L^3: A Linear Language with Locations

Greg Morrisett[1], Amal Ahmed[1], and Matthew Fluet[2]

[1] Harvard University
{greg,amal}@eecs.harvard.edu
[2] Cornell University
fluet@cs.cornell.edu

Abstract. We explore foundational typing support for *strong updates* — updating a memory cell to hold values of unrelated types at different points in time. We present a simple, but expressive type system based upon standard linear logic, one that also 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 make our calculus expressive enough to serve as a model for languages with 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.

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. 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.[3] 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

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

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. The Vault language [1, 2] 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. Typed Assembly Language [3, 4] 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 C, called CQUAL [5], which uses strong updates to track security-relevant properties in legacy C code.

Vault, later versions of TAL [6], and CQUAL all based their support for strong updates and alias control on the *Alias Types* formalism of Smith, Walker, and Morrisett [7, 8]. Though Alias Types were proven sound in a syntactic sense, we lacked an understanding of their *semantics*. 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 the logic of Bunched Implications (BI) [9]. 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. We do not envision the core calculi presented here to be used by end programmers. Instead, we intend these calculi to be expressive enough to serve as a target language for more palatable surface languages. Proofs of theorems, as well as extended discussion and examples, can be found in a companion technical report [10].

2 Core $\mathbf{L}^3$

Linear types, which are derived from Girard's linear logic [11], have proven useful for modeling imperative programming languages in a functional setting [12, 13]. For instance, the Clean programming language [14] 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.

Though a linear interpretation of reference cells supports strong updates, it is too restrictive for many, if not most, realistic programs. 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 [7, 8], 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.

In this section, we present a different formulation of Alias Types based on a relatively standard call-by-value 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 pointers 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.

2.1 Syntax

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

$$\begin{array}{ll} \textit{LocConsts}\ \ \ell \in \textit{LocConsts} & \\ \textit{LocVars}\ \ \ \rho \in \textit{LocVars} & \textit{Locs}\ \eta\ ::= \ell \mid \rho \end{array}$$

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

Most of the types, expressions, and values are based on a traditional, call-by-value, linear lambda calculus. In the following sections, we will explain the bits that are new or different.

Types The types $\mathbf{1}$, $\tau_1 \otimes \tau_2$, $\tau_1 \multimap \tau_2$, 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 $\mathtt{let}\ \langle\,\rangle = e_1\ \mathtt{in}\ e_2$ and $\mathtt{let}\ \langle x_1, x_2\rangle = e_1\ \mathtt{in}\ e_2$ are used to eliminate unit ($\mathbf{1}$) and tensor products ($\otimes$)

Stores

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

Evaluation Contexts

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

(let-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) $(\sigma, \texttt{drop}\, !v) \longmapsto (\sigma, \langle\,\rangle)$

(new) $(\sigma, \texttt{new}\, v) \longmapsto (\sigma \uplus \{\ell \mapsto v\}, \ulcorner \ell, \langle \texttt{cap}\, \ell, !(\texttt{ptr}\, \ell)\rangle \urcorner)$

(free) $(\sigma \uplus \{\ell \mapsto v\}, \texttt{free}\, \ulcorner \ell, \langle \texttt{cap}\, \ell, !(\texttt{ptr}\, \ell)\rangle \urcorner) \longmapsto (\sigma, \ulcorner \ell, v \urcorner)$

(swap) $(\sigma \uplus \{\ell \mapsto v_1\}, \texttt{swap}\, (\texttt{ptr}\, \ell)\ \langle \texttt{cap}\, \ell, v_2\rangle) \longmapsto (\sigma \uplus \{\ell \mapsto v_2\}, \langle \texttt{cap}\, \ell, v_1\rangle)$

Fig. 1. Core $\mathbf{L}^3$– Selected Operational Semantics

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. The link between these two components is the location's name: either a location constant ℓ or a location variable ρ. Location constants (e.g., 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. Note that location variables ρ may be bound in types and expressions and alpha-convert, while location constants ℓ do not.

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. 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.

Abstraction over locations within 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.

Expressions and 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), eliding some of the more obvious transitions. 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, inner-most 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.

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.

The expressions $\texttt{new}\,e$ and $\texttt{free}\,e$ perform the complementary actions of allocating and deallocating mutable references in the global store. $\texttt{new}\,e$ 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. $\texttt{free}\,e$ 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. Using $\texttt{swap}$ instead of dereference and update ensures that resources are not duplicated [15]. Thus, $\texttt{swap}$ is the appropriate primitive to ensure the linearity of resources. 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 $\ulcorner \eta, e \urcorner$ (read "pack η in e") 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 $\texttt{let}\ \ulcorner \rho, x \urcorner = e_1\ \texttt{in}\ e_2$.

