A Step-Indexed Model of Substructural State

Amal Ahmed
Harvard University
amal@eecs.harvard.edu

Matthew Fluet [*]
Cornell University
fluet@cs.cornell.edu

Greg Morrisett [*]
Harvard University
greg@eecs.harvard.edu

Abstract

The concept of a "unique" object arises in many emerging programming languages such as Clean, CQual, Cyclone, TAL, and Vault. In each of these systems, unique objects make it possible to perform operations that would otherwise be prohibited (e.g., deallocating an object) or to ensure that some obligation will be met (e.g., an opened file will be closed). However, different languages provide different interpretations of "uniqueness" and have different rules regarding how unique objects interact with the rest of the language.

Our goal is to establish a common model that supports each of these languages, by allowing us to encode and study the interactions of the different forms of uniqueness. The model we provide is based on a substructural variant of the polymorphic λ-calculus, augmented with four kinds of mutable references: unrestricted, relevant, affine, and linear. The language has a natural operational semantics that supports deallocation of references, strong (type-varying) updates, and storage of unique objects in shared references. We establish the strong soundness of the type system by constructing a novel, semantic interpretation of the types.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.3.3 [*Programming Language*]: Language Constructs and Features

General Terms Languages

Keywords substructural type system, mutable references, step-indexed model

1. Introduction

Consider the following imperative code fragment, written with SML syntax:

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

At line 1, we assume ref cells `r1` and `r2` whose contents are integers. At line 2, we update the first cell with a boolean. Then, at line 3, we read the second cell, using the contents in a context expecting an integer. If the function is called with actual arguments that are different ref cells, then there is nothing in the function that will cause a run-time type error.[1] Yet, if the same ref cell is passed for each formal argument, then the update on line 2 will change the contents of both `r1` and `r2`, causing a run-time type error to occur at line 3.

SML (and most imperative languages) reject the above program, because references are *unrestricted*, that is, they may be freely aliased. In general, reasoning about unrestricted references is hard because we need additional information to understand what other values are affected by an update. In the absence of this information, we must be conservative. For instance, in SML, we must assume that an update to an `int ref` could affect any other `int ref`. To ensure type soundness, we must therefore require the type of the ref's contents be preserved by the update. In other words, most type systems can only track invariants on refs, instead of program-point-specific properties. As a result, we are forced to weaken the type of the ref to cover all possible program points. In the example above, we must weaken `r1`'s type to "`(int + bool) ref`" and pay the costs of tagging values, and checking those tags when the pointer is dereferenced.

Unfortunately, in many settings, this weakened invariant is insufficient. Hence, researchers have turned to more powerful systems that do provide a means of ensuring exclusive access to state. In particular, many projects have introduced some form of linearity to "tame" state. Linear logic [15] and other substructural logics give rise to more expressive type systems, because they are designed to precisely account for resources.

For instance, the Clean programming language [26] relies upon a form of uniqueness to ensure equational reasoning in the presence of mutable data structures. The Cyclone programming language [17] uses unique pointers to allow fine-grained memory management. For example, a unique pointer may be updated from uninitialized to initialized, and its contents may also be deallocated:

```
x = malloc(4); // x: --- *‘U
*x = 3;        // x: int *‘U
free(x);       // x: undefined
```

In both of these languages, a unique object may be implicitly discarded, yielding a weak form of uniqueness called *affinity*.

The Vault programming language [13] uses tracked keys to enforce resource management protocols. For example, the following interface specifies that opening a file returns a new tracked key, which must be present when reading the file, and which is consumed when closing the file:

```
interface IO {
  type FILE;
  tracked($F) FILE open(string)  [ +$F ];
  char read (tracked($F) FILE)   [ $F ];
  void close (tracked($F) FILE)  [ -$F ]; }
```

[*] This material is based upon work supported by the Air Force Office of Scientific Research under Award No. F49620-03-1-0156 and Award No. F49620-01-1-0298 and by the Office of Naval Research under Award No. N00014-01-1-0968. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of these organizations or the U.S. Government.

ICFP'05 September 26–28, 2005, Tallinn, Estonia.
Copyright © 2005 ACM 1-59593-064-7/05/0009…$5.00.

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

Because tracked keys may be neither duplicated nor discarded, Vault supports a strong form of uniqueness technically termed *linearity*, which ensures that an opened file must be closed exactly once. Other projects [32, 12] have also incorporated linearity to ensure that memory is reclaimed.

Both forms of uniqueness (linearity and affinity) support *strong updates*, whereby the type of a stateful object is changed in response to stateful operations. For example, the Cyclone code fragment above demonstrates the type of the unique pointer changing from uninitialized to initialized (with an integer) in response to the assignment. The intuitive understanding is that a unique object cannot be duplicated, and thus there are no aliases to the object; hence, no other portion of the program may observe the change in the object's type, so it is safe to perform a strong update.

