{ok}]\!]_{rb}\ f\ C_0\ \Psi_1\ =\ (C',\ \Psi')$

The translation of a return block rb translates the code in the block and then adds it to the code region C_0 under the name f.

The translation takes the current code region C_0, its type Ψ_1, and the label of the current function f. Ψ_1 is a step ahead of C_0; it already contains a type for f. This allows a function to be recursive.

The first thing the block does upon entry is to set the frame pointer to be equal to the stack pointer. Then it executes the code resulting from the translations of lds and ss. Before the return statement, the n arguments and sz of local declarations and stack allocated data are popped off of the stack. Finally, the return value is pushed onto the stack.

The translation returns the code region C' and its type Ψ'. Notice that C' and Ψ' are in step.

Lemma 8 ensures that except for the entry for f, C_0 has type Ψ_1 and that when the translation of rb is finished, C' has type Ψ'.

Lemma 8 (Trans-rb)
If $[\![\Gamma \vdash^{\tau} rb\ \mathsf{ok}]\!]_{rb}\ f\ C_0\ \Psi_1\ =\ (C',\ \Psi')$ *and* $\vdash C_0 : (\Psi_1 \backslash f)$ *then* $\vdash C' : \Psi'$.

$$\frac{\begin{array}{c}[\![\Gamma \vdash lds : \Gamma']\!]_{lds}\ V\ 0\ = (V',\ sz',\ I_{lds}) \\ [\![\Gamma' \vdash ss\ \mathsf{ok}]\!]_s\ V'\ sz'\ f\ C_0\ \Psi_1\ cb\ (\mathtt{mov}\, r_{\mathrm{fp}}, r_{\mathrm{sp}};\ I_{lds}) = (C_0',\ \Psi_1',\ cb',\ I') \\ [\![\Gamma',\ V',\ sz',\ f]\!] = (\Theta,\ F) \qquad [\![\Gamma' \vdash v : \tau]\!]_v\ V'\ sz'\ f\ F\ r_1\ = I_v\end{array}}{[\![\Gamma \vdash^{\tau} \{lds; ss; \mathbf{return}\, v\}\ \mathsf{ok}]\!]_{rb}\ f\ C_0\ \Psi_1\ =\ (C_0'[cb' \mapsto I''],\ \Psi_1')}\ \textsf{(trans-rb)}$$

where $I'' = I';\ I_v;\ \mathtt{pop}\, r_2;\ \ldots\ \mathtt{pop}\, r_2;\ \mathtt{push}\, r_1;\ \mathtt{jmp}\, r_{\mathrm{ra}}$

Figure 18: Return Block Translation

Proof 8
By Lemma Trans-lds (similar to Lemma 9), Lemma Trans-ss (similar to Lemma 10), and Lemma 12.

5.4.4 Local Declaration Translation: $[\![\Gamma \vdash ld : \Gamma']\!]_{ld}\ V\ sz\ = (V',\ sz',\ I')$

The translation of local declarations allocates space on the stack for each new variable. The translation takes the variable map V and the size sz allocated so far for the local declaration.

For variables that are initialized to values, space for the local variable is pushed on the stack, and the value is moved into that stack slot. For declarations involving allocation, space is allocated in the appropriate place, the value is loaded into the allocated space, and then a space for the local variable is made on the stack, and the pointer to the newly allocated space is moved into that stack slot.

The translation returns the new variable map V including a mapping for the new variable declared in the local declaration, the new size sz of the local declarations on the stack (incremented by two if the value allocated stack space, and incremented by one otherwise), and the sequence of instructions that does the allocation and initialization.

