# The Logical Approach to Stack Typing

Amal Ahmed David Walker

Department of Computer Science, Princeton University
35 Olden Street, Princeton, NJ 08544

{amal,dpw}@cs.princeton.edu

## ABSTRACT

We develop a bunched logic for reasoning about adjacency. We lay out a memory model for our logic and present a sound set of natural deduction style inference rules. We deploy the logic in a simple type system for a stack-based assembly language. The connectives for the logic provide a flexible yet concise mechanism for reasoning about stack allocation, deallocation and aliasing.

## 1. INTRODUCTION

The study of type-directed certifying compilation is now approximately 8 years old and during this nascent period we have learned, among other things, the following two facts.

- Safety is not necessarily in conflict with performance. In principle, we have the theoretical techniques to equip programs with proofs that can justify just about any code optimization or compilation strategy.

- Easier said than done. Engineering a high-performance certifying compiler for a sophisticated language such as Java or ML is a time-consuming process, particularly if we attempt to design a target language that is somewhat independent of our compilation strategy and source language.

To continue to advance the state-of-the-art in this field, we need to develop convenient abstractions to help us structure proofs of common properties. These abstractions should be flexible yet concise and relatively easy for the compiler implementor to understand and manipulate. The most successful abstractions will make it easier to build certifying compilers. They may also find their way from low-level compiler intermediate languages into general-purpose programming languages. In fact, we have already seen such migration: Grossman et al.'s Cyclone [6] uses Tofte and Talpin's [26] region abstractions; DeLine and Fahndrich's Vault [4] incorporates Walker et al.'s capabilities [27]; and O'Callahan [13] improves the JVML type system using mechanisms from Morrisett et al.'s STAL [9].

In this paper, we focus on developing relatively flexible abstractions for reasoning about stack-allocated data. A stack is used to store activation records in almost every modern compiler and stack allocation of objects and records is an important optimization for all sorts of languages. Explicit stack data structures also appear in byte code languages such as the Java virtual machine and Common Language Runtime. Recently, projects such as Cyclone have investigated techniques for safe stack allocation at the source programming-language level.

Our main contribution is a logic for reasoning about *adjacency*. The logic was inspired by both the work of Polakow and Pfenning on ordered linear logic [20, 21, 19] and also the work of Ishtiaq, O'Hearn, Pym and Reynolds [14, 23, 7] on bunched logic and applications, but the details differ somewhat from each. We use the logic to reason about stack allocation in a simple typed assembly language.

Our presentation of typed assembly language is itself new. We encode the state of our abstract machine entirely using our new substructural logic and consequently, the language has the feel of Hoare logic. However, we also wrap logical formulae up in higher-order types, which provides us with a simple way to extend traditional first-order Hoare logic to the higher-order domain of type theory.

In the next section, we introduce our adjacency logic (AL). We discuss the meaning of judgments and formulae in terms of a concrete memory model. We also provide the logical inference rules and prove a few simple metatheoretic properties. In section 3, we introduce our typed assembly language and show how to use our logic to reason about the stack. We also prove standard progress and preservation lemmas to show that our type system is sound. In section 4, we discuss related work in greater detail and, in section 5, we suggest some directions for future research.

## 2. AL: AN ADJACENCY LOGIC

In this section, we describe AL, a logic for reasoning about adjacency. We can always reason about adjacency by equating locations with integers and reasoning using arithmetic. However, AL connectives incorporate information about adjacency directly. This leads to simpler specifications since auxiliary arithmetic equations can be omitted from formulae. Before introducing the logic itself, we will introduce the values and memories that will serve as a model for the logic.

### 2.1 Values

We will be reasoning about several different sorts of values including integers, code locations, which contain executable code, proper memory locations (aka heap locations) and a fixed, finite set of registers.

$$\begin{array}{lcll} \textit{Integers} & i & \in & \textit{Int} \\ \textit{Proper Locations} & \ell & \in & \textit{Loc} \\ \textit{Registers} & r & \in & \textit{Reg} \\ \textit{Code Locations} & c & \in & \textit{Codeloc} \end{array}$$

In order to discuss adjacent locations, we take the further step of organizing the set *Loc* in a total order given by the relation $\leq$. We also assume a total function $\mathsf{succ} : \textit{Loc} \to \textit{Loc}$, which maps a location $\ell$ to the location that immediately follows it. We write $\mathsf{adj}(\ell, \ell')$ when $\ell' = \mathsf{succ}(\ell)$. We write $\ell + i$ as syntactic sugar for $\mathsf{succ}^i(\ell)$ and $\ell - i$ for the location $\ell'$ such that $\ell = \mathsf{succ}^i(\ell')$.

**Types.** Our subsequent semantics for formulae will incorporate a simple typing judgment for values. We give integers and locations singleton types which identify them exactly. Code locations will be given code types as described by a code context. These code types have the form $(F) \to 0$, where $F$ is a logical formula that describes requirements that must be satisfied by the state of the abstract machine before it is safe to jump to the code. Code can only "return" by explicitly jumping to a continuation (a return address) that it has been passed as an argument and therefore our code types do not have proper return types. Abstract types $\alpha$ arise through existential or universal quantification in formulae.

$$\begin{array}{lcll} \textit{Types} & \tau & ::= & \alpha \mid \mathbf{S}(i) \mid \mathbf{S}(\ell) \mid (F) \to 0 \\ \textit{Code Contexts} & \Psi & ::= & \cdot \mid \Psi, c\!:\!(F) \to 0 \end{array}$$

Store values are given types via a judgment of the form $\vdash^{\Psi} h : \tau$. Since the code context never changes in a typing derivation, we normally omit it from the judgment form as in the rules below.

$$\frac{}{\vdash i : \mathbf{S}(i)}\ (\mathsf{int}) \qquad \frac{}{\vdash \ell : \mathbf{S}(\ell)}\ (\mathsf{loc})$$

$$\frac{\Psi(c) = (F) \to 0}{\vdash c : (F) \to 0}\ (\mathsf{code})$$

## 2.2 Memories

A memory is a partial map from generic locations $g$ to store values $h$.

$$\begin{array}{lcll} \textit{Generic Loc's} & g & \in & (\textit{Loc} \cup \textit{Reg}) \\ \textit{Store Values} & h & \in & \textit{Sval} = (\textit{Int} \cup \textit{Loc} \cup \textit{Codeloc}) \\ \textit{Memories} & m & \in & (\textit{Loc} \cup \textit{Reg}) \rightharpoonup \textit{Sval} \end{array}$$

Registers are special (second-class) locations that may appear in the domain of a memory but may not appear stored in other locations, and hence, are not included in the set of store values.

We use the following notation to manipulate memory.

- $\textit{dom}(m)$ denotes the domain of the memory $m$
- $m(g)$ denotes the store value at location $g$
- $m\,[g := h]$ denotes a memory $m'$ in which $g$ maps to $h$ but is otherwise the same as $m$. If $g \notin \textit{dom}(m)$ then the update is undefined.
- Given a set of locations $X \subseteq \textit{Loc}$, $\ulcorner X \urcorner$ is the greatest member (the *supremum*) of the set and $\llcorner X \lrcorner$ is the least member (the *infimum*) of the set according to the total order $\leq$. We also let $\ulcorner X \cup R \urcorner = \ulcorner X \urcorner$ and $\llcorner X \cup R \lrcorner = \llcorner X \lrcorner$ when $R \subseteq \textit{Reg}$. $\ulcorner \emptyset \urcorner$ and $\llcorner \emptyset \lrcorner$ (and hence $\ulcorner R \urcorner$ and $\llcorner R \lrcorner$) are undefined.
- $m_1 \# m_2$ indicates that the memories $m_1$ and $m_2$ have disjoint domains
- $m_1 \uplus m_2$ denotes the union of disjoint memories; if the domains of the two memories are not disjoint then this operation is undefined.
- $m_1 @ m_2$ denotes the union of disjoint memories with the additional caveat that either $\mathsf{adj}(\ulcorner \textit{dom}(m_1) \urcorner, \llcorner \textit{dom}(m_2) \lrcorner)$ or one of $m_1$ or $m_2$ is empty.

$$\begin{array}{lcll} \textit{Predicates} & p & ::= & (g\!:\!\tau) \mid \mathsf{more}^{\leftarrow} \mid \mathsf{more}^{\rightarrow} \\ \textit{Formulae} & F & ::= & p \mid \mathbf{1} \mid F_1 \otimes F_2 \mid F_1 \circ F_2 \mid \\ & & & F_1 \multimap F_2 \mid F_1 \rightarrowtail F_2 \mid F_1 \twoheadrightarrow F_2 \mid \\ & & & \top \mid F_1 \,\&\, F_2 \mid \mathbf{0} \mid F_1 \oplus F_2 \mid \\ & & & \phi \mid \forall b.F \mid \exists b.F \\ \textit{Bindings} & b & ::= & i\!:\!\mathsf{I} \mid \ell\!:\!\mathsf{L} \mid \alpha\!:\!\mathsf{T} \mid \phi\!:\!\mathsf{F} \end{array}$$

**Figure 1: AL Formulae**

## 2.3 Formulae

Figure 1 presents the syntax of formulae. We use a notation reminiscent of connectives from linear logic [5] and Polakow and Pfenning's ordered logic [20, 21, 19]. However, these logics should only be used as an approximate guide to the meaning of formulae. There are some differences as we will see below.