Yet, programming in a language with only unique (i.e., linear or affine) objects is much too painful. In such a setting, one can only construct tree-like data structures. Hence, it is not surprising that both Cyclone and Vault allow a programmer to put unique objects in shared objects, with a variety of restrictions to ensure that these mixed objects behave in a safe manner. In fact, understanding the various mechanisms by which unique objects (with strong updates) may safely coexist and mix with shared objects is currently an active area of research [5], though much of it has focused on high-level programming features, often without a complete formal account.

Therefore, it is natural to study a core language with mutable references of all sorts mentioned above: linear, affine, and unrestricted. The study of substructural logics immediately suggests one more sort — *relevant*, which describes data that may be duplicated but not implicitly discarded. Having made these distinctions, a number of design questions arise: What does it mean to duplicate or to discard a reference? What operations may be safely performed with the different sorts of references? What combinations of sorts for a reference and its contents are safe?

A major contribution of this paper is to answer these questions, giving an integrated design of references for all of these substructural sorts (Section 3). Our design allows unique (linear and affine) values to be stored in shared (unrestricted and relevant) references, while preserving the desirable feature that resources are tracked accurately. Our language extends a core λ-calculus with a straightforward type system that provides data of each of the substructural sorts mentioned above (Section 2). The key idea, present in other substructural type systems, is to break out the substructural sorts as type "qualifiers." Rather than prove soundness via a syntactic subject-reduction proof, we adopt an approach compatible with that used in Foundational Proof Carrying Code [6, 7]. We construct a step-indexed model (Section 4) where types are interpreted as sets of store description / value pairs, which are further refined using an index representing the number of steps available for future evaluation. We believe this model improves on previous models of mutable state, contributing a compositional notion of aliasing and ownership that directly addresses the subtleties of allowing unique values to be stored in shared references. Furthermore, we achieve a simple model, in comparison to denotational and domain-theoretic approaches, that easily extends to impredicative polymorphism and first-class references. Constructing a (well-founded) set-theoretic model means that our soundness and safety proofs are amenable to formalization in the higher-order logic of Foundational PCC. Hence, our work provides a useful foundation for future extensions of Foundational PCC, which currently only supports unrestricted references, but is an attractive target for source languages wishing to carry high-level security guarantees, enforced by type states and linear resources, through to machine code.

2. λ^{URAL}: A Substructural λ-Calculus

Advanced type systems for state rely upon limiting the ordering and number of uses of data and operations to ensure that state is handled in a safe manner. For example, (safely) deallocating a data structure requires that the data structure is never used in the future. In order to establish this property, a type system may ensure that the data structure is used *at most once*; after one use, the data structure may be safely deallocated, since there can be no further uses.

A *substructural* type system provides the core mechanisms necessary to restrict the number and order of uses of data and operations. A conventional type system, such as that employed by the simply-typed λ-calculus, with a typing judgement like $\Gamma \vdash e : \tau$, satisfies three structural properties:

Exchange *If* $\Gamma_1, x{:}\tau_x, y{:}\tau_y, \Gamma_2 \vdash e : \tau$,
then $\Gamma_1, y{:}\tau_y, x{:}\tau_x, \Gamma_2 \vdash e : \tau$.

Contraction *If* $\Gamma_1, x{:}\tau_z, y{:}\tau_z, \Gamma_2 \vdash e : \tau$,
then $\Gamma_1, z{:}\tau_z, \Gamma_2 \vdash e[z/x][z/y] : \tau$.

Weakening *If* $\Gamma \vdash e : \tau$, *then* $\Gamma, x{:}\tau_x \vdash e : \tau$.

In contrast, a substructural type system is designed so that one or more of these structural properties do not hold in general. Among the most widely studied substructural type systems are the *linear* type systems [29, 24], derived from Girard's linear logic [15], in which all variables satisfy **Exchange**, but linearly typed variables satisfy neither **Contraction** nor **Weakening**.

In this section, we present a *substructural* polymorphic λ-calculus, similar in spirit to Walker's linear lambda calculus [30]. In our calculus, types and variables are qualified as unrestricted (U), relevant (R), affine (A), or linear (L). All variables will satisfy **Exchange**, while only unrestricted variables will satisfy both **Contraction** and **Weakening**, allowing such variables to be used an arbitrary number of times. We will require

- linear variables to satisfy neither **Contraction** nor **Weakening**, ensuring that such variables are used exactly once,
- affine variables to satisfy **Weakening** (but not **Contraction**), ensuring that such variables are used at most once, and
- relevant variables to satisfy **Contraction** (but not **Weakening**), ensuring that such variables are used at least once.[2]

The diagram below demonstrates the relationship between these qualifiers, inducing a lattice ordering $\preceq$.

$$\begin{tikzcd}
 & \text{linear } (\mathsf{L}) \arrow[dl] \arrow[dr] & \\
\text{affine } (\mathsf{A}) \arrow[dr] & & \text{relevant } (\mathsf{R}) \arrow[dl] \\
 & \text{unrestricted } (\mathsf{U}) &
\end{tikzcd}$$

2.1 Syntax

Figure 1 presents the syntax for our core calculus, dubbed the λ^{URAL}-calculus. Most of the types, expressions, and values are based on a traditional polymorphic λ-calculus.

