Reasoning about Hierarchical Storage

Amal Ahmed Limin Jia David Walker

Princeton University
{amal,ljia,dpw}@cs.princeton.edu

Abstract

In this paper, we develop a new substructural logic that can encode invariants necessary for reasoning about hierarchical storage. *We show how the logic can be used to describe the layout of bits in a memory word, the layout of memory words in a region, the layout of regions in an address space, or even the layout of address spaces in a multiprocessing environment. We provide a semantics for our formulas and then apply the semantics and logic to the task of developing a type system for Mini-KAM, a simplified version of the abstract machine used in the ML Kit with regions.*

1 Introduction

The problem of establishing that programs allocate, initialize, use and deallocate memory safely has plagued programming language researchers for decades. Moreover, the relatively recent development of proof-carrying code [14, 15] and typed assembly language [13] and the widescale deployment of low-level safe virtual machines [10, 25] has provided still more incentive to study the wide variety of invariants that can be used to ensure safe memory management.

One of the most promising trends in this area is the use of substructural logics rather than conventional classical logics to describe the state of a computation [23, 8]. The expressive connectives of a substructural logic are able to capture the spatial orientation of a data structure in a concise fashion without having to rely upon the series of auxiliary predicates needed by conventional logics. For instance, Ishtiaq, O'Hearn and Reynolds have used the multiplicatives of bunched logic to capture *spatial separation* properties in data structures. Their formula $(\ell \mapsto 3) * (\ell' \mapsto 3)$ describes two *separate* locations ℓ and ℓ' that both contain the integer 3. Similar information can be captured in a classical formula, but only at the expense of having to introduce additional predicates that represent inequality information explicitly: $(\ell \mapsto 3) \wedge (\ell' \mapsto 3) \wedge (\ell \neq \ell')$. As the complexity of the spatial properties increases, the classical formulas become less and less wieldy. For example, as the number of locations in a formula grows linearly, the number of inequalities needed to specify that all locations are disjoint grows quadratically.

In this paper, we develop a new substructural logic that provides simple but general connectives for reasoning about *hierarchical storage*. We show how the logic can be used to describe the layout of bits in a memory word, the layout of memory words in a *region* [27], and the layout of regions in an address space. We then combine connectives that describe these hierarchical relationships with other substructural formulas that describe separation and adjacency of memory locations. We provide a semantics for our formulas and apply the semantics and logic to the task of developing a type system for Mini-KAM, a simplified version of the abstract machine used in the ML Kit with regions [26].

2 Preliminary Development

We develop our logic from first principles following the methodology set out by Martin Löf [11] and Frank Pfenning [18, 19]. The details of our logic were directly inspired by Cardelli and Gordon's ambient logic [3], and O'Hearn and Pym's logic of bunched implications (BI) [16].

We begin by considering not only *whether* a formula is true but also *where* it is true. Hence the primary judgment J of our logic has the form $F \mathbin{@} p$ where F is a logical formula and p is a place where that formula may or may not be true. For the purposes of the current paper, places are nodes in an edge-labeled tree, as in the ambient logic. We use the metavariable n to range over edge names and we use paths from the root ($*$) to refer to places.

$$\textit{Path/Place} \qquad p \quad ::= \quad * \mid p.n$$

With these primitive concepts in hand, we may proceed to develop a logic capable of expressing three main spatial properties:

- *Containment* of one place in another.

- *Separation* or disjointedness of one object from another.
- *Adjacency* of one object to another.

Containment. We say that one place p *contains* another place p' when $p' = p.n$ for some edge n. Our logic internalizes the notion of containment with a formula $n[F]$, which is defined by the following rules.

$$\frac{\vdash F \mathbin{@} p.n}{\vdash n[F] \mathbin{@} p}\ n[]I \qquad\qquad \frac{\vdash n[F] \mathbin{@} p}{\vdash F \mathbin{@} p.n}\ n[]E$$

As a preliminary check on the consistency of these rules, we note that the elimination rule is locally sound and complete with respect to the introduction rule. Local soundness ensures that the elimination rules for a connective are not too strong: it is impossible to gain extra information simply by introducing the connective and then immediately eliminating it. Local soundness of the above rules is witnessed by the following local reduction ($\Rightarrow_r$).

$$\frac{\dfrac{\dfrac{\mathcal{D}}{\vdash F \mathbin{@} p.n}\ n[]I}{\vdash n[F] \mathbin{@} p}\ n[]E}{\vdash F \mathbin{@} p.n} \quad \Rightarrow_r \quad \frac{\mathcal{D}}{\vdash F \mathbin{@} p.n}$$

Local completeness ensures that the elimination rules are not too weak: given an arbitrary proof of the connective, we can recover enough information through eliminations to be able to reintroduce the connective. Local completeness of the rules above is witnessed by the following local expansion ($\Rightarrow_e$).

$$\frac{\mathcal{E}}{\vdash n[F] \mathbin{@} p} \quad \Rightarrow_e \quad \frac{\dfrac{\dfrac{\mathcal{E}}{\vdash n[F] \mathbin{@} p}\ n[]E}{\vdash F \mathbin{@} p.n}\ n[]I}{\vdash n[F] \mathbin{@} p}$$

To illustrate the use of this simple connective, we assume the presence of a collection of logical predicates τ, which can be interpreted as saying that "a value with type τ is here." For example, the formula $\ell[\mathbf{int}]$ says "an integer is in the location ℓ." The judgment $\rho[r[\mathbf{bool}]] \mathbin{@} *$ says that "at the root, the process ρ contains a region r that contains a boolean."

Separation. Separation is most easily defined by extending our basic judgments to depend upon linear contexts with the following form.

$$\textit{Contexts} \quad \Delta \quad ::= \quad \cdot \mid (u{:}J) \mid \Delta_1, \Delta_2$$

We define contexts as a tree of hypotheses rather than a more conventional list of hypotheses in anticipation of further extensions for reasoning about adjacency. The nodes in these trees are labeled with the (linear) separator ",". The leaves of these trees are either empty (denoted by "$\cdot$") or a single judgment labeled with a variable u.[1] We treat these trees of hypotheses as equivalent up to associativity and commutativity of the "," separator and regard "$\cdot$" as the left and right identity for ",". However, these contexts are not subject to contraction or weakening.

Our new hypothetical judgments obey the following linear substitution principle, where Γ is a context containing a single hole and $\Gamma(\Delta)$ is notation for filling the hole in Γ with the context Δ. We consider $\Gamma(\Delta)$ to be undefined if Γ and Δ have any variables u in common. In our final logical system, this substitution principle can be proven as a lemma.

Principle 1 (Substitution)
If $\Delta \vdash F \mathbin{@} p$ *and* $\Gamma(u{:}F \mathbin{@} p) \vdash F' \mathbin{@} p'$ *then* $\Gamma(\Delta) \vdash F' \mathbin{@} p'$.

To internalize the notion of separation, we introduce a multiplicative conjunction $F_1 \otimes F_2$ (pronounced F_1 *tensor* F_2).

$$\frac{\Delta_1 \vdash F_1 \mathbin{@} p \quad \Delta_2 \vdash F_2 \mathbin{@} p}{\Delta_1, \Delta_2 \vdash (F_1 \otimes F_2) \mathbin{@} p}\ \otimes I$$

$$\frac{\Delta \vdash (F_1 \otimes F_2) \mathbin{@} p \quad \Gamma(u_1{:}F_1 \mathbin{@} p\,,\ u_2{:}F_2 \mathbin{@} p) \vdash F \mathbin{@} p}{\Gamma(\Delta) \vdash F \mathbin{@} p}\ \otimes E$$

The only thing to distinguish this connective from BI's multiplicative conjunction is the presence of the path p in our judgments. Since the path is everywhere the same, our multiplicative combines two separate objects in a particular place into a pair of objects in that place. As before, it is easy to verify that the elimination is locally sound and complete with respect to the introduction.

As an example of our new connective, consider the formula $r[\ell[\mathbf{int}] \otimes \ell'[\mathbf{int}]]$, which asserts that "region r contains two separate locations ℓ and ℓ' that both contain integers". The formula $r[\ell[\mathbf{int}]] \otimes r[\ell'[\mathbf{int}]]$ makes the same assertion.