In addition to multiplicative (linear and ordered) connectives, the logic contains additive connectives and quantification. The bindings in quantification formulae describe the sort, (integer $\mathsf{I}$, proper location $\mathsf{L}$, type $\mathsf{T}$, or formula $\mathsf{F}$), of the bound variable. We reuse the metavariable $\ell$ (for concrete locations) as a location variable below. There is, however, a reason to distinguish between them: if $\ell$ is a concrete location, we can compute $\ell + i$, which is not the case if $\ell$ is a location variable. From now on, the reader may assume that occurrences of $\ell$ denote a variable, unless otherwise noted in the text. We use abbreviations for the following common formulae.

- $\exists i{:}\mathsf{I}.\,(g\!:\!\mathbf{S}(i))$ is abbreviated $g\!:\!\mathbf{int}$
- $\exists \alpha{:}\mathsf{T}.\,(g\!:\!\alpha)$ is abbreviated $g\!:\!\_$
- $\exists \ell{:}\mathsf{L}.\,\exists \alpha{:}\mathsf{T}.\,(\ell\!:\!\alpha)$ is abbreviated $\mathsf{ns}$
- We use metavariable $R$ to range over formulae with shape $r_1\!:\!\tau_1 \otimes \cdots \otimes r_n\!:\!\tau_n$

Our logic contains quite a number of formulae, but one need not be intimidated. Each formula is defined orthogonally from all others and can be understood independently. Moreover, the logic makes sense if we choose to use any subset of the formulae. Consequently, a system designer need not understand or implement all of the connectives but can choose the subset that suits their application.

**Semantics.** We use formulae to describe memories and write $m \vDash^{\Psi} F$ when the formula $F$ describes the memory $m$. The reader can safely ignore the superscript $\Psi$ for now. The most basic formulae are predicates with the form $(g\,{:}\,\tau)$. These predicates describe memories that contain a single location $g$ that holds a value with type $\tau$. There are two other basic predicates $\mathsf{more}^{\leftarrow}$ and $\mathsf{more}^{\rightarrow}$. They describe an infinite sequence of locations that increases to the left (right). Later, we will use $\mathsf{more}^{\leftarrow}$ to indicate that the stack may be grown to the left. Analogously, $\mathsf{more}^{\rightarrow}$ may be used to indicate that some region of memory may be grown to the right, although we do not use it in this paper.

The key to reasoning about adjacent memories is a form of ordered conjunction, which we will call *fuse*. Semantically, $m \vDash^{\Psi} F_1 \circ F_2$ if and only if $m$ can be divided into two *adjacent* parts, $m_1$ and $m_2$ such that $m_1 \vDash^{\Psi} F_1$ and $m_2 \vDash^{\Psi} F_2$. More formally, we require $m = m_1 @ m_2$.

To get accustomed to some of the properties of fuse, we will reason about the following memories, which contain locations in the set $\{\ell_i \mid 0 \leq i \leq n\}$ where each location in this set is adjacent to the next in sequence (i.e., for all $i$, $\mathsf{adj}(\ell_i, \ell_{i+1})$).

| *Memory* | *Domain* | *Describing Formula* |
|---|---|---|
| $m_1$ | $\{\ell_1, \ell_3\}$ | $F_1$ |
| $m_2$ | $\{\ell_4, \ell_6\}$ | $F_2$ |
| $m_3$ | $\{\ell_2\}$ | $F_3$ |
| $m_4$ | $\emptyset$ | $F_4$ |

First, notice that $m_1 \cup m_2$ may be described using the formula $F_1 \circ F_2$ since the supremum of $m_1$ is adjacent to the infimum of $m_2$. This same memory cannot be described by $F_2 \circ F_1$ — fuse is not generally commutative. On the other hand, $m_1$ can be described by either $F_1 \circ F_4$ or $F_4 \circ F_1$ since $m_1 = m_1 @ \emptyset = \emptyset @ m_1$. Since neither the supremum nor the infimum of $m_3$ is adjacent to the infimum or supremum, respectively, of $m_1$ or $m_2$ we cannot readily use fuse to describe the relationship between these memories.

When we don't know or don't care about the ordering relationship between two disjoint memories we will use the unordered multiplicative conjunction $F_1 \otimes F_2$ (which we call tensor). A memory $m$ can be described by $F_1 \otimes F_2$ if and only if there exist $m_1, m_2$, such that $m_1 \vDash^{\Psi} F_1$ and $m_2 \vDash^{\Psi} F_2$ and $m = m_1 \uplus m_2$. This definition differs from the definition for $\circ$ only in that the disjoint union operator "$\uplus$" makes none of the ordering requirements of the adjacent disjoint union operator "$@$". To see the impact of the change, we consider the following further memories.

| *Memory* | *Domain* | *Describing Formula* |
|---|---|---|
| $m_1$ | $\{\ell_1, \ell_2\}$ | $F_1$ |
| $m_2$ | $\{\ell_3, \ell_4\}$ | $F_2$ |
| $m_3$ | $\{\ell_5\}$ | $F_3$ |
| $m_4$ | $\emptyset$ | $F_6$ |

The memory $m = m_1 \cup m_2 \cup m_3$ can be described by the formula $(F_1 \otimes F_2) \otimes F_3$ since $m$ can be broken into two disjoint parts, $m_1 \cup m_2$ and $m_3$, which satisfy the subformulae $(F_1 \otimes F_2)$ and $F_3$ respectively. The memory $m$ also satisfies the formulae $F_1 \otimes (F_2 \otimes F_3)$ and $(F_3 \otimes F_2) \otimes F_1$ since it is defined in terms of the associative and commutative disjoint union operator.

Our logic contains one more sort of conjunction, the additive $F_1 \& F_2$. A memory $m$ can be described by this formula if the memory (the whole thing!) can be described by both subformulae. Meanwhile, the additive disjunction of two formulae, $F_1 \oplus F_2$, describes a memory $m$ if the entire memory can be described either one of $F_1$ or $F_2$.

The multiplicative $\mathbf{1}$ describes the empty memory and serves as the (right and left) unit for both fuse and tensor. In other words, if $m \vDash^{\Psi} F$ then both $m \vDash^{\Psi} F \circ \mathbf{1}$ and $m \vDash^{\Psi} F \otimes \mathbf{1}$. The unit for additive conjunction $\top$ describes any memory, while the unit for additive disjunction $\mathbf{0}$ describes no memories. These properties can easily be verified from the semantic definitions of the connectives.

The semantics of the other connectives are largely standard. In the semantics of quantifiers, we use the notation $X[a/b]$ to denote capture-avoiding substitution of $a$ for the variable in $b$ in the object $X$. We extend this notation to substitution for a sequence of bindings as in $X[a_1, \ldots, a_n/\vec{b}]$ or $X[\vec{b}'/\vec{b}]$. In either case, the objects substituted for variables must have the correct sort (type, formula, location or integer) and the sequences must have the same length or else the substitution is undefined. The semantics of all formulae are collected in figure Figure 2. We include the semantics of linear and ordered implications ($\multimap$, $\rightarrowtail$, $\twoheadrightarrow$) but, in the interest of space, we do not describe them here. The semantics of these connectives follow the work of Ishtiaq and O'Hearn [7].

**An Extended Example.** Recall that it is safe to jump to a code location of type $(F) \rightarrow 0$ if the requirements described by $F$ are satisfied by the current state of the abstract machine. More specifically, we require that the current memory $m$ be described by $F$, that is, $m \vDash^{\Psi} F$.