$$\frac{[\![\Gamma \vdash v : \tau]\!]_v\ V\ sz\ f\ F\ r_1\ = I_v}{[\![\Gamma \vdash \tau\, x\ =\ v : \Gamma[x : \tau]]\!]_{ld}\ V\ sz\ = (V[x \mapsto sz+1],\ sz+1,\ I')}\ \textsf{(trans-ld-var)}$$
where $I' = I_v;\ \mathtt{push}\, r_1$

$$\frac{[\![\Gamma \vdash v : \tau]\!]_v\ V\ sz\ f\ F\ r_1\ = I_v}{[\![\Gamma \vdash \tau\, x\ =\ \mathbf{new}_S\, v : \Gamma[x : \tau]]\!]_{ld}\ V\ sz\ = (V[x \mapsto sz+2],\ sz+2,\ I')}\ \textsf{(trans-ld-}\tau *_S\textsf{)}$$
where $I' = I_v;\ \mathtt{push}\, r_1;\ \mathtt{mov}\, r_2, r_{\mathrm{sp}};\ \mathtt{push}\, r_2$

$$\frac{[\![\Gamma \vdash v : \tau]\!]_v\ V\ sz\ f\ F\ r_1\ = I_v}{[\![\Gamma \vdash \tau\, x\ =\ \mathbf{new}_H\, v : \Gamma']\!]_{ld}\ V\ sz\ = (V[x \mapsto sz+1],\ sz+1,\ I')}\ \textsf{(trans-ld-}\tau *_H\textsf{)}$$
where $I' = I_v;\ \mathtt{add}\, r_{\mathrm{alloc}}, 1;\ \mathtt{st}\, r_{\mathrm{alloc}}, r_1;\ \mathtt{push}\, r_{\mathrm{alloc}}$

Figure 19: Local Declaration Translation

Lemma 9 checks that the instruction sequence I' correctly allocates space for the the variable. If a block B is well-formed in the memory after the declaration, then a block consisting of the instruction sequence I' and the block B should be well-formed in the original memory. In other words, the instruction sequence I' takes us from the original memory shape to the memory shape after the declaration.

Lemma 9 (Trans-ld)
If $[\![\Gamma \vdash ld : \Gamma']\!]_{ld}\ V\ sz\ = (V',\ sz',\ I')$ *then* $\forall B.\ (\Theta' \parallel F' \vdash B\ \mathsf{ok} \Longrightarrow \Theta \parallel F \vdash (I';\ B)\ \mathsf{ok})$ *where* $[\![\Gamma,\ V,\ sz,\ f]\!] = (\Theta,\ F)$*, and* $[\![\Gamma',\ V',\ sz',\ f]\!] = (\Theta',\ F')$.

Proof 9
By cases on the translation.

5.4.5 Statement Translation: $[\![\Gamma \vdash s\ \mathsf{ok}]\!]_s\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I\ = (C_0',\ \Psi_1',\ cb',\ I')$

The translation of statements translates each statement into a sequence of assembly instructions. The translation takes the variable map V, local declaration size sz, function name f, code region C_0, code context Ψ_1, code label cb of the block currently being translated, and instruction sequence I of translated instructions so far.

To determine the state of memory we can use Γ, V, sz, and f: $[\![\Gamma,\ V,\ sz,\ f]\!] = (\Theta,\ F)$. The memory state will be constant between each statement since all allocation is done in lds. The temporary registers r_1 and r_2 may change within the translation of a statement, but they are not used across statements, and therefore can be assumed to be $\mathbf{ns}$.

The code region C_0 contains all the code blocks already translated. The code context Ψ_1 contains the types of all blocks already translated, and also includes the type of the code block cb, since the block may be recursive. Except for cb, the domains of C_0 and Ψ_1 are equal, and for every element x in the joint domain, $(\Psi_1(x) = \exists\Theta_x.F_x \to 0) \implies (\Theta_x \parallel F_x \vdash C_0(x)\ \mathsf{ok})$.

The code label cb is the name of the current block being translated and I accumulates the instructions that provide the translation of the previous statements in the block. The translation returns the new code region C_0', code context Ψ_1', code label cb', and sequence of instructions I'. Except for cb', the other return values are all either the same as or extensions of the equivalent inputs. cb' will either be the same as cb or it will be a fresh code label.

In the simple cases that do not create new code blocks (simple variable assignment, addition, subtraction, variable dereference, and update), C_0', Ψ_1', and cb' are the corresponding inputs and I' is the sequence of assembly instructions that that perform the operation. The cases for function calls and if statements involve control flow, so they create new code blocks, which means that C_0' and Ψ_1' are extensions of the corresponding inputs and cb' is a fresh label.

