$\mathbf{L}^3$: A Linear Language with Locations

Greg Morrisett
Harvard University
greg@eecs.harvard.edu

Amal Ahmed
Cornell University
aahmed@cs.cornell.edu

Matthew Fluet
Cornell University
fluet@cs.cornell.edu

Abstract

We present a simple, but expressive type system that supports *strong updates*—updating a memory cell to hold values of unrelated types at different points in time. Our formulation is based upon standard linear logic and, as a result, enjoys a simple semantic interpretation for types that is closely related to models for spatial logics. The typing interpretation is strong enough that, in spite of the fact that our core calculus supports shared, mutable references and cyclic graphs, every well-typed program terminates.

We then consider extensions needed to model ML-style references, where the capability to access a reference cell is unrestricted, but strong updates are disallowed. Our extensions include a `thaw` primitive for re-gaining the capability to perform strong updates on unrestricted references. The `thaw` primitive is closely related to other mechanisms that support strong updates, such as CQUAL's `restrict`.

1 Introduction

The goal of this work is to explore foundational typing support for *strong updates*. In type systems for imperative languages, a strong update corresponds to changing the type of a mutable object whenever the contents of the object is changed. That is, the type of a mutable location can change from program point to program point. As an example, consider the following code fragment written with SML syntax:

```
1.  let val r = ref () in
2.      r := true;
3.      if (!r) then
4.        r := 42
5.      else
6.        r := 15;
7.      !r + 12
8.  end
```

At line 1, we create a ref cell `r` whose contents are initialized with `unit`. At line 2, we change the contents so that `r` holds a `bool`. Then at lines 4 and 6, we change the contents of `r` again, this time to `int`. In spite of the fact that at different program points `r` holds values of different, incompatible types, there is nothing in the program that will cause a run-time type error.[1] This is because subsequent reads of the reference are type-compatible with the immediately preceding writes.

Unfortunately, most imperative languages, including SML and Java, do not support strong updates. For instance, SML rejects the above program since it requires that reference cells hold values of exactly one type. The reason for this is that tracking the current type of a reference cell at each program point is hindered by the potential for aliasing. Consider, for instance the following function:

```
1.  fun f(r1:int ref, r2:int ref):int =
2.    (r1 := true;
3.     !r2 + 42)
```

In order to avoid a typing error, this function can only be called in contexts where `r1` and `r2` are different ref cells. The reason is that if we passed in the same cell for each formal argument, then the update on line 2 should change not only the type of `r1` but also the type of `r2`, causing a type error to occur at line 3.

Thus, any type system that supports strong updates needs some control over aliasing. In addition, it is clear that the hidden side-effects of a function, such as the change in type to `f`'s first argument in the example above, must be reflected in the interface of the function to achieve modular type-checking. In short, strong updates seem to need a lot of technical machinery to ensure soundness and reasonable accuracy.

Lately, there have been a number of languages, type systems, and analyses that have supported some form of strong updates as discussed more thoroughly in Section 5. For example, the Vault language [19, 20] was designed for coding low-level systems code, such as device drivers. The ability to track strong updates was crucial for ensuring that driver code respected certain protocols. As another example, Typed Assembly Language [30, 29] used strong updates to track the types of registers and stack slots. More recently, Foster and Aiken have presented a flow-sensitive qualifier system for

[1] We assume that values are represented uniformly so that, for instance, unit, booleans, and integers all take up one word of storage.

C, called CQUAL [5], which uses strong updates to track security-relevant properties in legacy C code.

Vault, later versions of TAL [28], and CQUAL all based their support for strong updates and alias control on the *Alias Types* formalism of Smith, Walker, and Morrisett [34, 38]. Though Alias Types were proven sound in a syntactic sense, we lacked an understanding of their *semantics*. For instance, it was noted that Alias Types seemed strongly related to the logic of Bunched Implications (BI) [26] and Reynolds' separation logic [32], but the lack of a semantic interpretation for Alias Types precluded a formal connection.

Furthermore, Vault, TAL, and CQUAL added a number of new extensions that were not handled by Alias Types. For instance, the `restrict` operator of CQUAL is unusual in that it allows a computation to temporarily gain exclusive ownership of a reference cell and perform strong updates, in spite of the fact that there may be unknown aliases to the object.

In this paper, we re-examine strong updates from a more foundational standpoint. In particular, we give an alternative formulation of Alias Types in the form of a core calculus based on standard linear logic, which yields an extremely clean semantic interpretation of the types that is directly related to the semantic model of BI. We show that our core calculus is sound and that every well-typed program terminates, in spite of the fact that the type system supports first-class, shared, mutable references with strong updates. We then show how the calculus can be extended to support a combination of ML-style references with uncontrolled aliasing and a `restrict`-like primitive for temporarily gaining exclusive ownership over such references to support strong updates. Proofs of theorems can be found in a companion technical report [3].

2 Linearity and Strong Updates

Linear types, which are derived from Girard's linear logic [21], have proven useful for modeling imperative programming languages in a functional setting [36, 31]. For instance, the Clean programming language [1] relies upon a form of linearity (or uniqueness) to ensure equational reasoning in the presence of mutable data structures (and other effects such as IO). The intuitive understanding is that a linear object cannot be duplicated, and thus there are no aliases to the object, so it is safe to perform updates in-place while continuing to reason equationally. For example, consider the following function:

```
fun setx (r : {x:T1,y:T2,z:T3}) (x1 : T1) =
  let val {x=x0,y=y0,z=z0} = r
  in  drop x0;
      {x=x1, y=y0, z=z0}
  end
```

The function takes a record `r` and a `T1` value, pulls out the components of the record, explicitly discards the old `x` component by passing it to `drop`, and then builds a new record out of the new `T1` value and the old `y` and `z` components. In a linear setting, this code should type-check since each non-trivial resource is used exactly once. Furthermore, from an operational standpoint, this code is almost equivalent to doing an in-place update of the `x` component of the point:

```
fun setx (r: {x:T1,y:T2,z:T3}) (x1 : T1) =
  r.x := x1
```

The reason in-place update is safe is that no one from the outside can have a copy of (a reference to) the argument record `r`. So, when `r` is deconstructed, we can safely re-use its storage to build the resulting record.

Note, however, that the imperative version is not exactly equivalent to the functional one. In particular, we must explicitly `drop` the old value of `x` to be faithful. As suggested by Baker [10], we can replace the update with a `swap`:

```
fun setx (r: {x:T1,y:T2,z:T3}) (x1 : T1) =
  let val x0 = swap r.x x1
  in  drop x0;
      r
  end
```

The expression `swap r.x x1` is meant to write `x1` into `r.x` and return the original value in `r.x`. Using `swap` instead of update ensures that we do not forget to use an old value. Furthermore, using `swap` instead of dereference for mutable cells ensures that resources are not duplicated. Thus, `swap` is the appropriate primitive to ensure the linearity of resources.

We remark that for resources that can be freely duplicated (e.g., integers), we can always simulate reads and writes by using swap in conjunction with a dummy value. Thus, in this paper, we will focus on `swap` as the only primitive for manipulating the contents of mutable objects.

From our standpoint, the biggest advantage of a linear interpretation is that there is no need for the type of a mutable object to remain invariant. In particular, there is nothing wrong with changing the functional version of `setx` so that we build a record where the `x` component has a completely different type from the original:

```
fun setx (r : {x:T1,y:T2,z:T3}) (x1 : T4) =
  let val {x=x0,y=y0,z=z0} = r
  in  drop x0;
    {x=x1, y=y0, z=z0}
  end
```

In this instance, `setx` has the type

$$\{\mathtt{x}:\mathtt{T1},\mathtt{y}:\mathtt{T2},\mathtt{z}:\mathtt{T3}\} \multimap \{\mathtt{x}:\mathtt{T4},\mathtt{y}:\mathtt{T2},\mathtt{z}:\mathtt{T3}\}$$

reflecting that the type of the `x` component changes. This suggests that the imperative version with `swap`, which is operationally equivalent, can also be allowed to change the type:

```
fun setx (r: {x:T1,y:T2,z:T3}) (x1 : T4) =
  let val x0 = swap r.x x1
  in  drop x0;
      r
  end
```

Changing the type of the contents of a mutable object is called a *strong update*. Strong updates are useful for reflecting protocols on resources whose state changes over time. Linear typing makes it particularly easy to support strong updates, as we showed above.

3 Core $\mathbf{L}^3$

Though a linear interpretation of reference cells supports strong updates, it is too restrictive for many, if not most, realistic programs. In particular, there is no facility for creating cyclic or even shared data structures, since each mutable object can have at most one reference to it. On the other hand, the presence of unrestricted aliasing breaks the soundness of strong updates, reflecting that the functional and imperative interpretations no longer coincide. Rather, in languages such as ML where references can be freely duplicated, we are forced to preserve the type of the contents at updates.

What is needed is some way to support the controlled duplication of references to mutable objects, while supporting strong updates. One approach, suggested by Alias Types [34], is to separate the typing components of a mutable object into two pieces: a *pointer* to the object, which can be freely duplicated, and a *capability* for accessing the contents of the object. The type of the capability records the current type of the contents of the object and thus must remain linear to ensure the soundness of strong updates.

As an example, consider a function that is to be passed a pair where the two components are aliases for the same location, and we wish to update the contents of the location, changing its type. In a language based on the calculus we describe below, the function might look something like this:

```
fun f (c1 : Cap ρ τ1)
      (p : !Ptr ρ ⊗ !Ptr ρ)
      (v : τ2) =
  let val (x,y) = p
      val (c2,z) = swap x (c1,v)
  in
      (c2, y, z)
  end
```

The pair p includes two pointers to the same abstract location ρ, and these pointers may be freely duplicated or forgotten since their types are under the "of course" constructor (i.e., !Ptr ρ). Note, however, that the types of the pointers say nothing about the type of the value to which they point, for this is ephemeral. Only the name of the object (ρ) is in any sense persistent.

In contrast, the value c1 is a capability for accessing the location ρ and records the current type of the contents of the location. Such a capability must be presented whenever we attempt to perform a swap operation on a pointer to ρ. The capability is then consumed and a new capability is returned. The original capability tells us what type of value was in the location before the swap (τ_1), whereas the newly returned capability tells us what type of value now resides in the location (τ_2, since that is the type of v). Thus, the function above has a signature like this:

$$\mathtt{f} : \forall\rho.\mathsf{Cap}\,\rho\,\tau_1 \multimap (!\mathsf{Ptr}\,\rho \otimes !\mathsf{Ptr}\,\rho) \multimap \tau_2 \multimap (\mathsf{Cap}\,\rho\,\tau_2 \otimes !\mathsf{Ptr}\,\rho \otimes \tau_1)$$

Our intention is that, like types, capabilities can be erased; at run-time, they play no computationally significant role.

In the original version of Alias Types, capabilities were collected into a global, implicit parameter that was threaded through the computation and assigned a global store type. Consequently, capabilities were not first-class and a complicated, second-order type system was needed to achieve sufficient abstraction for code re-use. Furthermore, the capabilities were effectively restricted to describing a finite frontier of reachable reference cells, so inductive data structures (e.g., a list of reference cells) were beyond the reach of the formalism. In later work, Walker and Morrisett [38] extended the approach to support first-class capabilities and inductive data structures, but the resulting language was extremely complicated.

In this section, we present a different formulation of Alias Types based on a relatively standard linear lambda calculus; we name our calculus $\mathbf{L}^3$(*Linear Language with Locations*). In $\mathbf{L}^3$, capabilities are explicit and first-class, which makes it simple to support inductively defined data structures. Furthermore, as in Alias Types, $\mathbf{L}^3$ supports multiple references to a given mutable object as well as strong updates. Somewhat surprisingly, the core language retains a simple semantics, which, for instance, allows us to prove that well-typed programs terminate. Thus, we believe that $\mathbf{L}^3$ is an appropriate foundation for strong updates in the presence of sharing.