Adjacency. The last main concept in our logic is adjacency. To model adjacency at the level of judgments, we extend our hypothetical context one more time with an adjacency separator ";".

$$\textit{Contexts} \quad \Delta \quad ::= \quad \cdots \mid \Delta_1; \Delta_2$$

We extend the equivalence relation on contexts so that ";" is associative, has the empty context "$\cdot$" as its identity but, unlike "," is *not* commutative. Neither separator distributes over the other. In summary, the equivalence relation on contexts is the reflexive, symmetric and transitive closure of the following axioms.

[1] We label hypotheses with distinct variables u to facilitate the proof of the substitution principle (Principle 1).

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 $\Delta \equiv \Delta'$

We internalize adjacency with an ordered conjunction (called *fuse*).

$$\frac{\Delta_1 \vdash F_1 \mathbin{@} p \quad \Delta_2 \vdash F_2 \mathbin{@} p}{\Delta_1; \Delta_2 \vdash (F_1 \circ F_2) \mathbin{@} p} \circ I$$

$$\frac{\Delta \vdash (F_1 \circ F_2) \mathbin{@} p \quad \Gamma(u_1{:}F_1 \mathbin{@} p;\, u_2{:}F_2 \mathbin{@} p) \vdash F \mathbin{@} p}{\Gamma(\Delta) \vdash F \mathbin{@} p} \circ E$$

Our ordered conjunction allows us to specify a sequence of objects lined up one next to the other. For example, we can specify a sequence of 32 bits in a word at location ℓ as $\ell[0[\mathbf{bit}] \circ 1[\mathbf{bit}] \circ \cdots \circ 31[\mathbf{bit}]]$ where 0 through 31 are the names of the bit locations; or we can specify three objects in sequence on the top of the stack as $stack[l_1[\mathbf{int}] \circ l_2[\mathbf{bool}] \circ l_3[\mathbf{int}] \circ tail]$ where the formula $tail$ describes the tail of the stack.

Adjacent locations are not only next to one another they are also separate from one another. Formally, adjacency and separation are related by the following principle, which states that we may view a proof of $\Gamma(\Delta_1, \Delta_2) \vdash F \mathbin{@} p$ as a proof of $\Gamma(\Delta_1; \Delta_2) \vdash F \mathbin{@} p$.[2]

Principle 2 (Disorder)
If $\Gamma(\Delta_1, \Delta_2) \vdash F \mathbin{@} p$ *then* $\Gamma(\Delta_1; \Delta_2) \vdash F \mathbin{@} p$.

The ordered and linear conjunctions ($\circ$ and $\otimes$, respectively) share a single identity **1** defined by familiar inference rules.

$$\frac{}{\cdot \vdash \mathbf{1} \mathbin{@} p}\ 1I \qquad \frac{\Delta \vdash \mathbf{1} \mathbin{@} p \quad \Gamma(\cdot) \vdash F \mathbin{@} p}{\Gamma(\Delta) \vdash F \mathbin{@} p}\ 1E$$

Quantifiers and Other Connectives. Our logic includes formulae for universal and existential quantification that have the form $\forall b.F$ and $\exists b.F$, respectively. The bindings in quantification formulae describe the sort (integer I, path P, type T, formula F, or name N) of the bound variable.

$$\textit{Bindings} \quad b ::= i{:}\mathsf{I} \mid p{:}\mathsf{P} \mid \alpha{:}\mathsf{T} \mid \phi{:}\mathsf{F} \mid n{:}\mathsf{N}$$

To support universal and existential quantification we extend the judgments of our logic to additionally depend on a variable context Θ. The final form of the basic judgment of our logic is $\Theta \parallel \Delta \vdash F \mathbin{@} p$ where Θ describes the variables that appear free in Δ, F, or p.

$$\textit{Variable Contexts} \quad \Theta ::= \cdot \mid \Theta, b$$

[2]Unlike our Substitution Principle, the Disorder Principle cannot be proven as a lemma in our final system unless we add the corresponding structural inference rule. We choose not to add an explicit structural rule, but implicitly include one sort of proof for another wherever necessary in a derivation, following a similar idea in Pfenning and Davies' development of modal logic [19].

Rules for the quantification connectives are given in Figure 1, along with rules for additive conjunction ($\&$) and its unit ($\top$), and additive disjunction ($\oplus$) and its unit (**0**). Our logic extends readily to handle linear, left-ordered and right-ordered implications ($\multimap, \rightarrowtail, \twoheadrightarrow$) though we do not describe these here in the interest of space.

Our latest hypothetical judgments obey all the principles described thus far as well as the following variable substitution principle. We use the metavariable K to range over sorts, x to range over variables in general and a to range over objects of each different sort.

Principle 3 (Variable Substitution)
If $\Theta, x{:}K \parallel \Delta \vdash F \mathbin{@} p$ *then for all* $a{\in}K$, $\Theta \parallel \Delta[a/x] \vdash F[a/x] \mathbin{@} p[a/x]$.

Summary of Logical Formulae and Deduction Rules The syntax of formulae in our logic is summarized below. The simplest formulae are predicates q. Thus far we have only encountered predicates of the form τ, which can be interpreted as "a value of type τ is here." Additional predicates will be introduced in Section 3.2.

$$\begin{array}{llll} \textit{Predicates} & q & ::= & \tau \mid \cdots \\ \textit{Formulae} & F & ::= & q \mid n[F] \mid \mathbf{1} \mid F_1 \otimes F_2 \mid F_1 \circ F_2 \mid \\ & & & \top \mid F_1 \mathbin{\&} F_2 \mid \mathbf{0} \mid F_1 \oplus F_2 \mid \\ & & & \phi \mid \forall b.F \mid \exists b.F \end{array}$$

The natural deduction rules of our logic are collected in Figure 1.

3 Store Semantics

In this section, we describe a model for our logic based on hierarchical stores.

3.1 Stores

We assume the existence of an abstract set of values Val. A store is a partial map from paths p to values v.

$$\textit{Stores} \quad s \in Path \rightharpoonup Val$$

Values will be given types via a judgment of the form $\vdash^{\Psi} v : \tau$, where Ψ is an abstract type assignment.

In order to discuss adjacent places, we assume the existence of a partial function $\mathsf{succ} : Path \rightharpoonup Path$, which maps a path p to the path that immediately follows it. We write $\mathsf{adj}(p, p')$ when $p' = \mathsf{succ}(p)$. We use $p + i$ and $p\langle i \rangle$ as syntactic sugar for $\mathsf{succ}^i(p)$ and $p - i$ for the path p' such that $p = \mathsf{succ}^i(p')$. We define the relation $\leq$ in terms of succ as follows: $p \leq p'$ iff there exists a natural number i such that $\mathsf{succ}^i(p) = p'$. We say that a set $P \subseteq Path$ is *ordered* if and only if it can be organized in a total order given by the relation $\leq$. Otherwise we say that P is *unordered*.

We use the following notation to manipulate stores.

$\boxed{\Theta \parallel \Delta \vdash F \mathbin{@} p}$

Hypothesis

$$\frac{}{\Theta \parallel u{:}F \mathbin{@} p \vdash F \mathbin{@} p}\ \textit{Hyp (u)}$$

Containment

$$\frac{\Theta \parallel \Delta \vdash F \mathbin{@} p.n}{\Theta \parallel \Delta \vdash n[F] \mathbin{@} p}\ n[]I \qquad \frac{\Theta \parallel \Delta \vdash n[F] \mathbin{@} p}{\Theta \parallel \Delta \vdash F \mathbin{@} p.n}\ n[]E$$

Linear and Ordered Unit

$$\frac{}{\Theta \parallel \cdot \vdash \mathbf{1} \mathbin{@} p}\ 1I$$

$$\frac{\Theta \parallel \Delta \vdash \mathbf{1} \mathbin{@} p \quad \Theta \parallel \Gamma(\cdot) \vdash F \mathbin{@} p}{\Theta \parallel \Gamma(\Delta) \vdash F \mathbin{@} p}\ 1E$$

Linear Conjunction

$$\frac{\Theta \parallel \Delta_1 \vdash F_1 \mathbin{@} p \quad \Theta \parallel \Delta_2 \vdash F_2 \mathbin{@} p}{\Theta \parallel \Delta_1, \Delta_2 \vdash (F_1 \otimes F_2) \mathbin{@} p}\ \otimes I$$