For function calls $x = f(avs)$, we create a sequence of instructions consisting of the current instructions I, instructions to save the frame pointer and return address on the stack, the instructions to push the arguments onto the stack, and then instructions to call the function f. We add this entire sequence to the code region C_0 under the name cb. Then we begin the next code block cb_{cont} which represents the return point of the function call. The first thing the code will do when it returns is pop the result off of the stack and store it into the space for x as well as restore the frame pointer and return address. The translation returns at this point, so the next statement translated will be part of cb_{cont}.

For if statements **if** v **then** ss_1 **else** ss_2, we create a sequence of instructions consisting of the current instructions I and instructions to check the condition v and branch appropriately to either cb_{true} or cb_{false}. We then translate the blocks ss_{true} and ss_{false}, using the code labels cb_{true} and cb_{false} respectively. These translations will each return a new code region (C_0^t and C_0^f), code context (Ψ_1^t and Ψ_1^f), code label (cb^t and cb^f), and instruction sequence (I^t and I^f). Each translation may have added an arbitrary number of code blocks, depending on the number of nested if statements and function calls it contained. The final code region we return contains the union of C_0^t and C_0^f with the addition of two blocks to finish the code from each translation and terminate it in a jump to cb_{cont}. The final code context is the union of Ψ_1^t and Ψ_1^f with the addition of the type of cb_{cont}. The translation returns with an empty block with label cb_{cont} that will be the control flow merge point after the if statement.

$$\frac{\begin{array}{c}[\![\Gamma,\ V,\ sz,\ f]\!] = (\Theta,\ F) \qquad \Gamma \vdash x : \tau \\ [\![\Gamma \vdash v : \tau]\!]_v\ V\ sz\ f\ F\ r_1\ = I_v \qquad (\tau \neq \tau' *_S)\end{array}}{[\![\Gamma \vdash x\ =\ v\ \mathsf{ok}]\!]_s\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I\ = (C_0,\ \Psi_1,\ cb,\ (I;\ I_v;\ \mathtt{update}\, x))} \text{ (trans-s-varassign)}$$

$$\frac{\begin{array}{c}[\![\Gamma,\ V,\ sz,\ f]\!] = (\Theta,\ F) \qquad \Gamma \vdash x : \mathbf{int} \\ [\![\Gamma \vdash v_1 : \mathbf{int}]\!]_v\ V\ sz\ f\ F\ r_1\ = I_1 \qquad [\![\Gamma \vdash v_2 : \mathbf{int}]\!]_v\ V\ sz\ f\ (F[r_1 := \mathbf{int}])\ r_2\ = I_2\end{array}}{[\![\Gamma \vdash x\ =\ v_1 + v_2\ \mathsf{ok}]\!]_s\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I\ = (C_0,\ \Psi_1,\ cb,\ (I;\ I_1;\ I_2;\ \mathtt{add}\, r_1, r_1, r_2;\ \mathtt{update}\, x))} \text{ (trans-s-add)}$$

$$\frac{\begin{array}{c}[\![\Gamma,\ V,\ sz,\ f]\!] = (\Theta,\ F) \qquad \Gamma \vdash x : \mathbf{int} \\ [\![\Gamma \vdash v_1 : \mathbf{int}]\!]_v\ V\ sz\ f\ F\ r_1\ = I_1 \qquad [\![\Gamma \vdash v_2 : \mathbf{int}]\!]_v\ V\ sz\ f\ (F[r_1 := \mathbf{int}])\ r_2\ = I_2\end{array}}{[\![\Gamma \vdash x\ =\ v_1 - v_2\ \mathsf{ok}]\!]_s\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I\ = (C_0,\ \Psi_1,\ cb,\ (I;\ I_1;\ I_2;\ \mathtt{sub}\, r_1, r_1, r_2;\ \mathtt{update}\, x))} \text{ (trans-s-sub)}$$