Kind and Type Levels We structure our types τ as a qualifier ξ applied to a pre-type $\overline{\tau}$, yielding the four sorts of types noted above. The qualifier of a type dictates the structural operations that may be applied to values of the type, while the pre-type dictates the introduction and elimination forms. The pre-types $\mathbf{1}_\otimes$, $\tau_1 \otimes \tau_2$, and $\tau_1 \multimap \tau_2$ correspond to the unit, pair, and function types of the polymorphic λ-calculus.

[2] In the logic community, it is perhaps more accurate to use the qualifier "strict" for such variables. However, "strict" is already an overloaded term in the functional programming community; so, like Walker [30], we use "relevant."

Kind Level:			
Kinds	κ	$::=$	$\mathsf{QUAL} \mid \overline{\star} \mid \star$
Type Level:			
Constant Qualifiers	q	$\in$	$\mathit{Quals} = \{\mathsf{U}, \mathsf{R}, \mathsf{A}, \mathsf{L}\}$
Qualifiers	ξ	$::=$	$\alpha \mid q$
PreTypes	$\overline{\tau}$	$::=$	$\alpha \mid \mathbf{1}_\otimes \mid \tau_1 \otimes \tau_2 \mid \tau_1 \multimap \tau_2 \mid \forall\alpha{:}\kappa.\, \tau$
Types	τ	$::=$	$\alpha \mid {}^{\xi}\overline{\tau}$
Type-level Terms	ι	$::=$	$\xi \mid \overline{\tau} \mid \tau$
Expression Level:			
Values	v	$::=$	$x \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid \lambda x.\, e \mid \Lambda.\, e$
Expressions	e	$::=$	$v \mid \mathtt{let}\ \langle\rangle = e_1\ \mathtt{in}\ e_2 \mid \mathtt{let}\ \langle x_1, x_2\rangle = e_1\ \mathtt{in}\ e_2 \mid e_1\, e_2 \mid e\,[]$

Figure 1. λ^{URAL} Syntax

Polymorphism over qualifiers, pre-types, and types is provided by a single pre-type $\forall\alpha{:}\kappa.\,\tau$; we introduce a kind level to distinguish among the type-level terms that may be used to instantiate a polymorphic pre-type (with kinds QUAL, $\overline{\star}$, and $\star$ for qualifiers, pre-types, and types, respectively).

In an accompanying technical report [3], we show that it is also easy to extend our results to include sum ($\tau_1 \oplus \tau_2$), existential ($\exists\alpha{:}\kappa.\,\tau$), and recursive ($\mu\alpha{:}\overline{\star}.\,\tau$) pre-types and recursive functions in the calculus, though we elide such constructs in this presentation.

This structuring of types as a qualifier applied to a pre-type follows that of Walker [30], but differs from other presentations of linear lambda calculi that use exactly one modality ($!\tau$) to distinguish unrestricted from linear types. It seems possible to introduce alternative modalities (e.g, $-\tau$ for affine and $+\tau$ for relevant), but then we would have to consider their interaction (e.g., what does $-!{+}\tau$ denote?). Also, with four distinct qualifiers, it is natural to introduce qualfier polymorphism, which is best formulated by separating qualifiers from pre-types.

Expression Level Each pre-type has an associated value introduction form. 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 units ($\mathbf{1}_\otimes$) and pairs ($\otimes$), respectively. As usual, a function with pre-type $\tau_1 \multimap \tau_2$ is eliminated via application $e_1\, e_2$, while a type-level abstraction $\forall\alpha{:}\kappa.\,\tau$ is eliminated via instantiation $e\,[]$.

Note that expressions are not decorated with type-level terms. This simplifies the semantic model presented in Section 4, where soundness is with respect to typing derivations, and is appropriate for an expressive “internal” language. We leave as an open problem the formulation of appropriate inference and elaboration algorithms yielding derivations in the type system of the next section, which would likely require some type-level annotations on expressions in a “surface” language.

2.2 Static Semantics

The goal of the type system for λ^{URAL} is to approximate the requirements of languages like Vault and Cyclone, which ensure that linear values are used exactly once, affine values are used at most once, and relevant values are used at least once. Dually, the type system should ensure that only unrestricted and relevant values are duplicated and only unrestricted and affine values are discarded. To prevent values from being implicitly copied or dropped when their containing value is duplicated or discarded, the type system must also ensure that a (functional) value with a qualifier lower in the lattice may not contain values with qualifiers higher in the lattice. For example, an affine (A) pair may not contain linear (L) components, since we could end up dropping the linear components by dropping the pair, so the type sytem must rule out expressions of type ${}^{\mathsf{A}}({}^{\mathsf{L}}\overline{\tau}_1 \otimes {}^{\mathsf{L}}\overline{\tau}_2)$.