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

Ordered Conjunction

$$\frac{\Theta \parallel \Delta_1 \vdash F_1 \mathbin{@} p \quad \Theta \parallel \Delta_2 \vdash F_2 \mathbin{@} p}{\Theta \parallel \Delta_1; \Delta_2 \vdash (F_1 \circ F_2) \mathbin{@} p}\ \circ I$$

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

Additive Conjunction and Unit

$$\frac{}{\Theta \parallel \Delta \vdash \top \mathbin{@} p}\ \top I \qquad \frac{\Theta \parallel \Delta \vdash F_1 \mathbin{@} p \quad \Theta \parallel \Delta \vdash F_2 \mathbin{@} p}{\Theta \parallel \Delta \vdash (F_1 \,\&\, F_2) \mathbin{@} p}\ \&I$$

$$\frac{\Theta \parallel \Delta \vdash (F_1 \,\&\, F_2) \mathbin{@} p}{\Theta \parallel \Delta \vdash F_1 \mathbin{@} p}\ \&E1 \qquad \frac{\Theta \parallel \Delta \vdash (F_1 \,\&\, F_2) \mathbin{@} p}{\Theta \parallel \Delta \vdash F_2 \mathbin{@} p}\ \&E2$$

Additive Disjunction and Unit

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

$$\frac{\Theta \parallel \Delta \vdash F_1 \mathbin{@} p}{\Theta \parallel \Delta \vdash (F_1 \oplus F_2) \mathbin{@} p}\ \oplus I1 \qquad \frac{\Theta \parallel \Delta \vdash F_2 \mathbin{@} p}{\Theta \parallel \Delta \vdash (F_1 \oplus F_2) \mathbin{@} p}\ \oplus I2$$

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

Universal Quantification

$$\frac{\Theta, x{:}K \parallel \Delta \vdash F \mathbin{@} p}{\Theta \parallel \Delta \vdash (\forall x{:}K.F) \mathbin{@} p}\ \forall I$$

$$\frac{\Theta \parallel \Delta \vdash (\forall x{:}K.F) \mathbin{@} p \quad a \in K}{\Theta \parallel \Delta \vdash F\,[a/x] \mathbin{@} p}\ \forall E$$

Existential Quantification

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

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

Figure 1. Natural Deduction Rules

- $dom(s)$ denotes the domain of the store s
- $s(p)$ denotes the value stored at path p
- $s\,[p := v]$ denotes a store s' in which p maps to v but is otherwise the same as s. If $p \notin dom(s)$ then $s' = s \cup \{p \mapsto v\}$
- Given an ordered set of paths $X \subseteq Path$, $\ulcorner X \urcorner$ 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 relation $\leq$. Given any unordered set $Y \subseteq Path$, $\ulcorner Y \urcorner$ and $\llcorner Y \lrcorner$ are undefined. $\ulcorner \emptyset \urcorner$ and $\llcorner \emptyset \lrcorner$ are also undefined.
- $s_1 \# s_2$ indicates that the stores s_1 and s_2 have disjoint domains
- $s_1 \uplus s_2$ denotes the union of disjoint stores; if the domains of the two stores are not disjoint then this operation is undefined.
- $s_1 \circledcirc s_2$ denotes the union of disjoint stores with the additional caveat that either $\mathsf{adj}(\ulcorner dom(s_1) \urcorner, \llcorner dom(s_2) \lrcorner)$ or one of s_1 or s_2 is empty.

3.2 Semantics of Judgments

We use judgments to describe stores and write $s \models^{\Psi} F \mathbin{@} p$ when the judgment $F \mathbin{@} p$ describes the store s. The most basic formulae are the predicates τ, $\mathsf{more}^{\leftarrow}$, and $\mathsf{more}^{\rightarrow}$. The judgment $\tau \mathbin{@} p$ describes a store with a single place p that holds a value of type τ. The judgments $\mathsf{more}^{\leftarrow} \mathbin{@} p$ and $\mathsf{more}^{\rightarrow} \mathbin{@} p$ describe an infinite sequence of adjacent places to the left (and right, respectively) of some place contained in p. When we develop a type system for Mini-KAM in section 4, we will use $\mathsf{more}^{\leftarrow}$ to indicate that the stack may be grown to the left, and $\mathsf{more}^{\rightarrow}$ to indicate that heap regions may be grown to the right.

The formula $n[F] \mathbin{@} p$ describes a store s if and only if s may be described by the formula $F \mathbin{@} p.n$. To illustrate the

semantics of the containment connective we consider the store $s = \{*.n_1.n_2 \mapsto 5\}$. The judgment $(n_1[n_2[\mathbf{int}]]) \mathbin{@} *$ describes s, as do the judgments $(n_2[\mathbf{int}]) \mathbin{@} * .n_1$ and $\mathbf{int} \mathbin{@} * .n_1.n_2$.

The semantics of the multiplicative conjunction $F_1 \otimes F_2$ follows from the work of Ishtiaq and O'Hearn [8]: A store s can be described by $(F_1 \otimes F_2) \mathbin{@} p$ if and only if there exist s_1, s_2, such that $s_1 \vDash^\Psi F_1 \mathbin{@} p$ and $s_2 \vDash^\Psi F_2 \mathbin{@} p$ and $s = s_1 \uplus s_2$. To get accustomed to some of the properties of tensor and to contrast it with fuse, we will reason about the following stores which contain locations in the set $\{*.m.n_i \mid 0 \leq i \leq k\}$ where each path in this set is adjacent to the next in sequence (i.e., for all i, $\mathsf{adj}(*.m.n_i, *.m.n_{i+1})$).

Store	*Domain*	*Describing Judgment*
s_1	$\{*.m.n_1, *.m.n_3\}$	$F_1 \mathbin{@} *$
s_2	$\{*.m.n_4, *.m.n_6\}$	$F_2 \mathbin{@} *$
s_3	$\{*.m.n_2\}$	$F_3 \mathbin{@} *$
s_4	$\emptyset$	$F_4 \mathbin{@} *$
s_5	$\{*.m.n_7\}$	$F_5 \mathbin{@} *.m$
s_6	$\{*.m.n_6\}$	$F_6 \mathbin{@} *.m$

The store $s = s_1 \cup s_2 \cup s_3$ can be described by the judgment $((F_1 \otimes F_2) \otimes F_3) \mathbin{@} *$ since s can be broken into two disjoint parts, $s_1 \cup s_2$ and s_3, which satisfy the subformulae $(F_1 \otimes F_2) \mathbin{@} *$ and $F_3 \mathbin{@} *$ respectively. The store s also satisfies the judgments $(F_1 \otimes (F_2 \otimes F_3)) \mathbin{@} *$ and $(F_3 \otimes (F_2 \otimes F_1)) \mathbin{@} *$ since it is defined in terms of the associative and commutative disjoint union operator.

For fuse, we have $s \vDash^\Psi (F_1 \circ F_2) \mathbin{@} p$ if and only if s can be divided into two *adjacent* parts, s_1 and s_2 such that $s_1 \vDash^\Psi F_1 \mathbin{@} p$ and $s_2 \vDash^\Psi F_2 \mathbin{@} p$. More formally, we require $s = s_1 \circledcirc s_2$. Now consider our example from above. The store $s_1 \cup s_2$ may be described using the judgment $(F_1 \circ F_2) \mathbin{@} *$ since the supremum of s_1 is adjacent to the infimum of s_2. This same store cannot be described by $(F_2 \circ F_1) \mathbin{@} *$ — fuse is not generally commutative. On the other hand, s_1 can be described by either $(F_1 \circ F_4) \mathbin{@} *$ or $(F_4 \circ F_1) \mathbin{@} *$ since $s_1 = s_1 \circledcirc \emptyset = \emptyset \circledcirc s_1$. Since neither the supremum nor the infimum of s_3 is adjacent to the infimum or supremum, respectively, of s_1 or s_2 we cannot readily use fuse to describe the relationship between these memories. Finally, the supremum of s_2 is adjacent to the infimum of s_5 and $s_2 \cup s_5$ can be described by $(F_2 \circ m[F_5]) \mathbin{@} *$.

To see how fuse interacts with the containment connective, consider the stores s_5 and s_6. The supremum of s_6 is adjacent to the infimum of s_5 so it follows that the store $s_6 \cup s_5$ can be described by $(F_6 \circ F_5) \mathbin{@} * .m$. Moreover, this same store $s_6 \cup s_5$ can be described by $m[F_6 \circ F_5] \mathbin{@} *$ and also by $(m[F_6] \circ m[F_5]) \mathbin{@} *$. The interaction between tensor and containment is analogous: $s_6 \cup s_5$ satisfies both $m[F_5 \otimes F_6] \mathbin{@} *$ and $(m[F_5] \otimes m[F_6]) \mathbin{@} *$. The fact that the same store satisfies both these judgments is a point of departure from the ambient logic [3].