$$\frac{\begin{array}{c}[\![\Gamma,\ V,\ sz,\ f]\!] = (\Theta,\ F) \qquad \Gamma \vdash x : \tau \\ [\![\Gamma \vdash v : \tau *_q]\!]_v\ V\ sz\ f\ F\ r_1\ = I_v \qquad (\tau \neq \tau' *_S)\end{array}}{[\![\Gamma \vdash x\ =\ !\, v\ \mathsf{ok}]\!]_s\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I\ = (C_0,\ \Psi_1,\ cb,\ (I;\ I_v;\ \mathtt{ld}\, r_1, r_1;\ \mathtt{update}\, x))} \text{ (trans-s-deref)}$$

$$\frac{\begin{array}{c}[\![\Gamma,\ V,\ sz,\ f]\!] = (\Theta,\ F) \qquad \Gamma \vdash x : \tau \qquad (\tau \neq \tau' *_S) \\ [\![\Gamma \vdash v_1 : \mathbf{int}]\!]_v\ V\ sz\ f\ F\ r_1\ = I_1 \qquad [\![\Gamma \vdash v_2 : \mathbf{int}]\!]_v\ V\ sz\ f\ (F[r_1 := [\![\tau]\!]_\tau\ \_])\ r_2\ = I_2\end{array}}{[\![\Gamma \vdash v_1\ :=\ v_2\ \mathsf{ok}]\!]_s\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I\ = (C_0,\ \Psi_1,\ cb,\ (I;\ I_1;\ I_2;\ \mathtt{st}\, r_1, r_2))} \text{ (trans-s-update)}$$

$$\frac{\begin{array}{l}[\![\Gamma,\ V,\ sz,\ f]\!] = (\Theta,\ F) \qquad [\![\Gamma \vdash v : \mathbf{int}]\!]_v\ V\ sz\ f\ F\ r_1\ = I_v \\ [\![\Gamma \vdash ss_1\ \mathsf{ok}]\!]_s\ V\ sz\ f\ C_1\ (\Psi_1[cb_{true} \mapsto (\exists\Theta.F \to 0)])\ cb_{true}\ \emptyset\ = (C_0^t,\ \Psi_1^t,\ cb^t,\ I^t) \\ [\![\Gamma \vdash ss_2\ \mathsf{ok}]\!]_s\ V\ sz\ f\ C_1\ (\Psi_1[cb_{false} \mapsto (\exists\Theta.F \to 0)])\ cb_{false}\ \emptyset\ = (C_0^f,\ \Psi_1^f,\ cb^f,\ I^f)\end{array}}{[\![\Gamma \vdash \mathbf{if}\, v\, \mathbf{then}\, ss_1\, \mathbf{else}\, ss_2\ \mathsf{ok}]\!]_s\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I\ = (C_0',\ \Psi_1',\ cb_{cont},\ \emptyset)} \text{ (trans-s-if)}$$

where
$$\begin{array}{rcl} C_1 & = & (C_0[cb \mapsto (I;\ I_v;\ \mathtt{bz}\, r_1, cb_{false};\ \mathtt{jmp}\, cb_{true})]) \\ C_0' & = & (C_0^t \cup C_0^f)[cb^t \mapsto (I^t;\ \mathtt{jmp}\, cb_{cont})][cb^f \mapsto (I^f;\ \mathtt{jmp}\, cb_{cont})] \\ \Psi_1' & = & (\Psi_1^t \cup \Psi_1^f)[cb_{cont} \mapsto (\exists\Theta.F \to 0)] \end{array}$$

$$\frac{\begin{array}{c}\Gamma \vdash f : \tau list \to \tau \qquad \Gamma \vdash x : \tau \qquad (\tau \neq \tau' *_S) \\ [\![\Gamma,\ V,\ sz,\ f]\!] = (\Theta,\ F) \\ [\![\Gamma; \tau list \vdash avs\ \mathsf{ok}]\!]_{avs}\ V\ sz\ f\ [\,] = I_{avs}\end{array}}{[\![\Gamma \vdash x\ =\ f(avs)\ \mathsf{ok}]\!]_s\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I\ = (C_1,\ \Psi_2,\ cb_{cont},\ \emptyset)} \text{ (trans-s-funcall)}$$