3.1 Syntax

The syntax for core $\mathbf{L}^3$ is as follows:

$$\begin{array}{lll}
 & \textit{LocConsts} & \ell \in \textit{LocConsts} \\
 & \textit{LocVars} & \rho \in \textit{LocVars} \\
 & \textit{Locs} & \eta ::= \ell \mid \rho \\
\textit{Types} & & \\
\tau & ::= & \mathbf{1} \mid \tau_1 \otimes \tau_2 \mid \tau_1 \multimap \tau_2 \mid !\tau \mid \\
 & & \mathsf{Ptr}\,\eta \mid \mathsf{Cap}\,\eta\,\tau \mid \forall\rho.\tau \mid \exists\rho.\tau \\
\textit{Exprs} & & \\
e & ::= & \langle\rangle \mid \mathtt{let}\,\langle\rangle = e_1\,\mathtt{in}\,e_2 \mid \\
 & & \langle e_1, e_2\rangle \mid \mathtt{let}\,\langle x_1, x_2\rangle = e_1\,\mathtt{in}\,e_2 \mid \\
 & & x \mid \lambda x.e \mid e_1\,e_2 \mid \\
 & & !v \mid \mathtt{let}\,!x = e_1\,\mathtt{in}\,e_2 \mid \mathtt{dup}\,e \mid \mathtt{drop}\,e \mid \\
 & & \mathtt{ptr}\,\ell \mid \mathtt{cap}\,\ell \mid \\
 & & \mathtt{create}\,e \mid \mathtt{destroy}\,e \mid \mathtt{swap}\,e_1\,e_2 \mid \\
 & & \Lambda\rho.e \mid e[\eta] \mid \ulcorner\eta, e\urcorner \mid \mathtt{let}\,\ulcorner\rho, x\urcorner = e_1\,\mathtt{in}\,e_2 \\
\textit{Values} & & \\
v & ::= & \langle\rangle \mid \langle v_1, v_2\rangle \mid x \mid \lambda x.e \mid !v \mid \\
 & & \mathtt{ptr}\,\ell \mid \mathtt{cap}\,\ell \mid \Lambda\rho.e \mid \ulcorner\eta, v\urcorner
\end{array}$$

Most of the types, expressions, and values are based on a traditional, call-by-value, linear lambda calculus. In the fol-

Stores

$\sigma \quad ::= \quad \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\}$

Evaluation Contexts

$E \quad ::= \quad [\,] \mid \texttt{let}\ \langle\rangle = E\ \texttt{in}\ e \mid \langle E, e\rangle \mid \langle v, E\rangle \mid \texttt{let}\ \langle x_1, x_2\rangle = E\ \texttt{in}\ e \mid$
$E\,e \mid v\,E \mid \texttt{let}\ !x = E\ \texttt{in}\ e \mid \texttt{dup}\,E \mid \texttt{drop}\,E \mid$
$\texttt{create}\,E \mid \texttt{destroy}\,E \mid \texttt{swap}\,E\,e \mid \texttt{swap}\,v\,E \mid E\,[\ell] \mid \ulcorner \ell, E \urcorner \mid \texttt{let}\ \ulcorner \rho, x \urcorner = E\ \texttt{in}\ e$

(let-unit)	$(\sigma, \texttt{let}\ \langle\rangle = \langle\rangle\ \texttt{in}\ e)$	$\longmapsto$	(σ, e)
(let-pair)	$(\sigma, \texttt{let}\ \langle x_1, x_2\rangle = \langle v_1, v_2\rangle\ \texttt{in}\ e)$	$\longmapsto$	$(\sigma, e[v_1/x_1][v_2/x_2])$
(app)	$(\sigma, (\lambda x.\, e)\, v)$	$\longmapsto$	$(\sigma, e[v/x])$
(let-bang)	$(\sigma, \texttt{let}\ !x = !v\ \texttt{in}\ e)$	$\longmapsto$	$(\sigma, e[v/x])$
(dup)	$(\sigma, \texttt{dup}\,!v)$	$\longmapsto$	$(\sigma, \langle !v, !v\rangle)$
(drop)	$(\sigma, \texttt{drop}\,!v)$	$\longmapsto$	$(\sigma, \langle\rangle)$
(cre)	$(\sigma, \texttt{create}\,v)$	$\longmapsto$	$(\sigma \uplus \{\ell \mapsto v\}, \ulcorner \ell, \langle \texttt{cap}\,\ell, !(\texttt{ptr}\,\ell)\rangle \urcorner)$
(des)	$(\sigma \uplus \{\ell \mapsto v\}, \texttt{destroy} \ulcorner \ell, \langle \texttt{cap}\,\ell, !(\texttt{ptr}\,\ell)\rangle \urcorner)$	$\longmapsto$	$(\sigma, \ulcorner \ell, v \urcorner)$
(swap)	$(\sigma \uplus \{\ell \mapsto v_1\}, \texttt{swap}\,(\texttt{ptr}\,\ell)\,\langle \texttt{cap}\,\ell, v_2\rangle)$	$\longmapsto$	$(\sigma \uplus \{\ell \mapsto v_2\}, \langle \texttt{cap}\,\ell, v_1\rangle)$
(lapp)	$(\sigma, (\Lambda\rho.\, e)\,[\ell])$	$\longmapsto$	$(\sigma, e[\ell/\rho])$
(let-lpack)	$(\sigma, \texttt{let}\ \ulcorner \rho, x \urcorner = \ulcorner \ell, v \urcorner\ \texttt{in}\ e)$	$\longmapsto$	$(\sigma, e[\ell/\rho][v/x])$
(ctxt)	$\dfrac{(\sigma, e) \longmapsto (\sigma', e')}{(\sigma, E[e]) \longmapsto (\sigma', E[e'])}$		

Figure 1: Core $\mathbf{L}^3$– Operational Semantics

lowing sections, we will explain the bits that are new or different.

3.1.1 Types

The types **1**, $\tau_1 \otimes \tau_2$, $\tau_1 \multimap \tau_2$, and $!\tau$ are those found in the linear lambda calculus. The first three types are linear and must be eliminated exactly once. The pattern matching expression forms $\texttt{let}\ \langle\rangle = e_1\ \texttt{in}\ e_2$ and $\texttt{let}\ \langle x_1, x_2\rangle = e_1\ \texttt{in}\ e_2$ are used to eliminate unit (**1**) and tensor products ($\otimes$) respectively. As usual, a linear function $\tau_1 \multimap \tau_2$ is eliminated via application. The "of course" type $!\tau$ can be used to relax the linear restriction. A value of type $!\tau$ may be explicitly duplicated ($\texttt{dup}\,e$) or dropped ($\texttt{drop}\,e$). To put it another way, *weakening* and *contraction* of unrestricted $!\tau$ values is explicit, rather than implicit.

As mentioned earlier, we break a mutable object into two components: pointers ($\mathsf{Ptr}\,\eta$) to the object's location and a capability ($\mathsf{Cap}\,\eta\,\tau$) for accessing the contents of the location, via `swap`, to obtain a value of type τ. The link between these two components is the location's name: either a location constant ℓ or a location variable ρ. Location constants, comparable to physical memory addresses, are used internally by our operational semantics, but are not allowed in source programs. Instead, source programs manipulate location variables, which abstract over location constants. We use the meta-variable η to range over both location constants and location variables.

As noted above, we wish to allow the pointer to a location to be freely duplicated and discarded, but we must treat the capability as a linear value. The static semantics given in the next section will allow the introduction of values of type $!(\mathsf{Ptr}\,\eta)$ but will prevent the introduction of values of type $!(\mathsf{Cap}\,\eta\,\tau)$; furthermore, this will be consistent with the semantic interpretation of types, which will establish that $!(\mathsf{Ptr}\,\eta)$ is inhabited, while $!(\mathsf{Cap}\,\eta\,\tau)$ is uninhabited.

Quantification of location variables over types are given by the universal $\forall\rho.\tau$ and existential $\exists\rho.\tau$ types. Values of universal type must be instantiated and values of existential type must be opened. Types are considered equivalent up to renaming of bound location variables; in particular, note that location constants do not alpha-convert. We use the notation $FLV(\tau)$ to denote the free location variables of a type and the notation $\tau[\eta/\rho]$ to denote the standard capture-avoiding substitution of locations for location variables in types.

3.1.2 Expressions and Operational Semantics

It is useful to consider expressions along with their operational semantics. Figure 1 gives the small-step operational semantics for core $\mathbf{L}^3$ as a relation between configurations of the form (σ, e). In the configuration, σ is a global store mapping locations to closed values; note that a closed value has no free variables or location variables, but may have arbitrary location constants—even locations not in the domain of σ. The notation $\sigma_1 \uplus \sigma_2$ denotes the disjoint union of the

stores σ_1 and σ_2; the operation is undefined unless the domains of σ_1 and σ_2 are disjoint. We use evaluation contexts E to lift the primitive rewriting rules to a left-to-right, innermost to outer-most, call-by-value interpretation of the language.

Our calculus adopts many familiar terms from the linear lambda calculus. We already explained the introduction and elimination forms for unit, tensor products, and functions; their semantics is straightforward.

Four expression forms manipulate terms of !-type. If v is a value of type τ, then, under certain conditions specified by the static semantics, we can form $!v$ having the type $!\tau$. The restriction of $!\cdot$ to values corresponds to our desire to give the language a call-by-value operational semantics; it is possible to write expressions that can be (syntactically) duplicated, but which reduce to values that cannot be duplicated.[2] Note that $!\cdot$ is principally applied to functions and location abstractions, which are values. Note further that the static semantics prevents the application of $!\cdot$ to capabilities.

The elimination form $\texttt{let}\ !x = e_1\ \texttt{in}\ e_2$ evaluates e_1 to a value with type $!\tau$ and substitutes the value for x in e_2. The expressions $\texttt{dup}e$ and $\texttt{drop}e$ are the means by which values of !-type are explicitly duplicated and discarded. $\texttt{dup}e$ evaluates e to a value $!v$ and returns a linear pair, where each component has the value $!v$. Similarly, $\texttt{drop}e$ evaluates e to a value, which is discarded, and returns $\langle\rangle$.

There are expression forms for both the pointer to a location ($\texttt{ptr}\,\ell$) and the capability for a location ($\texttt{cap}\,\ell$). However, neither expression form is available in source programs. It may be somewhat surprising that the expression $\texttt{cap}\,\ell$ makes no mention of the *type* of the value stored at the location ℓ. However, as we will see shortly, $\texttt{cap}\,\ell$ terms have no operational significance; hence, any token would serve as the run-time representation of the knowledge of the type of the value stored at a location. On the other hand, we expect a well-typed program to give rise to exactly one $\texttt{cap}\,\ell$ term for each allocated location. By choosing a representation that includes the location name, this property can be checked "at a glance," as it were, of the untyped syntax of a program.

The expressions $\texttt{create}e$ and $\texttt{destroy}e$ perform the complementary actions of allocating and deallocating mutable references in the global store. $\texttt{create}e$ evaluates e to a value, allocates a fresh (unallocated) location ℓ, stores the value at that location, and returns the pair $\langle \texttt{cap}\,\ell, !(\texttt{ptr}\,\ell)\rangle$ in an existential package that hides the particular location ℓ. The static semantics will ensure that the type of $\texttt{cap}\,\ell$ "knows" the type of the value stored at ℓ. $\texttt{destroy}e$ performs the reverse. It evaluates e to the pair $\langle \texttt{cap}\,\ell, !(\texttt{ptr}\,\ell)\rangle$, extracts the value stored at ℓ, deallocates the location ℓ, and returns the value. We remark that deallocation can result in dangling pointers to a location, but that since the (unique) capability for that location is destroyed, those pointers can never be dereferenced.