2.2 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:

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

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

$\Delta; \Gamma \vdash e : \tau$

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

$$\text{(Dup)}\ \frac{\Delta; \Gamma \vdash e : !\tau}{\Delta; \Gamma \vdash \texttt{dup}\, e : !\tau \otimes !\tau} \qquad \text{(Drop)}\ \frac{\Delta; \Gamma \vdash e : !\tau}{\Delta; \Gamma \vdash \texttt{drop}\, e : \mathbf{1}}$$

$$\text{(New)}\ \frac{\Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \texttt{new}\, e : \exists\rho.(\mathsf{Cap}\ \rho\ \tau \otimes !(\mathsf{Ptr}\ \rho))} \qquad \text{(Free)}\ \frac{\Delta; \Gamma \vdash e : \exists\rho.(\mathsf{Cap}\ \rho\ \tau \otimes !(\mathsf{Ptr}\ \rho))}{\Delta; \Gamma \vdash \texttt{free}\, e : \exists\rho.\tau}$$

$$\text{(Swap)}\ \frac{\Delta; \Gamma_1 \vdash e_1 : \mathsf{Ptr}\ \rho \qquad \Delta; \Gamma_2 \vdash e_2 : \mathsf{Cap}\ \rho\ \tau_1 \otimes \tau_2}{\Delta; \Gamma_1 \boxplus \Gamma_2 \vdash \texttt{swap}\ e_1\ e_2 : \mathsf{Cap}\ \rho\ \tau_2 \otimes \tau_1}$$

Fig. 2. Core $\mathbf{L}^3$– Selected Static Semantics

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 $\Gamma_1 \boxplus \Gamma_2 = \Gamma$ that splits the assumptions in Γ between the contexts Γ_1 and Γ_2:

$$\begin{aligned} \bullet \boxplus \bullet &= \bullet \\ (\Gamma_1, x{:}\tau) \boxplus \Gamma_2 &= (\Gamma_1 \boxplus \Gamma_2), x{:}\tau \quad (x \notin dom(\Gamma_2)) \\ \Gamma_1 \boxplus (\Gamma_2, x{:}\tau) &= (\Gamma_1 \boxplus \Gamma_2), x{:}\tau \quad (x \notin dom(\Gamma_1)) \end{aligned}$$

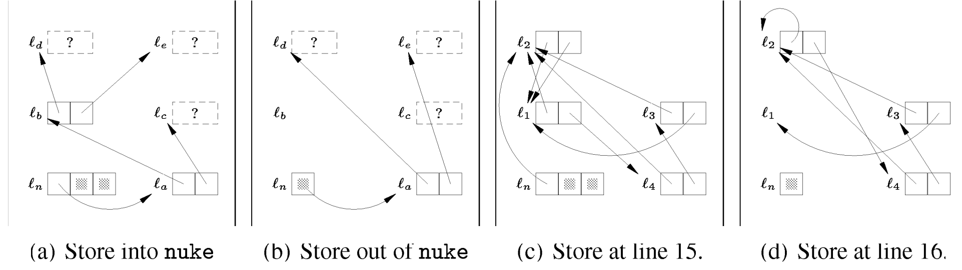
Splitting the context is necessary to ensure that a given resource is used by at most one sub-expression. Note that $\boxplus$ splits all assumptions, even those of !-type. However, recall that contraction and weakening is supported for !-types through explicit operations.

Figure 2 presents the typing rules for $\mathbf{L}^3$, eliding the normal rules for a linear lambda calculus. The (Bang) rule uses an auxiliary function $|\cdot|$ on contexts to extract the linear components:

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

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 $\texttt{ptr}\ \ell$ or $\texttt{cap}\ \ell$, 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 (New), (Free), and (Swap) rules act as introduction and elimination rules for $\mathsf{Ptr}\ \rho$ and $\mathsf{Cap}\ \rho\ \tau$ types. Both (New) and (Free) 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 $\mathsf{Cap}\ \rho\ \tau_2$.



(a) Store into `nuke` (b) Store out of `nuke` (c) Store at line 15. (d) Store at line 16.

Fig. 3. Store during the evaluation of the example program.