$s \vDash^\Psi F \mathbin{@} p$ if and only if

- $F = \tau$ and $dom(s) = \{p\}$ and $s(p) = v$ and $\vdash^\Psi v : \tau$
- $F = \mathsf{more}^{\leftarrow}$ and there exists a non-empty set X such that $dom(s) = \{p.x \mid x{\in}X\}$ and $\forall x{\in}X.\, \exists y{\in}dom(s).\, \mathsf{adj}(y, p.x)$
- $F = \mathsf{more}^{\rightarrow}$ and there exists a non-empty set X such that $dom(s) = \{p.x \mid x{\in}X\}$ and $\forall x{\in}X.\, \exists y{\in}dom(s).\, \mathsf{adj}(p.x, y)$
- $F = n[F']$ and $s \vDash^\Psi F' \mathbin{@} p.n$
- $F = \mathbf{1}$ and $dom(s) = \emptyset$
- $F = F_1 \otimes F_2$ and there exist s_1, s_2, such that $s = s_1 \uplus s_2$ and $s_1 \vDash^\Psi F_1 \mathbin{@} p$ and $s_2 \vDash^\Psi F_2 \mathbin{@} p$
- $F = F_1 \circ F_2$ and there exist s_1, s_2, such that $s = s_1 \circledcirc s_2$ and $s_1 \vDash^\Psi F_1 \mathbin{@} p$ and $s_2 \vDash^\Psi F_2 \mathbin{@} p$
- $F = \top$ (and no other conditions need be satisfied)
- $F = F_1 \mathbin{\&} F_2$ and $s \vDash^\Psi F_1 \mathbin{@} p$ and $s \vDash^\Psi F_2 \mathbin{@} p$
- $F = \mathbf{0}$ and false (this formula can never be satisfied)
- $F = F_1 \oplus F_2$ and either
  1. $s \vDash^\Psi F_1 \mathbin{@} p$, or
  2. $s \vDash^\Psi F_2 \mathbin{@} p$.
- $F = \forall x{:}K.F'$ and $s \vDash^\Psi F'[a/x] \mathbin{@} p$ for all $a{\in}K$
- $F = \exists x{:}K.F'$ and there exists some $a{\in}K$ such that $s \vDash^\Psi F'[a/x] \mathbin{@} p$

Figure 2. Semantics of Judgments

The semantics for the rest of the formulae are collected in Figure 2. 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. The objects substituted for variables must have the correct sort (integer, path, type, formula, or name) or else the substitution is undefined.

3.3 Semantics of Contexts & Soundness

Like individual formulae, contexts can describe stores. The semantics of contexts appears below. 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.

$s \vDash^\Psi_C \Delta$ if and only if

- $\Delta = \cdot$ and $dom(s) = \emptyset$
- $\Delta = u{:}F \mathbin{@} p$ and $s \vDash^\Psi F \mathbin{@} p$
- $\Delta = \Delta_1, \Delta_2$ and $s = s_1 \uplus s_2$ and $s_1 \vDash^\Psi_C \Delta_1$ and $s_2 \vDash^\Psi_C \Delta_2$
- $\Delta = \Delta_1; \Delta_2$ and $s = s_1 \circledcirc s_2$ and $s_1 \vDash^\Psi_C \Delta_1$ and $s_2 \vDash^\Psi_C \Delta_2$

We have proven the following lemma which states that deduction is sound with respect to our semantic model. The proof follows by induction on the derivation that $\cdot \parallel \Delta \vdash F \mathbin{@} p$.

Lemma 4 (Soundness of Logical Deduction)
If $s \vDash^\Psi_C \Delta$ and $\cdot \parallel \Delta \vdash F \mathbin{@} p$, then $s \vDash^\Psi F \mathbin{@} p$.

4 Mini-KAM

In this section, we present the syntax and the static and dynamic semantics of Mini-KAM, a simplified and idealized version of the Kit Abstract Machine [6] used in the ML Kit with regions [26]. Mini-KAM is a stack-based machine that consists of three registers: two general-purpose registers $acc1$ and $acc2$, and a stack pointer sp that points to the last allocated cell at the top of the stack. In addition to the stack and registers, Mini-KAM has a set of *infinite* regions. Infinite regions are so named to distinguish them from *finite* regions [26]. The latter are regions whose maximum size is known *a priori*, which means that they can be allocated on the stack.

We model Mini-KAM using the hierarchical stores we introduced in Section 3.1. Figure 3 illustrates the store hierarchy in Mini-KAM. The three registers are "contained" in the root ($*$), as is the stack (named $stack$) and a set of (infinite) regions r_i. The stack in Mini-KAM grows with decreasing addresses and contains an infinite set of locations n_i that correspond to memory cells. The regions, on the other hand, contain an infinite set of increasing locations, starting with the distinguished location name $start$ — i.e., $*.r.start$ is the path to the first location in region r.

By comparison, the KAM has an infinite number of fixed-size pages each of which is contained in either the *free list* or in an allocated region. We could easily model the KAM by adding an extra level to our store hierarchy, such that regions contain pages which in turn contain locations, and also maintain a free list contained in root ($*$) that contains similar pages. The KAM also has a notion of *region pointers* which are special in that the two least significant bits are used to indicate whether the region is finite or infinite, and whether allocation should be performed at the top or the bottom (overwrite mode) of the region. The remaining 30 bits store the actual pointer to the region. We could imagine yet another level in our store hierarchy, such that locations each contain 32 bits as we sketched out in an example in Section 2. We have chosen to abstract away some of these details in this short paper.

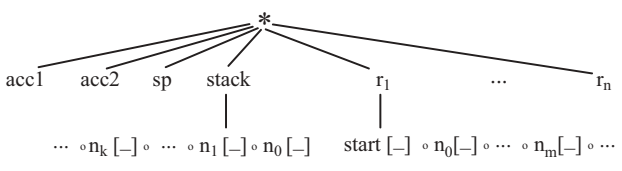

Figure 3. Mini-KAM Store Hierarchy

4.1 Syntax

We will be reasoning about several different sorts of values v including integers $i \in Int$, code locations $c \in Codeloc$, which contain executable code, places or paths $p \in Path$, and two special values `live` and `dead` that are useful for reasoning about whether or not an existing region is safe to access. Therefore, the set of values Val (which was left abstract in Section 3.1) may be defined as follows.

$$Val \quad = \quad Int \cup Path \cup Codeloc \cup \{\texttt{live}\} \cup \{\texttt{dead}\}$$

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 ι terminated by a jump instruction. Finally, the operands that appear in instructions are simply values v.

The state Σ of the Mini-KAM is a 3-tuple containing a code region C, a store s, and the block of code B that is currently being executed. The store hierarchy must match that shown in Figure 3.

Values	v	$::=$	$i \mid p \mid c \mid \texttt{live} \mid \texttt{dead}$
Instructions	ι	$::=$	$\texttt{immed1}\,(v) \mid \texttt{immed2}\,(v) \mid \texttt{swap} \mid$ $\texttt{add} \mid \texttt{sub} \mid \texttt{push} \mid \texttt{pop} \mid$ $\texttt{selectStack}\,(i) \mid \texttt{storeStack}\,(i) \mid$ $\texttt{select}\,(i) \mid \texttt{store}\,(i) \mid$ $\texttt{letRgnFin}\,(i) \mid \texttt{letRgnInf} \mid$ $\texttt{endRgnInf} \mid \texttt{alloc}\,(i)$
Blocks	B	$::=$	$\texttt{jmp} \mid \iota; B$
Code Region	C	$::=$	$\cdot \mid C, c \mapsto B$
Machine State	Σ	$::=$	(C, s, B)

4.2 Operational Semantics

We define execution of our abstract machine using a small-step operational semantics $\Sigma \longmapsto \Sigma'$. We briefly describe the instructions here. Readers desiring further details should consult the formal operational semantics given in Figure 4.