$\boxed{\Delta \vdash \xi_1 \preceq \xi_2}$

$$\frac{\Delta \vdash \alpha : \mathsf{QUAL}}{\Delta \vdash \mathsf{U} \preceq \alpha} \qquad \frac{q_1 \preceq q_2}{\Delta \vdash q_1 \preceq q_2} \qquad \frac{\Delta \vdash \alpha : \mathsf{QUAL}}{\Delta \vdash \alpha \preceq \mathsf{L}}$$

$$\frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \xi \preceq \xi} \qquad \frac{\Delta \vdash \xi_1 \preceq \xi' \qquad \Delta \vdash \xi' \preceq \xi_2}{\Delta \vdash \xi_1 \preceq \xi_2}$$

$\boxed{\Delta \vdash \tau \preceq \xi}$

$$\frac{\Delta \vdash \alpha : \star}{\Delta \vdash \alpha \preceq \mathsf{L}} \qquad \frac{\Delta \vdash \overline{\tau}' : \overline{\star} \qquad \Delta \vdash \xi' \preceq \xi}{\Delta \vdash {}^{\xi'}\overline{\tau}' \preceq \xi}$$

$\boxed{\Delta \vdash \Gamma \preceq \xi}$

$$\frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \bullet \preceq \xi} \qquad \frac{\Delta \vdash \Gamma \preceq \xi \qquad \Delta \vdash \tau \preceq \xi}{\Delta \vdash \Gamma, x{:}\tau}$$

Figure 4. λ^{URAL} Statics (Sub-Qual Rules)

Despite these requirements, the type system is relatively simple. λ^{URAL} typing judgements have the form $\Delta; \Gamma \vdash e : \tau$ where the contexts Δ and Γ are defined as follows:

Type-level Term Context	Δ	$::=$	$\bullet \mid \Delta, \alpha{:}\kappa$
Value Context	Γ	$::=$	$\bullet \mid \Gamma, x{:}\tau$

Thus, Δ is used to track the set of type-level variables in scope (along with their kinds), whereas Γ, as usual, is used to track the set of (expression-level) variables in scope (along with their types). There may be at most one occurrence of a type-level variable α in Δ and, similarly, at most one occurrence of a variable x in Γ.

Figure 2 presents the λ^{URAL} kinding rules and Figure 3 presents the λ^{URAL} typing rules. In order to ensure the correct relationship between a data structure and its components, we extend the lattice ordering on constant qualifiers to types and contexts (see Figure 4). In the presence of qualifier and type polymorphism, we include the rules $\Delta \vdash \mathsf{U} \preceq \alpha$ and $\Delta \vdash \alpha \preceq \mathsf{L}$, a conservative extension, since U and L are the bottom and top of the lattice. A more general approach would incorporate bounded qualifier constraints, which we believe is straightforward, but doing so does not add to the discussion at hand.

As is usual in a substructural setting, our type system relies upon a judgement $\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2$ that splits the assumptions in Γ between the contexts Γ_1 and Γ_2 (see Figure 5). Splitting the context is necessary to ensure that variables are used appropriately by sub-expressions. Note that $\boxplus$ ensures that an A or L assumption appears in exactly one sub-context. On the other hand, U and R assumptions may appear in both sub-contexts, corresponding to implicit duplication of the variables.

$\boxed{\Delta \vdash \iota : \kappa}$

$$(\mathsf{VarKn})\,\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa} \qquad (\mathsf{Qual})\,\frac{}{\Delta \vdash q : \mathsf{QUAL}} \qquad (\mathsf{Type})\,\frac{\Delta \vdash \xi : \mathsf{QUAL} \qquad \Delta \vdash \overline{\tau} : \overline{\star}}{\Delta \vdash {}^{\xi}\overline{\tau} : \star}$$

$$(\mathsf{MUnitPTy})\,\frac{}{\Delta \vdash \mathbf{1}_{\otimes} : \overline{\star}} \qquad (\mathsf{MPairPTy})\,\frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \otimes \tau_2 : \overline{\star}} \qquad (\mathsf{FnPTy})\,\frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \multimap \tau_2 : \overline{\star}} \qquad (\mathsf{AllPTy})\,\frac{\Delta, \alpha{:}\kappa \vdash \tau : \star}{\Delta \vdash \forall \alpha{:}\kappa.\, \tau : \overline{\star}}$$

Figure 2. λ^{URAL} Statics (Kinding Rules)

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

$$(\mathsf{Var})\,\frac{\Delta \vdash \tau : \star}{\Delta; \bullet, x{:}\tau \vdash x : \tau} \qquad (\mathsf{MUnit})\,\frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta; \bullet \vdash \langle\rangle : {}^{\xi}\mathbf{1}_{\otimes}} \qquad (\mathsf{MPair})\,\frac{\begin{array}{cc} \Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 & \Delta \vdash \xi : \mathsf{QUAL} \\ \Delta; \Gamma_1 \vdash v_1 : \tau_1 & \Delta \vdash \tau_1 \preceq \xi \\ \Delta; \Gamma_2 \vdash v_2 : \tau_2 & \Delta \vdash \tau_2 \preceq \xi \end{array}}{\Delta; \Gamma \vdash \langle v_1, v_2\rangle : {}^{\xi}(\tau_1 \otimes \tau_2)}$$