2.3 Examples and Discussion

This core language is expressive enough to approximate the examples given in Section 1. 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

$$\begin{array}{lll}
\texttt{lrswap} \equiv \lambda\texttt{r}{:}\mathsf{LRef}\ \tau.\ \lambda\texttt{x}{:}\tau'. & & \\
\quad \texttt{let}\ \ulcorner\rho, \texttt{cp}\urcorner = \texttt{r}\ \texttt{in} & & \#\ \texttt{cp}{:}\mathsf{Cap}\ \rho\ \tau \otimes !\mathsf{Ptr}\ \rho \\
\quad \texttt{let}\ \langle \texttt{c}_0, \texttt{p}_0\rangle = \texttt{cp}\ \texttt{in} & & \#\ \texttt{c}_0{:}\mathsf{Cap}\ \rho\ \tau,\ \texttt{p}_0{:}!\mathsf{Ptr}\ \rho \\
\quad \texttt{let}\ \langle \texttt{p}_1, \texttt{p}_2\rangle = \texttt{dup}\,\texttt{p}_0\ \texttt{in} & & \#\ \texttt{p}_1{:}!\mathsf{Ptr}\ \rho,\ \texttt{p}_2{:}!\mathsf{Ptr}\ \rho \\
\quad \texttt{let}\ !\texttt{p}_2' = \texttt{p}_2\ \texttt{in} & & \#\ \texttt{p}_2'{:}\mathsf{Ptr}\ \rho \\
\quad \texttt{let}\ \langle \texttt{c}_1, \texttt{y}\rangle = \texttt{swap}\,\texttt{p}_2'\ \langle \texttt{c}_0, \texttt{x}\rangle\ \texttt{in} & & \#\ \texttt{c}_1{:}\mathsf{Cap}\ \rho\ \tau',\ \texttt{y}{:}\tau \\
\quad \langle \ulcorner\rho, \langle \texttt{c}_1, \texttt{p}_1\rangle\urcorner, \texttt{y}\rangle & &
\end{array}$$

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 `x` with !-type is annotated as $\texttt{x}^!$. Second, we elide $\texttt{let}\ !x = \cdot\ \texttt{in}\ \cdot$, $\texttt{dup}\ \cdot$, and $\texttt{drop}\ \cdot$ expressions, leaving the duplication, dereliction, and linearization of !-type variables implicit. Taken together, these two conventions mean that a variable introduced as $\texttt{x}^!$ of type $!\tau$ may be used zero, one, or many times in its scope, including contexts requiring the type τ. Third, we annotate successive "versions" of a linear variable with an integer superscript. Finally, we introduce the expression form $\texttt{let}\ p = e_1\ \texttt{in}\ e_2$, where p is a pattern, to abbreviate multiple elimination forms.

We consider well-typed core $\mathbf{L}^3$ program given below. Line 1 allocates a mutable reference, with pointer $\texttt{p}_n$. Lines 2 through 8 define a function, `nuke`, parameterized by five locations. Note that the type of $\texttt{c}_n$ 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 $\texttt{p}_n$ 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 ($\texttt{p}_a$) 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. Note that the type of $\mathtt{c}_n^2$ on line 8 is

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

The final shape of the store is given in Figure 3(b). The whole program is given as follows:

1. $\mathtt{let}\ \ulcorner \rho_n, \langle \mathtt{c}_n^0, \mathtt{p}_n^! \rangle \urcorner = \mathtt{new}\ \langle\,\rangle\ \mathtt{in}$
2. $\mathtt{let}\ \mathtt{nuke}^! = !\Lambda \rho_a, \rho_b, \rho_c, \rho_d, \rho_e.\ \lambda \mathtt{c}_n^0.$
3. $\quad \mathtt{let}\ \langle \mathtt{c}_n^1, \langle \mathtt{p}_a^!, \langle \mathtt{c}_a^0, \mathtt{c}_b^0 \rangle\rangle\rangle = \mathtt{swap}\ \mathtt{p}_n\ \langle \mathtt{c}_n^0, \langle\,\rangle\rangle\ \mathtt{in}$
   $\quad \#\ \mathtt{c}_n^1{:}\mathbf{1},\ \mathtt{c}_a^0{:}\mathsf{Cap}\ \rho_a\ (!(\mathsf{Ptr}\ \rho_b) \otimes !(\mathsf{Ptr}\ \rho_c)),\ \mathtt{c}_b^0{:}\mathsf{Cap}\ \rho_b\ (!(\mathsf{Ptr}\ \rho_d) \otimes !(\mathsf{Ptr}\ \rho_e))$
4. $\quad \mathtt{let}\ \langle \mathtt{c}_a^1, \langle \mathtt{p}_b^!, \mathtt{p}_c^! \rangle\rangle = \mathtt{swap}\ \mathtt{p}_a\ \langle \mathtt{c}_a^0, \langle\,\rangle\rangle\ \mathtt{in} \quad \#\ \mathtt{c}_a^1{:}\mathbf{1}$
5. $\quad \mathtt{let}\ \langle \mathtt{p}_d^!, \mathtt{p}_e^! \rangle = \mathtt{free}\ \ulcorner \rho_b, \langle \mathtt{c}_b^0, \mathtt{p}_b \rangle \urcorner\ \mathtt{in}$
6. $\quad \mathtt{let}\ \langle \mathtt{c}_a^2, \langle\,\rangle\rangle = \mathtt{swap}\ \mathtt{p}_a\ \langle \mathtt{c}_a^1, \langle \mathtt{p}_d, \mathtt{p}_e \rangle\rangle\ \mathtt{in} \quad \#\ \mathtt{c}_a^2{:}\mathsf{Cap}\ \rho_a\ (!(\mathsf{Ptr}\ \rho_d) \otimes !(\mathsf{Ptr}\ \rho_e))$
7. $\quad \mathtt{let}\ \langle \mathtt{c}_n^2, \langle\,\rangle\rangle = \mathtt{swap}\ \mathtt{p}_n\ \langle \mathtt{c}_n^1, \mathtt{c}_a^2 \rangle\ \mathtt{in}$
8. $\quad \mathtt{c}_n^2\ \mathtt{in}$
9. $\mathtt{let}\ \ulcorner \rho_1, \langle \mathtt{c}_1^0, \mathtt{p}_1^! \rangle \urcorner = \mathtt{new}\ \langle\,\rangle\ \mathtt{in}$
10. $\mathtt{let}\ \ulcorner \rho_2, \langle \mathtt{c}_2^0, \mathtt{p}_2^! \rangle \urcorner = \mathtt{new}\ \langle \mathtt{p}_1, \mathtt{p}_1 \rangle\ \mathtt{in}$
11. $\mathtt{let}\ \ulcorner \rho_3, \langle \mathtt{c}_3^0, \mathtt{p}_3^! \rangle \urcorner = \mathtt{new}\ \langle \mathtt{p}_2, \mathtt{p}_1 \rangle\ \mathtt{in}$
12. $\mathtt{let}\ \ulcorner \rho_4, \langle \mathtt{c}_4^0, \mathtt{p}_4^! \rangle \urcorner = \mathtt{new}\ \langle \mathtt{p}_2, \mathtt{p}_3 \rangle\ \mathtt{in}$
13. $\mathtt{let}\ \langle \mathtt{c}_1^1, \langle\,\rangle\rangle = \mathtt{swap}\ \mathtt{p}_1\ \langle \mathtt{c}_1^0, \langle \mathtt{p}_2, \mathtt{p}_4 \rangle\rangle\ \mathtt{in}$
14. $\mathtt{let}\ \langle \mathtt{c}_n^1, \langle\,\rangle\rangle = \mathtt{swap}\ \mathtt{p}_n\ \langle \mathtt{c}_n^0, \langle \mathtt{p}_2, \langle \mathtt{c}_2^0, \mathtt{c}_1^1 \rangle\rangle\rangle\ \mathtt{in}$
15. $\mathtt{let}\ \mathtt{c}_n^2 = \mathtt{nuke}\ [\rho_2, \rho_1, \rho_1, \rho_2, \rho_4]\ \mathtt{c}_n^1\ \mathtt{in}$
16. $\mathtt{let}\ \langle \mathtt{c}_n^3, \mathtt{c}_2^1 \rangle = \mathtt{swap}\ \mathtt{p}_n\ \langle \mathtt{c}_n^2, \langle\,\rangle\rangle\ \mathtt{in}$
17. $\mathtt{let}\ \langle \mathtt{c}_n^4, \langle\,\rangle\rangle = \mathtt{swap}\ \mathtt{p}_n\ \langle \mathtt{c}_n^3, \langle \mathtt{p}_3, \langle \mathtt{c}_3^1, \mathtt{c}_2^1 \rangle\rangle\rangle\ \mathtt{in}$
18. $\mathtt{let}\ \mathtt{c}_n^5 = \mathtt{nuke}\ [\rho_3, \rho_2, \rho_1, \rho_2, \rho_4]\ \mathtt{c}_n^4\ \mathtt{in}$
19. $\mathtt{let}\ \mathtt{c}_3^2 = \mathtt{free}\ \ulcorner \rho_n, \langle \mathtt{c}_n^5, \mathtt{p}_n \rangle \urcorner\ \mathtt{in}$
20. $\mathtt{let}\ \langle \mathtt{p}_a^!, \mathtt{p}_b^! \rangle = \mathtt{free}\ \ulcorner \rho_3, \langle \mathtt{c}_3^2, \mathtt{p}_3 \rangle \urcorner\ \mathtt{in}$
21. $\mathtt{let}\ \langle \mathtt{p}_c^!, \mathtt{p}_d^! \rangle = \mathtt{free}\ \ulcorner \rho_4, \langle \mathtt{c}_4^0, \mathtt{p}_4 \rangle \urcorner\ \mathtt{in}\ \langle\,\rangle$