- `immed1` and `immed2` load the operand v into the registers $acc1$ and $acc2$ respectively, while `swap` swaps the contents of these two registers.
- `add` and `sub` assume that $acc1$ and $acc2$ contain integer operands and place the result in $acc1$.
- `push` stores the value in $acc1$ at the top of the stack and increments the stack pointer, while `pop` pops the value at the top of the stack and places it in $acc1$.
- `selectStack`(i) loads the contents of $p_{top} + i$ (where p_{top} is the top of the stack) into $acc1$, while `storeStack`(i) stores the value in $acc1$ at $p_{top} + i$.
- `select`(i) loads the contents of $p + i$ (where p is the address in $acc1$) into $acc1$, while `store`(i) stores the value in $acc1$ at $p + i$.

$(C, s, B) \longmapsto \Sigma$ where	
If $B =$	then $\Sigma =$
`immed1` $(v); B'$	$(C, s\,[*.acc1 := v], B')$
`immed2` $(v); B'$	$(C, s\,[*.acc2 := v], B')$
`swap` $; B'$	$(C, s\,[*.acc1 := s(*.acc2)]\,[*.acc2 := s(*.acc1)], B')$
`add` $; B'$	$(C, s\,[*.acc1 := s(*.acc1) + s(*.acc2)], B')$
`sub` $; B'$	$(C, s\,[*.acc1 := s(*.acc1) - s(*.acc2)], B')$
`push` $; B'$	$(C, s\,[p := s(*.acc1)]\,[*.sp := p], B')$ where $p = s(*.sp) - 1$
`pop` $; B'$	$(C, s\,[*.acc1 := s(s(*.sp))]\,[*.sp := s(*.sp) + 1], B')$
`selectStack` $(i); B'$	$(C, s\,[*.acc1 := s(s(*.sp) + i)], B')$
`storeStack` $(i); B'$	$(C, s\,[p := s(*.acc1)], B')$ where $p = s(*.sp) + i$
`select` $(i); B'$	$(C, s\,[*.acc1 := s(p)], B')$ where $p = s(*.acc1) + i$ and $\exists r, n.\, p = *.r.n \ \wedge\ s(*.r) =$ `live`
`store` $(i); B'$	$(C, s\,[p := s(*.acc2)], B')$ where $p = s(*.acc1) + i$ and $\exists r, n.\, p = *.r.n \ \wedge\ s(*.r) =$ `live`
`letRgnFin` $(i); B'$	$(C, s\,[*.sp := s(*.sp) - i]\,[*.acc1 := s(*.sp) - i], B')$
`letRgnInf` $; B'$	$(C, s\,[*.r :=$ `live`$]\,[*.r.start + j := 0]\,[p1 := *.r.start]\,[p2 := *.r.start]\,[*.sp := s(*.sp) - 2], B')$ for all $j \geq 0$ where $p1 = s(*.sp) - 1, p2 = s(*.sp) - 2, *.r \notin dom(s)$ and $\forall n. * .r.n \notin dom(s)$
`endRgnInf` $; B'$	$(C, s\,[*.r :=$ `dead`$]\,[*.sp := s(*.sp) + 2], B')$ where $s(s(*.sp) + 1) = *.r.start$
`alloc` $(i); B'$	$(C, s\,[p := s(p) + i]\,[*.acc1 := s(p) + 1], B')$ where $p = s(*.acc1)$ and $\exists r, n.\, s(p) = *.r.n \ \wedge\ s(*.r) =$ `live`
`jmp`	(C, s, B'') where $C(s(*.acc1)) = B''$

Figure 4. Operational Semantics

- `letRgnFin` (i) allocates a finite region of size i on the stack; it simply decrements the stack pointer by i and places a pointer to the beginning of the finite region in $acc1$.
- `letRgnInf` and `endRgnInf` allocate and free infinite regions, while `alloc` (i) allocates i consecutive memory locations in an existing infinite region. We shall describe these instructions in detail when we discuss their typing rules in Section 4.3.

4.3 Types and Typing Rules

In this section, we describe the typing rules for Mini-KAM, with an emphasis on how we use the store logic to describe the state of the abstract machine. Formal typing rules appear in Figures 5 and 6.

We give integers, paths, `live` and `dead` singleton types which identify them exactly. For example, the integer i may be given the type $\mathbf{S}(i)$. Code locations are given code types as described by a code context. These code types have the form $(F \mathbin{@} p) \rightarrow 0$, where $F \mathbin{@} p$ is a judgment 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 α arise through existential or universal quantification in formulae. The syntax of types is as follows.

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \alpha \mid \mathbf{S}(v) \mid (F \mathbin{@} p) \rightarrow 0 \\ \textit{Code Contexts} & \Psi & ::= & \cdot \mid \Psi, c{:}(F \mathbin{@} p) \rightarrow 0 \end{array}$$

The type system for Mini-KAM is defined by the following judgments.

$\vdash^{\Psi} v : \tau$	Value v has type τ
$\Theta \parallel J \vdash^{\Psi} \iota : J'$	Instruction ι requires a context $\Theta \parallel J$ and yields J'
$\Theta \parallel J \vdash^{\Psi} B$ ok	Block B is well-formed in context $\Theta \parallel J$
$\vdash C : \Psi$	Code region C has type Ψ
$\vdash \Sigma$ ok	State Σ is well-formed

The static semantics is given in Figures 5 and 6. Once again, although judgments for operands, instructions and blocks are formally parameterized by Ψ, we normally omit this annotation.

We use abbreviations for the following common formulae: $\exists i{:}\mathsf{I}.\, \mathbf{S}(i)$ is abbreviated $\mathbf{int}$; $\exists \alpha{:}\mathsf{T}.\, n[\alpha]$ is abbreviated $n[\_]$; $\exists n{:}\mathsf{N}.\, \exists \alpha{:}\mathsf{T}.\, n[\alpha])$ is abbreviated ns; $m_1[m_2 \cdots [m_k[F]] \cdots]$ is abbreviated $m_1.m_2.\cdots.m_k[F]$.

Definitions (Lookup and Update)
Our typing rules make extensive use of an operation to *look up* the formula describing the contents at a place offset by an index ($p\langle i \rangle$) in a judgment $F \mathbin{@} p$. To facilitate this operation we use the notation $J(p\langle i \rangle)$ which is defined as follows.

$$J(p.n\langle i \rangle) = F_i \ \ \textit{if}\ \ \cdot \parallel J \vdash (\top \otimes (n[F_0] \circ n_1[F_1] \circ \cdots \circ n_i[F_i])) \mathbin{@} p$$

Let us take a closer look at the above definition. Assume that s is the subset of the store that satisfies $(n[F_0] \circ n_1[F_1] \circ \cdots \circ n_i[F_i]) \mathbin{@} p$. Notice that in the above definition, the formula $\top$ "consumes" everything in the store that is not in s. This allows us to ignore parts of the store that are not relevant to the information we want to look up. For instance,

$\boxed{\vdash^{\Psi} v : \tau}$

$$\frac{}{\vdash v : \mathbf{S}(v)}\ (v \notin Codeloc) \qquad \frac{\Psi(c) = (F \mathbin{@} p) \to 0}{\vdash c : (F \mathbin{@} p) \to 0}\ \textsf{(code)}$$

$\boxed{\Theta \parallel F \mathbin{@} p \vdash B\ \textsf{ok}}$

$$\frac{J(*.acc_1) = (J') \to 0 \qquad \Theta \parallel J \vdash J'}{\Theta \parallel J \vdash \texttt{jmp}\ \textsf{ok}}\ \textsf{(b-jmp)}$$

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

$$\frac{\Theta \parallel (n[\mathsf{more}^{\leftarrow} \circ F_1] \otimes F) \mathbin{@} p \vdash B\ \textsf{ok}}{\Theta \parallel (n[\mathsf{more}^{\leftarrow} \circ \mathsf{ns} \circ F_1] \otimes F) \mathbin{@} p \vdash B\ \textsf{ok}}\ \textsf{(b-stackcut)}$$

$$\frac{\Theta \parallel (n[\mathsf{more}^{\leftarrow} \circ \mathsf{ns} \circ F_1] \otimes F) \mathbin{@} p \vdash B\ \textsf{ok}}{\Theta \parallel (n[\mathsf{more}^{\leftarrow} \circ F_1] \otimes F) \mathbin{@} p \vdash B\ \textsf{ok}}\ \textsf{(b-stackgrow)}$$

$$\frac{\Theta \parallel (n[F_1 \circ \mathsf{ns} \circ \mathsf{more}^{\rightarrow}] \otimes F) \mathbin{@} p \vdash B\ \textsf{ok}}{\Theta \parallel (n[F_1 \circ \mathsf{more}^{\rightarrow}] \otimes F) \mathbin{@} p \vdash B\ \textsf{ok}}\ \textsf{(b-regiongrow)}$$

$$\frac{\Theta, b \parallel F \mathbin{@} p \vdash B\ \textsf{ok}}{\Theta \parallel \exists b.F \mathbin{@} p \vdash B\ \textsf{ok}}\ \textsf{(b-unpack)}$$

$$\frac{\Theta \parallel J \vdash J' \qquad \Theta \parallel J' \vdash B\ \textsf{ok}}{\Theta \parallel J \vdash B\ \textsf{ok}}\ \textsf{(b-weaken)}$$