where
$$\begin{array}{rcl} F & = & \mathsf{more}^{\leftarrow}(x\ell) \otimes \mathsf{first}(xk) \otimes (r_{\mathrm{sp}} \Rightarrow \mathbf{S}(xk.x\ell{+}1)) \otimes (r_{\mathrm{fp}} \Rightarrow \mathbf{S}(xk_{\mathrm{an}}.x\ell{+}sz)) \otimes (r_{\mathrm{ra}} \Rightarrow F_{ra}) \otimes F_{mem} \\ F_{cont} & = & \mathsf{more}^{\leftarrow}(x\ell{-}3) \otimes (xk_{\mathrm{ret}}.x\ell{-}2 \Rightarrow [\![\tau]\!]_\tau\ xk_{\mathrm{ret}}) \otimes (xk_{\mathrm{ra}}.x\ell{-}1 \Rightarrow F_{ra}) \\ & & \otimes (xk_{\mathrm{rfp}}.x\ell \Rightarrow \mathbf{S}(xk_{\mathrm{an}}.x\ell)) \otimes \mathsf{first}(xk_{\mathrm{ret}}) \\ & & \otimes !(xk_{\mathrm{ra}}{=}xk_{\mathrm{ret}}{+}1) \otimes !(xk_{\mathrm{rfp}}{=}xk_{\mathrm{ra}}{+}1) \otimes !(xk{=}xk_{\mathrm{rfp}}{+}1) \\ & & \otimes (r_{\mathrm{sp}} \Rightarrow \mathbf{S}(xk_{\mathrm{ret}}.x\ell{-}2)) \otimes (r_{\mathrm{ra}} \Rightarrow \mathbf{ns}) \otimes (r_{\mathrm{fp}} \Rightarrow \mathbf{ns}) \\ & & \otimes F_{mem} \\ \Theta_{cont} & = & \Theta, xk_{\mathrm{ret}}, xk_{\mathrm{ra}}, xk_{\mathrm{rfp}} \\ C_1 & = & C_0[cb \mapsto (I;\ \mathtt{push}\, r_{\mathrm{fp}};\ \mathtt{push}\, r_{\mathrm{ra}};\ I_{avs};\ \mathtt{mov}\, r_{\mathrm{ra}}, cb_{cont};\ \mathtt{jmp}\, f)] \\ \Psi_2 & = & \Psi_1[cb_{cont} \mapsto (\exists\Theta_{cont}.F_{cont} \to 0)] \end{array}$$

Figure 20: Statement Translation

Lemma 10 ensure that before and after translating the statement, the code region is one step ahead of the code context and that the instruction sequence preserves the shape of memory. We use the notation $(\Psi_1 \backslash cb)$ to mean the function Ψ_1 without the mapping for cb.

Lemma 10 (Trans-s)
If $[\![\Gamma \vdash s \ \mathsf{ok}]\!]_s \ V \ sz \ f \ C_0 \ \Psi_1 \ cb \ I \ = (C_0', \ \Psi_1', \ cb', \ I')$
and $\vdash C_0 : (\Psi_1 \backslash cb)$ *and* $\forall B.\ (\Theta \parallel F \vdash B \ \mathsf{ok} \Longrightarrow \Theta_{cb} \parallel F_{cb} \vdash (I; \ B) \ \mathsf{ok})$
then $\vdash C_0' : (\Psi_1' \backslash cb')$ *and* $\forall B.\ (\Theta \parallel F \vdash B \ \mathsf{ok} \Longrightarrow \Theta'_{cb} \parallel F'_{cb} \vdash (I'; \ B) \ \mathsf{ok})$
where $[\![\Gamma, \ V, \ sz, \ f]\!] = (\Theta, \ F)$, $\Psi_1(cb) = \exists \Theta_{cb}.F_{cb} \to 0$, *and* $\Psi_1'(cb') = \exists \Theta'_{cb}.F'_{cb} \to 0$.

Proof 10
By cases on the translation using Lemma Trans-ss (similar to Lemma 10) and Lemma 12.