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.

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 $\mathtt{c}_n^{(i)}$. Note 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

$$
\begin{aligned}
\mathcal{V}\|\mathbf{1}\| &= \{(\{\,\}, \langle\,\rangle)\} \\
\mathcal{V}\|\tau_1 \otimes \tau_2\| &= \{(\sigma_1 \uplus \sigma_2, \langle v_1, v_2\rangle) \mid (\sigma_1, v_1) \in \mathcal{V}\|\tau_1\| \ \wedge\ (\sigma_2, v_2) \in \mathcal{V}\|\tau_2\|\} \\
\mathcal{V}\|\tau_1 \multimap \tau_2\| &= \{(\sigma_2, \lambda x.\, e) \mid \forall \sigma_1, v_1.\, (\sigma_1, v_1) \in \mathcal{V}\|\tau_1\| \ \wedge\ \sigma_1 \uplus \sigma_2 \text{ defined } \Rightarrow\ (\sigma_1 \uplus \sigma_2, e[v_1/x]) \in \mathcal{C}\|\tau_2\|\} \\
\mathcal{V}\|!\tau\| &= \{(\{\,\}, !v) \mid (\{\,\}, v) \in \mathcal{V}\|\tau\|\} \\
\mathcal{V}\|\mathsf{Ptr}\,\ell\| &= \{(\{\,\}, \mathtt{ptr}\,\ell)\} \\
\mathcal{V}\|\mathsf{Cap}\,\ell\,\tau\| &= \{(\sigma \uplus \{\ell \mapsto v\}, \mathtt{cap}\,\ell) \mid (\sigma, v) \in \mathcal{V}\|\tau\|\} \\
\mathcal{V}\|\forall \rho.\tau\| &= \{(\sigma, \Lambda\rho.\, e) \mid \forall \ell.\, (\sigma, e[\ell/\rho]) \in \mathcal{C}\|\tau[\ell/\rho]\|\} \\
\mathcal{V}\|\exists \rho.\tau\| &= \{(\sigma, \ulcorner \ell, v \urcorner) \mid (\sigma, v) \in \mathcal{V}\|\tau[\ell/\rho]\|\}
\end{aligned}
$$

$$
\mathcal{C}\|\tau\| = \{(\sigma_s, e_s) \mid \forall \sigma_r.\, \sigma_s \uplus \sigma_r \text{ defined } \Rightarrow\ \exists n, \sigma_f, v_f.\, (\sigma_s \uplus \sigma_r, e_s) \longmapsto^n (\sigma_f \uplus \sigma_r, v_f) \ \wedge\ (\sigma_f, v_f) \in \mathcal{V}\|\tau\|\}
$$

$$
\begin{aligned}
\mathcal{S}\|\bullet\|\delta &= \{(\{\,\}, \emptyset)\} \\
\mathcal{S}\|\Gamma, x{:}\tau\|\delta &= \{(\sigma \uplus \sigma_x, \gamma[x \mapsto v_x]) \mid (\sigma, \gamma) \in \mathcal{S}\|\Gamma\|\delta \ \wedge\ (\sigma_x, v_x) \in \mathcal{V}\|\delta(\tau)\|\}
\end{aligned}
$$

$$
|\Delta; \Gamma \vdash e : \tau\| = \forall \delta, \sigma, \gamma.\ dom(\delta) = dom(\Delta) \ \wedge\ (\sigma, \gamma) \in \mathcal{S}\|\Gamma\|\delta \ \Rightarrow\ (\sigma, \gamma(\delta(e))) \in \mathcal{C}\|\delta(\tau)\|
$$

Fig. 4. Core $\mathbf{L}^3$– Semantic Interpretations

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 $\mathtt{c}_b$.

2.4 Semantic Interpretations

In this section, we give semantic interpretations to types and prove that the typing rules of Section 2.2 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 3 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 [16]. 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.