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

$$\frac{\begin{array}{c} dom(C) = dom(\Psi) \\ \forall c \in dom(C).\ \Psi(c) = (F \mathbin{@} p) \to 0 \\ \textit{implies}\ \cdot \parallel F \mathbin{@} p \vdash^{\Psi} C(c)\ \textsf{ok} \end{array}}{\vdash C : \Psi}\ \textsf{(codergn)}$$

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

$$\frac{\vdash C : \Psi \qquad s \vDash^{\Psi} F \mathbin{@} * \qquad \cdot \parallel F \mathbin{@} * \vdash^{\Psi} B\ \textsf{ok}}{\vdash (C, s, B)\ \textsf{ok}}\ \textsf{(state)}$$

Figure 5. Static Semantics (except instrs.)

any objects immediately to the right of n_i are simply consumed by $\top$. We shall use the abbreviation $J(p)$ for the lookup $J(p\langle 0 \rangle)$.

We *update* the type of a path $p\langle i \rangle$ in a judgment J using the notation $J\,[p\langle i \rangle := \tau]$ which is defined as follows.

$$\begin{array}{l} J\,[*.p.n\langle i \rangle := \tau]\ = \\ \qquad (F_1 \otimes (F_2 \circ p[n[\tau_0] \circ n_1[\tau_1] \circ \cdots \circ n_i[\tau] \circ F_3)) \mathbin{@} * \\ \textit{if}\ \cdot \parallel J \vdash (F_1 \otimes (F_2 \circ p[n[\tau_0] \circ n_1[\tau_1] \circ \cdots \circ n_i[\tau_i]] \circ F_3)) \mathbin{@} * \\ \textit{where } p \textit{ is a sequence of names.} \end{array}$$

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 J, which describes the current store s.

Block Typing. 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 the stack (b-stackgrow), retract our view of the stack (b-stackcut) or extend our view of a region (b-regiongrow). Typically, when we wish to push more data on the stack, we will first use the b-stackgrow rule (as many times as necessary), and then `push` data onto the stack. Similarly, to allocate i new cells in an infinite region, typically we would first use the b-regiongrow rule i times and then perform an `alloc`(i). To pop the stack, we reverse the order, using `pop` one or more times, followed by as many uses of the b-stackcut rule.

Instruction Typing. Instruction typing is performed in a context in which the free variables are described by Θ and the current state of the store is described by the input judgement J. An instruction will generally transform the state of the store and result in a new state described by the judgement J'. For instance, if the initial state is described by J and we can verify that $\vdash v : \tau$, then the instruction `immed1`(v) transforms the store so that the new state is described by $J\,[*.acc1 := \tau]$. The rule for typing `immed2` is identical. The rule for swapping the contents of $acc1$ and $acc2$ makes use of our judgment lookup operation. In general, the lookup operation $J(p) = F$ suffices to verify that the path p exists in the store and $F \mathbin{@} p$ describes some portion of the store. The rules for integer addition and subtraction are similar to that for `swap`.

To type check the `push` instruction, if sp points to the location n in $stack$ and we can verify that some *portion* of the current store can be described by the judgment $(n'[\_] \circ n[\_]) \mathbin{@} *.stack$, then after the stack pointer has been decremented it should point to $*.stack.n'$. We can come to this conclusion even though we do not know exactly which locations n and n' we're dealing with. The fuse operator allows us to conclude that $\mathsf{adj}(*.stack.n', *.stack.n)$. In this way, we can replace arithmetic reasoning (that is, reasoning about incrementing and decrementing pointers) with reasoning about adjacency within our logic. The typing rule for `pop` is almost identical to that for `push`.

Typing `selectStack`(i) and `storeStack`(i) requires reasoning similar to that for `push`: the stack pointer points at $*.stack.n_0$ and we can verify that some part of the store is described by $(n_0[\_] \circ n_1[\_] \circ \cdots \circ n_i[\_]) \mathbin{@} *.stack$, allowing us to conclude that the result of adding i to the stack pointer would be the path $*.stack.n_i$.

To type check `select` and `store` (which involve accessing a region other than the stack), if $acc1$ points to an address in region r, then we must verify that region r is live — i.e., we check that $J(*.r) = \mathbf{S}(\texttt{live})$. The rest of the reasoning for these instructions is similar to that for their

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

$$\frac{\vdash v : \tau}{\Theta \parallel J \vdash \texttt{immed1}\,(v) : J\,[*.acc1 := \tau]}\ \text{(immed1)} \qquad \frac{\vdash v : \tau}{\Theta \parallel J \vdash \texttt{immed2}\,(v) : J\,[*.acc2 := \tau]}\ \text{(immed2)}$$

$$\frac{J(*.acc1) = \tau_1 \qquad J(*.acc2) = \tau_2}{\Theta \parallel J \vdash \texttt{swap} : J\,[*.acc1 := \tau_2]\,[*.acc2 := \tau_1]}\ \text{(swap)}$$

$$\frac{J(*.acc1) = \mathbf{int} \qquad J(*.acc2) = \mathbf{int}}{\Theta \parallel J \vdash \texttt{add} : J\,[*.acc1 := \mathbf{int}]}\ \text{(add)} \qquad \frac{J(*.acc1) = \mathbf{int} \qquad J(*.acc2) = \mathbf{int}}{\Theta \parallel J \vdash \texttt{sub} : J\,[*.acc1 := \mathbf{int}]}\ \text{(sub)}$$

$$\frac{J(*.acc1) = \tau \qquad J(*.sp) = \mathbf{S}(*.stack.n) \qquad J(*.stack) = n'[\_] \circ n[\_]}{\Theta \parallel J \vdash \texttt{push} : J\,[*.sp := \mathbf{S}(*.stack.n')]\,[*.stack.n' := \tau]}\ \text{(push)}$$

$$\frac{J(*.sp) = \mathbf{S}(*.stack.n') \qquad J(*.stack) = n'[\tau] \circ n[\_]}{\Theta \parallel J \vdash \texttt{pop} : J\,[*.sp := \mathbf{S}(*.stack.n)]\,[*.acc1 := \tau]}\ \text{(pop)}$$

$$\frac{J(*.sp) = \mathbf{S}(*.stack.n_0) \qquad J(*.stack) = n_0[\_] \circ n_1[\_] \circ \cdots \circ n_i[\tau]}{\Theta \parallel J \vdash \texttt{selectStack}\,(i) : J\,[*.acc1 := \tau]}\ \text{(selectStack)}$$

$$\frac{J(*.sp) = \mathbf{S}(*.stack.n_0) \qquad J(*.acc1) = \tau \qquad J(*.stack) = n_0[\_] \circ n_1[\_] \circ \cdots \circ n_i[\_]}{\Theta \parallel J \vdash \texttt{storeStack}\,(i) : J\,[*.stack.n_i := \tau]}\ \text{(storeStack)}$$

$$\frac{J(*.acc1) = \mathbf{S}(*.r.n_0) \qquad J(*.r) = \mathbf{S}(\texttt{live}) \otimes (n_0[\_] \circ n_1[\_] \circ \cdots \circ n_i[\tau])}{\Theta \parallel J \vdash \texttt{select}\,(i) : J\,[*.acc1 := \tau]}\ \text{(select)}$$

$$\frac{J(*.acc1) = \mathbf{S}(*.r.n_0) \qquad J(*.acc2) = \tau \qquad J(*.r) = \mathbf{S}(\texttt{live}) \otimes (n_0[\_] \circ n_1[\_] \circ \cdots \circ n_i[\_])}{\Theta \parallel J \vdash \texttt{store}\,(i) : J\,[*.r.n_i := \tau]}\ \text{(store)}$$