Consider the type of code location $c$ which, among other things, requires that registers $r_1$ and $r_2$ point to locations in memory and that $r_3$ contain the address of the continuation:

$$\begin{aligned} c \;:\; (\exists\, \ell{:}\mathsf{L},\ \ell'{:}\mathsf{L},\ \mathit{mem}{:}\mathsf{F}.\ & (((\ell\,{:}\,\mathbf{int}) \otimes \top) \,\&\, ((\ell'\,{:}\,\mathbf{int}) \otimes \top) \,\&\, \mathit{mem}) \\ & \otimes (r_1\,{:}\,\mathbf{S}(\ell)) \\ & \otimes (r_2\,{:}\,\mathbf{S}(\ell')) \\ & \otimes (r_3\,{:}\,\tau_{cont})) \rightarrow 0 \end{aligned}$$

$$\text{and} \quad \tau_{cont} = (\mathit{mem} \otimes (r_1\,{:}\,\_) \otimes (r_2\,{:}\,\_) \otimes (r_3\,{:}\,\_)) \rightarrow 0$$

There are a number of things to note about the calling convention described by the above type. First, the semantics of $\&$ dictate that to jump to $c$ the caller must pack the variable $\mathit{mem}$ with a formula that describes all the proper (non-register) locations in memory. Second, since the formula describing the proper locations is abstracted using the variable $\mathit{mem}$, the code at $c$ can only jump to the continuation when the register-free subset of memory is in the same state as it was upon jumping to $c$. Finally, a caller may jump to code location $c$ if there exists some location $\ell$ that register $r_1$ points to, and some location $\ell'$ that register $r_2$ points to. The locations $\ell$ and $\ell'$ may or may not be the same location, which we illustrate next by considering two different scenarios.

**Scenario 1** Consider the following memory where $\ell_i$ are concrete locations such that $\mathsf{adj}(\ell_1, \ell_2)$ and $\mathsf{adj}(\ell_2, \ell_3)$:

$$\begin{aligned} & m_1 = \{\ell_1 \mapsto \mathsf{5},\ \ell_2 \mapsto \mathsf{3},\ \ell_3 \mapsto \mathsf{9},\ r_1 \mapsto \ell_2,\ r_2 \mapsto \ell_2,\ r_3 \mapsto c'\} \\ & \text{where } c' : (\,((\ell_1\,{:}\,\mathbf{int}) \circ (\ell_2\,{:}\,\mathbf{int}) \circ (\ell_3\,{:}\,\mathbf{int})) \\ & \qquad\qquad \otimes\ (r_1\,{:}\,\_) \otimes (r_2\,{:}\,\_) \otimes (r_3\,{:}\,\_)\,) \rightarrow 0 \end{aligned}$$

The three consecutive proper locations in $m_1$ may be thought of as a stack. The memory also consists of three registers,

$m \vDash^\Psi F$ if and only if

- $F = (g : \tau)$ and $dom(m) = \{g\}$ and $m(g) = h$ and $\vdash^\Psi h : \tau$
- $F = \mathsf{more}^{\leftarrow}$ and $dom(m) = \{g \mid \exists g'. g \leq g'\}$
- $F = \mathsf{more}^{\rightarrow}$ and $dom(m) = \{g \mid \exists g'. g \geq g'\}$
- $F = \mathbf{1}$ and $dom(m) = \emptyset$
- $F = F_1 \otimes F_2$ and there exist $m_1, m_2$, such that $m = m_1 \uplus m_2$ and $m_1 \vDash^\Psi F_1$ and $m_2 \vDash^\Psi F_2$
- $F = F_1 \circ F_2$ and there exist $m_1, m_2$, such that $m = m_1 @ m_2$ and $m_1 \vDash^\Psi F_1$ and $m_2 \vDash^\Psi F_2$
- $F = F_1 \multimap F_2$ and for all memories $m_1$ such that $m_1 \vDash^\Psi F_1$, $m_1 \uplus m \vDash^\Psi F_2$
- $F = F_1 \rightarrowtail F_2$ and for all memories $m_1$ such that $m_1 \vDash^\Psi F_1$, $m_1 @ m \vDash^\Psi F_2$
- $F = F_1 \twoheadrightarrow F_2$ and for all memories $m_1$ such that $m_1 \vDash^\Psi F_1$, $m @ m_1 \vDash^\Psi F_2$
- $F = \top$ (and no other conditions need be satisfied)
- $F = F_1 \& F_2$ and $m \vDash^\Psi F_1$ and $m \vDash^\Psi F_2$
- $F = \mathbf{0}$ and false (this formula can never be satisfied)
- $F = F_1 \oplus F_2$ and either
  1. $m \vDash^\Psi F_1$, or
  2. $m \vDash^\Psi F_2$.
- $F = \forall x{:}K.F'$ and $m \vDash^\Psi F'[a/x]$ for all $a \in K$
- $F = \exists x{:}K.F'$ and there exists some $a \in K$ such that $m \vDash^\Psi F'[a/x]$

**Figure 2: Semantics of Formulae**

two of which ($r_1$ and $r_2$) contain aliases of location $\ell_2$ in the stack. Register $r_3$ contains a code location $c'$. We can conclude that we may safely jump to code location $c$ when $m_1$ is the current memory as follows.

- Since the formula $(\ell_2 : \mathbf{int})$ describes $\{\ell_2 \mapsto 3\}$ and $\top$ describes $\{\ell_1 \mapsto 5\} \uplus \{\ell_3 \mapsto 9\}$, the formula $((\ell_2 : \mathbf{int}) \otimes \top)$ describes the memory $\{\ell_2 \mapsto 3\} \uplus \{\ell_1 \mapsto 5\} \uplus \{\ell_3 \mapsto 9\}$.
- Similarly, we can conclude that the formula $((\ell_2 : \mathbf{int}) \otimes \top)$ describes $\{\ell_1 \mapsto 5,\ \ell_2 \mapsto 3,\ \ell_3 \mapsto 9\}$.
- The formula $(\ell_1 : \mathbf{int}) \circ (\ell_2 : \mathbf{int}) \circ (\ell_3 : \mathbf{int})$ describes $\{\ell_1 \mapsto 5,\ \ell_2 \mapsto 3,\ \ell_3 \mapsto 9\}$ since $\ell_1$, $\ell_2$ and $\ell_3$ are consecutive locations.
- $(r_1 : \mathbf{S}(\ell_2))$ describes $\{r_1 \mapsto \ell_2\}$.
- $(r_2 : \mathbf{S}(\ell_2))$ describes $\{r_2 \mapsto \ell_2\}$.
- $(r_3 : \tau_{cont}[((\ell_1 : \mathbf{int}) \circ (\ell_2 : \mathbf{int}) \circ (\ell_3 : \mathbf{int}))/mem])$ describes $\{r_3 \mapsto c'\}$.
- Then, the formula

$$\begin{array}{l} (((\ell_2 : \mathbf{int}) \otimes \top) \& ((\ell_2 : \mathbf{int}) \otimes \top) \\ \quad \& ((\ell_1 : \mathbf{int}) \circ (\ell_2 : \mathbf{int}) \circ (\ell_3 : \mathbf{int}))) \\ \otimes (r_1 : \mathbf{S}(\ell_2)) \\ \otimes (r_2 : \mathbf{S}(\ell_2)) \\ \otimes (r_3 : \tau_{cont})) \to 0 \end{array}$$

  describes

$$\{\ell_1 \mapsto 5,\ \ell_2 \mapsto 3,\ \ell_3 \mapsto 9\} \uplus \{r_1 \mapsto \ell_2\} \uplus \{r_2 \mapsto \ell_2\} = m_1.$$

- Using existential introduction, where location $\ell_2$ serves as a witness for both $\ell$ and $\ell'$, and $((\ell_1 : \mathbf{int}) \circ (\ell_2 : \mathbf{int}) \circ (\ell_3 : \mathbf{int}))$ serves as a witness for $mem$, we can conclude the following.

$$\begin{array}{ll} m_1 \vDash^\Psi & \exists \ell{:}\mathsf{L},\ \ell'{:}\mathsf{L},\ mem{:}\mathsf{F}. \\ & (((\ell : \mathbf{int}) \otimes \top) \& ((\ell' : \mathbf{int}) \otimes \top) \& mem) \\ & \otimes (r_1 : \mathbf{S}(\ell)) \\ & \otimes (r_2 : \mathbf{S}(\ell')) \\ & \otimes (r_3 : \tau_{cont}) \end{array}$$

Consequently, if $m_1$ is our memory, we can jumped to $c$.