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.

The semantic interpretation of a typing judgment $\|\Delta; \Gamma \vdash e : \tau\|$ 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.

Theorem 1 (Core L^3 Soundness). *If* $\Delta; \Gamma \vdash e : \tau$*, then* $[\![\Delta; \Gamma \vdash e : \tau]\!]$.

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

3 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:[4]

```
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

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

$$\text{(freeze)} \quad (\phi, \sigma \uplus \{\ell \mapsto !v\}, \texttt{freeze}\ \langle \texttt{cap}\ \ell, \texttt{thwd}\ L\rangle\ v') \longmapsto (\phi \uplus \{\ell \mapsto !v\}, \sigma, \langle !(\texttt{frzn}\ \ell), \texttt{thwd}\ L\rangle)$$

$$\text{(thaw)} \quad (\phi \uplus \{\ell \mapsto !v\}, \sigma, \texttt{thaw}\ \langle !(\texttt{frzn}\ \ell), \texttt{thwd}\ L\rangle\ v') \longmapsto (\phi, \sigma \uplus \{\ell \mapsto !v\}, \langle \texttt{cap}\ \ell, \texttt{thwd}\ (L \uplus \{\ell\})\rangle)$$

$$\text{(refreeze)} \quad (\phi, \sigma \uplus \{\ell \mapsto !v\}, \texttt{refreeze}\ \langle \texttt{cap}\ \ell, \texttt{thwd}\ (L \uplus \{\ell\})\rangle) \longmapsto (\phi \uplus \{\ell \mapsto !v\}, \sigma, \langle !(\texttt{frzn}\ \ell), \texttt{thwd}\ L\rangle)$$

Fig. 5. Extended $\mathbf{L}^3$– Additional Operational Semantics

$\Delta; \Gamma \vdash e : \tau$

$$\text{(Freeze)} \quad \frac{\Delta; \Gamma_1 \vdash e_1 : \mathsf{Cap}\ \rho\ !\tau \otimes \mathsf{Thwd}\ \theta \qquad \Delta; \Gamma_2 \vdash e_2 : \mathsf{Notin}\ \rho\ \theta}{\Delta; \Gamma_1 \boxplus \Gamma_2 \vdash \texttt{freeze}\ e_1\ e_2 : !(\mathsf{Frzn}\ \rho\ !\tau) \otimes \mathsf{Thwd}\ \theta}$$

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

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

Fig. 6. Extended $\mathbf{L}^3$– Additional Static Semantics

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 re-proving soundness.

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. Note that it would be unsound to freeze a thawed location at a type other than the original frozen type, because other frozen capabilities expect the location to hold a

value of the original type. Therefore, we provide a separate operation that requires the original type to 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.

3.1 Changes to Support the Extensions

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

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

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 (new) 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, $\mathsf{Notin}\,\eta\,\bullet$.

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; thawed locations must 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.

3.2 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.

```
read ≡ λr^!:Ref !τ. λt^0:Thwd • .
         let ⌜ρ, ⟨f_a^!, l^!⟩⌝ = r in                  # f_a:!Frzn ρ !τ, l:!Ptr ρ
         let ⟨c^1, t^1⟩ = thaw ⟨f_a, t^0⟩ void_ρ in    # c^1:Cap ρ !τ, t^1:Thwd •, ρ:!τ
         let ⟨c^2, x^!⟩ = swap l ⟨c^1, ⟨⟩⟩ in          # c^2:Cap ρ 1, x:!τ
         let ⟨c^3, ⟨⟩⟩ = swap l ⟨c^2, x⟩ in            # c^3:Cap ρ !τ
         let ⟨f_b^!, t^2⟩ = refreeze ⟨c^3, t^1⟩ in     # f_b:!Frzn ρ !τ, t^2:Thwd •
         ⟨x, t^2⟩