The expression $\texttt{swap}\,e_1\,e_2$ combines the operations of dereferencing and updating a mutable reference as explained previously. The first expression evaluates to a pointer $\texttt{ptr}\,\ell$ and the second to a pair $\langle \texttt{cap}\,\ell, v\rangle$. The operation then swaps v for v' where v' is the value stored at ℓ, and returns $\langle \texttt{cap}\,\ell, v'\rangle$. Again, the static semantics will ensure that the type of the input $\texttt{cap}\,\ell$ "knows" the type of v' and the type of the output $\texttt{cap}\,\ell$ "knows" the type of v.

It is easily seen that the $\texttt{cap}\,\ell$ terms have no operational significance. That is, we could erase these terms without affecting our ability to evaluate the programs.

Finally, there are introduction and elimination forms for universal and existential location quantification. The expression $\Lambda\rho.\,e$ provides universal abstraction over a location and is eliminated with an explicit application of the form $e[\eta]$. The expression form $\ulcorner \eta, e \urcorner$ has the type $\exists\rho.\tau$ when e has the type τ with η substituted for ρ. The package can be opened with the expression form $\texttt{let}\ \ulcorner \rho, x \urcorner = e_1\ \texttt{in}\ e_2$. Note that the existential package $\ulcorner \eta, e \urcorner$ elides the $\texttt{as}\,\exists\rho.\tau$ clause found in most presentations of existential types. As we will invariably intend to hide all occurrences of η in the type of e, we adopt the more concise notation.

As usual, expressions are considered equivalent up to renaming of bound variables and bound location variables. We use the notation $e[v/x]$ (resp. $e[\eta/\rho]$) to denote the standard capture avoiding substitution of values (resp. locations) for variables (resp. location variables) in expressions.

3.2 Static Semantics

The type system for $\mathbf{L}^3$ must ensure that critical resources, such as capabilities, are not duplicated or dropped. Our type system is based on the linear lambda calculus and is thus relatively simple.

$\mathbf{L}^3$ typing judgments have the form $\Delta;\Gamma \vdash e : \tau$ where the contexts Δ and Γ are defined as follows:

$$\begin{array}{lrcl} \textit{Location Contexts} & \Delta & ::= & \bullet \mid \Delta, \rho \\ \textit{Value Contexts} & \Gamma & ::= & \bullet \mid \Gamma, x{:}\tau \end{array}$$

Thus, Δ is used to track the set of location variables in scope, whereas Γ, as usual, is used to track the set of variables in scope, as well as their types. We consider contexts to be ordered lists of assumptions. There may be at most one occurrence of a location variable ρ in Δ and, similarly, at most one occurrence of a variable x in Γ.

As is usual in a linear setting, our type system relies upon an operator $\Gamma_1 \uplus \Gamma_2 = \Gamma$ that splits the assumptions in Γ between the contexts Γ_1 and Γ_2:

$$\begin{array}{lcll} \bullet \uplus \bullet & = & \bullet & \\ (\Gamma_1, x{:}\tau) \uplus \Gamma_2 & = & (\Gamma_1 \uplus \Gamma_2), x{:}\tau & (x \notin \mathit{dom}(\Gamma_2)) \\ \Gamma_1 \uplus (\Gamma_2, x{:}\tau) & = & (\Gamma_1 \uplus \Gamma_2), x{:}\tau & (x \notin \mathit{dom}(\Gamma_1)) \end{array}$$

Splitting the context is necessary to ensure that a given resource is used by at most one sub-expression. Note that $\uplus$

[2] An alternative approach would be to take $!e$ as an irreducible value and to reduce e at the !-type elimination. However, this implicit suspension is less in the spirit of call-by-value.

$\boxed{\Delta;\Gamma \vdash e : \tau}$

$$(\mathsf{Unit})\ \frac{}{\Delta;\bullet \vdash \langle\rangle : \mathbf{1}} \qquad (\mathsf{Let\text{-}Unit})\ \frac{\Delta;\Gamma_1 \vdash e_1 : \mathbf{1} \qquad \Delta;\Gamma_2 \vdash e_2 : \tau}{\Delta;\Gamma_1 \uplus \Gamma_2 \vdash \mathtt{let}\ \langle\rangle = e_1\ \mathtt{in}\ e_2 : \tau} \qquad (\mathsf{Pair})\ \frac{\Delta;\Gamma_1 \vdash e_1 : \tau_1 \qquad \Delta;\Gamma_2 \vdash e_2 : \tau_2}{\Delta;\Gamma_1 \uplus \Gamma_2 \vdash \langle e_1, e_2\rangle : \tau_1 \otimes \tau_2}$$

$$(\mathsf{Let\text{-}Pair})\ \frac{\Delta;\Gamma_1 \vdash e_1 : \tau_1 \otimes \tau_2 \quad \Delta;\Gamma_2, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau}{\Delta;\Gamma_1 \uplus \Gamma_2 \vdash \mathtt{let}\ \langle x_1, x_2\rangle = e_1\ \mathtt{in}\ e_2 : \tau} \qquad (\mathsf{Var})\ \frac{FLV(\tau) \subseteq \Delta}{\Delta;\bullet, x{:}\tau \vdash x : \tau} \qquad (\mathsf{Fun})\ \frac{\Delta;\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Delta;\Gamma \vdash \lambda x.e : \tau_1 \multimap \tau_2}$$

$$(\mathsf{App})\ \frac{\Delta;\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Delta;\Gamma_2 \vdash e_2 : \tau_1}{\Delta;\Gamma_1 \uplus \Gamma_2 \vdash e_1\, e_2 : \tau_2} \qquad (\mathsf{Bang})\ \frac{\Delta;\Gamma \vdash v : \tau \qquad |\Gamma| = \bullet}{\Delta;\Gamma \vdash\ !v : !\tau} \qquad (\mathsf{Let\text{-}Bang})\ \frac{\Delta;\Gamma_1 \vdash e_1 : !\tau_1 \qquad \Delta;\Gamma_2, x{:}\tau_1 \vdash e_2 : \tau_2}{\Delta;\Gamma_1 \uplus \Gamma_2 \vdash \mathtt{let}\ !x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

$$(\mathsf{Dup})\ \frac{\Delta;\Gamma \vdash e : !\tau}{\Delta;\Gamma \vdash \mathtt{dup}\, e : !\tau \otimes !\tau} \qquad (\mathsf{Drop})\ \frac{\Delta;\Gamma \vdash e : !\tau}{\Delta;\Gamma \vdash \mathtt{drop}\, e : \mathbf{1}} \qquad (\mathsf{Create})\ \frac{\Delta;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \mathtt{create}\, e : \exists\rho.(\mathsf{Cap}\,\rho\,\tau \otimes !(\mathsf{Ptr}\,\rho))}$$

$$(\mathsf{Destroy})\ \frac{\Delta;\Gamma \vdash e : \exists\rho.(\mathsf{Cap}\,\rho\,\tau \otimes !(\mathsf{Ptr}\,\rho))}{\Delta;\Gamma \vdash \mathtt{destroy}\, e : \exists\rho.\tau} \qquad (\mathsf{Swap})\ \frac{\Delta;\Gamma_1 \vdash e_1 : \mathsf{Ptr}\,\rho \qquad \Delta;\Gamma_2 \vdash e_2 : \mathsf{Cap}\,\rho\,\tau_1 \otimes \tau_2}{\Delta;\Gamma_1 \uplus \Gamma_2 \vdash \mathtt{swap}\, e_1\, e_2 : \mathsf{Cap}\,\rho\,\tau_2 \otimes \tau_1} \qquad (\mathsf{LFun})\ \frac{\Delta,\rho;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \Lambda\rho.e : \forall\rho.\tau}$$

$$(\mathsf{LApp})\ \frac{\Delta;\Gamma \vdash e : \forall\rho.\tau \qquad \rho' \in \Delta}{\Delta;\Gamma \vdash e[\rho'] : \tau[\rho'/\rho]} \qquad (\mathsf{LPack})\ \frac{\rho' \in \Delta \qquad \Delta;\Gamma \vdash e : \tau[\rho'/\rho]}{\Delta;\Gamma \vdash \ulcorner\rho', e\urcorner : \exists\rho.\tau}$$

$$(\mathsf{Let\text{-}LPack})\ \frac{\Delta;\Gamma_1 \vdash e_1 : \exists\rho.\tau_1 \qquad FLV(\tau_2) \subseteq \Delta \qquad \Delta,\rho;\Gamma_2, x{:}\tau_1 \vdash e_2 : \tau_2}{\Delta;\Gamma_1 \uplus \Gamma_2 \vdash \mathtt{let}\ \ulcorner\rho, x\urcorner = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

Figure 2: Core $\mathbf{L}^3$– Static Semantics

splits all assumptions, even those of !-type. However, recall that contraction and weakening is supported for !-types through explicit operations.

Figure 2 presents the typing rules for $\mathbf{L}^3$. The first seven rules are the normal typing rules for a linear lambda calculus.

The (Bang) rule uses an auxiliary function on contexts to extract the linear components:

$$\begin{array}{lcl} |\bullet| & = & \bullet \\ |\Gamma, x{:}!\tau| & = & |\Gamma| \\ |\Gamma, x{:}\tau| & = & |\Gamma|, x{:}\tau \quad (\tau \neq\ !\tau') \end{array}$$

The rule requires that $|\Gamma|$ is empty. This ensures that the value v can be freely duplicated and discarded, without implicitly duplicating or discarding linear assumptions.

Note that there are no rules for $\mathtt{ptr}\,\ell$ or $\mathtt{cap}\,\ell$, as these expression forms are not present in the surface language. Likewise, all of the rules are given in terms of location variables ρ, and not in terms of location constants ℓ. Instead, the (Create), (Destroy), and (Swap) rules act as introduction and elimination rules for $\mathsf{Ptr}\,\rho$ and $\mathsf{Cap}\,\rho\,\tau$ types. Both (Create) and (Destroy) operate on existentially quantified capability/pointer pairs, which hides the location constant present in the operational semantics. Note that (Swap) maintains the linear invariant on capabilities by consuming a value of type $\mathsf{Cap}\,\rho\,\tau_1$ and producing a value of type $\mathsf{Cap}\,\rho\,\tau_2$.

The final four rules deal with universally and existentially quantified location variables. Antecedents of the form $\rho' \in \Delta$ assert that location variables in terms are bound. The (Let-LPack) rule contains the standard scoping condition that prevents ρ from appearing free in the result type τ_2.

3.3 Examples and Discussion

The core language thus presented is expressive enough to approximate the examples given in Sections 1 and 2. A simple linear reference can be viewed as a value of type

$$\mathsf{LRef}\,\tau \equiv \exists\rho.(\mathsf{Cap}\,\rho\,\tau \otimes\ !\mathsf{Ptr}\,\rho),$$

and we can lift the primitive `swap` to update a reference with

$$\begin{array}{l} \mathtt{lrswap} \equiv \lambda \mathtt{r}{:}\mathsf{LRef}\,\tau.\lambda \mathtt{x}{:}\tau'. \\ \quad \mathtt{let}\ \ulcorner\rho, \mathtt{cl}\urcorner = \mathtt{r}\ \mathtt{in} \\ \quad \mathtt{let}\ \langle \mathtt{c}_0, \mathtt{l}_0\rangle = \mathtt{cl}\ \mathtt{in} \\ \quad \mathtt{let}\ \langle \mathtt{l}_1, \mathtt{l}_2\rangle = \mathtt{dup}\,\mathtt{l}_0\ \mathtt{in} \\ \quad \mathtt{let}\ !\mathtt{l}_2' = \mathtt{l}_2\ \mathtt{in} \\ \quad \mathtt{let}\ \langle \mathtt{c}_1, \mathtt{y}\rangle = \mathtt{swap}\,\mathtt{l}_2'\,\langle \mathtt{c}_0, \mathtt{x}\rangle\ \mathtt{in} \\ \quad \langle \ulcorner\rho, \langle \mathtt{c}_1, \mathtt{l}_1\rangle\urcorner, \mathtt{y}\rangle \end{array}$$

However, by keeping $\mathsf{Cap}\,\rho\,\tau$ and $!\mathsf{Ptr}\,\rho$ packaged together, we lose any benefits of making $\mathsf{Ptr}\,\rho$ unrestricted. So, we consider an extended example, demonstrating the power of treating capabilities and pointers separately. In the interest of brevity and readability, we adopt the following conventions. First, the binding occurrence of a variable $\mathtt{x}$ with !-type is annotated as $\mathtt{x}^!$. Second, we elide $\mathtt{let}\ !x = \cdot\ \mathtt{in}\ \cdot$, $\mathtt{dup}\,\cdot$, and $\mathtt{drop}\,\cdot$ expressions, leaving the duplication, dereliction, and linearization of !-type variables implicit. Taken together, these two conventions mean that a variable introduced as $\mathtt{x}^!$ of type $!\tau$ may be used zero, one, or many times in its scope, including contexts requiring the type τ. Third, to make it easier for a reader to manually type-check expressions, we annotate successive "versions" of a linear variable with an integer superscript. Finally, we introduce the expression form $\mathtt{let}\ p = e_1\ \mathtt{in}\ e_2$, where p is a pattern, to abbreviate multiple elimination forms. All of these conventions can

easily be desugared back into the core language presented in the previous sections.