$$\frac{J(*.sp) = \mathbf{S}(*.stack.n_i) \qquad J(*.stack) = n_0[\_] \circ n_1[\_] \circ \cdots \circ n_i[\_]}{\Theta \parallel J \vdash \texttt{letRgnFin}\,(i) : J\,[*.sp := \mathbf{S}(*.stack.n_0)]\,[*.acc1 := \mathbf{S}(*.stack.n_0)]}\ \text{(letRgnFin)}$$

$$\frac{\Theta \parallel J \vdash J' \qquad J' = F_1 \otimes (F_2 \circ (stack[n_0[\_] \circ n_1[\_] \circ n_2[\tau]]) \circ F_3) \otimes sp[\mathbf{S}(*.stack.n_2)] \mathbin{@} *}{\begin{array}{l}\Theta \parallel J \vdash \texttt{letRgnInf} : \exists r.\, F_1 \otimes (F_2 \circ (stack[n_0[\mathbf{S}(*.r.start)] \circ n_1[\mathbf{S}(*.r.start)] \circ n_2[\tau]]) \circ F_3) \\ \qquad \otimes sp[\mathbf{S}(*.stack.n_0)] \otimes r[\mathbf{S}(\texttt{live}) \otimes (start[\_] \circ \mathsf{more}^{\rightarrow})] \mathbin{@} *\end{array}}\ \begin{array}{l}(r \notin FV(J')) \\ \text{(letRgnInf)}\end{array}$$

$$\frac{\begin{array}{c}J(*.sp) = \mathbf{S}(*.stack.n_0) \qquad J(*.r) = \mathbf{S}(\texttt{live}) \\ J(*.stack) = n_0[\mathbf{S}(*.r.n_{curr})] \circ n_1[\mathbf{S}(*.r.start)] \circ n_2[\_]\end{array}}{\Theta \parallel J \vdash \texttt{endRgnInf} : J\,[*.sp := \mathbf{S}(*.stack.n_2)]\,[*.r := \mathbf{S}(\texttt{dead})]}\ \text{(endRgnInf)}$$

$$\frac{J(*.acc1) = \mathbf{S}(p) \qquad J(p) = \mathbf{S}(*.r.n_0) \qquad J(*.r) = \mathbf{S}(\texttt{live}) \otimes (n_0[\_] \circ \cdots \circ n_i[\_])}{\Theta \parallel J \vdash \texttt{alloc}\,(i) : J\,[*.acc1 := \mathbf{S}(*.r.n_1)]\,[p := \mathbf{S}(*.r.n_i)]}\ \text{(alloc)}$$

Figure 6. Static Semantic (Instructions)

stack counterparts.

To allocate a finite region of size i on the stack we simply verify that that there are i locations to the left of the current top of the stack and decrement the stack pointer by i, using fuse to reason about adjacency as in the rule for `selectStack`. Register $acc1$ is updated with a pointer to the beginning of the finite region.

We illustrate how some of the above instructions may be used through an example. The code sequence in Figure 7 stores i values (v_0 through v_{i-1} of types τ_0 through τ_{i-1}) on the stack, uses these values in some computation A and then pops them off the stack. The judgments to the right of each instruction describe the state of the store after that instruction has been executed. The annotation "$\times i$" that follows some instructions indicates that the instruction should be performed i times.

Allocating a new infinite region is the only operation in Mini-KAM that involves extending the existing store. This means that the typing rule for `letRgnInf` is very different from the rules we have considered till now. Figure 8 depicts the situations before and after we allocate an infinite region. When an infinite region is allocated, a *start* pointer that points to the beginning of the region and a *current* pointer that points to the last allocated cell in the region

Code	*Describing Judgment*
	$(\top \otimes acc1[\_] \otimes stack[\mathsf{more}^{\leftarrow} \circ n[\tau] \circ F_1] \otimes sp[\mathbf{S}(*.stack.n)]) \mathbin{@} *$
(b-stackgrow) $\times i$	$(\top \otimes acc1[\_] \otimes stack[\mathsf{more}^{\leftarrow} \circ \mathsf{ns} \circ \cdots \circ \mathsf{ns} \circ n[\tau] \circ F_1] \otimes sp[\mathbf{S}(*.stack.n)]) \mathbin{@} *$
(b-unpack) $\times i$	$(\top \otimes acc1[\_] \otimes stack[\mathsf{more}^{\leftarrow} \circ n_0[\_] \circ \cdots \circ n_{i-1}[\_] \circ n[\tau] \circ F_1] \otimes sp[\mathbf{S}(*.stack.n)]) \mathbin{@} *$
`letRgnFin` (i)	$(\top \otimes acc1[\_] \otimes stack[\mathsf{more}^{\leftarrow} \circ n_0[\_] \circ \cdots \circ n_{i-1}[\_] \circ n[\tau] \circ F_1] \otimes sp[\mathbf{S}(*.stack.n_0)]) \mathbin{@} *$
`immed1` (v_0)	$(\top \otimes acc1[\tau_0] \otimes stack[\mathsf{more}^{\leftarrow} \circ n_0[\_] \circ \cdots \circ n_{i-1}[\_] \circ n[\tau] \circ F_1] \otimes sp[\mathbf{S}(*.stack.n_0)]) \mathbin{@} *$
`storeStack` (0)	$(\top \otimes acc1[\tau_0] \otimes stack[\mathsf{more}^{\leftarrow} \circ n_0[\tau_0] \circ \cdots \circ n_{i-1}[\_] \circ n[\tau] \circ F_1] \otimes sp[\mathbf{S}(*.stack.n_0)]) \mathbin{@} *$
$\vdots$	$\vdots$
`immed1` (v_{i-1})	$(\top \otimes acc1[\tau_{i-1}] \otimes stack[\mathsf{more}^{\leftarrow} \circ n_0[\tau_0] \circ \cdots \circ n_{i-2}[\tau_{i-2}] \circ n_{i-1}[\_] \circ n[\tau] \circ F_1] \otimes sp[\mathbf{S}(*.stack.n_0)]) \mathbin{@} *$
`storeStack` $(i-1)$	$(\top \otimes acc1[\tau_{i-1}] \otimes stack[\mathsf{more}^{\leftarrow} \circ n_0[\tau_0] \circ \cdots \circ n_{i-1}[\tau_{i-1}] \circ n[\tau] \circ F_1] \otimes sp[\mathbf{S}(*.stack.n_0)]) \mathbin{@} *$
% *BEGIN Computation A*	
$\vdots$	$\vdots$
`selectStack` (j) (where $0 \leq j < i$)	$(\top \otimes acc1[\tau_j] \otimes stack[\mathsf{more}^{\leftarrow} \circ n_0[\tau_0] \circ \cdots \circ n_j[\tau_j] \circ \cdots \circ n_{i-1}[\tau_{i-1}] \circ n[\tau] \circ F_1] \otimes sp[\mathbf{S}(*.stack.n_0)]) \mathbin{@} *$
$\vdots$	$\vdots$
% *END Computation A*	
`pop`	$(\top \otimes acc1[\tau_0] \otimes stack[\mathsf{more}^{\leftarrow} \circ \mathsf{ns} \circ n_1[\tau_1] \circ \cdots \circ n_{i-1}[\tau_{i-1}] \circ n[\tau] \circ F_1] \otimes sp[\mathbf{S}(*.stack.n_1)]) \mathbin{@} *$
$\vdots$	$\vdots$
`pop`	$(\top \otimes acc1[\tau_{i-1}] \otimes stack[\mathsf{more}^{\leftarrow} \circ \mathsf{ns} \circ \cdots \circ \mathsf{ns} \circ n[\tau] \circ F_1] \otimes sp[\mathbf{S}(*.stack.n)]) \mathbin{@} *$
(b-stackcut) $\times i$	$(\top \otimes acc1[\tau_{i-1}] \otimes stack[\mathsf{more}^{\leftarrow} \circ n[\tau] \circ F_1] \otimes sp[\mathbf{S}(*.stack.n)]) \mathbin{@} *$

Figure 7. Saving Values on the Stack

are added to the top of the stack, so the typing rule must verify that there exist two locations immediately to the left of the top of the stack. Operationally, to create the new region r we add $*.r \mapsto \mathtt{live}$ to the store to indicate that the region is *live*. We also extend the store with the infinite sequence of adjacent paths $*.r.start, *.r.start+1, \ldots$, mapped to some initial value, say 0. The operational semantics requires that $*.r$ and paths of the form $*.r.n$ (where n is a name) do not appear in the domain of the original store. We give a type to this operation by existentially quantifying the region name r in the conclusion of the typing rule and by requiring that r not appear free in the premise. The new portion of the store is described by $r[\mathbf{S}(\mathtt{live}) \otimes (start[\_] \circ \mathsf{more}^{\rightarrow})] \mathbin{@} *$ — the judgment describing the transformed state in the typing rule reflects this extension. This judgment also reflects the fact that a start pointer ($*.r.start$) and a current pointer (also $*.r.start$ at this point) are pushed onto the stack and the stack pointer is decremented by two.