$$(\mathsf{Fn})\,\frac{\Delta \vdash \xi : \mathsf{QUAL} \qquad \Delta \vdash \Gamma \preceq \xi \qquad \Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x.\, e : {}^{\xi}(\tau_1 \multimap \tau_2)} \qquad (\mathsf{All})\,\frac{\Delta \vdash \xi : \mathsf{QUAL} \qquad \Delta \vdash \Gamma \preceq \xi \qquad \Delta, \alpha{:}\kappa; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda.\, e : {}^{\xi}\forall \alpha{:}\kappa.\, \tau}$$

$$(\mathsf{Let\text{-}MUnit})\,\frac{\begin{array}{c} \Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \\ \Delta; \Gamma_1 \vdash e_1 : {}^{\xi}\mathbf{1}_{\otimes} \qquad \Delta; \Gamma_2 \vdash e_2 : \tau \end{array}}{\Delta; \Gamma \vdash \mathtt{let}\ \langle\rangle = e_1\ \mathtt{in}\ e_2 : \tau} \qquad (\mathsf{Let\text{-}MPair})\,\frac{\begin{array}{c} \Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \\ \Delta; \Gamma_1 \vdash e_1 : {}^{\xi}(\tau_1 \otimes \tau_2) \qquad \Delta; \Gamma_2, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau \end{array}}{\Delta; \Gamma \vdash \mathtt{let}\ \langle x_1, x_2\rangle = e_1\ \mathtt{in}\ e_2 : \tau}$$

$$(\mathsf{App})\,\frac{\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \qquad \Delta; \Gamma_1 \vdash e_1 : {}^{\xi}(\tau_1 \multimap \tau_2) \qquad \Delta; \Gamma_2 \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1\, e_2 : \tau_2} \qquad (\mathsf{Inst})\,\frac{\Delta; \Gamma \vdash e : {}^{\xi}\forall \alpha{:}\kappa.\, \tau \qquad \Delta \vdash \iota : \kappa}{\Delta; \Gamma \vdash e\,[] : \tau[\iota/\alpha]}$$

$$(\mathsf{Weak})\,\frac{\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \qquad \Delta; \Gamma_1 \vdash e : \tau \qquad \Delta \vdash \Gamma_2 \preceq \mathsf{A}}{\Delta; \Gamma \vdash e : \tau}$$

Figure 3. λ^{URAL} Static Semantics (Typing Rules)

$\boxed{\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2}$

$$\frac{}{\Delta \vdash \bullet \leadsto \bullet \boxplus \bullet} \qquad \frac{\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \qquad \Delta \vdash \tau : \star}{\Delta \vdash \Gamma, x{:}\tau \leadsto \Gamma_1, x{:}\tau \boxplus \Gamma_2}$$

$$\frac{\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \qquad \Delta \vdash \tau : \star}{\Delta \vdash \Gamma, x{:}\tau \leadsto \Gamma_1 \boxplus \Gamma_2, x{:}\tau}$$

$$\frac{\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta \vdash \Gamma, x{:}\tau \leadsto \Gamma_1, x{:}\tau \boxplus \Gamma_2, x{:}\tau}$$

Figure 5. λ^{URAL} Statics (Context Split Rules)

The rule (MPair) is representative: the context is split by $\boxplus$ to type each of the pair components, and the types of each component are bounded by the qualifier assigned to the pair. Intuitively, the L and A assumptions in the context are exclusively "owned" by exactly one of the two components. Likewise, in the rule (Fn), the free variables of Γ, which constitute the closure of the function, must be bounded by the qualifier assigned to the function. Note that the qualifier assigned to a function type is unrelated to the types of the argument and result; rather, it is related to the abstracted components that are used when the function is executed.

The rule (Weak) splits the context into a sub-context used to type the expression e and a discardable sub-context, consisting of U and A variables, that are not required to type the expression. Note that the rule (Weak) acts as a strengthened **Weakening** property, allowing an arbitrary number of U and A variables to be dropped at once. The corresponding strengthened **Contraction** property is incorporated into the judgement $\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2$, which allows an arbitrary number of U and R variables to be copied at once.

3. $\lambda^{\mathsf{refURAL}}$: A Substructural λ-Calculus with References

Languages like Vault and Cyclone include objects that change state (e.g., file descriptors), so it is natural to include some stateful values. We consider the difficult case of references, which can serve as mutable containers for both functional values and stateful values. Hence, we extend the λ^{URAL}-calculus with mutable references, to yield the $\lambda^{\mathsf{refURAL}}$-calculus. The reference pre-type $\mathsf{ref}\ \tau$ may be combined with a qualifier ξ to yield the four sorts (U, R, A, L) of references discussed earlier. We also introduce operations to allocate ($\mathtt{new}_q$) and deallocate ($\mathtt{free}$) references, as well as to read ($\mathtt{rd}$), write ($\mathtt{wr}$), and swap ($\mathtt{sw}$) their contents. Not all of these operations can be safely performed with all sorts of references, as we discuss in Section 3.2. The syntactic extensions to support references are as follows:

Type Level:			
PreTypes	$\overline{\tau}$	$::=$	$\ldots \mid \mathsf{ref}\ \tau$
Expression Level:			
Locations	l	$\in$	*Locs*
Values	v	$::=$	$\ldots \mid l$
Expressions	e	$::=$	$\ldots \mid \mathtt{new}_q\, e \mid \mathtt{free}\, e \mid$
			$\mathtt{rd}\, e \mid \mathtt{wr}\, e_1\, e_2 \mid \mathtt{sw}\, e_1\, e_2$

3.1 Operational Semantics

Figure 6 gives the small-step operational semantics for $\lambda^{\mathsf{refURAL}}$ as a relation between configurations of the form (s, e), where

$$\begin{array}{llcl}
\textit{Store} & s & ::= & \{l_1 \mapsto (q_1, v_1), \ldots, l_n \mapsto (q_n, v_n)\}
\end{array}$$

$$\begin{array}{ll}
\text{(let-munit)} & (s, \mathtt{let}\ \langle\rangle = \langle\rangle\ \mathtt{in}\ e) \longmapsto (s, e) \\
\text{(let-mpair)} & (s, \mathtt{let}\ \langle x_1, x_2\rangle = \langle v_1, v_2\rangle\ \mathtt{in}\ e) \longmapsto \\
& \qquad (s, e[v_1/x_1][v_2/x_2]) \\
\text{(app)} & (s, (\lambda x.\, e)\, v) \longmapsto (s, e[v/x]) \\
\text{(inst)} & (s, (\Lambda.\, e)\, []) \longmapsto (s, e) \\
\text{(new)} & (s, \mathtt{new}_q\, v) \longmapsto (s \uplus \{l \mapsto (q, v)\}, l) \\
\text{(free)} & (s \uplus \{l \mapsto (q, v)\}, \mathtt{free}\, l) \longmapsto (s, v) \\
\text{(read)} & (s \uplus \{l \mapsto (q, v)\}, \mathtt{rd}\, l) \longmapsto \\
& \qquad (s \uplus \{l \mapsto (q, v)\}, \langle l, v\rangle) \\
\text{(write)} & (s \uplus \{l \mapsto (q, v_1)\}, \mathtt{wr}\, l\, v_2) \longmapsto \\
& \qquad (s \uplus \{l \mapsto (q, v_2)\}, l) \\
\text{(swap)} & (s \uplus \{l \mapsto (q, v_1)\}, \mathtt{sw}\, l\, v_2) \longmapsto \\
& \qquad (s \uplus \{l \mapsto (q, v_2)\}, \langle l, v_1\rangle) \\
\text{(ctxt)} & \dfrac{(s, e) \longmapsto (s', e')}{(s, E[e]) \longmapsto (s', E[e'])}
\end{array}$$

Figure 6. $\lambda^{\mathsf{refURAL}}$ Operational Semantics

	Ref	Ops	Contents and Ops			
			U	R	A	L
shared	U	`new`$_\mathsf{U}$ *(weak updates)*	`rd` `wr` `sw`	X	`wr` `sw`	X
shared	R	`new`$_\mathsf{R}$ *(weak updates)*	`rd` `wr` `sw`	`rd` `sw`	`wr` `sw`	`sw`
unique	A	`new`$_\mathsf{A}$ `free` *(strong updates)*	`rd` `wr` `sw`	X	`wr` `sw`	X
unique	L	`new`$_\mathsf{L}$ `free` *(strong updates)*	`rd` `wr` `sw`	`rd` `sw`	`wr` `sw`	`sw`

Figure 7. Operations for Substructural State

s is a global store mapping locations to qualifiers and values.[3] The notation $s_1 \uplus s_2$ denotes the disjoint union of the stores s_1 and s_2; the operation is undefined if the domains of s_1 and s_2 are not disjoint. We use evaluation contexts E (omitted in this presentation) to lift the primitive rewriting rules to a standard, left-to-right, innermost-to-outermost, call-by-value interpretation of the language.

Most of the rules are standard, so we highlight only those involving references. The expressions `new`$_q\, e$ and `free` e perform the complementary actions of allocating and deallocating mutable references in the global store. Specifically, the expression `new`$_q\, e$ evaluates e to a value v, allocates a fresh (unallocated) location l to store the qualifier q and value v, and returns l. The expression `free` e performs the reverse: it evaluates e to a location l, deallocates l, and returns the value previously stored at l.

The expressions for reading and writing a mutable reference *implicitly* duplicate and discard (respectively) the contents of the reference. The expression `rd` e evaluates e to a location l, duplicates the value v stored at l, and returns $\langle l, v\rangle$, leaving the value stored at l unchanged. Meanwhile, `wr` $e_1\, e_2$ evaluates e_1 to a location l and e_2 to value v_2, stores v_2 at location l, discards the value previously stored at l, and returns l.

In languages with only unrestricted (ML-style) references, it is customary for `rd` to return only the contents of l and for `wr` to return $\langle\rangle$. However, we do not wish to consider reading or writing a linear (resp. affine) reference as the exactly-one-use (resp. at-least-one-use) of the value. Therefore, the `rd` and `wr` (and `sw`) operations return the location l that was read or written, which remains available for future use. The behavior of ML-style references may be recovered by implicitly discarding the returned location.