**Scenario 2.** Consider the memory $m_2$ which is identical to $m_1$ except that location $\ell_2$ is no longer aliased by registers $r_1$ and $r_2$:

$$m_2 = \{\ell_1 \mapsto 5,\ \ell_2 \mapsto 3,\ \ell_3 \mapsto 9,\ r_1 \mapsto \ell_2,\ r_2 \mapsto \ell_3,\ r_3 \mapsto c'\}$$

Using reasoning similar to the above, we can conclude that the formula

$$\begin{array}{l} (((\ell_2 : \mathbf{int}) \otimes \top) \& ((\ell_3 : \mathbf{int}) \otimes \top) \\ \quad \& ((\ell_1 : \mathbf{int}) \circ (\ell_2 : \mathbf{int}) \circ (\ell_3 : \mathbf{int}))) \\ \otimes (r_1 : \mathbf{S}(\ell_2)) \\ \otimes (r_2 : \mathbf{S}(\ell_3)) \\ \otimes (r_3 : \tau_{cont})) \to 0 \end{array}$$

describes

$$\{\ell_1 \mapsto 5,\ \ell_2 \mapsto 3,\ \ell_3 \mapsto 9\} \uplus \{r_1 \mapsto \ell_2\} \uplus \{r_2 \mapsto \ell_3\} = m_2.$$

Then, to conclude that it is safe to jump to $c$ when the current memory is $m_2$ we use existential introduction with locations $\ell_2$ and $\ell_3$ as witnesses for $\ell$ and $\ell'$ respectively.

It should be noted that the type ascribed to $c$ is somewhat simplistic. Normally we would like to abstract the formula that describes the stack using the variable $mem$, but the above type prohibits the code at $c$ from allocating additional space on the stack. We rectify the problem by using $\mathsf{more}^{\leftarrow}$ to describe the part of memory where additional stack cells may be allocated, and by requiring a register $r_{sp}$ that points to the top of the stack as follows.

$$\begin{array}{ll} c : & (\exists \ell_{hd}{:}\mathsf{L},\ \alpha{:}\mathsf{T},\ \ell{:}\mathsf{L},\ \ell'{:}\mathsf{L},\ tail{:}\mathsf{F}. \\ & \quad (\mathsf{more}^{\leftarrow} \circ (\ell_{hd} : \alpha) \\ & \qquad \circ (((\ell : \mathbf{int}) \otimes \top) \& ((\ell' : \mathbf{int}) \otimes \top) \& tail) \\ & \quad \otimes (r_1 : \mathbf{S}(\ell)) \\ & \quad \otimes (r_2 : \mathbf{S}(\ell')) \\ & \quad \otimes (r_3 : \tau_{cont}) \\ & \quad \otimes (r_{sp} : \mathbf{S}(\ell_{hd}))) \to 0 \end{array}$$

$$\text{and} \quad \tau_{cont} = ((\ell_{hd} : \alpha) \circ tail) \begin{array}{l} \otimes (r_1 : \_) \\ \otimes (r_2 : \_) \\ \otimes (r_3 : \_)) \to 0 \end{array}$$

## 2.4 Contexts

Following O'Hearn and Pym [14], our logical contexts are bunches (trees) rather than simple lists. The nodes in our bunches are labeled either with an ordered separator ";" or an (unordered) linear separator ",". The leaves of our bunches are either empty or a single formula labeled with a variable $u$. We write our contexts as follows.

$$\Delta \quad ::= \quad \cdot \mid u{:}F \mid \Delta_1, \Delta_2 \mid \Delta_1; \Delta_2$$

We will also frequently have reason to work with a context containing a single hole that may be filled by another context. We use the metavariable $\Gamma$ to range over contexts with a hole and write $\Gamma(\Delta)$ to fill the hole in $\Gamma$ with $\Delta$.

$$\Gamma \quad ::= \quad (\,) \mid \Gamma, \Delta \mid \Delta, \Gamma \mid \Gamma; \Delta \mid \Delta; \Gamma$$

We require that no variables are repeated in a context and consider $\Gamma(\Delta)$ to be undefined if $\Gamma$ and $\Delta$ have any variables in common. Again following O'Hearn and Pym, we define an equivalence relation on contexts. It is the reflexive, symmetric and transitive closure of the following axioms.

1. $\cdot, \Delta \equiv \Delta$
2. $\cdot; \Delta \equiv \Delta$
3. $\Delta; \cdot \equiv \Delta$
4. $(\Delta_1, \Delta_2), \Delta_3 \equiv \Delta_1, (\Delta_2, \Delta_3)$
5. $(\Delta_1; \Delta_2); \Delta_3 \equiv \Delta_1; (\Delta_2; \Delta_3)$
6. $\Delta_1, \Delta_2 \equiv \Delta_2, \Delta_1$
7. $\Gamma(\Delta) \equiv \Gamma(\Delta')$ if $\Delta \equiv \Delta'$

**Semantics.** Like individual formulae, contexts can describe memories. The semantics of contexts appears in Figure 3. Notice that the semantics of the ordered separator ";" mirrors the semantics of fuse whereas the semantics of the linear separator "," mirrors the semantics of tensor.

Our proofs require that we also give semantics to contexts with a hole (also in Figure 3). This semantic judgment has the form $(m, L) \vDash^{\Psi}_{C} \Gamma$ which may be read as saying that the memory $m$ is described by $\Gamma$ when the hole in $\Gamma$ is filled in by a context $\Delta$ that describes the memory defined on the locations in the set $L$.

## 2.5 Basic Semantic Properties

In this section, we outline some simple, but necessary properties of our semantics.

We will need to reason about the decomposition of a memory $m$ described by the context $\Gamma(\Delta)$ into two parts, the part described by $\Gamma$ and the part described by $\Delta$. The following lemma allows us to decompose and then recompose memories.

**Lemma 1 (Semantic Decomposition)**

- *If $m \vDash^{\Psi}_{C} \Gamma(\Delta)$ then there exist $m_1$ and $m_2$, such that $m = m_1 \uplus m_2$ and $(m_1, dom(m_2)) \vDash^{\Psi}_{C} \Gamma$ and $m_2 \vDash^{\Psi}_{C} \Delta$.*
- *If $(m_1, dom(m_2)) \vDash^{\Psi}_{C} \Gamma$ and $m_2 \vDash^{\Psi}_{C} \Delta$, then $m_1 \uplus m_2 \vDash^{\Psi}_{C} \Gamma(\Delta)$.*

Proof. By induction on the structure of $\Gamma$. □

$m \vDash^{\Psi}_{C} \Delta$ if and only if

- $\Delta = \cdot$ and $dom(m) = \emptyset$
- $\Delta = u{:}F$ and $m \vDash^{\Psi} F$
- $\Delta = \Delta_1, \Delta_2$ and $m = m_1 \uplus m_2$ and $m_1 \vDash^{\Psi}_{C} \Delta_1$ and $m_2 \vDash^{\Psi}_{C} \Delta_2$
- $\Delta = \Delta_1; \Delta_2$ and $m = m_1 @ m_2$ and $m_1 \vDash^{\Psi}_{C} \Delta_1$ and $m_2 \vDash^{\Psi}_{C} \Delta_2$

$(m, L) \vDash^{\Psi}_{C} \Gamma$ if and only if

- $\Gamma = (\,)$ and $dom(m) = \emptyset$
- $\Gamma = \Gamma', \Delta'$ and $(m_1, L) \vDash^{\Psi}_{C} \Gamma'$ and $m_2 \vDash^{\Psi}_{C} \Delta'$ and $m = m_1 \uplus m_2$
- $\Gamma = \Gamma'; \Delta'$ and $(m_1, L) \vDash^{\Psi}_{C} \Gamma'$ and $m_2 \vDash^{\Psi}_{C} \Delta'$ and $m = m_1 \uplus m_2$ and one of $\mathsf{adj}(\ulcorner dom(m_1) \uplus L \urcorner, \llcorner dom(m_2) \lrcorner)$, or $(dom(m_1) \uplus L) = \emptyset$, or $dom(m_2) = \emptyset$.
- $\Gamma = \Delta'; \Gamma'$ and $m_1 \vDash^{\Psi}_{C} \Delta'$ and $(m_2, L) \vDash^{\Psi}_{C} \Gamma'$ and $m = m_1 \uplus m_2$ and one of $\mathsf{adj}(\ulcorner dom(m_1) \urcorner, \llcorner dom(m_2) \uplus L \lrcorner)$ or $dom(m_1) = \emptyset$, or $(dom(m_2) \uplus L) = \emptyset$.