We consider the following well-typed core $\mathbf{L}^3$ program.

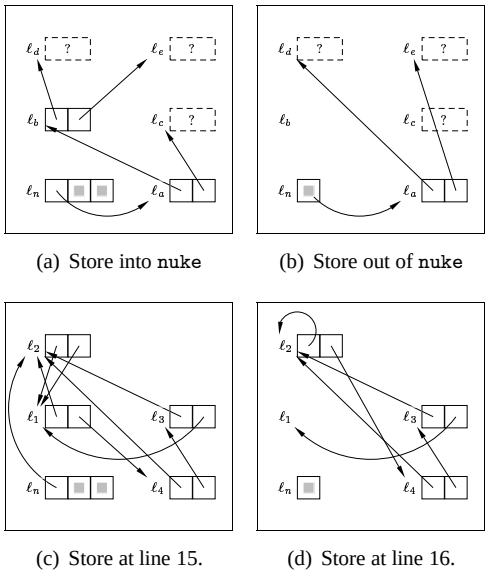
```
1.  let ⌜ρn,⟨c0n,p!n⟩⌝ = create ⟨⟩ in
2.  let nuke! = !Λρa,ρb,ρc,ρd,ρe.λc0n.
3.                let ⟨c1n,⟨p!a,⟨c0a,c0b⟩⟩⟩ = swap pn ⟨c0n,⟨⟩⟩ in
4.                let ⟨c1a,⟨p!b,p!c⟩⟩ = swap pa ⟨c0a,⟨⟩⟩ in
5.                let ⟨p!d,p!e⟩ = destroy ⌜ρb,⟨c0b,pb⟩⌝ in
6.                let ⟨c2a,⟨⟩⟩ = swap pa ⟨c1a,⟨pd,pe⟩⟩ in
7.                let ⟨c2n,⟨⟩⟩ = swap pn ⟨c1n,c2a⟩ in
8.                c2n in
9.  let ⌜ρ1,⟨c01,p!1⟩⌝ = create ⟨⟩ in
10. let ⌜ρ2,⟨c02,p!2⟩⌝ = create ⟨p1,p1⟩ in
11. let ⌜ρ3,⟨c03,p!3⟩⌝ = create ⟨p2,p1⟩ in
12. let ⌜ρ4,⟨c04,p!4⟩⌝ = create ⟨p2,p3⟩ in
13. let ⟨c11,⟨⟩⟩ = swap p1 ⟨c01,⟨p2,p4⟩⟩ in
14. let ⟨c1n,⟨⟩⟩ = swap pn ⟨c0n,⟨p2,⟨c02,c11⟩⟩⟩ in
15. let c2n = nuke[ρ2,ρ1,ρ1,ρ2,ρ4] c1n in
16. let ⟨c3n,c12⟩ = swap pn ⟨c2n,⟨⟩⟩ in
17. let ⟨c4n,⟨⟩⟩ = swap pn ⟨c3n,⟨p3,⟨c13,c12⟩⟩⟩ in
18. let c5n = nuke[ρ3,ρ2,ρ1,ρ2,ρ4] c4n in
19. let c23 = destroy ⌜ρn,⟨c5n,pn⟩⌝ in
20. let ⟨p!a,p!b⟩ = destroy ⌜ρ3,⟨c23,p3⟩⌝ in
21. let ⟨p!c,p!d⟩ = destroy ⌜ρ4,⟨c04,p4⟩⌝ in
22. ⟨⟩
```

Line 1 allocates a mutable reference, with pointer $\mathtt{p}_n$. Lines 2 through 8 define a function, `nuke`, parameterized by five locations. Note that the type of $\mathtt{c}_n$ on line 2 is

$$\mathsf{Cap}\,\rho_n\,(!(\mathsf{Ptr}\,\rho_a)\otimes(\mathsf{Cap}\,\rho_a\,(!(\mathsf{Ptr}\,\rho_b)\otimes!(\mathsf{Ptr}\,\rho_c))\otimes \mathsf{Cap}\,\rho_b\,(!(\mathsf{Ptr}\,\rho_d)\otimes!(\mathsf{Ptr}\,\rho_e)))).$$

This capability type describes the shape of the store reachable from $\mathtt{p}_n$ that will be necessary as a pre-condition for calling `nuke` (see Figure 3(a)). Note that there are no requirements on the contents of the locations named by ρ_c, ρ_d, and ρ_e, or even that those locations be allocated. In line 3, a pointer ($\mathtt{p}_a$) and two capabilities are read out of $\mathtt{p}_n$. In line 4, two additional pointers ($\mathtt{p}_b$ and $\mathtt{p}_c$) are read out of $\mathtt{p}_a$. At line 5, the reference pointed to by $\mathtt{p}_b$ is destroyed, which also extracts two pointers ($\mathtt{p}_d$ and $\mathtt{p}_e$) stored there. These last two pointers are written into $\mathtt{p}_a$. The final capability for the reference pointed to by $\mathtt{p}_a$ is written into $\mathtt{p}_n$, and the final capability for the reference pointed to by $\mathtt{p}_n$ is returned as the result of the function. The final shape of the store is given in Figure 3(b). Note that this function essentially uses the global reference pointed to by $\mathtt{p}_n$ as a means of passing arguments and returning results.

Lines 9 through 13 allocate four additional mutable references and construct a complex pointer graph. Line 14 copies a pointer ($\mathtt{p}_2$) and moves two capabilities ($\mathtt{c}_2$ and $\mathtt{c}_1$) into $\mathtt{p}_n$ in preparation for calling `nuke` at line 15. Figures 3(c) and 3(d) show the global store immediately before and after the function call. Note that the store contains cyclic pointers and that the function `nuke` performs a non-trivial alteration to the pointer graph, leaving a dangling pointer and introducing a self-loop. The remainder of the program makes one more call to `nuke` and then proceeds to destroy the remaining allocated references.



(a) Store into `nuke` (b) Store out of `nuke`

(c) Store at line 15. (d) Store at line 16.

Figure 3: Store during the evaluation of the example program.

As this example shows, the core language can give rise to complex pointer graphs. Also, note that in passing arguments and returning results through the reference pointed to by $\mathtt{p}_n$, the type of the reference necessarily changes; this is reflected in the successive types of $\mathtt{c}_n^{(i)}$. Recall that there are no extraneous antecedents in the typing judgments for location abstraction and instantiation. This means that there is no *a priori* reason to believe that the location variables $\rho_a,\ldots,\rho_e$ will not be instantiated with aliasing locations. In fact, lines 15 and 18 demonstrate that such is often the case. Hence, it is by no means clear that after deallocating the reference at $\mathtt{p}_b$ (line 5), it will be possible to swap into $\mathtt{p}_a$ (line 6). However, the linearity of capabilities ensures that it will not be possible to instantiate `nuke` with aliasing locations for ρ_a and ρ_b, since doing so would necessarily entail having two occurrences of the same capability – once through $\mathtt{c}_a$ and once through $\mathtt{c}_b$.

At this point (and probably at the end of Section 3.1.2), the astute reader might well question our claim that $\mathtt{cap}\,\ell$ terms have no operational significance. Couldn't we have equally well claimed that $\mathtt{ptr}\,\ell$ terms have no operational significance, as we could have chosen to let the $\mathtt{cap}\,\ell$ term denote the particular part of the store to be modified? Alternatively, couldn't we have given a store-passing semantics, in which a capability is represented as the store over which it has access? While these criticisms are valid, we believe that there are compelling reasons to take the view that $\mathtt{ptr}\,\ell$ terms are computationally significant and $\mathtt{cap}\,\ell$ are not.

Note, for instance, that line 16 simply shuffles capabilities for the purpose of type checking; an optimizing compiler might well elide this operation. Likewise, the argument and

$$
\begin{array}{rcl}
\mathcal{V}[\![\mathbf{1}]\!] & = & \{(\{\},\langle\rangle)\} \\
\mathcal{V}[\![\tau_1 \otimes \tau_2]\!] & = & \{(\sigma_1 \uplus \sigma_2, \langle v_1, v_2\rangle) \mid (\sigma_1, v_1) \in \mathcal{V}[\![\tau_1]\!] \wedge (\sigma_2, v_2) \in \mathcal{V}[\![\tau_2]\!]\} \\
\mathcal{V}[\![\tau_1 \multimap \tau_2]\!] & = & \{(\sigma_2, \lambda x.\, e) \mid \forall \sigma_1, v_1.\ (\sigma_1, v_1) \in \mathcal{V}[\![\tau_1]\!] \wedge \sigma_1 \uplus \sigma_2 \text{ defined} \Rightarrow (\sigma_1 \uplus \sigma_2, e[v_1/x]) \in \mathcal{C}[\![\tau_2]\!]\} \\
\mathcal{V}[\![!\tau]\!] & = & \{(\{\}, !v) \mid (\{\}, v) \in \mathcal{V}[\![\tau]\!]\} \\
\mathcal{V}[\![\mathsf{Ptr}\,\ell]\!] & = & \{(\{\}, \mathtt{ptr}\,\ell)\} \\
\mathcal{V}[\![\mathsf{Cap}\,\ell\,\tau]\!] & = & \{(\sigma \uplus \{\ell \mapsto v\}, \mathtt{cap}\,\ell) \mid (\sigma, v) \in \mathcal{V}[\![\tau]\!]\} \\
\mathcal{V}[\![\forall\rho.\tau]\!] & = & \{(\sigma, \Lambda\rho.\, e) \mid \forall \ell.\ (\sigma, e[\ell/\rho]) \in \mathcal{C}[\![\tau[\ell/\rho]]\!]\} \\
\mathcal{V}[\![\exists\rho.\tau]\!] & = & \{(\sigma, \ulcorner \ell, v \urcorner) \mid (\sigma, v) \in \mathcal{V}[\![\tau[\ell/\rho]]\!]\} \\
 & & \\
\mathcal{C}[\![\tau]\!] & = & \{(\sigma_s, e_s) \mid \forall \sigma_r.\ \sigma_s \uplus \sigma_r \text{ defined} \Rightarrow \exists n, \sigma_f, v_f.\ (\sigma_s \uplus \sigma_r, e_s) \longmapsto^n (\sigma_f \uplus \sigma_r, v_f) \wedge (\sigma_f, v_f) \in \mathcal{V}[\![\tau]\!]\} \\
 & & \\
\mathcal{S}[\![\bullet]\!]\delta & = & \{(\{\}, \emptyset)\} \\
\mathcal{S}[\![\Gamma, x{:}\tau]\!]\delta & = & \{(\sigma \uplus \sigma_x, \gamma[x \mapsto v_x]) \mid (\sigma, \gamma) \in \mathcal{S}[\![\Gamma]\!]\delta \wedge (\sigma_x, v_x) \in \mathcal{V}[\![\delta(\tau)]\!]\} \\
 & & \\
[\![\Delta; \Gamma \vdash e : \tau]\!] & = & \forall \delta, \sigma, \gamma.\ dom(\delta) = dom(\Delta) \wedge (\sigma, \gamma) \in \mathcal{S}[\![\Gamma]\!]\delta \Rightarrow (\sigma, \gamma(\delta(e))) \in \mathcal{C}[\![\delta(\tau)]\!]
\end{array}
$$

Figure 4: Core $\mathbf{L}^3$– Semantic Interpretations

result of `nuke` need not be passed at run time. Copying values into the reference pointed to by $\mathtt{p}_n$ need only copy the pointer, and not the two capabilities (lines 14 and 17).