5.4.6 Value Translation: $[\![\Gamma \vdash v : \tau]\!]_v \ V \ sz \ f \ F_1 \ r \ = I'$

The translation of a value simply copies the value into the provided register. The translation takes the variable map V, lds size sz, function name f, current formula F_1, and result register r. The first three are used to reconstruct the state of memory (Θ, F) between statements. The current formula F_1 is the state of memory right before the call to translate-value.

The memory state is always consistent between sequences of instructions corresponding to distinct statements, but within the commands for the translation of a statement, the memory state may change because of values in the temporary registers r_1 and r_2. So the only possible difference between F and F_1 is the values in the temporary register other than the one that is about to be written into.

The value translation returns the sequence of instructions I' that move the value into r.

$$\frac{}{[\![\Gamma \vdash n : \mathbf{int}]\!]_v \ V \ sz \ f \ F \ r \ = \mathtt{mov}\, r, n} \ \text{(trans-v-int)}$$

$$\frac{\Gamma(x) = \tau}{[\![\Gamma \vdash x : \tau]\!]_v \ V \ sz \ f \ F \ r \ = (\mathtt{mov}\, r, r_{\mathfrak{hp}}; \ \mathtt{add}\, r, V(x); \ \mathtt{ld}\, r, r)} \ \text{(trans-v-var)}$$

$$\frac{[\![\Gamma \vdash v : \tau']\!]_v \ V \ sz \ f \ F \ r \ = I_v \qquad \tau \leq \tau'}{[\![\Gamma \vdash v : \tau]\!]_v \ V \ sz \ f \ F \ r \ = I_v} \ \text{(trans-v-subtype)}$$

Figure 21: Value Translation

Lemma 11 proves that the subtyping relationship between heap and stack pointers exists at the assembly level. In other words, given two source level types that are subtypes, we can make them look equivalent at the assembly level.

Lemma 11 (Subtype)
If $\tau \leq \tau'$ *then* $[\![\tau']\!]_\tau \ H \equiv [\![\tau]\!]_\tau \ H$.

Proof 11
By cases on the subtyping relationship.

Lemma 12 checks that the current formula F_1 is just the shape of memory with possibly an updated temporary register and that the sequence of instructions I' terminates in a state where F_1 is updated with the translation of value v in r.

If the value is not a $\tau' *_S$ then the type translation $[\![\tau]\!]_\tau\ xk$ is the same whether xk is H or k_x. However, in the case where τ is $\tau' *_S$, we have the option of using a "real" stack pointer or a heap pointer that is being treated as a stack pointer (using the subtyping relationship). We only need to prove the lemma holds in one of these cases, but any use of this lemma may not assume which case we are using unless it has information about the structure of τ.

Lemma 12 (Trans-v)
If $[\![\Gamma \vdash v : \tau]\!]_v\ V\ sz\ f\ F_1\ r\ = I'$
and $\exists \tau'.F_1 = F[r' := \tau']$
then $\forall B.\ (\Theta \parallel F'[r := [\![\tau]\!]_\tau\ xk] \vdash B\ \mathsf{ok} \Longrightarrow \Theta'_{cb} \parallel F'_{cb} \vdash (I';\ B)\ \mathsf{ok})$
where $[\![\Gamma,\ V,\ sz,\ f]\!] = (\Theta,\ F)$, xk *is either* H *or* k_x, r *is either* r_1 *or* r_2, *and* r' *is the other one of* r_1 *or* r_2.

Proof 12
By cases on the translation and Lemma 11.

5.4.7 Argument Translation: $[\![\Gamma; v \vdash \tau]\!]_{av}\ V\ sz\ f\ \tau done = I'$

The translation of arguments to a function call pushes the each argument onto to the stack. The translation takes the variable map V, lds size sz, function name f, and list of translated argument types $\tau done$. It uses this information to reconstruct the state of memory (Θ, F) before the current argument is pushed.