To deallocate an infinite region we must first verify that the region is live and that the current pointer ($*.r.n_{curr}$) and start pointer ($*.r.start$) for the region are at the top of the stack. We deallocate region r by updating the contents of $*.r$ with the value `dead` and popping the current and start pointers off the stack.

When we allocate memory in a region we must make sure that the current pointer for the region (which we saved on the stack when we created the region) is updated correctly. As illustrated in Figure 9, the `alloc` instruction assumes that register $acc1$ contains the address of the location where the current pointer is stored. To type check allocation, if the current pointer points to $*.r.n_0$, we must verify that region r is live, and that there exists a sequence of $i+1$ contiguous locations in region r starting with n_0. Furthermore, the transformed state should be described by the initial judgment J altered to reflect the fact that the cur-

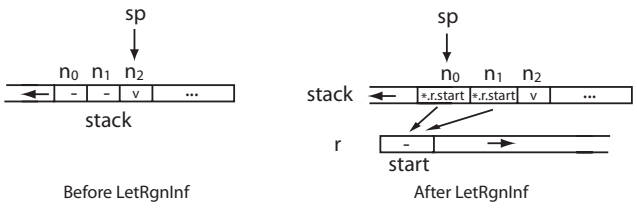

Figure 8. letRgnInf : Before and After

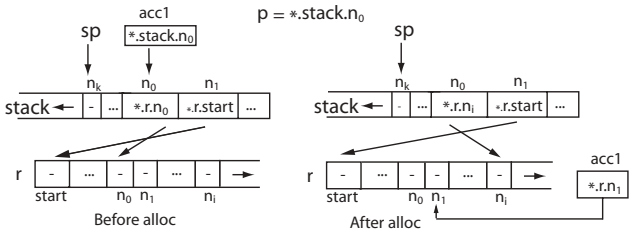

Figure 9. alloc(i) : Before and After

rent pointer on the stack is incremented by i and register $acc1$ is updated with a pointer to the beginning of the memory just allocated.

Soundness. To demonstrate that our language is sound we have proven standard progress and preservation lemmas.

Theorem 5 (Preservation)
If $\vdash (C, s, B)$ ok *and* $(C, s, B) \longmapsto (C, s', B')$ *then* $\vdash (C, s', B')$ ok.

Theorem 6 (Progress)
If $\vdash (C, s, B)$ ok *then* $(C, s, B) \longmapsto (C, s', B')$.

5 Related Work

Our logic and type system was inspired by a number of previous efforts to handle explicit memory management in a safe language. However, as far as we are aware, this is the first time a logic or type system has been used to describe a general memory hierarchy. It is also the first time that the concepts of containment, separation and adjacency have been combined in a single logic. We break down related work in terms of these three central concepts.

Containment. Cardelli and Gordon's ambient logic [3] was a direct source of inspiration for this work, but our two logics differ considerably:

- Their logic is classical and is presented as a sequent calculus whereas our logic is intuitionistic and is given in natural deduction.
- Their logic contains negation and modal necessity, but they do not consider adjacency.
- They use processes as a model for their logic whereas we use a hierarchical store.
- Their semantics for containment formulae differ slightly from ours. As a result, they may have two ambients with the same name in the same location (for instance, $m[F] \mid m[F]$, which is *not* equivalent to $m[F \mid F]$), whereas we only ever have one region with a given name in any location (that is, $m[F_1] \otimes m[F_2] \mathbin{@} p$ is equivalent to $m[F_1 \otimes F_2] \mathbin{@} p$).

Recently, Cardelli, Gardner and Ghelli [2] have developed a related spatial logic for reasoning about trees and graphs. They have used this logic to develop a query language for semi-structured data such as XML or web documents. Our logic, on the other hand, is intended to be used in a proof-carrying code system. Once again, there are a variety of differences between the connectives and the semantics of our two logics.

In the area of memory management, Tofte and Talpin's original work on region-based memory management [27] helped pave the way for this research. Their regions act as a fixed one-level hierarchy. Our logic extends the idea of regions to the general case of a multi-level hierarchy. More recent work on region-based memory management has considered integration of ideas from linear logic and linear type systems with regions [29, 5, 31], but no one has considered regions together with adjacency before and no one has considered a general memory containment type constructor.

Separation. Immediately after Girard developed linear logic [7], researchers rushed to investigate computational interpretations of the logic that take advantage of its separation properties to safely manage memory [9, 28, 4]. These projects used linear logic or some variant as a type system for a lambda calculus with explicit allocation and deallocation of memory. More recently, a new approach was suggested by Reynolds [23] and Ishtiaq and O'Hearn [8]. Rather than using a substructural logic to type lambda terms, they use a logic to describe the shape of the store. They have focused on using O'Hearn and Pym's bunched logic with multiplicatives and additives, but not ordered or containment connectives. Smith, Walker and Morrisett [24, 30] have worked out related ideas in a type-theoretic framework and we borrow their idea of using singleton types to reason about pointer aliasing.

Adjacency. Morrisett et al. [12] developed an algebra of lists to reason about adjacent locations on the stack. However, this discipline is quite inflexible when compared with our logic and it is impossible to use Morrisett's stack types to reason about regions.

Polakow and Pfenning's ordered linear logic [21, 22, 20] allows them to reason about the ordering of objects in memory. Polakow and Pfenning have applied their logic to the problem of reasoning about continuations allocated and deallocated on a stack. Petersen et al. [17] further observed that Polakow and Pfenning's mobility modality could be interpreted as pointer indirection and their fuse connective could join two adjacent structs. These observations allow them to use ordered logic as a type system for a language with explicit data layout. Petersen et al. do not consider dependency, which would allow them to reason accurately about aliasing, or the properties of separation or containment.

In a related paper [1] we presented a fragment of this logic with adjacency and separation connectives, but not containment. We used the logic to provide a type system for a stack-based assembly language, but were unable to capture region-based memory management in that system.

Acknowledgments. We are grateful to Peter O'Hearn for explaining the details of the logic of bunched implications to us. We would also like to thank Neal Glew, Peter O'Hearn and the anonymous referees for helpful comments on previous drafts of this paper.

References

[1] 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.

[2] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In *29th International Colloquium on Automata, Languages, and Programming, ICALP 2002*, volume 2380 of *Lecture Notes in Computer Science*, pages 597–610, Malaga, Spain, July 2002. Springer-Verlag.

[3] L. Cardelli and A. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM Press, Jan. 2000.

[4] J. Chirimar, C. A. Gunter, and J. G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, Mar. 1996.

[5] 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.

[6] M. Elsman and N. Hallenberg. A region-based abstract machine for the ML Kit. Technical Report TR-2002-18, Royal Veterinary and Agricultural University of Denmark and IT University of Copenhagen, August 2002. IT University Technical Report Series.

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

[8] 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.

[9] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[11] M. Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, University of Siena, 1985.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] L. Petersen, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout. In *ACM Symposium on Principles of Programming Languages*, Jan. 2003. To appear.

[18] F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 16, pages 977–1061. Elsevier Science and MIT Press, 2001.

[19] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

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

[21] 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.

[22] 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.

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

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

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

[26] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, P. Sestoft, and P. Bertelsen. Programming with regions in the ML Kit (for version 3). Technical Report 98/25, Computer Science Department, University of Copenhagen, 1998.

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

[28] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Progarmming Concepts and Methods*, Sea of Galilee, Israel, Apr. 1990. North Holland. IFIP TC 2 Working Conference.

[29] 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.

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

[31] D. Walker and K. Watkins. On linear types and regions. In *ACM International Conference on Functional Programming*, Florence, Sept. 2001. ACM Press.