Essentially, we note that linear values, by their very nature, must closely follow the control-flow path of a program. If $\mathtt{cap}\,\ell$ terms were operationally significant, represented, say, as a machine pointer, then copying them along the control-flow path of a program could be computationally expensive. On the other hand, $!(\mathtt{ptr}\,\ell)$ terms can be dispersed through a program by means of shared data-structures. As in the example above, it may be cheaper to "seed" shared data-structures with pointers early in a program, which can then be efficiently traversed, without incurring the (computational) overhead of threading linear capabilities to and from the points of use.

3.4 Semantic Interpretations

In this section, we give semantic interpretations to types and prove that the typing rules of Section 3.2 are sound with respect to these interpretations. We have also sketched a conventional syntactic proof of soundness, but found a semantic interpretation more satisfying for a few reasons. First, while shared $\mathtt{ptr}\,\ell$ values can be used to create cyclic pointer graphs, the linearity of $\mathtt{cap}\,\ell$ values prevents the construction of recursive functions through the standard "back-patching" technique. (The extension in Section 4 will relax this restriction, giving rise to a more powerful language.) Hence, our core language has the property that every well-typed term terminates, just as in linear lambda calculus without references [11]. Our semantic proof captures this property in the definition of the types, whereas the syntactic approach is too weak to show that this property holds. Second, the semantic approach avoids the need to define typing rules for various intermediate structures including stores. Rather, stores consistent with a particular type will be incorporated into the semantic interpretation of the type. Finally, the semantic interpretation will allow us some extra flexibility when we consider extensions to the language in the next section.

Figure 4 gives our semantic interpretations of types as values ($\mathcal{V}[\![\tau]\!]$), types as computations ($\mathcal{C}[\![\tau]\!]$), contexts as substitutions ($\mathcal{S}[\![\Gamma]\!]$), and finally a semantic interpretation of typing judgments. We remark that these definitions are well-founded since the interpretation of a type is defined in terms of the interpretations of strictly smaller types.

For any closed type τ, we choose its semantic value interpretation $\mathcal{V}[\![\tau]\!]$ to be a set (i.e., unary logical relation) of tuples of the form (σ, v), where v is a closed value and σ a store. We can think of σ as the "exclusive store" of the value, corresponding to the portion of the store over which the value has exclusive rights. This exclusivity is conveyed by the linear $\mathsf{Cap}\,\ell\,\tau$ type, whose interpretation demands that σ includes ℓ and maps it to a value of the appropriate type. This corresponds to the primitive "points-to" relation in BI.

On the other hand, the $\mathsf{Ptr}\,\ell$ type makes no demands on the structure of σ. It simply asserts that we have a copy of a reference to a particular location. As in BI, this is what allows us to have "dangling pointers".

Note that the interpretation of $\tau_1 \otimes \tau_2$ demands that the exclusive stores of the sub-values are disjoint. Thus, this type corresponds to the spatial conjunction of BI.

An unrestricted $!\tau$ corresponds to those values that have no exclusive store; hence, duplicating or discarding of $!\tau$ values does not implicitly duplicate or discard portions of the store. This is similar to the non-spatial relations and connectives embedded into BI.

Finally, the interpretations of function and abstraction types are given in terms of the interpretation of types as computations $\mathcal{C}[\![\tau]\!]$. The definition of $\mathcal{C}[\![\tau]\!]$ combines both termination and type preservation. A starting store and expression (σ_s, e_s) is a member of $\mathcal{C}[\![\tau]\!]$ if for every disjoint (rest of the) store σ_r, a finite number of reductions leads to a final store

and value (σ_f, v_f) that is a member of $\mathcal{V}[\![\tau]\!]$ and leaves σ_r unmodified. Notice that the computation interpretation corresponds to the frame axiom of BI, whereas the interpretation of linear implication is, as expected, in correspondence with BI's magic wand.

These interpretations on closed terms are lifted to open terms by means of an appropriate substitution, and additional piece of store. If δ is a mapping from location variables to location constants, then $\mathcal{S}[\![\Gamma]\!]\delta$ returns a store σ and a substitution γ. The substitution γ is a mapping from the variables in Γ to closed values, whereas σ is the disjoint union of the exclusive stores corresponding to each value in the range of γ. Intuitively, if we consider Γ as a set of assumptions, then occurrences of $\mathsf{Cap}\,\rho\,\tau$ types correspond to assumptions about the structure of the store. Thus, $\mathcal{S}[\![\Gamma]\!]\delta$ gives all consistent stores and substitutions that are compatible with the assumptions in Γ.

Finally, the semantic interpretation of a typing judgment $\Delta;\Gamma \vdash e : \tau$ is given by $[\![\Delta;\Gamma \vdash e : \tau]\!]$, which is a logical formula asserting that for all substitutions δ and γ and all stores σ compatible with Δ and Γ, $(\sigma, \gamma(\delta(e)))$ is a member of the interpretation of $\delta(\tau)$ as a computation.

Type soundness of the static semantics given in Section 3.2 amounts to showing that whenever we can derive $\Delta;\Gamma \vdash e : \tau$, we can prove $[\![\Delta;\Gamma \vdash e : \tau]\!]$. As an immediate corollary, for any well-typed closed expression e of type τ, we know that evaluating $(\{\}, e)$ terminates with a configuration (σ, v) in the value interpretation of τ.

Theorem 1 (Core L^3 Soundness)

If $\Delta;\Gamma \vdash e : \tau$, *then* $[\![\Delta;\Gamma \vdash e : \tau]\!]$.

Proof

By induction on the derivation $\Delta;\Gamma \vdash e : \tau$. □

Corollary 2

If $\bullet;\bullet \vdash e : \tau$, *then* $(\{\}, e) \longmapsto^* (\sigma_f, v_f)$ *and* $(\sigma_f, v_f) \in \mathcal{V}[\![\tau]\!]$.

Another interesting corollary is that if we run any closed, well-typed term of base type (e.g., **1**), then the resulting store will be empty. Thus, the expression will be forced to destroy any locations that it creates before terminating.

4 Extended L^3

Thus far, our language only supports linear capabilities. While this gives us the ability to do strong updates, and the separation of pointers and capabilities allows us to build interesting store graphs, we still cannot simulate ML-style references which are completely unrestricted. Such references are strictly more powerful than the linear references considered in the previous sections. Although an ML-style reference requires the cell to hold values of exactly one type, this is sufficient for building recursive computations. For example, we can write a divergent expression as follows:[3]

```
1.  let val r = ref (fn () => ())
2.      val g = fn () => (!r) ()
3.  in  r := g;
4.      g ()
5.  end
```

The unrestricted nature of ML-style references is crucial in this example: the reference `r` (holding a function of type `unit -> unit`), is used both in g's closure (line 2) and in the assignment at line 3.

In this section, we consider some minimal extensions needed for unrestricted references. At the same time, we are interested in modeling more recent languages, such as CQUAL, that support regaining (if only temporarily) a unique capability on an unrestricted reference so as to support strong updates.

One approach to modeling ML-style references is to add a new kind of unrestricted capability, with its own version of `swap`. To ensure soundness, the new swap would require that the value being swapped in to the location have the same type as the value currently in the location. This would ensure that the other capabilities for the location remained consistent with the current world. That is, unrestricted capabilities must have types that are *frozen* throughout their lifetime. An unrestricted, frozen capability could be created from a normal, linear capability. However, there could be no support for destroying a frozen location since this would invalidate the other capabilities for that location.

These additions to the language would be relatively straightforward, but we are also interested in supporting strong updates for unrestricted references. The extensions described below are inspired by CQUAL's `restrict` operator in that they allow an unrestricted, frozen capability to be temporarily "thawed" to a linear capability. This allows us to perform strong updates on the location.

In fact, these extensions obviate the need for a new `swap` on frozen capabilities – only thawed (linear) capabilities permit a swap, regardless of whether the content's type changes. Hence, the process of thawing a location demands exclusive access and thus the programmer must present evidence that no other frozen capability for the same location is currently thawed. In our extended language, this evidence is a value representing a proof that no other thawed location aliases the location on which we would like to do strong updates. There are many possible ways to prove such a fact, based on types or regions or some other partitioning of objects. Here, we do not commit to a particular logic so that the framework can be used in various settings. Rather, we use our semantic interpretation of types to specify a general condition so that admissible rules can be added to the type system without reproving soundness.

[3] Please note that this example is written with SML syntax, where `!` is the function to read the contents of a reference.

A thawed location can also be "re-frozen" in our extended language. This is meant to re-enable access to the location along a different frozen capability. However, to ensure soundness, the original frozen type must be re-established when we re-freeze. Together, thawing and re-freezing a location correspond to the lexically-scoped `restrict` of CQUAL. However, we are not limited to the last-in-first-out thawing and re-freezing imposed by a lexically-scoped discipline, and, indeed, there is no real requirement that a thawed location ever be re-frozen.

Finally, because frozen capabilities are unrestricted, we will require a frozen location to hold a value of !-type. This prevents a program from discarding a linear value by placing the (one and only) reference to the value in a frozen location and then discarding all capabilities to access the location.

4.1 Changes to Support the Extensions

The syntactic changes to support the extensions described above are as follows:

$$
\begin{array}{llll}
\textit{LocSets} & L & \in & \mathcal{P}(\textit{LocConsts}) \\
\textit{Thawed Contexts} & & & \\
 & \theta & ::= & \bullet \mid \theta, \eta{:}\tau \\
\textit{Types} & \tau & ::= & \ldots \mid \mathsf{Frzn}\,\eta\,\tau \mid \mathsf{Thwd}\,\theta \mid \mathsf{Notin}\,\eta\,\theta \\
\textit{Exprs} & e & ::= & \ldots \mid \mathtt{freeze}\,e_1\,e_2 \mid \mathtt{thaw}\,e_1\,e_2 \mid \\
 & & & \quad \mathtt{refreeze}\,e \mid \mathtt{frzn}\,\ell \mid \mathtt{thwd}\,L \\
\textit{Values} & v & ::= & \ldots \mid \mathtt{frzn}\,\ell \mid \mathtt{thwd}\,L \\
\textit{Evaluation Contexts} & & & \\
 & E & ::= & \ldots \mid \mathtt{freeze}\,E\,e \mid \mathtt{freeze}\,v\,E \mid \\
 & & & \quad \mathtt{thaw}\,E\,e \mid \mathtt{thaw}\,v\,E \mid \\
 & & & \quad \mathtt{refreeze}\,E
\end{array}
$$

The extended language is evaluated in the presence of a *frozen store* ϕ, which contains type-invariant mutable references, and a *linear store* σ. Figure 5 gives the small-step operational semantics for extended $\mathbf{L}^3$ as a relation between configurations of the form (ϕ, σ, e), where the two stores are necessarily disjoint. All of the operational semantics rules of core $\mathbf{L}^3$ carry over to the extended language by passing ϕ along unmodified. (However, note that (cre) must choose a fresh location not in the domain of either ϕ or σ.) The static semantics for the extended language consist of all the rules for the core language and the rules given in Figure 6.

The type $\mathsf{Frzn}\,\eta\,\tau$ is the type of a frozen capability for location η which in turn holds a value of type τ. The (internal) term $\mathtt{frzn}\,\ell$ represents such a capability. We allow frozen capabilities to occur under the !-constructor, and thus they can be duplicated or forgotten.