**Figure 3: Semantics of Contexts**

Our semantics does not distinguish between equivalent contexts.

**Lemma 2 (Soundness of Context Equivalence)**

- *If $m \vDash^{\Psi}_{C} \Delta$ and $\Delta \equiv \Delta'$ then $m \vDash^{\Psi}_{C} \Delta'$.*
- *If $(m, L) \vDash^{\Psi}_{C} \Gamma$ and $\Gamma \equiv \Gamma'$ then $(m, L) \vDash^{\Psi}_{C} \Gamma'$.*

Proof. By induction on the definition of context equivalence. In the case for equivalence rule 7, we use Lemma 1. □

Finally, since the append operator "@" makes more requirements of a context than the disjoint union operator "$\uplus$" we may easily prove that whenever a memory can be described by the context $\Delta_1; \Delta_2$ it can also be described by the context $\Delta_1, \Delta_2$. This notion is formalized by the following lemma.

**Lemma 3 (Semantic Disorder)**
*If $m \vDash^{\Psi}_{C} \Gamma(\Delta_1; \Delta_2)$ then $m \vDash^{\Psi}_{C} \Gamma(\Delta_1, \Delta_2)$.*

Proof. By induction on the structure of $\Gamma$. □

## 2.6 Logical Deduction

The basic judgment for the natural deduction formulation of our logic has the form $\Theta \parallel \Delta \vdash F$ where $\Theta$ is the set of variables that may appear free in $\Delta$ or $F$. These variables may be integer, location, type, or formula variables.

$$\textit{Variable Contexts} \quad \Theta \quad ::= \quad \cdot \mid \Theta, i{:}\mathsf{I} \mid \Theta, \ell{:}\mathsf{L} \mid \Theta, \alpha{:}\mathsf{T} \mid \Theta, \phi{:}\mathsf{F}$$

The inference rules (see Figures 4, 5) are very similar to the rules given by O'Hearn and Pym [14] so we only highlight the central differences. The most important difference, of course, is the presence of our ordered separator and linear separator as opposed to the linear separator and additive separator that is the focus of most of O'Hearn and Pym's work. Moreover, there is only a single unit for the two contexts rather than two units as in O'Hearn and Pym's work. Finally, since we have no additive separator in the context, our elimination form for the additive conjunction is slightly different from the elimination form given by O'Hearn and Pym. It seems likely that the additives suggested by O'Hearn and Pym are compatible with this system, but we have not investigated this possibility.

We do not have an explicit structural rule to mark the movement between equivalent contexts. Instead, we treat equivalence implicitly: the logical judgments and inference rules operate over equivalence classes of contexts. For example, when we write

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

we implicitly assume the presence of equivalence as in the following more explicit rule.

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

The rules of Figures 4,5 define a sound logical system. However, for our application, we would like one further property: It should be possible to forget adjacency information. We saw in the previous section (Lemma 3) that, if a memory satisfies a context that imposes ordering conditions via ";" then the same memory will satisfy a context that does not impose these ordering conditions (in other words, it is sound for "," to replace ";"). In order to include this principle in our deductive system, we allow any proof of a judgment with the form $\Theta \parallel \Gamma(\Delta_1, \Delta_2) \vdash F$ to be considered a proof of $\Theta \parallel \Gamma(\Delta_1; \Delta_2) \vdash F$. To mark the inclusion of one proof for another in the premise of a rule, we put a asterisk beside the name of that rule, as in the following derivation.

$$\frac{\Theta \parallel u_1 : F_1 \vdash F_1 \quad \dfrac{\Theta \parallel u_3 : F_3 \vdash F_3 \quad \Theta \parallel u_2 : F_2 \vdash F_2}{\Theta \parallel u_2 : F_2, u_3 : F_3 \vdash F_3 \otimes F_2}\ \otimes I}{\Theta \parallel u_1 : F_1; u_2 : F_2; u_3 : F_3 \vdash F_1 \circ (F_3 \otimes F_2)}\ \circ I*$$

These inclusions give rise the following principle.

**Principle 4 (Logical Disorder)**
*If $\Theta \parallel \Gamma(\Delta_1, \Delta_2) \vdash F$ then $\Theta \parallel \Gamma(\Delta_1; \Delta_2) \vdash F$.*

We borrow the idea of including one sort of proof for another from Pfenning and Davies' judgmental reconstruction of modal logic [18]. In that work, they include proofs of truth directly as proofs of possibility. An alternative to this approach would be to add an explicit structural rule, but we prefer to avoid structural rules as every use of such a rule changes the structure and height of a derivation.

Our logic obeys standard substitution principles and deduction is sound with respect to our semantic model.

**Lemma 5 (Substitution)**
*If $\mathrm{FV}(\Delta) \cap \mathrm{FV}(\Gamma) = \emptyset$ then*

- *If $\Gamma(u{:}F) \equiv \Gamma'(u{:}F)$ then $\Gamma(\Delta) \equiv \Gamma'(\Delta)$.*
- *If $\Theta \parallel \Delta \vdash F$ and $\Theta \parallel \Gamma(u{:}F) \vdash F'$ then $\Theta \parallel \Gamma(\Delta) \vdash F'$.*
- *If $\Theta, x : K \parallel \Delta \vdash F$ then for all $a \in K$, $\Theta \parallel \Delta[a/x] \vdash F[a/x]$.*

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

*Hypothesis*

$$\frac{}{\Theta \parallel u{:}F \vdash F}\ \mathit{Hyp}\ (u)$$

*Linear and Ordered Unit*

$$\frac{}{\Theta \parallel \cdot \vdash \mathbf{1}}\ 1I \qquad \frac{\Theta \parallel \Delta \vdash \mathbf{1} \quad \Theta \parallel \Gamma(\cdot) \vdash C}{\Theta \parallel \Gamma(\Delta) \vdash C}\ 1E$$

*Linear Conjunction*

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

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

*Ordered Conjunction*

$$\frac{\Theta \parallel \Delta_1 \vdash F_1 \quad \Theta \parallel \Delta_2 \vdash F_2}{\Theta \parallel \Delta_1; \Delta_2 \vdash F_1 \circ F_2}\ \circ I$$

$$\frac{\Theta \parallel \Delta \vdash F_1 \circ F_2 \quad \Theta \parallel \Gamma(u_1{:}F_1; u_2{:}F_2) \vdash C}{\Theta \parallel \Gamma(\Delta) \vdash C}\ \circ E$$

*Linear Implication*

$$\frac{\Theta \parallel \Delta, u{:}F_1 \vdash F_2}{\Theta \parallel \Delta \vdash F_1 \multimap F_2}\ \multimap I$$

$$\frac{\Theta \parallel \Delta \vdash F_1 \multimap F_2 \quad \Theta \parallel \Delta_1 \vdash F_1}{\Theta \parallel \Delta, \Delta_1 \vdash F_2}\ \multimap E$$

*Ordered Implications*

$$\frac{\Theta \parallel u{:}F_1; \Delta \vdash F_2}{\Theta \parallel \Delta \vdash F_1 \rightarrowtail F_2}\ \rightarrowtail I$$

$$\frac{\Theta \parallel \Delta \vdash F_1 \rightarrowtail F_2 \quad \Theta \parallel \Delta_1 \vdash F_1}{\Theta \parallel \Delta_1; \Delta \vdash F_2}\ \rightarrowtail E$$

$$\frac{\Theta \parallel \Delta; u{:}F_1 \vdash F_2}{\Theta \parallel \Delta \vdash F_1 \twoheadrightarrow F_2}\ \twoheadrightarrow I$$

$$\frac{\Theta \parallel \Delta \vdash F_1 \twoheadrightarrow F_2 \quad \Theta \parallel \Delta_1 \vdash F_1}{\Theta \parallel \Delta; \Delta_1 \vdash F_2}\ \twoheadrightarrow E$$

**Figure 4: AL : Multiplicative Connectives**

Proof. By induction on the appropriate derivation in each case. □

**Lemma 6 (Soundness of Logical Deduction)**
*If $m \vDash^{\Psi}_{C} \Delta$ and $\cdot \parallel \Delta \vdash F$, then $m \vDash^{\Psi} F$.*