The expression `sw` $e_1\, e_2$ combines the operations of dereferencing and updating a mutable reference, but has the attractive property that it neither duplicates nor discards a value. Notice that performing a write or swap operation on a location may change the type of the location's contents. The static semantics will permit weak (type-invariant) updates on all references (with some additional caveats), but will restrict strong (type-varying) updates to unique references.

[3] We write $s^{\mathsf{qual}}(l)$ and $s^{\mathsf{val}}(l)$ for the respective projections of $s(l)$.

The reader may well wonder why each reference is "stamped" with a qualifier at its allocation when the remainder of the operational rules are entirely agnostic with respect to a reference's qualifier. Essentially, the qualifier is a form of instrumentation, which, when combined with the semantic model presented in Section 4, allows us to guarantee that linear and relevant references cannot be implicitly discarded. Such a property is difficult to capture exclusively in the operational semantics (i.e., by ensuring that the abstract machine "gets stuck" when a linear or relevant reference is implicitly dropped). On the other hand, the abstract machine does "get stuck" when attempting to access a reference after it has been deallocated.

3.2 Static Semantics

As with the type system for λ^{URAL}, we would like the type system for $\lambda^{\mathsf{refURAL}}$ to ensure the property that no linear or affine value is implicitly duplicated and no linear or relevant value is implicitly discarded. With that in mind — and noting that only unrestricted and relevant references may be implicitly copied (by the $\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$ judgement), while only unrestricted and affine references may be implicitly dropped (by the (Weak) rule) — we now answer the questions we laid out in Section 1: What operations may be safely performed with the different sorts of references? What combinations of sorts for a reference and its contents are safe? These answers are summarized in Figure 7.

First, consider what it means to duplicate a reference. Operationally, a reference is a location in the global store. Therefore, duplicating an unrestricted or relevant reference l, simply yields two copies of l — while the value stored at l is *not* duplicated. Since duplicating a shared reference does not alter the uniqueness of its contents, it is not only reasonable but also extremely useful to allow shared references to store unique values. In particular, it permits the sharing of (large) unique data structures without expensive copying.

On the other hand, dropping an unrestricted or affine reference l effectively drops its contents, since this reference may (must, in the case of affine) have been the only copy of l. If the contents were a linear or relevant value, then the exactly-one-use and at-least-one-use invariants (respectively) would be violated. Hence, we cannot allow linear and relevant values (which cannot be discarded) to be stored in unrestricted or affine references (which can be discarded).

Considering yet another axis, we note that linear and affine references must be unique. Hence, we can `free` unique references, and also perform strong updates on them. Shared references, on the other hand, can never be deallocated and can only support weak updates.

As we noted above, the `rd` operator induces an implicit copy while the `wr` operator induces an implicit drop. Therefore, whether

$\boxed{\Delta \vdash \iota : \kappa}$

$$(\mathsf{RefPTy})\ \frac{\Delta \vdash \tau : \star}{\Delta \vdash \mathsf{ref}\ \tau : \overline{\star}}$$

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

$$(\mathsf{New(U,A)})\ \frac{q \preceq \mathsf{A} \qquad \Delta;\Gamma \vdash e : \tau \qquad \Delta \vdash \tau \preceq \mathsf{A}}{\Delta;\Gamma \vdash \mathtt{new}_q\, e : {}^{q}\mathsf{ref}\ \tau} \qquad (\mathsf{New(R,L)})\ \frac{\mathsf{R} \preceq q \qquad \Delta;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \mathtt{new}_q\, e : {}^{q}\mathsf{ref}\ \tau}$$

$$(\mathsf{Free})\ \frac{\Delta;\Gamma \vdash e : {}^{\xi}\mathsf{ref}\ \tau \qquad \Delta \vdash \mathsf{A} \preceq \xi}{\Delta;\Gamma \vdash \mathtt{free}\, e : \tau} \qquad (\mathsf{Read})\ \frac{\Delta;\Gamma \vdash e : {}^{\xi}\mathsf{ref}\ \tau \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta;\Gamma \vdash \mathtt{rd}\, e : {}^{\mathsf{L}}({}^{\xi}\mathsf{ref}\ \tau \otimes \tau)}$$

$$(\mathsf{Write(Strong)})\ \frac{\begin{array}{c}\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \\ \Delta;\Gamma_1 \vdash e_1 : {}^{\xi}\mathsf{ref}\ \tau_1 \qquad \Delta \vdash \tau_1 \preceq \mathsf{A} \qquad \Delta \vdash \mathsf{A} \preceq \xi \\ \Delta;\Gamma_2 \vdash e_2 : \tau_2 \qquad \Delta \vdash \tau_2 \preceq \xi\end{array}}{\Delta;\Gamma \vdash \mathtt{wr}\, e_1\, e_2 : {}^{\xi}\mathsf{ref}\ \tau_2} \qquad (\mathsf{Write(Weak)})\ \frac{\begin{array}{c}\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \\ \Delta;\Gamma_1 \vdash e_1 : {}^{\xi}\mathsf{ref}\ \tau \qquad \Delta \vdash \tau \preceq \mathsf{A} \\ \Delta;\Gamma_2 \vdash e_2 : \tau\end{array}}{\Delta;\Gamma \vdash \mathtt{wr}\, e_1\, e_2 : {}^{\xi}\mathsf{ref}\ \tau}$$

