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

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 [16, 15, 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 [15] and proof-carrying code [4] allows reasoning about data allocated in the current stack frame, but not much more. STAL [15] 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, 17, 18, 19], 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 [20]. In alias types [20], 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. [14] 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 [21, 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 $\forall \ell_1, \ell_2.((\ell_1 \Rightarrow \mathbf{int}) \otimes (\ell_2 \Rightarrow \mathbf{int})) \rightarrow \cdots$. 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 [20] and L^3 [14]. 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 $\forall \ell_1, \ell_2.((\mathsf{frzn}\ \ell_1\ \mathbf{int}) \otimes (\mathsf{frzn}\ \ell_2\ \mathbf{int})) \rightarrow \cdots$ 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 $\forall \ell_1, \ell_2.(!(\mathsf{frzn}\ \ell_1\ \mathbf{int}) \otimes !(\mathsf{frzn}\ \ell_2\ \mathbf{int})) \rightarrow \cdots$.

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 $(r_{\mathsf{sp}} \Rightarrow \mathbf{S}(\ell)) \otimes (\ell \Rightarrow \mathbf{int}) \otimes \ldots$ 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 $(r_{\mathsf{sp}} \Rightarrow \mathbf{S}(\ell)) \otimes !(\mathsf{frzn}\ \ell\ \mathbf{int}) \otimes \ldots$. 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

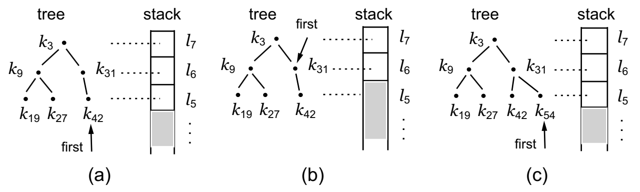

Figure 1. Example Memory Stack Tag Tree

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.

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 rules for logical deduction and formula equivalence are given in the companion technical report [13].

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.

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

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.

Store Value	sv	$::=$	$i \mid k.\ell \mid c$
Memory	m	$\in$	$\mathit{Loc} \cup \mathit{Reg} \rightharpoonup \mathit{Sval}$
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.

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.

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.

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

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 $\forall k.\ell \in \mathit{dom}(\Pi_{old} \cup \Pi)$:
$k \in \mathsf{Live}\,(t)$ iff $k.\ell \in \mathit{dom}(\Pi)$

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 $k.\ell \in \mathit{dom}(\Pi)$
$\Pi_{old}; \Pi; \ell \to m(\ell); t \vDash^{\Psi} \Pi(k.\ell)\, at\, k.\ell$

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.

Linear Judgment The main semantic judgment is the linear satisfaction judgment. It is defined in Figure 2. 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 2, if we substitute G for F.

- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \tau\, at\, p$ iff $p \neq *$ and $t.\mathit{fst} = \mathit{absent}$, 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 $dom(m) = \emptyset$, $t.\mathit{fst} = \mathit{absent}$, 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 $t = (H, T, k, \mathit{present})$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \mathsf{frzn}\ k.\ell\ G\, at\, p$ iff $dom(m) = \emptyset$, $t.\mathit{fst} = \mathit{absent}$, $(\Pi_{old} \cup \Pi)(k.\ell) = G'$, 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 $t.\mathit{fst} = \mathit{absent}$, and $dom(m) = \{\ell - n | n \in \mathsf{Nat}\}$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \mathsf{more}^{\rightarrow}(\ell)\, at\, p$ iff $t.\mathit{fst} = \mathit{absent}$, and $dom(m) = \{\ell + n | n \in \mathsf{Nat}\}$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} \mathbf{1}\, at\, p$ iff $t.\mathit{fst} = \mathit{absent}$, 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 $t.\mathit{fst} = \mathit{absent}$, for all memory m', tree t' and $t'.\mathit{fst} = \mathit{absent}$, $\Pi_{old}; \Pi; m'; t' \vDash^{\Psi} F_1\, at\, p$ 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 satisfi es $\mathbf{0}$.
- $\Pi_{old}; \Pi; m; t \vDash^{\Psi} F_1 \oplus F_2\, at\, p$ 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 $t.\mathit{fst} = \mathit{absent}$, 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 2. The Semantics of Formulas

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:

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

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

Semantic Entailment $F_1 \vDash^{\Psi} F_2$ iff for all m and t, $(m, t) \vDash^{\Psi} F_1\, at\, p$ overall implies $(m, t) \vDash^{\Psi} F_2\, at\, p$ 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 $\exists xk{:}\mathsf{TG}.F_2 \otimes\, !(xk\ \mathsf{outlives}\ H)$.