The type $\mathsf{Notin}\,\eta\,\theta$ represents a proof that the location η is not in the thawed context θ. As presented, our language has no terms of this type. Rather, our intention is that the type should only be inhabited by some value when indeed, the given location is not in the locations given by θ. For instance, in the next section, we will make use of a constant $\mathtt{void}_\eta$, which we could add to the language as a proof of the trivial fact that for all locations η, $\mathsf{Notin}\,\eta\,\bullet$.

$$
\begin{array}{ll}
\text{(freeze)} & (\phi, \sigma \uplus \{\ell \mapsto\, !v\}, \\
 & \quad \mathtt{freeze}\,\langle \mathtt{cap}\,\ell, \mathtt{thwd}\,L\rangle\, v') \\
 & \quad\quad \longmapsto (\phi \uplus \{\ell \mapsto\, !v\}, \sigma, \langle\, !(\mathtt{frzn}\,\ell), \mathtt{thwd}\,L\rangle) \\
\text{(thaw)} & (\phi \uplus \{\ell \mapsto\, !v\}, \sigma, \mathtt{thaw}\,\langle\, !(\mathtt{frzn}\,\ell), \mathtt{thwd}\,L\rangle\, v') \\
 & \quad\quad \longmapsto (\phi, \sigma \uplus \{\ell \mapsto\, !v\}, \langle \mathtt{cap}\,\ell, \mathtt{thwd}\,(L \uplus \{\ell\})\rangle) \\
\text{(refreeze)} & (\phi, \sigma \uplus \{\ell \mapsto\, !v\}, \\
 & \quad \mathtt{refreeze}\,\langle \mathtt{cap}\,\ell, \mathtt{thwd}\,(L \uplus \{\ell\})\rangle) \\
 & \quad\quad \longmapsto (\phi \uplus \{\ell \mapsto\, !v\}, \sigma, \langle\, !(\mathtt{frzn}\,\ell), \mathtt{thwd}\,L\rangle) \\
\text{(ctxt)} & \dfrac{(\phi, \sigma, e) \;\longmapsto\; (\phi', \sigma', e')}{(\phi, \sigma, E[e]) \;\longmapsto\; (\phi', \sigma', E[e'])}
\end{array}
$$

Figure 5: Extended $\mathbf{L}^3$– Operational Semantics

A value of type $\mathsf{Thwd}\,\theta$ is called a *thaw token* and is used to record the current set of frozen locations that have been thawed, as well as their original types. The term $\mathtt{thwd}\,L$ is used to represent a thaw token. In a given program, there will be at most one thaw token value that must be effectively threaded through the execution. Thus, $\mathsf{Thwd}\,\theta$ values must be treated linearly. An initial thaw token of type $\mathsf{Thwd}\,\bullet$ is made available at the start of a program's execution.

The `thaw` operation takes as its first argument a pair of a frozen capability for a location ($!\mathsf{Frzn}\,\eta\,\tau$) and the current thaw token ($\mathsf{Thwd}\,\theta$). The second argument is a proof that the location has not already been thawed ($\mathsf{Notin}\,\eta\,\theta$). The operation returns a linear capability ($\mathsf{Cap}\,\eta\,\tau$) and a new thaw token of type $\mathsf{Thwd}\,(\theta, \eta{:}\tau)$. In thawing a location, the operational semantics transfers the location from the frozen store to the linear store. This is a technical device that keeps the current state of a location manifest in the semantics; a real implementation would maintain a single, global store with all locations.

The `refreeze` operation takes a linear capability of type $\mathsf{Cap}\,\eta\,\tau$ and a thaw token of type $\mathsf{Thwd}\,(\theta, \eta{:}\tau)$ and returns a frozen capability with type $!\mathsf{Frzn}\,\eta\,\tau$ and the updated thaw token of type $\mathsf{Thwd}\,\theta$. Note that to re-freeze, the type of the capability's contents must match the type associated with the location in the thaw token.

Finally, a frozen capability of type $!\mathsf{Frzn}\,\eta\,\tau$ is created with the `freeze` operation. The first argument to `freeze` is a pair of a linear capability for a location ($\mathsf{Cap}\,\eta\,\tau$) and the current thaw token ($\mathsf{Thwd}\,\theta$). The other argument is a value of type $\mathsf{Notin}\,\eta\,\theta$ ensuring that the location being frozen is not in the current thawed set; currently thawed locations should be re-frozen to match the type of any frozen aliases. Note that `freeze` returns the thaw token unchanged.

Both `freeze` and `refreeze` have the operational effect of moving a location from the linear store to the frozen store.

$\boxed{\Delta;\Gamma \vdash e : \tau}$

$$(\textsf{Freeze})\ \frac{\Delta;\Gamma_1 \vdash e_1 : \mathsf{Cap}\,\rho\,!\tau \otimes \mathsf{Thwd}\,\theta \qquad \Delta;\Gamma_2 \vdash e_2 : \mathsf{Notin}\,\rho\,\theta}{\Delta;\Gamma_1 \boxplus \Gamma_2 \vdash \mathtt{freeze}\,e_1\,e_2 : !(\mathsf{Frzn}\,\rho\,!\tau) \otimes \mathsf{Thwd}\,\theta} \qquad (\textsf{Thaw})\ \frac{\Delta;\Gamma_1 \vdash e_1 : !(\mathsf{Frzn}\,\rho\,!\tau) \otimes \mathsf{Thwd}\,\theta \qquad \Delta;\Gamma_2 \vdash e_2 : \mathsf{Notin}\,\rho\,\theta}{\Delta;\Gamma_1 \boxplus \Gamma_2 \vdash \mathtt{thaw}\,e_1\,e_2 : \mathsf{Cap}\,\rho\,!\tau \otimes \mathsf{Thwd}\,(\theta,\rho{:}!\tau)}$$

$$(\textsf{Refreeze})\ \frac{\Delta;\Gamma \vdash e : \mathsf{Cap}\,\rho\,!\tau \otimes \mathsf{Thwd}\,(\theta,\rho{:}!\tau)}{\Delta;\Gamma \vdash \mathtt{refreeze}\,e : !(\mathsf{Frzn}\,\rho\,!\tau) \otimes \mathsf{Thwd}\,\theta}$$

Figure 6: Extended $\mathbf{L}^3$– Additional Static Semantics

4.2 Examples and Discussion

The extended language is now expressive enough to encode the example given at the beginning of this section. An ML-style reference can be viewed as a value of type:

$$\mathsf{Ref}\,!\tau \equiv !\exists\rho.(!\mathsf{Frzn}\,\rho\,!\tau \otimes !\mathsf{Ptr}\,\rho).$$

Next, we need to give `read` and `write` operations on references. We consider a simple scenario in which a frozen capability is thawed exactly for the duration of a `read` or `write`; hence, we will assume that the thaw token has type $\mathsf{Thwd}\,\bullet$ at the start of the operation and we will return the thaw token with this type at the conclusion of the operation. Recall that we take $\mathtt{void}_\eta$ as a constant term of type $\mathsf{Notin}\,\eta\,\bullet$, which suffices given our assumed type of the thawed token.

$$\begin{aligned}
\mathtt{read} \equiv\ & \lambda \mathtt{r}^!{:}\mathsf{Ref}\,!\tau.\,\lambda \mathtt{t}^0{:}\mathsf{Thwd}\,\bullet.\\
& \mathtt{let}\ \ulcorner \rho, \langle \mathtt{f}_a^!, \mathtt{l}^! \rangle \urcorner = \mathtt{r}\ \mathtt{in}\\
& \mathtt{let}\ \langle \mathtt{c}^1, \mathtt{t}^1 \rangle = \mathtt{thaw}\,\langle \mathtt{f}_a, \mathtt{t}^0 \rangle\,\mathtt{void}_\rho\ \mathtt{in}\\
& \mathtt{let}\ \langle \mathtt{c}^2, \mathtt{x}^! \rangle = \mathtt{swap}\,\mathtt{l}\,\langle \mathtt{c}^1, \langle\rangle \rangle\ \mathtt{in}\\
& \mathtt{let}\ \langle \mathtt{c}^3, \langle\rangle \rangle = \mathtt{swap}\,\mathtt{l}\,\langle \mathtt{c}^2, \mathtt{x} \rangle\ \mathtt{in}\\
& \mathtt{let}\ \langle \mathtt{f}_b^!, \mathtt{t}^2 \rangle = \mathtt{refreeze}\,\langle \mathtt{c}^3, \mathtt{t}^1 \rangle\ \mathtt{in}\\
& \langle \mathtt{x}, \mathtt{t}^2 \rangle
\end{aligned}$$

$$\begin{aligned}
\mathtt{write} \equiv\ & \lambda \mathtt{r}^!{:}\mathsf{Ref}\,!\tau.\,\lambda \mathtt{z}^!{:}!\tau.\,\lambda \mathtt{t}^0{:}\mathsf{Thwd}\,\bullet.\\
& \mathtt{let}\ \ulcorner \rho, \langle \mathtt{f}_a^!, \mathtt{l}^! \rangle \urcorner = \mathtt{r}\ \mathtt{in}\\
& \mathtt{let}\ \langle \mathtt{c}^1, \mathtt{t}^1 \rangle = \mathtt{thaw}\,\langle \mathtt{f}_a, \mathtt{t}^0 \rangle\,\mathtt{void}_\rho\ \mathtt{in}\\
& \mathtt{let}\ \langle \mathtt{c}^2, \mathtt{x}^! \rangle = \mathtt{swap}\,\mathtt{l}\,\langle \mathtt{c}^1, \mathtt{z} \rangle\ \mathtt{in}\\
& \mathtt{let}\ \langle \mathtt{f}_b^!, \mathtt{t}^2 \rangle = \mathtt{refreeze}\,\langle \mathtt{c}^2, \mathtt{t}^1 \rangle\ \mathtt{in}\\
& \mathtt{t}^2
\end{aligned}$$

It is easy to see how these operations can be combined to reconstruct the divergent computation by "back-patching" an unrestricted reference.

As the extended $\mathbf{L}^3$ is strictly more powerful than the core language given previously, the semantic interpretation given in Section 3.4 will not suffice as a model. We turn our attention to a more sophisticated semantic interpretation in the next section.

4.3 Semantic Interpretations

The model we use here is based on the indexed model of references developed by Ahmed, Appel, and Virga [2, 4], which in turn extends Appel and McAllester's indexed model [7] of recursive types. We begin by summarizing the intuitions underlying these models. Appel and McAllester's semantic interpretation of a (closed) type $\mathcal{V}[\![\tau]\!]$ is a set of pairs (k,v), where k is a natural number and v is a value. The intuitive idea is that in any computation running for no more than k steps, the value v cannot be distinguished from values of type τ. For instance, no computation that runs for only one step can distinguish between a pair of booleans and a pair of integers, because in one step the computation can only extract the values in the pair and not perform any type-specific operation on the values. A pair (k,e) is a member of $\mathcal{C}[\![\tau]\!]$ if e cannot get stuck within k steps, and furthermore, if e reduces in $j < k$ steps to a value v then $(k-j,v) \in \mathcal{V}[\![\tau]\!]$.They call k the *approximation index*.

To model ML-style references, one could define the semantic interpretation of a type $\mathcal{V}[\![\tau]\!]$ as a set of pairs (Ψ,v), where Ψ is a *store typing* that maps store locations to (the semantic interpretations of) their designated types. This, however, would mean that *the interpretations of types* must be predicates on partial functions from locations to *the interpretations of types*. A simple diagonalization argument will show that the set of type interpretations has an inconsistent cardinality.

Ahmed, Appel, and Virga show how to eliminate this circularity using the notion of approximations from the indexed model. They define the semantic interpretation of a type $\mathcal{V}[\![\tau]\!]$ as a set of tuples of the form (k,Ψ,v), where Ψ maps locations to (the interpretations of) their designated types to approximation $k-1$. Intuitively, this says that to determine whether v has type τ for k steps, we only need to know what type the contents of each store location will have for $k-1$ steps. This ensures that the model is well-founded.