write ≡ λr^!:Ref !τ. λz^!:!τ. λt^0:Thwd • .
          let ⌜ρ, ⟨f_a^!, l^!⟩⌝ = r in                  # f_a:!Frzn ρ !τ, l:!Ptr ρ
          let ⟨c^1, t^1⟩ = thaw ⟨f_a, t^0⟩ void_ρ in    # c^1:Cap ρ !τ, t^1:Thwd •, ρ:!τ
          let ⟨c^2, x^!⟩ = swap l ⟨c^1, z⟩ in           # c^2:Cap ρ !τ, x:!τ
          let ⟨f_b^!, t^2⟩ = refreeze ⟨c^2, t^1⟩ in     # f_b:!Frzn ρ !τ, t^2:Thwd •
          t^2
```

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

3.3 Semantic Interpretations

As the extended $\mathbf{L}^3$ is strictly more powerful that the core language given previously, the semantic interpretation given in Section 2.4 will not suffice as a model. We describe the essential intuitions underlying our semantic interpretation here; details are given in the technical report [10].

Our model for extended $\mathbf{L}^3$ is based on the indexed model of general references by Ahmed, Appel, and Virga [17, 18] where the semantic interpretation of a (closed) type $\mathcal{V}\|\tau\|$ is a set of triples of the form (k, Ψ, v). Here k is a natural number (called the *approximation index*), Ψ is a store typing that maps locations to (the interpretation of) their designated types, and v is a value. Intuitively, $(k, \Psi, v) \in \mathcal{V}\|\tau\|$ says that in any computation running for no more than k steps, v cannot be distinguished from values of type τ. Furthermore, Ψ need only specify the types of locations to approximation $k - 1$ — to determine whether v has type τ for k steps, it suffices to know the type of each store location 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 $(k, \Psi, \zeta, \sigma, v)$. 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 $k-1$); v is a value. As for core $\mathbf{L}^3$, we consider σ to be the exclusive store of the value v. The lifted thaw set $\zeta \in \mathcal{P}(\mathit{LocConsts})_\perp$ denotes either the set of currently thawed locations (if v has exclusive rights to the thaw token) or $\perp$ (if v has no such rights). We define $\mathcal{V}\|\mathsf{Thwd}\ \theta\|$ as the set of all tuples of the form $(k, \Psi, L, \{\}, \texttt{thwd}\ L)$ such that the type of every currently thawed location ($\ell \in L$) in θ is consistent (to approximation k) with the type of the location in Ψ. This ensures that when we move a location from the linear store back to the frozen store, we end up with a frozen store where every location contains the type mandated by Ψ.

In order to track how far "out of synch" the frozen store ϕ is with respect to the frozen store typing Ψ, we define the relation $\phi :_k \Psi \setminus \zeta$. Informally, this says that the frozen store ϕ is well-typed with respect to the store typing Ψ modulo the current set of thawed locations ζ — that is, the contents of locations in the frozen store must have the types specified by Ψ, but the contents of thawed locations do not have to have the types mandated by Ψ.

As for core $\mathbf{L}^3$, we have established the following theorem which shows the soundness of the typing rules with respect to the model.

Theorem 2 (Extended $\mathbf{L}^3$ Soundness). *If* $\Delta; \Gamma \vdash e : \tau$*, then* $[\![\Delta; \Gamma \vdash e : \tau]\!]$.

4 Related Work

A number of researchers have noted that linearity and strong updates can be used to effectively manage memory (c.f. [15, 19–23]). 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 [7]. 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. [24] 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.

The Vault programming language [1] extended the ideas of the Capability Calculus [25] 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 [2] 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. We are confident that it will be possible to extend $\mathbf{L}^3$ to handle these features.

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 [26, 27], uniqueness types [20, 28–30], confinement types [31–33], balloon types [34], islands [35], and roles [36]. 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.

A more distantly related body of work concerns the typing of process calculi [37, 38]. In this work, a kind of strong update is allowed in the type of channels, where a single communication port can be used for sending values of different types. While a connection with linearity has been established [39], the intuition seems to be more closely related to type-states than to strong updates. A potentially fruitful direction for future work would be to investigate both the application of process types to this work and to extend this work to apply in a concurrent setting.

5 Future Work

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. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: Proc. ACM Conference on Programming Language Design and Implementation (PLDI). (2001) 59–69
2. Fähndrich, M., DeLine, R.: Adoption and focus: Practical linear types for imperative programming. In: Proc. ACM Conference on Programming Language Design and Implementation (PLDI). (2002) 13–24
3. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Transactions on Programming Languages and Systems **21** (1999) 528–569