Proof. By induction on the natural deduction derivation. □

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

*Additive Conjunction and Unit*

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

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

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

*Additive Disjunction and Unit*

$$\frac{\Theta \parallel \Delta \vdash \mathbf{0}}{\Theta \parallel \Delta \vdash F}\ 0E$$

$$\frac{\Theta \parallel \Delta \vdash F_1}{\Theta \parallel \Delta \vdash F_1 \oplus F_2}\ \oplus I1 \qquad \frac{\Theta \parallel \Delta \vdash F_2}{\Theta \parallel \Delta \vdash F_1 \oplus F_2}\ \oplus I2$$

$$\frac{\Theta \parallel \Delta \vdash F_1 \oplus F_2 \quad \Theta \parallel \Gamma(u_1{:}F_1) \vdash C \quad \Theta \parallel \Gamma(u_2{:}F_2) \vdash C}{\Theta \parallel \Gamma(\Delta) \vdash C}\ \oplus E$$

*Universal Quantification*

$$\frac{\Theta, x{:}K \parallel \Delta \vdash F}{\Theta \parallel \Delta \vdash \forall x{:}K.F}\ \forall I \qquad \frac{\Theta \parallel \Delta \vdash \forall x{:}K.F \quad a \in K}{\Theta \parallel \Delta \vdash F[a/x]}\ \forall E$$

*Existential Quantification*

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

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

**Figure 5: AL : Additive Connectives and Quantifiers**

## 2.7 Operations on Formulae

Our typing rules will make extensive use of an operation to *look up* the type of a location offset by an index ($g[i]$) in a formula. To facilitate this operation we use the notation $F(g[i])$ which is defined as follows.

**Definition (Formula Lookup)**

$$F(g_0[i]) = \tau \quad \text{if} \quad \cdot \parallel F \vdash \top \otimes ((g_0 : \tau_0) \circ \cdots \circ (g_i : \tau_i))$$

We *update* the type of a location $g[i]$ in a formula $F$ using the notation $F\,[g[i] := \tau]$ which is defined as follows.

**Definition (Formula Update)**

$$\begin{array}{l} F\,[g_0[i] := \tau] \overset{\text{def}}{=} F_1 \otimes (F_2 \circ (g_0 : \tau_0) \circ \cdots \circ (g_i : \tau) \circ F_3) \\ \quad \text{if} \quad \cdot \parallel F \vdash F_1 \otimes (F_2 \circ (g_0 : \tau_0) \circ \cdots \circ (g_i : \tau_i) \circ F_3) \end{array}$$

The following lemma states that formula lookup and update are sound with respect to the semantics. This lemma is used extensively in the proof of soundness of our type system.

**Lemma 7**

- *If $m \vDash^{\Psi} F$ and $F(g[i]) = \tau$, then $m(g[i]) = h$ and $\vdash^{\Psi} h : \tau$.*
- *If $m \vDash^{\Psi} F$ and $\vdash^{\Psi} h : \tau$, then $m\,[g[i] := h] \vDash^{\Psi} F\,[g[i] := \tau]$.*

PROOF. By the soundness of logical deduction and the semantics of formulae. □

## 3. A SIMPLE ASSEMBLY LANGUAGE

In this section, we develop the simplest possible assembly language that allows us to write stack-based programs.

### 3.1 Syntax

There are four main components of a program. A code region $C$ is a finite partial map from code values to blocks of code $B$. Each block is a sequence of instructions $\iota$ terminated by a jump instruction. Finally, the operands ($v$) which appear in instructions are made up of the values that we have seen before.

$$\begin{array}{llcl} \textit{Operands} & v & ::= & h \mid r \\ \textit{Instructions} & \iota & ::= & \texttt{add}\, r_d, r_s, v \mid \texttt{sub}\, r_d, r_s, v \mid \\ & & & \texttt{blte}\, r, v \mid \texttt{mov}\, r_d, v \mid \\ & & & \texttt{ld}\, r_d, r_s[i] \mid \texttt{st}\, r_d[i], r_s \\ \textit{Blocks} & B & ::= & \texttt{jmp}\, v \mid \iota; B \\ \textit{Code Region} & C & ::= & \cdot \mid C, c \mapsto B \end{array}$$

### 3.2 Types and Typing Rules

The type system for our assembly language is defined by the following judgments.

| | |
|---|---|
| $F \vdash^{\Psi} v : \tau$ | Operand $v$ has type $\tau$ in $F$ |
| $\Theta \parallel F \vdash^{\Psi} \iota : F'$ | Instruction $\iota$ requires a context $\Theta \parallel F$ and yields $F'$ |
| $\Theta \parallel F \vdash^{\Psi} B$ ok | Block $B$ is well-formed in context $\Theta \parallel F$ |
| $\vdash C : \Psi$ | Code region $C$ has type $\Psi$ assuming $\Psi$ |
| $\vdash \Sigma$ ok | State $\Sigma$ is well-formed |

Once again, although judgments for operands, instructions and blocks are formally parameterized by $\Psi$, we normally omit this annotation. The static semantics is given in Figures 6 and 7.

We assume our type system will be used in the context of proof-carrying code. More specifically, we assume a complete derivation will be attached to any assembly language program that needs to be checked for safety. To check that a program is well formed one need only check that it corresponds to the attached derivation and that the derivation uses rules from our type system and associated logic. The problem of inferring a derivation from an unannotated program is surely undecidable, but not relevant to a proof-carrying code application as a compiler can generate the appropriate derivation from a more highly structured source-level program.

**Operand Typing.** The rules for giving types to store values are extended to allow us to give types to operands, which include both store values and registers. The rule for registers requires that we look up the type of the register in the formula that describes the current state.

**Instruction Typing.** Instruction typing is performed in a context in which the free variables are described by $\Theta$ and the current state of the memory is described by the input formula $F$. An instruction will generally transform the state of the memory and result in a new state described by the formula $F'$. For instance, if the initial state is described by $F$, and we can verify that $r_s$ and $v$ contain integers then the instruction $\mathtt{add}\, r_d, r_s, v$ transforms the state so that the result is described by $F[r_d := \mathbf{int}]$. This resulting formula uses our formula update notation, which we defined in Section 2.7. The simple rule for integer subtraction is identical. To type check the conditional branch we must show that the second operand has code type and that the formula describing the current state entails the requirements specified in the function type. The rule for typing move is straightforward.

The rules for typing load and store instructions make use of our formula lookup operations. The formula lookup operation $F(\ell[i]) = \tau$ suffices to verify that the location $\ell + i$ exists in memory and contains a value with type $\tau$ (recall Lemma 7).

Finally, our type system allows simple pointer arithmetic, which can be used to bump up the stack pointer. The rules addr-add and addr-sub provide alternate typing rules for addition and subtraction operations. If $r_s$ contains the (constant or variable) location $\ell_0$ and $v$ is the constant integer $i$ and we can prove that the current memory can be described as

$$\top \otimes ((\ell_0 : \_) \circ (\ell_1 : \_) \circ \cdots \circ (\ell_i : \_))$$

then the results of addition is the location $g_i$. We can come to this conclusion even though we do not know exactly which locations $g_0$ and $g_i$ that we're dealing with. The fuse operator allows us to reason that each of the locations in the sequence in the formula are adjacent to one another and therefore that $g_i$ is $i$ locations from $g_0$. We may reason about the address subtraction typing rule analogously.

**Block Typing.** Block typing is described in Figure 7. The basic block typing rules are b-instr, which processes one instruction in a block and then the rest of the block, and b-jmp which types the jump instruction that ends a block.

Block typing also includes rules to extend our view of memory (b-stackgrow) or retract our view of memory (b-stackcut). Typically, when we wish to push more data on the stack, we will first use the b-stackgrow rule (as many times as necessary), then we will add the appropriate amount to the stack pointer and finally we will store onto the stack the data we wish to keep there. To pop the stack, we do the reverse, the first loading data off the stack into registers, subtracting the appropriate amount from the stack pointer and using the b-stackcut rule.

**State Typing.** The rule for typing code is the standard rule for a mutually recursive set of functions. The rule for typing an overall machine state requires that we type check our program $C$ and then check the code we are currently executing ($B$) under the assumption $F$, which describes the current memory $m$.

**An Example.** The stack may be used to save temporary values during the course of a computation. The code sequence in Figure 8 saves registers $r_1$ through $r_n$, which contain values of types $\tau_1$ through $\tau_n$, on the stack, performs a computation $A$, and then restores the $n$ values to their original registers. The formulae to the right of each instruction describe the state of the memory at each step.