For any closed type τ in extended $\mathbf{L}^3$, its semantic interpretation $\mathcal{V}[\![\tau]\!]$ is a set of tuples of the form (k,Ψ,ζ,σ,v). Here k is the approximation index; Ψ is a store typing that maps frozen locations (including locations that are currently thawed) to the semantic interpretations of their frozen types (to approximation $k-1$); ζ denotes either the set of currently thawed locations or $\perp$; σ is a linear store; and v is a value.

As for core $\mathbf{L}^3$, we consider σ to be the exclusive store of the value. The lifted thaw set ζ models the thaw token which the value may or may not have. Since there is at most one thaw token that must be threaded through the program, there is just one global thaw set. Hence, we define a partial

$$(k,\Psi) \sqsubseteq (j,\Psi') \stackrel{\text{def}}{=} j \le k \wedge \forall \ell \in dom(\Psi).\ \lfloor \Psi \rfloor_j(\ell) = \lfloor \Psi' \rfloor_j(\ell)$$

$$\begin{array}{rcl}
\mathcal{V}[\![\mathbf{1}]\!] & = & \{(k,\Psi,\bot,\{\},\langle\rangle)\} \\
\mathcal{V}[\![\tau_1 \otimes \tau_2]\!] & = & \{(k,\Psi,\zeta_1 \circledast \zeta_2,\sigma_1 \uplus \sigma_2,\langle v_1,v_2\rangle) \mid (k,\Psi,\zeta_1,\sigma_1,v_1) \in \mathcal{V}[\![\tau_1]\!] \wedge (k,\Psi,\zeta_2,\sigma_2,v_2) \in \mathcal{V}[\![\tau_2]\!]\} \\
\mathcal{V}[\![\tau_1 \multimap \tau_2]\!] & = & \{(k,\Psi,\zeta_2,\sigma_2,\lambda x.e) \mid \\
 & & \quad \forall \Psi',\zeta_1,\sigma_1,v_1,j<k. \\
 & & \quad\ (\,(k,\Psi) \sqsubseteq (j,\Psi') \wedge (j,\Psi',\zeta_1,\sigma_1,v_1) \in \mathcal{V}[\![\tau_1]\!] \wedge \zeta_1 \circledast \zeta_2 \text{ defined} \wedge \sigma_1 \uplus \sigma_2 \text{ defined}\,) \Rightarrow \\
 & & \quad\ (j,\Psi',\zeta_1 \circledast \zeta_2,\sigma_1 \uplus \sigma_2,e[v_1/x]) \in \mathcal{C}[\![\tau_2]\!]\} \\
\mathcal{V}[\![!\tau]\!] & = & \{(k,\Psi,\bot,\{\},!v) \mid (k,\Psi,\bot,\{\},v) \in \mathcal{V}[\![\tau]\!]\} \\
\mathcal{V}[\![\mathsf{Ptr}\,\ell]\!] & = & \{(k,\Psi,\bot,\{\},\mathtt{ptr}\,\ell)\} \\
\mathcal{V}[\![\mathsf{Cap}\,\ell\,\tau]\!] & = & \{(k,\Psi,\zeta,\sigma \uplus \{\ell \mapsto v\},\mathtt{cap}\,\ell) \mid \forall j<k.\ (j,\lfloor\Psi\rfloor_j,\zeta,\sigma,v) \in \mathcal{V}[\![\tau]\!]\} \\
\mathcal{V}[\![\forall\rho.\tau]\!] & = & \{(k,\Psi,\zeta,\sigma,\Lambda\rho.e) \mid \forall j,\Psi',\ell.\ (k,\Psi) \sqsubseteq (j,\Psi') \Rightarrow (j,\Psi',\zeta,\sigma,e[\ell/\rho]) \in \mathcal{C}[\![\tau[\ell/\rho]]\!]\} \\
\mathcal{V}[\![\exists\rho.\tau]\!] & = & \{(k,\Psi,\zeta,\sigma,\ulcorner\ell,v\urcorner) \mid (k,\Psi,\zeta,\sigma,v) \in \mathcal{V}[\![\tau[\ell/\rho]]\!]\} \\
\mathcal{V}[\![\mathsf{Frzn}\,\ell\,\tau]\!] & = & \{(k,\Psi,\bot,\{\},\mathtt{frzn}\,\ell) \mid \lfloor\Psi\rfloor_k(\ell) = \lfloor\mathcal{V}[\![\tau]\!]\rfloor_k\} \\
\mathcal{V}[\![\mathsf{Thwd}\,\theta]\!] & = & \{(k,\Psi,L,\{\},\mathtt{thwd}\,L) \mid \forall \ell \in L.\lfloor\Psi\rfloor_k(\ell) = \lfloor\mathcal{V}[\![\theta(\ell)]\!]\rfloor_k\} \\
\mathcal{V}[\![\mathsf{Notin}\,\ell\,\theta]\!] & = & \{(k,\Psi,\bot,\{\},v) \mid \ell \notin dom(\theta)\}
\end{array}$$

Figure 7: Extended $\mathbf{L}^3$: Semantic Interpretations (Values)

function for merging thaw sets, written $\zeta_1 \circledast \zeta_2$, as follows:

$$\zeta \in (\mathcal{P}(LocConsts))_\bot$$

$$\begin{array}{c} \zeta \circledast \bot = \zeta \\ \bot \circledast \zeta = \zeta \end{array}$$

We use the meta-variable D to denote sets of tuples of the form (k,Ψ,ζ,σ,v). For any such set D, we define the k-approximation of the set as the subset of its elements whose index is less than k. We also extend this notion point-wise to mappings Ψ from locations to sets D.

$$\begin{array}{rcl}
\lfloor D \rfloor_k & \stackrel{\text{def}}{=} & \{(j,\Psi,\zeta,\sigma,v) \mid j<k \wedge (j,\Psi,\zeta,\sigma,v) \in D\} \\
\lfloor \Psi \rfloor_k & \stackrel{\text{def}}{=} & \{\ell \mapsto \lfloor D \rfloor_k \mid \ell \in dom(\Psi) \wedge \Psi(\ell) = D\}
\end{array}$$

For the model to be well-founded, it must be the case that $(k,\Psi,\zeta,\sigma,v) \in \mathcal{V}[\![\tau]\!]$ if and only if $(k,\lfloor\Psi\rfloor_k,\zeta,\sigma,v) \in \mathcal{V}[\![\tau]\!]$. Figure 7 gives our semantic interpretation of (closed) types as values ($\mathcal{V}[\![\tau]\!]$). The reader may check that for each type τ, if the definition of $(k,\Psi,\zeta,\sigma,v) \in \mathcal{V}[\![\tau]\!]$ needs to look up the type of a location in Ψ, this is always written $\lfloor\Psi\rfloor_i(\ell)$ where $i \le k$.

Note that the interpretation of $\mathbf{1}$ makes no demands of the thaw set or the linear store. The same is true of the interpretations of $\mathsf{Ptr}\,\ell$ and $!\tau$, and also of the interpretations of $\mathsf{Frzn}\,\ell\,\tau$ and $\mathsf{Notin}\,\ell\,\tau$. Values of each of these types may be (explicitly converted to values of !-type and then) duplicated or discarded without duplicating or discarding the thaw token or portions of the store.

The interpretation of the type $\tau_1 \otimes \tau_2$ demands that at least one of the thaw sets of its sub-values be $\bot$, just as it demands that their exclusive stores be disjoint.

The interpretation of $\mathsf{Frzn}\,\ell\,\tau$ requires that the type of ℓ specified by Ψ match τ to approximation k. Intuitively, we cannot access the value stored at location ℓ in the frozen store without at least a `swap`, which consumes one step so that we only need to know that the contents at ℓ approximates type τ for another $k-1$ steps.

The interpretation of $\mathsf{Thwd}\,\theta$ requires simply that the type of every thawed location according to θ is consistent (to approximation k) with the type of the location according to Ψ. This ensures that when we move a location from the linear store back to the frozen store— recall that this requires that the location contain a value of type $\theta(\ell)$ — we end up with a frozen store where every location contains the type mandated by Ψ (to an appropriate approximation).

The interpretation of $\mathsf{Notin}\,\ell\,\theta$ demands that ℓ not be mapped by θ. This means that whenever θ is identical to the thawed context for the current thaw token, we essentially have a proof that ℓ is not a thawed location. Note that the static semantics for `freeze` and `thaw` ensure that the θ in the type of the current thaw token matches the θ for which we demand the proof.

The interpretation of $\mathsf{Cap}\,\ell\,\tau$ places similar demands on the linear store as it did for core $\mathbf{L}^3$. Also, its thaw set is precisely the thaw set of the value stored at ℓ in σ. Furthermore, it only requires that this value have the type τ for upto $k-1$ steps, intuitively, because accessing the value stored at ℓ in σ requires a `swap`, which consumes one step.

Since functions and type abstractions are suspended computations, their interpretations are a bit more involved. An abstraction $\Lambda\rho.e$ is in the interpretation of a type $\forall\rho.\tau$ for k steps if, at some future point in the program, say when we have $j<k$ steps left to execute, $e[\ell/\rho]$ is in the interpretation of computations of type $\tau[\ell/\rho]$. Before we reach that future point, however, the program may have frozen new locations, so that the frozen store typing Ψ' at that future point may map more locations than the current Ψ.

We define a relation $(k,\Psi) \sqsubseteq (j,\Psi')$ (see Figure 7) which specifies the relationship between frozen store typings as we go from a state in which we can safely execute k more steps

$$
\begin{array}{rcl}
\phi :_k \Psi \setminus \zeta & \stackrel{\text{def}}{=} & \zeta \neq \bot \wedge dom(\Psi) = dom(\phi) \uplus \zeta \wedge \forall j < k.\, \forall \ell \in dom(\phi).\, (j, \lfloor \Psi \rfloor_j, \bot, \{\}, \phi(\ell)) \in \lfloor \Psi \rfloor_k(\ell) \\
\\
\mathcal{C}[\![\tau]\!] & = & \{(k, \Psi_s, \zeta_s, \sigma_s, e_s) \mid \\
& & \quad \forall j, \phi_s, \phi_f, \zeta_r, \sigma_r, \sigma', e_f. \\
& & \quad\; (0 \leq j < k \wedge \phi_s :_k \Psi_s \setminus (\zeta_s \circledast \zeta_r) \wedge (\phi_s, \sigma_s \uplus \sigma_r, e_s) \longmapsto^j (\phi_f, \sigma', e_f) \wedge irred(\phi_f, \sigma', e_f)) \Rightarrow \\
& & \quad\; \exists \Psi_f, \zeta_f, \sigma_f. \\
& & \quad\;\; (k, \Psi_s) \sqsubseteq (k-j, \Psi_f) \wedge \phi_f :_{k-j} \Psi_f \setminus (\zeta_f \circledast \zeta_r) \wedge \sigma' = \sigma_f \uplus \sigma_r \wedge (k-j, \Psi_f, \zeta_f, \sigma_f, e_f) \in \mathcal{V}[\![\tau]\!]\} \\
\\
\mathcal{S}[\![\bullet]\!]\delta & = & \{(k, \Psi, \bot, \{\}, \emptyset)\} \\
\mathcal{S}[\![\Gamma, x{:}\tau]\!]\delta & = & \{(k, \Psi, \zeta \circledast \zeta_x, \sigma \uplus \sigma_x, \gamma[x \mapsto v_x]) \mid (k, \Psi, \zeta, \sigma, \gamma) \in \mathcal{S}[\![\Gamma]\!]\delta \wedge (k, \Psi, \zeta_x, \sigma_x, v_x) \in \mathcal{V}[\![\delta(\tau)]\!]\} \\
\\
[\![\Delta; \Gamma \vdash e : \tau]\!] & = & \forall k \geq 0.\, \forall \delta, \gamma, \Psi, \zeta, \sigma.\, dom(\delta) = dom(\Delta) \wedge (k, \Psi, \zeta, \sigma, \gamma) \in \mathcal{S}[\![\Gamma]\!]\delta \Rightarrow (k, \Psi, \zeta, \sigma, \gamma(\delta(e))) \in \mathcal{C}[\![\delta(\tau)]\!]
\end{array}
$$

Figure 8: Extended $\mathbf{L}^3$: Semantic Interpretations (Computations & Judgments)

to a state where we can safely execute $j \leq k$ more steps. We require that the types of all locations in Ψ be preserved in Ψ', though only to approximation j. Note that $dom(\Psi')$ may be a superset of $dom(\Psi)$.