4. Morrisett, G., Crary, K., Glew, N., Walker, D.: Stack-based typed assembly language. Journal of Functional Programming **12** (2002) 43–88
5. Aiken, A., Foster, J.S., Kodumal, J., Terauchi, T.: Checking and inferring local non-aliasing. In: Proc. ACM Conference on Programming Language Design and Implementation (PLDI). (2003) 129–140
6. Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., Zdancewic, S.: TALx86: A realistic typed assembly language. In: Workshop on Compiler Support for Systems Software, Atlanta, GA (1999) 25–35 Published as INRIA Technical Report 0288, March, 1999.
7. Smith, F., Walker, D., Morrisett, G.: Alias types. In: Proc. European Symposium on Programming (ESOP). (2000) 366–381
8. Walker, D., Morrisett, G.: Alias types for recursive data structures. In: Proc. Workshop on Types in Compilation (TIC). (2000) 177–206
9. Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: 28th ACM Symposium on Principles of Programming Languages (POPL), London, UK (2001) 14–26
10. Ahmed, A., Fluet, M., Morrisett, G.: $\mathbf{L}^3$: A linear language with locations. Technical Report TR-24-04, Harvard University (2004)
11. Girard, J.Y.: Linear logic. Theoretical Computer Science **50** (1987) 1–102
12. Wadler, P.: Linear types can change the world! In: Programming Concepts and Methods. (1990) IFIP TC 2 Working Conference.
13. O'Hearn, P.W., Reynolds, J.C.: From Algol to polymorphic linear lambda-calculus. Journal of the ACM **47** (2000) 167–223
14. Achten, P., Plasmeijer, R.: The ins and outs of Clean I/O. Journal of Functional Programming **5** (1995) 81–110
15. Baker, H.: Lively linear LISP—look ma, no garbage. ACM SIGPLAN Notices **27** (1992) 89–98
16. Benton, P.N.: Strong normalisation for the linear term calculus. Journal of Functional Programming **5** (1995) 65–80
17. Ahmed, A., Appel, A.W., Virga, R.: An indexed model of impredicative polymorphism and mutable references. Available at `http://www.cs.princeton.edu/~appel/papers/impred.pdf` (2003)
18. Ahmed, A.J.: Semantics of Types for Mutable State. PhD thesis, Princeton University (2004)
19. Hofmann, M.: A type system for bouned space and functional in-place update. In: Proc. European Symposium on Programming (ESOP). (2000) 165–179
20. Smetsers, S., Barendsen, E., van Eekelen, M.C.J.D., Plasmeijer, M.J.: 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., Springer-Verlag (1994) 358–379
21. Aspinall, D., Hofmann, M.: Another type system for in-place update. In Metayer, D.L., ed.: Proc. European Symposium on Programming, Springer-Verlag (2002) 36–52 LNCS 2305.
22. Cheney, J., Morrisett, G.: A linearly typed assembly language. Technical Report 2003-1900, Department of Computer Science, Cornell University (2003)
23. Aspinall, D., Compagnoni, A.: Heap bounded assembly language. Journal of Automated Reasoning **31** (2003) 261–302
24. Boyland, J., Noble, J., Retert, W.: Capabilities for aliasing: A generalization of uniqueness and read-only. In: European Conference on Object-Oriented Programming (ECOOP). (2001)
25. Walker, D., Crary, K., Morrisett, G.: Typed memory management in a calculus of capabilities. ACM Transactions on Programming Languages and Systems **24** (2000) 701–771
26. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (1998)
27. Boyapati, C., Sălcianu, A., Beebee, W., Rinard, M.: Ownership types for safe region-based memory management in real-time Java. In: Proc. ACM Conference on Programming Language Design and Implementation (PLDI). (2003) 324–337
28. Boyland, J.: Alias burying: Unique variables without destructive reads. Software – Practice and Experience **31** (2001) 533–553
29. Clarke, D., Wrigstad, T.: External uniqueness is unique enough. In: European Conference on Object-Oriented Programming (ECOOP). (2003) 176–200
30. Hicks, M., Morrisett, G., Grossman, D., Jim, T.: Experience with safe manual memory-management in Cyclone. In: Proc. International Symposium on Memory Management. (2004) 73–84
31. Clarke, D.: Object Ownership and Containment. PhD thesis, School of Computer Science and Engineering, University of New South Wales (2001)
32. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (2001)
33. Vitek, J., Bokowski, B.: Confined types in Java. Software – Practice and Experience **31** (2001) 507–532
34. Almeida, P.S.: Balloon types: Controlling sharing of state in data types. In: European Conference on Object-Oriented Programming (ECOOP). (1997)
35. Hogg, J.: Islands: Aliasing protection in object-oriented languages. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (1991)
36. Kuncak, V., Lam, P., Rinard, M.: Role analysis. In: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (2002) 17–32
37. Igarashi, A., Kobayashi, N.: A generic type system for the Pi-calculus. In: Proc. ACM Symposium on Principles of Programming Languages (POPL). (2001) 128–141

38. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Proc. Parallel Architectures and Languages Europe. (1994) 398–413
39. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the Pi-Calculus. ACM Transactions on Programming Languages and Systems **21** (1999) 914–947