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

$$\frac{\vdash h : \tau}{F \vdash h : \tau}\ (\mathsf{sval}) \qquad \frac{}{F \vdash r : F(r[0])}\ (\mathsf{reg})$$

$\boxed{\Theta \parallel F \vdash \iota : F'}$

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

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

$$\frac{F \vdash r : \mathbf{int} \qquad F \vdash v : (F') \to 0 \qquad \Theta \parallel F \vdash F'}{\Theta \parallel F \vdash \mathtt{blte}\, r, v : F}\ (\mathsf{blte})$$

$$\frac{F \vdash v : \tau}{\Theta \parallel F \vdash \mathtt{mov}\, r_d, v : F\,[r_d := \tau]}\ (\mathsf{mov})$$

$$\frac{F \vdash r_s : \mathbf{S}(\ell) \qquad F(\ell[i]) = \tau}{\Theta \parallel F \vdash \mathtt{ld}\, r_d, r_s[i] : F\,[r_d := \tau]}\ (\mathsf{ld})$$

$$\frac{F \vdash r_d : \mathbf{S}(\ell) \qquad F \vdash r_s : \tau \qquad F(\ell[i]) = \tau'}{\Theta \parallel F \vdash \mathtt{st}\, r_d[i], r_s : F\,[\ell[i] := \tau]}\ (\mathsf{st})$$

$$\frac{\begin{array}{c} F \vdash r_s : \mathbf{S}(\ell_0) \qquad F \vdash v : \mathbf{S}(i) \\ \Theta \parallel F \vdash \top \otimes ((\ell_0 : \_) \circ (\ell_1 : \_) \circ \cdots \circ (\ell_i : \_)) \end{array}}{\Theta \parallel F \vdash \mathtt{add}\, r_d, r_s, v : F\,[r_d := \mathbf{S}(\ell_0)]}\ (\mathsf{addr\text{-}add})$$

$$\frac{\begin{array}{c} F \vdash r_s : \mathbf{S}(\ell_i) \qquad F \vdash v : \mathbf{S}(i) \\ \Theta \parallel F \vdash \top \otimes ((\ell_0 : \_) \circ \cdots \circ (\ell_i : \_)) \end{array}}{\Theta \parallel F \vdash \mathtt{sub}\, r_d, r_s, v : F\,[r_d := \mathbf{S}(\ell_0)]}\ (\mathsf{addr\text{-}sub})$$

**Figure 6: Static Semantics (Values and Instructions)**

## 3.3 Operational Semantics

**Abstract Machine States.** An abstract machine state $\Sigma$ is a 3-tuple containing a code region $C$, a *complete* memory $m$ and the block of code $B$ that is currently being executed. A complete memory is a *total* function from generic locations to store values (i.e., $(Loc \cup Reg) \to Sval$). We require memories to be complete in order to justify the b-stackcut rule.

**Operational Semantics.** We define execution of our abstract machine using a small-step operational semantics $\Sigma \longmapsto \Sigma'$.

The operational semantics is given in Figure 9. In the semantics, we use $\widehat{m}$ to convert an operand to a value that may be stored at a location.

$$\begin{array}{rcl} \widehat{m}(i) & = & i \\ \widehat{m}(c) & = & c \\ \widehat{m}(\ell) & = & \ell \\ \widehat{m}(r) & = & m(r) \end{array}$$

We also make use of the fact that "+" and "−" are overloaded so they operate both on integers and locations. This allows us to write a single specification for the execution of addition and subtraction operations. Aside from these

| *Code* | *Describing Formula* |
|---|---|
| | $(\mathsf{more}^{\leftarrow} \circ (\ell : \tau) \circ F_1) \otimes (r_{sp} : \mathbf{S}(\ell))$ |
| (b-stackgrow) *Repeat n times* | $(\mathsf{more}^{\leftarrow} \circ \mathsf{ns} \circ \cdots \circ \mathsf{ns} \circ (\ell : \tau) \circ F_1) \otimes (r_{sp} : \mathbf{S}(\ell))$ |
| (b-unpack) *Repeat n times* | $(\mathsf{more}^{\leftarrow} \circ (\ell_1 : \_) \circ \cdots \circ (\ell_n : \_) \circ (\ell : \tau) \circ F_1) \otimes (r_{sp} : \mathbf{S}(\ell))$ |
| $\mathtt{sub}\, r_{sp}, r_{sp}, n$ | $(\mathsf{more}^{\leftarrow} \circ (\ell_1 : \_) \circ (\ell_2 : \_) \circ \cdots \circ (\ell_n : \_) \circ (\ell : \tau) \circ F_1) \otimes (r_{sp} : \mathbf{S}(\ell_1))$ |
| $\mathtt{st}\, r_{sp}[0], r_1$ | $(\mathsf{more}^{\leftarrow} \circ (\ell_1 : \tau_1) \circ (\ell_2 : \_) \circ \cdots \circ (\ell_n : \_) \circ (\ell : \tau) \circ F_1) \otimes (r_{sp} : \mathbf{S}(\ell_1))$ |
| $\vdots$ | $\vdots$ |
| $\mathtt{st}\, r_{sp}[n-1], r_n$ | $(\mathsf{more}^{\leftarrow} \circ (\ell_1 : \tau_1) \circ (\ell_2 : \tau_2) \circ \cdots \circ (\ell_n : \tau_n) \circ (\ell : \tau) \circ F_1) \otimes (r_{sp} : \mathbf{S}(\ell_1))$ |
| *Code for A* | |
| $\mathtt{ld}\, r_1, r_{sp}[0]$ | $(\mathsf{more}^{\leftarrow} \circ (\ell_1 : \tau_1) \circ (\ell_2 : \tau_2) \circ \cdots \circ (\ell_n : \tau_n) \circ (\ell : \tau) \circ F_1) \otimes (r_{sp} : \mathbf{S}(\ell_1))$ |
| $\vdots$ | $\vdots$ |
| $\mathtt{ld}\, r_n, r_{sp}[n-1]$ | $(\mathsf{more}^{\leftarrow} \circ (\ell_1 : \tau_1) \circ (\ell_2 : \tau_2) \circ \cdots \circ (\ell_n : \tau_n) \circ (\ell : \tau) \circ F_1) \otimes (r_{sp} : \mathbf{S}(\ell_1))$ |
| $\mathsf{add}\, r_{sp}, r_{sp}, n$ | $(\mathsf{more}^{\leftarrow} \circ (\ell_1 : \tau_1) \circ (\ell_2 : \tau_2) \circ \cdots \circ (\ell_n : \tau_n) \circ (\ell : \tau) \circ F_1) \otimes (r_{sp} : \mathbf{S}(\ell))$ |
| (b-stackcut) | $(\mathsf{more}^{\leftarrow} \circ (\ell : \tau) \circ F_1) \otimes (r_{sp} : \mathbf{S}(\ell))$ |

**Figure 8: Saving Temporaries on the Stack**

details, the operational semantics is quite intuitive.

## 3.4 Progress & Preservation

To demonstrate that our language is sound, we have proven progress and preservation lemmas. Preservation requires the following lemma in the case for the b-stackcut and b-stackgrow rules.

**Lemma 8 (Stack Cut / Stack Grow Soundness)**

- *If $m$ is a complete memory and $m \models^{\Psi} (\mathsf{more}^{\leftarrow} \circ F_1 \circ F_2) \otimes R$ then $m \models^{\Psi} (\mathsf{more}^{\leftarrow} \circ F_2) \otimes R$.*
- *If $m$ is a complete memory and $m \models^{\Psi} (\mathsf{more}^{\leftarrow} \circ F_2) \otimes R$ then $m \models^{\Psi} (\mathsf{more}^{\leftarrow} \circ \mathsf{ns} \circ F_2) \otimes R$.*

Proof. By inspection of the semantics of formulae. □

**Theorem 9 (Progress & Preservation)**
*If $\vdash (C, m, B)$ ok then*

1. *$(C, m, B) \longmapsto (C, m', B')$.*
2. *if $(C, m, B) \longmapsto (C, m', B')$ then $\vdash (C, m', B')$ ok.*

## 4. RELATED WORK

Our logic and type system for assembly language grew out of a number of previous efforts to handle explicit memory management in a safe language. Among the first to investigate this area were Tofte and Talpin who developed a provably sound system of region-based memory management [26]. At approximately the same time, Reynolds and O'Hearn [22, 16] were investigating the semantics of Algol and its translation to low-level intermediate languages with explicit memory management.