A property crucial for the soundness of our model is the transitivity of the $\sqsubseteq$ relation. This allows us to prove Kripke monotonicity which we require since our model is a possible-worlds semantics. Intuitively, to model ML-style references, we must ensure that types are preserved. Since Ψ tells us what the frozen type of each frozen location should be, the types of locations in Ψ must be preserved. This is the reason that, when we thaw a location ℓ, although the dynamic semantics moves ℓ from the linear to the frozen store, we do not remove ℓ from the store typing.

In order to track how far "out of synch" the frozen store ϕ is with respect to the store typing Ψ, we define the relation $\phi :_k \Psi \setminus \zeta$ (see Figure 8). Informally, this says that the frozen store ϕ is well-typed (to approximation k) with respect to the store typing Ψ modulo the current set of thawed locations ζ. The relation requires that Ψ specify types for all locations in the frozen store as well as locations that are currently thawed. Also, it says that the contents of every location in the frozen store must have the type specified by Ψ. Note that it does not require that the contents of thawed locations have the types specified by Ψ. That condition is, however, checked upon `refreeze`.

The interpretation of functions makes use of reasoning similar to that for abstractions and also demands that the thaw set and exclusive store of the argument to the function be disjoint from the thaw set and exclusive store of the function itself. Furthermore, note that it requires that the argument to the function (v_1) must have type τ_1 for only $j < k$ steps (rather than $j \leq k$ steps). Intuitively, this is sufficient since beta-reduction consumes a step.

Informally, a tuple $(k, \Psi_s, \zeta_s, \sigma_s, e_s)$ is a member of $\mathcal{C}[\![\tau]\!]$ if for every frozen store ϕ_s and every disjoint (rest of the) thaw set ζ_r and linear store σ_r, such that ϕ_s is well-typed with respect to Ψ_s (ignoring the set of currently thawed locations $\zeta_s \circledast \zeta_r$), and the computation reaches an irreducible (final) state (ϕ_f, σ', e_f) in j steps, then the following conditions hold. First, it must be that σ_r is unmodified, that is, σ' must equal $\sigma_f \uplus \sigma_r$. Second, the (rest of the) thaw set ζ_r should also be unmodified. Note that this ensures that a computation whose starting exclusive thaw set ζ_s is $\bot$ cannot modify the global set of thawed locations represented by ζ_r. Third, the final frozen store ϕ_f must be well-typed with respect to a new frozen store typing Ψ_f (ignoring the set of final thawed locations $\zeta_f \circledast \zeta_r$) where Ψ_f is a valid extension of Ψ_s . Finally, $(k-j, \Psi_f, \zeta_f, \sigma_f, e_f)$ must be a member of $\mathcal{V}[\![\tau]\!]$.

The semantic interpretation of a typing is given by $[\![\Delta; \Gamma \vdash e : \tau]\!]$, which asserts that for all $k \geq 0$, all substitutions δ and γ, and all frozen store typings Ψ, thaw sets ζ, and linear stores σ that are compatible with Δ and Γ, $(k, \Psi, \zeta, \sigma, e)$ is a member of $\mathcal{C}[\![\delta(\tau)]\!]$.

We have established the following theorem which shows the soundness of the typing rules with respect to the model [3].

Theorem 3 (Extended $\mathbf{L}^3$ Soundness)

If $\Delta; \Gamma \vdash e : \tau$, *then* $[\![\Delta; \Gamma \vdash e : \tau]\!]$.

5 Related Work

A number of researchers have noted that linearity and strong updates can be used to provide more effective memory management (c.f. [10, 24, 33, 9, 15, 8]). Our work is complementary, in the sense that it provides a foundational standpoint for expressing such memory management in the presence of both linear and unrestricted data.

Our core $\mathbf{L}^3$ language is most directly influenced by Alias Types [34]. Relative to that work, the main contributions of our core language are (a) a simplification of the typing rules by treating capabilities as first-class linear objects, and (b) a model for the types that makes the connections with models for spatial logics clear. Of course, the extended version of $\mathbf{L}^3$ goes well beyond what Alias Types provided, with

its support for thawing and re-freezing locations. As noted earlier, these primitives are inspired by the lexically-scoped `restrict` of CQUAL [5], though they are strictly more powerful.

The work of Boyland et al. [14] considers another application of capabilities as a means to regulate sharing of mutable state. They present an operational semantics for an untyped calculus in which every pointer is annotated with a set of capabilities, which are checked at each read or write through the pointer. Capabilities can also be asserted, which revokes capabilities from aliasing pointers; this revocation can stall the abstract machine by removing necessary access rights for future pointer accesses. They leave as an open problem the specification of policies and type-systems to ensure that execution does not get stuck. While $\mathbf{L}^3$ does not solve this problem, the comparison points to an interesting phenomenon. We might naïvely assert that a $\mathsf{Cap}\,\rho\,\tau$ corresponds to a unique pointer with full capabilities; however, doing so would revoke capabilities from aliasing frozen pointers. Further investigation into the relation between these language should be pursued.

The Vault programming language [19] extended the ideas of the Capability Calculus [37] and Alias Types to enforce type-state protocols. As far as we are aware, there is no published type soundness proof of Vault's type system. Later work [20] added the `adoption` and `focus` constructs. The former takes linear references to an adoptee and an adopter, installs an internal pointer from the adopter to the adoptee, and returns a non-linear reference to the adoptee. This permits unrestricted aliasing of the adoptee through the non-linear reference; however, linear components of the adoptee may not be accessed directly through the non-linear reference. Instead, the `focus` construct temporarily provides a linear view of an object of non-linear type.

While there is no simple translation from Vault to our extended language, it should be clear that their `adoption`/`focus` play a similar role to our `freeze`/`thaw`/`refreeze`. We believe that we can approximate Vault's behavior by reinterpreting our $\mathsf{Frzn}\,\rho\,\tau$ and $\mathsf{Thwd}\,\theta$ types and modifying the proof obligations at `freeze`, `thaw`, and `refreeze`. Rather than interpreting $\mathsf{Frzn}\,\rho\,\tau$ as an indication that the location ρ stores a value of type τ, we interpret it as an indication that *some* location, adopted by the location ρ, stores a value of type τ. Likewise, $\mathsf{Thwd}\,\theta$ records the adopters of locations that have been thawed, rather than the locations themselves. `freeze` and `thaw` are parameterized by the adopter and require a proof that the adopter is not in the thawed set; `thaw` returns an existential package, which provides access to the "*some* location" hidden by $\mathsf{Frzn}\,\rho\,\tau$. An important avenue of future work is to validate this approximation and to consider ways in which it can coexist with our original interpretation.

There has been a great deal of work on adapting some notion of linearity to real programming languages such as Java. Examples include ownership types [18, 12], uniqueness types [33, 13, 17, 23], confinement types [16, 22, 35], balloon types [6], islands [25], and roles [27]. Each of these mechanisms is aimed at supporting local reasoning in the presence of aliasing and updates. Most of these approaches relax the strong requirements of linearity to make programming more convenient. We believe that $\mathbf{L}^3$ could provide a convenient foundation for modeling many of these features, because we have made the distinction between a reference and a capability to use the reference.

6 Summary and Future Work

We have presented $\mathbf{L}^3$ which is a simple, linearly-typed calculus that supports shared references and strong updates. The core language provides all the power of Alias Types but has a much simpler semantics that can be directly related to models of spatial logics such as BI. The extended language can model unrestricted references, as found in conventional imperative languages, where the type of the reference must be frozen. In addition, following CQUAL, our extended language allows "thawing" a frozen reference so that the location can be strongly updated, and then re-freezing a location once the type has been restored.

A key open issue is what logic to use for proving that it is safe to thaw a given location. For instance, one could imagine a logic that allows us to conclude two locations do not alias because their types are incompatible. In CQUAL, locations are placed in different conceptual regions, and the regions are used to abstract sets of thawed locations.

Another open issue is how to lift the ideas in $\mathbf{L}^3$ to a surface level language. Clearly, explicitly threading linear capabilities and a thaw token through a computation is too painful to contemplate. We are currently working on adapting ideas from indexed monads and type-and-effects systems to support implicit threading of these mechanisms.

References

[1] P. Achten and R. Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[2] A. Ahmed, A. W. Appel, and R. Virga. An indexed model of impredicative polymorphism and mutable references. Available at `http://www.cs.princeton.edu/~appel/papers/impred.pdf`, Jan. 2003.

[3] A. Ahmed, M. Fluet, and G. Morrisett. $\mathbf{L}^3$: A linear language with locations. Technical Report TR-24-04, Harvard University, Oct. 2004.

[4] A. J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[5] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 129–140, June 2003.

[6] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[7] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.

[8] D. Aspinall and A. Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning*, 31:261–302, 2003.

[9] D. Aspinall and M. Hofmann. Another type system for in-place update. In D. L. Metayer, editor, *Proc. European Symposium on Programming*, pages 36–52. Springer-Verlag, 2002. LNCS 2305.

[10] H. Baker. Lively linear LISP—look ma, no garbage. *ACM SIGPLAN Notices*, 27(8):89–98, 1992.

[11] P. N. Benton. Strong normalisation for the linear term calculus. *Journal of Functional Programming*, 5(1):65–80, January 1995.

[12] C. Boyapati, A. Sălcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 324–337, June 2003.

[13] J. Boyland. Alias burying: Unique variables without destructive reads. *Software – Practice and Experience*, 31(6):533–553, 2001.

[14] J. Boyland, J. Noble, and W. Retert. Capabilities for aliasing: A generalization of uniqeuness and read-only. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[15] J. Cheney and G. Morrisett. A linearly typed assembly language. Technical Report 2003-1900, Department of Computer Science, Cornell University, 2003.

[16] D. Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, 2001.

[17] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 176–200, July 2003.

[18] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[19] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, June 2001.

[20] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24, June 2002.

[21] J.-Y. Girard. Linear logic. *Theoretical Comput. Sci.*, 50:1–102, Theoretical Computer Science.

[22] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[23] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in Cyclone. In *Proc. International Symposium on Memory Management*, pages 73–84, October 2004.

[24] M. Hofmann. A type system for bounded space and functional in-place update. In *Proc. European Symposium on Programming (ESOP)*, pages 165–179, mar 2000.

[25] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1991.

[26] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *28th ACM Symposium on Principles of Programming Languages (POPL)*, pages 14–26, London, UK, Jan. 2001.

[27] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–32. ACM Press, 2002.

[28] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Workshop on Compiler Support for Systems Software*, pages 25–35, Atlanta, GA, May 1999. Published as INRIA Technical Report 0288, March, 1999.

[29] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, Jan. 2002.

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

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

[32] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 55–74, Los Alamitos, CA, USA, July 22–25 2002. IEEE Computer Society.

[33] S. Smetsers, E. Barendsen, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In *Dagstuhl Seminar on Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 358–379. Springer-Verlag, 1994.

[34] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proc. European Symposium on Programming (ESOP)*, pages 366–381, Mar. 2000.

[35] J. Vitek and B. Bokowski. Confined types in Java. *Software – Practice and Experience*, 31(6):507–532, 2001.

[36] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*, Apr. 1990. IFIP TC 2 Working Conference.

[37] D. Walker, K. Crary, and G. Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 24(4):701–771, July 2000.

[38] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Proc. Workshop on Types in Compilation (TIC)*, pages 177–206, Sept. 2000.