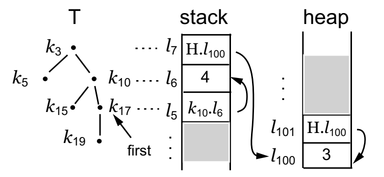

Figure 3. Example Memory

Example For this example, we consider just memory and not the registers. The memory m in Figure 3 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:

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

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

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

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

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: $\Pi_{old}; \Pi; \ell_7 \mapsto H.\ell_{100}; (H, T, k_{17}, \textit{absent}) \vDash^{\Psi} F_4\, at\, k_3.\ell_7$, which is true with ℓ_{100} as the witness of the existential.

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.

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:

$$\begin{array}{llll} \textit{Operands} & v & ::= & sv \mid r \\ \textit{Instructions} & \iota & ::= & \mathtt{add}\, r_d, r_s, v \mid \mathtt{sub}\, r_d, r_s, v \\ & & & \mid \mathtt{mov}\, r_d, v \mid \mathtt{bz}\, r, v \mid \mathtt{ld}\, r_d, r_s \\ & & & \mid \mathtt{st}\, r_d, r_s \mid \mathtt{stackgrow} \mid \mathtt{stackcut} \\ \textit{Blocks} & B & ::= & \mathtt{halt} \mid \mathtt{jmp}\, v \mid \iota; B \\ \textit{Code Regions} & C & ::= & \cdot \mid C, c \mapsto B \\ \textit{Machine States} & \Sigma & ::= & (C, m, t, B) \end{array}$$

The operational semantics for our language is standard and is similar to those in previous papers. The semantics of the two stack instructions is to apply the newFirst and delFirst operations to the tag tree.

4.2 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 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. Figure 4 shows some of the more interesting rules, which we describe in the following sections.[1] The complete rules are listed in the companion technical report [13].

The notation $(F\,[\,g := G\,])_{\Theta}$ 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.

Instruction Typing 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.

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

[1]For simplicity, we omit the code context Ψ from the typing judgments.

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

$$\frac{}{\Theta \parallel F \vdash^{\Psi} i : \mathbf{S}(i)}\ \textit{int} \qquad \frac{}{\Theta \parallel F \vdash^{\Psi} k.\ell : \mathbf{S}(k.\ell)}\ \textit{loc} \qquad \frac{\Psi(c) = (F') \to 0 \quad \cdot \parallel \cdot\,; u : F \vdash F'}{\Theta \parallel F \vdash^{\Psi} c : (F) \to 0}\ \textit{code} \qquad \frac{\Theta \parallel \cdot\,; F \vdash (r \Rightarrow \tau) \otimes \top}{\Theta \parallel F \vdash^{\Psi} r : \tau}\ \textit{reg}$$

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

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

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

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

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

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

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

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

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

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

Figure 4. Selected Static Semantics Rules

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 new 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 its 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 & State 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. 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.

Theorem 1 (Type Safety)
If $\vdash^{\Psi} (C, m, t, B)$ ok *then:*

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

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. The full details of translation are available in the companion technical report [13].

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

qualifi ers	q	$::=$	$S \mid H$
types	τ	$::=$	$\mathbf{int} \mid \tau *_q$
value	v	$::=$	$n \mid x$
program	p	$::=$	$fd \dots fd\, rb$
function decls	fd	$::=$	$\tau\, f(\tau\, x, \dots, \tau\, x)\, rb$
return block	rb	$::=$	$\{ld; \dots; ld; ss; \mathbf{return}\, v\}$
local decls	ld	$::=$	$\tau\, x\ =\ v \mid \tau\, x\ =\ \mathbf{new}_q\, v$
statement list	ss	$::=$	$\cdot \mid s;\ ss$
statement	s	$::=$	$\mathbf{if}\, v\, \mathbf{then}\, ss\, \mathbf{else}\, ss \mid x\ =\ v$
			$\mid x\ =\ v_1 + v_2 \mid x\ =\ v_1 - v_2$
			$\mid x\ =\ f(v, \dots, v) \mid x\ =\ !\, v$
			$\mid v_1\ :=\ v_2$