The translation returns the sequence of instructions I' that push the value onto the stack.

$$\frac{[\![\Gamma,\ V,\ sz,\ f]\!] = (\Theta,\ F) \qquad [\![\Gamma \vdash v : \tau']\!]_v\ V\ sz\ f\ F\ r_1\ = I_v}{[\![\Gamma; v \vdash \tau]\!]_{av}\ V\ sz\ f\ \tau done = I_v;\ \mathtt{push}\, r_1}\ \text{(trans-av)}$$

Figure 22: Argument Translation

Lemma 13 determines what the shape of memory should be before and after pushing the value and then proves that the instruction sequence I' links these two memory states.

Lemma 13 (Trans-av)
If $[\![\Gamma; av \vdash \tau]\!]_{av}\ V\ sz\ f\ \tau done = I'$
then $\forall B.\ (\Theta_\tau \parallel F_\tau \vdash B\ \mathsf{ok} \Longrightarrow \Theta \parallel F \vdash (I';\ B)\ \mathsf{ok}\ )$
where $[\![\Gamma, V, sz, f, \tau done@[\tau]]\!] = (\Theta_\tau,\ F_\tau)$, *and* $[\![\Gamma, V, sz, f, \tau done]\!] = (\Theta,\ F)$.

Proof 13
By cases on the structure of τ *and Lemma 12.*

5.5 Proof of Type Preservation

We prove that our translation is type preserving by proving that the translation of a well typed source program is also well typed.

Theorem 14 (Type Preservation of the Translation)
If $[\![p]\!] = (C, \Psi, B)$ *then there exists an initial memory* m *and initial tag tree* t *such that* $\vdash^\Psi (C, m, t, B)\ \mathsf{ok}$

Proof 14
This follows immediately from Lemma 6.

6 Conclusion

In this paper we have presented a general logic with domain-specific predicates that is powerful enough to express general heap and stack allocation. We have demonstrated this expressiveness by defining a translation from a language that abstracts the important stack allocation features of CLI to our assembly language.

The choice of a tag tree matches well with a stack. We plan to investigate if it generalizes to other schemes that are not LIFO. The combination of unrestricted capabilities with versioning and a validity scheme for the versions seems promising as a general logic for memory management.

Acknowledgments This research was supported in part by ARDA Grant no. NBCHC030106, National Science Foundation grants CCR-0238328 and CCR-0208601 and an Alfred P. Sloan Fellowship. This work does not necessarily reflect the opinions or policy of the federal government or Sloan foundation and no official endorsement should be inferred.

References

[1] A. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *IEEE Symposium on Logic in Computer Science*, pages 33–44, Ottawa, Canada, June 2003.

[2] A. Ahmed and D. Walker. The logical approach to stack typing. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, New Orleans, Jan. 2003.

[3] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM Conference on Programming Language Design and Implementation*, pages 174–185, La Jolla, California, 1995.

[4] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107. ACM Press, 2000.

[5] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Jan. 1999.

[6] ECMA. Standard ECMA-335: Common Language Infrastructure (CLI), Dec. 2001. http://www.ecma.ch.

[7] M. Fähndrich and R. Deline. Adoption and focus: Practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation*, pages 13–24, Berlin, June 2002. ACM Press.

[8] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, Berlin, June 2002. ACM Press.

[9] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 248–260. ACM Press, 2001.

[10] D. Grossman. *Safe Programming at the C Level of Abstraction*. PhD thesis, Cornell University, 2003.

[11] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based Memory Management in Cyclone. In *ACM Conference on Programming Language Design and Implementation*, Berlin, June 2002. ACM Press.

[12] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 14–26, London, UK, Jan. 2001.

[13] G. Morrisett, A. Ahmed, and M. Fluet. L^3: A linear language with locations. In *Seventh International Conference on Typed Lambda Calculi and Applications*, 2005.

[14] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based Typed Assembly Language. *Journal of Functional Programming*, 12(1):43–88, Jan. 2002.

[15] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 3(21):528–569, May 1999.

[16] P. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[17] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, number 2142 in LNCS, pages 1–19, Paris, 2001.

[18] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[19] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381, Berlin, Mar. 2000.

[20] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, Jan. 1994.