Later, the development of proof-carrying code [12, 11] and typed assembly language [10, 8] provided new motivation to study safe memory management at a low-level of abstraction. Morrisett et al. [8] looked specifically at a type system for stack allocation, but their type system is quite ad hoc and inflexible when compared with the work here. In particular, they cannot hide the relative order of objects on the stack since they have no analogue of our tensor connective. This deficiency often makes it impossible to store data deep on the stack. They also have no analogue of our additive connectives which allow us to specify different "views" of the stack.

Recently, researchers have developed very rich logics that are capable of expressing essentially any compile-time property of programs. For instance, Appel et al. [1, 2] use higher-order logic to code up the semantics of a flexible type system and Shao et al. [24] and Crary and Vanderwaart [3] incorporate logical frameworks into their type systems. With enough effort, implementors could surely code up our abstractions in these systems. However, our logic and language still serves a purpose in these settings: it may be used as a convenient and concise logical intermediate language. No matter how powerful the logic, it is still necessary to think carefully about how to structure one's proofs. Our research, which defines a logic at just the right level of abstraction for reasoning about the stack, provides that structure.

Our semantics for formulae is heavily influenced by the work of Ishtiaq, O'Hearn and Reynolds [7, 23]. However, Ishtiaq and O'Hearn work with a bunched logic [14] that contains additive and linear separators rather than linear and ordered separators as we have done. There are also a variety of other more minor differences. We also took inspiration from Polakow and Pfenning's ordered linear logic [20, 21, 19]. In fact, we initially attempted to encode stack invariants using their logic directly. However, we found the linear, ordered judgment $\Delta; \Omega \vdash F$ that they use incompatible with the adjacency property we desired. The semantics of this judgment is such that the linear formulae in $\Delta$ may be placed anywhere in the ordered context $\Omega$. Consequently two formulae may be juxtaposed in $\Omega$, but need not describe adjacent memory cells. There may be another formula in $\Delta$ that sits between them but has not yet been placed. Still, Petersen et al. [17] have managed to develop a language for reasoning about memory layout and adjacency that is derived from ordered logic.

Some of the technical typing machinery, including the use of singleton types to encode aliasing relationships comes from work by Smith, Walker and Morrisett [25, 28], and its precursor, the capability calculus [27].

## 5. FUTURE WORK

There are several directions for future work. First, we would like to continue to investigate substructural logics for

$\boxed{\Theta \parallel F \vdash B \ \mathsf{ok}}$

$$\frac{F \vdash v : (F') \rightarrow 0 \qquad \Theta \parallel F \vdash F'}{\Theta \parallel F \vdash \mathtt{jmp}\, v \ \mathsf{ok}} \ (\text{b-jmp})$$

$$\frac{\Theta \parallel F \vdash \iota : F' \qquad \Theta \parallel F' \vdash B \ \mathsf{ok}}{\Theta \parallel F \vdash \iota ; B \ \mathsf{ok}} \ (\text{b-instr})$$

$$\frac{\Theta \parallel (\mathsf{more}^{\leftharpoondown} \circ F_2) \otimes R \vdash B \ \mathsf{ok}}{\Theta \parallel (\mathsf{more}^{\leftharpoondown} \circ F_1 \circ F_2) \otimes R \vdash B \ \mathsf{ok}} \ (\text{b-stackcut})$$

$$\frac{\Theta \parallel (\mathsf{more}^{\leftharpoondown} \circ \mathsf{ns} \circ F_2) \otimes R \vdash B \ \mathsf{ok}}{\Theta \parallel (\mathsf{more}^{\leftharpoondown} \circ F_2) \otimes R \vdash B \ \mathsf{ok}} \ (\text{b-stackgrow})$$

$$\frac{\Theta, b \parallel F' \vdash B \ \mathsf{ok}}{\Theta \parallel \exists b.F' \vdash B \ \mathsf{ok}} \ (\text{b-unpack})$$

$$\frac{\Theta \parallel F \vdash F' \qquad \Theta \parallel F' \vdash B \ \mathsf{ok}}{\Theta \parallel F \vdash B \ \mathsf{ok}} \ (\text{b-weaken})$$

$\boxed{\vdash C : \Psi}$

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

$\boxed{\vdash \Sigma \ \mathsf{ok}}$

$$\frac{\vdash C : \Psi \qquad m \vDash^{\Psi} F \qquad \cdot \parallel F \vdash^{\Psi} B \ \mathsf{ok}}{\vdash (C, m, B) \ \mathsf{ok}} \ (\text{state})$$

**Figure 7: Static Semantics (Blocks and States)**

| $(C, m, B) \longmapsto \Sigma$ where | |
|---|---|
| If $B =$ | then $\Sigma =$ |
| $\mathtt{add}\, r_d, r_s, v; B'$ | $(C, m\,[r_d := (m(r_s) + \widehat{m}(v))], B')$ |
| $\mathtt{sub}\, r_d, r_s, v; B'$ | $(C, m\,[r_d := (m(r_s) - \widehat{m}(v))], B')$ |
| $\mathtt{blte}\, r, v; B'$ and $m(r) > 0$ | $(C, m, B')$ |
| $\mathtt{blte}\, r, v; B'$ and $m(r) \leq 0$ | $(C, m, B'')$ where $C(\widehat{m}(v)) = B''$ |
| $\mathtt{mov}\, r_d, v; B'$ | $(C, m\,[r_d := \widehat{m}(v)], B')$ |
| $\mathtt{ld}\, r_d, r_s[i]; B'$ | $(C, m\,[r_d := m(m(r_s) + i)], B')$ |
| $\mathtt{st}\, r_d[i], r_s; B'$ | $(C, m\,[g := m(r_s)], B')$ where $g = m(r_d) + i$ |
| $\mathtt{jmp}\, v$ | $(C, m, B'')$ where $C(\widehat{m}(v)) = B''$ |

**Figure 9: Operational Semantics**

reasoning about explicit memory management. In particular, we would like to determine whether we can find a logic in which deduction is complete for our memory model, or a related model such as the store model used by Ishtiaq and O'Hearn. Second, we would like to study certifying compilation for stack allocation algorithms in more detail. What are the limitations of our stack logic? Are there current optimization techniques which we cannot capture? Third, can we extend our logic with formulae to describe allocation of resources in different *regions* of memory? Such an extension would be aimed at encoding Tofte and Talpin's region calculus. Fourth, we feel confident that we will be able to use Walker and Morrisett's recursive formulas [28] or O'Hearn et al.'s inductive definitions [15] to code up recursive data structures, which we do not handle here, but the topic deserves deeper investigation. Finally, we have not discussed arrays in this paper. They would have to be handled either through a special macro that does the bounds check, or the system would need to be augmented with arithmetic formulae as in the work by Xi and Pfenning [29].

## Acknowledgments

We would like to thank Bob Harper and Leaf Petersen for stimulating conversations early in the development of this work.

## 6. REFERENCES

[1] A. W. Appel. Foundational proof-carrying code. In *Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 247–258. IEEE, 2001.

[2] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Twenty-seventh ACM Symposium on Principles of Programming Languages*, pages 243–253. ACM Press, Jan. 2000.

[3] K. Crary and J. Vanderwaart. An expressive, scalable type theory for certified code. In *ACM International Conference on Functional Programming*, Pittsburgh, Oct. 2002. ACM Press.

[4] R. Deline and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, June 2001. ACM Press.

[5] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[6] 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.

[7] 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.

[8] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based Typed Assembly Language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, Mar. 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28-52. Springer-Verlag, 1998.

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

[10] 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.

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

[12] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, Oct. 1996.

[13] R. O'Callahan. A simple, comprehensive type system for java bytecode subroutines. In *ACM Symposium on*

*Principles of Programming Languages*, pages 70–78, San Antonio, jan 1999.

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

[15] 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.

[16] P. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *Journal of the ACM*, 47(1):167–223, 2000.

[17] L. Peterson, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout. Unpublished manuscript, July 2002.

[18] F. Pfenning and R. Davies. A judgment reconstruction of modal logic. *Mathematical Structures in Computer Science*, 2000. To appear.

[19] J. Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001. Available As Technical Report CMU-CS-01-152.

[20] J. Polakow and F. Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In J.-Y. Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, pages 295–309, Berlin, 1999. Springer-Verlag.

[21] J. Polakow and F. Pfenning. Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic. *Electronic Notes in Theoretical Computer Science*, 20, 1999.

[22] J. Reynolds. Using functor categories to generate intermediate code. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 25–36, San Francisco, Jan. 1995.

[23] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial perspectives in computer science*, Palgrove, 2000.

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

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

[26] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[27] D. Walker, K. Crary, and G. Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, May 2000.

[28] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, Montreal, Sept. 2000.

[29] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.