Figure 5. The syntax of Micro-CLI

of objects in the heap, but they cannot (in verifiable code) be returned from methods nor stored into fields of objects. These restrictions make sure that a managed pointer outlives its target thus prevent dereferencing dangling pointers. 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. A value of type $\tau *_S$ is a pointer to a location on the stack or in the heap; and it is restricted in similar ways to managed pointers. We also disallow updating a stack pointer if it points to another stack pointer since such update may create dangling pointers. A value of type $\tau *_H$ is a pointer only into the heap. Type $\tau *_H$ corresponds to references in CLI and is a subtype of $\tau *_S$. Figure 5 shows the syntax for this language. The declaration $\tau *_S x\ =\ \mathbf{new}_S\, v$ allocates a new cell on the stack with initial value v of type τ and binds the address of the cell to x; $\tau *_H x\ =\ \mathbf{new}_H\, v$ is similar but allocates on the heap.

The translation of programs to code regions and an initial block is straightforward. The interesting part is how Micro-CLI types are translated into our memory logic, and how the logic rules are used to verify the safety of the instructions. We discuss only the type translation as it contains the key invariants.

The type translation, shown in Figure 6, has the form $[\![\tau]\!]_\tau\, ek$ where τ is the type to translate and ek is the tag of the stack location where the value resides (in the translation, we put all local variables on the stack). The tag ek is used only in the translation of stack pointers to state the tag of the location pointed to is valid.

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 a predicate $!\,(\,xk\ \mathsf{outlives}\ ek_c)$ to state the validity of the tag, and also to specify the restriction that the contents of a stack pointer always "outlives" the stack pointer itself. 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 $\to 0$. 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, 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 is pushed onto the stack, the stack pointer is in r_{sp}, the return address is in r_{ra}, and the frontier pointer is in r_{alloc}.

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

Theorem 2 (Type Preservation of the Translation)
If $[\![p]\!]\ =\ (C, \Psi, B)$ then there exists an initial memory m and initial tag tree t such that $\vdash^{\Psi}\ (C, m, t, B)$ ok

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.

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

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

$$\frac{[\![\tau_1]\!]_\tau\, xk_{\mathrm{a1}} \;=\; G_1 \quad \ldots \quad [\![\tau_n]\!]_\tau\, xk_{\mathrm{an}} \;=\; G_n \quad [\![\tau]\!]_\tau\, xk_{\mathrm{ret}} \;=\; G_{\mathrm{ret}}}{\begin{array}{l} [\![\tau_1 * \ldots * \tau_n \to \tau]\!]_\tau \, \_ \;=\; \\ \quad (\exists\, x\ell{:}\mathsf{L}, x\ell'{:}\mathsf{L}, xk{:}\mathsf{TG}, xk_{\mathrm{a1}}{:}\mathsf{TG}, \ldots, xk_{\mathrm{an}}{:}\mathsf{TG}, \mathit{store}{:}\mathsf{F}. \\ \quad\ \mathsf{more}^{\leftarrow}(x\ell) \otimes (xk_{\mathrm{an}}.x\ell{+}1 \Rightarrow G_n) \otimes \ldots \otimes (xk_{\mathrm{a1}}.x\ell{+}n \Rightarrow G_1) \otimes \mathsf{more}^{\rightarrow}(x\ell') \otimes (r_{\mathrm{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_{\mathrm{fp}} \Rightarrow \mathbf{ns}) \otimes (r_{\mathrm{sp}} \Rightarrow \mathbf{S}(xk_{an}.x\ell{+}1)) \\ \quad\ \otimes (r_{\mathrm{ra}} \Rightarrow \quad (\exists\, x\ell''{:}\mathsf{L}, F_{\mathrm{H}}{:}\mathsf{F}, xk_{\mathrm{ret}}{:}\mathsf{TG}. \\ \qquad\qquad\quad \mathsf{more}^{\leftarrow}(x\ell{+}n{-}1) \otimes (xk_{\mathrm{ret}}.x\ell{+}n \Rightarrow F_{\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} \\ \qquad\qquad\quad \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{+}n)) \\ \qquad\qquad\quad \otimes\ \mathsf{first}(xk_{\mathrm{ret}}) \otimes !(xk{=}xk_{\mathrm{ret}}{+}1)\ ) \to 0\ ) \\ \quad\ \otimes\ \mathsf{first}(xk_{\mathrm{an}}) \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)\ ) \to 0 \end{array}}\ \text{(trans-fun)}$$

Figure 6. Type Translation

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 qualifi ers. 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] L. Jia, F. Spalding, D. Walker, and N. Glew. Certifying compilation for a language with stack allocation. Technical Report TR-724-05, Princeton University, March 2005.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.