$$(\mathsf{Swap(Strong)})\ \frac{\begin{array}{c}\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \\ \Delta;\Gamma_1 \vdash e_1 : {}^{\xi}\mathsf{ref}\ \tau_1 \qquad \Delta \vdash \mathsf{A} \preceq \xi \\ \Delta;\Gamma_2 \vdash e_2 : \tau_2 \qquad \Delta \vdash \tau_2 \preceq \xi\end{array}}{\Delta;\Gamma \vdash \mathtt{sw}\, e_1\, e_2 : {}^{\mathsf{L}}({}^{\xi}\mathsf{ref}\ \tau_2 \otimes \tau_1)} \qquad (\mathsf{Swap(Weak)})\ \frac{\begin{array}{c}\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \\ \Delta;\Gamma_1 \vdash e_1 : {}^{\xi}\mathsf{ref}\ \tau \\ \Delta;\Gamma_2 \vdash e_2 : \tau\end{array}}{\Delta;\Gamma \vdash \mathtt{sw}\, e_1\, e_2 : {}^{\mathsf{L}}({}^{\xi}\mathsf{ref}\ \tau \otimes \tau)}$$

Figure 8. $\lambda^{\mathsf{refURAL}}$ Static Semantics (Kinding and Typing Rules)

we can read from or write to a reference depends entirely on the qualifier of its contents: `rd` is permitted if the contents are unrestricted or relevant (i.e., duplicable), `wr` is permitted if the contents are unrestricted or affine (i.e., discardable). The operation `sw` is permitted on any sort of reference, regardless of the qualifier of its contents. As noted above, strong writes and strong swaps, which change the type of the contents of the location, are only permitted on unique references.

Figure 8 gives the additional typing rules for $\lambda^{\mathsf{refURAL}}$. We note that the typing rules for core λ^{URAL} terms remain unchanged. There is no rule for locations, as locations are not allowed in the external language. Also note that the (New) and (Free) rules act as the introduction and elimination rules for ${}^{\xi}\mathsf{ref}\ \tau$ types, while the (Read), (Write), and (Swap) rules maintain an exactly-one-use invariant on references by consuming a value of type ${}^{\xi}\mathsf{ref}\ \tau_1$ and by producing a value of type ${}^{\xi}\mathsf{ref}\ \tau_2$ (possibly with $\tau_1 = \tau_2$).

Finally, we note that `wr` may be encoded using an explicit `sw` and an implicit drop:[4]

$$(\mathsf{Write(Weak)})\ \frac{\begin{array}{c}\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \\ \Delta;\Gamma_1 \vdash e_1 : {}^{\xi}\mathsf{ref}\ \tau \qquad \Delta \vdash \tau \preceq \mathsf{A} \\ \Delta;\Gamma_2 \vdash e_2 : \tau\end{array}}{\Delta;\Gamma \vdash \mathtt{wr}\, e_1\, e_2 : {}^{\xi}\mathsf{ref}\ \tau}$$

$$\begin{array}{ll}\overset{\text{def}}{=} & \mathtt{let}\ \langle r, x\rangle = \mathtt{sw}\, e_1\, e_2\ \mathtt{in}\quad \text{// using (Swap(Weak))} \\ & \text{// drop } x \text{, noting } \Delta \vdash \tau \preceq \mathsf{A} \\ & r\end{array}$$

However, `rd` may not be encoded using an explicit `sw` and an implicit copy, as a suitable (discardable) dummy value cannot in general be synthesized.

$$(\mathsf{Read})\ \frac{\Delta;\Gamma \vdash e : {}^{\xi}\mathsf{ref}\ \tau \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta;\Gamma \vdash \mathtt{rd}\, e : {}^{\mathsf{L}}({}^{\xi}\mathsf{ref}\ \tau \otimes \tau)}$$

$$\begin{array}{ll}\overset{\text{def}}{=} & \mathtt{let}\ \langle r, x\rangle = \mathtt{sw}\, e\ ?\ \mathtt{in}\quad \text{// where } \Delta;\Gamma \vdash\ ? : \tau \\ & \text{// copy } x \text{, noting } \Delta \vdash \tau \preceq \mathsf{R} \\ & \mathtt{let}\ \langle r, y\rangle = \mathtt{sw}\, r\, x\ \mathtt{in}\quad \text{// using (Swap(Weak))} \\ & \text{// drop } y \text{, but not necessarily } \Delta \vdash \tau \preceq \mathsf{A} \\ & \langle r, x\rangle\end{array}$$

[4] The encoding of a `wr` typed by the (Write(Strong)) rule makes use of the same term, but an alternate typing derivation.

4. A Step-Indexed Model

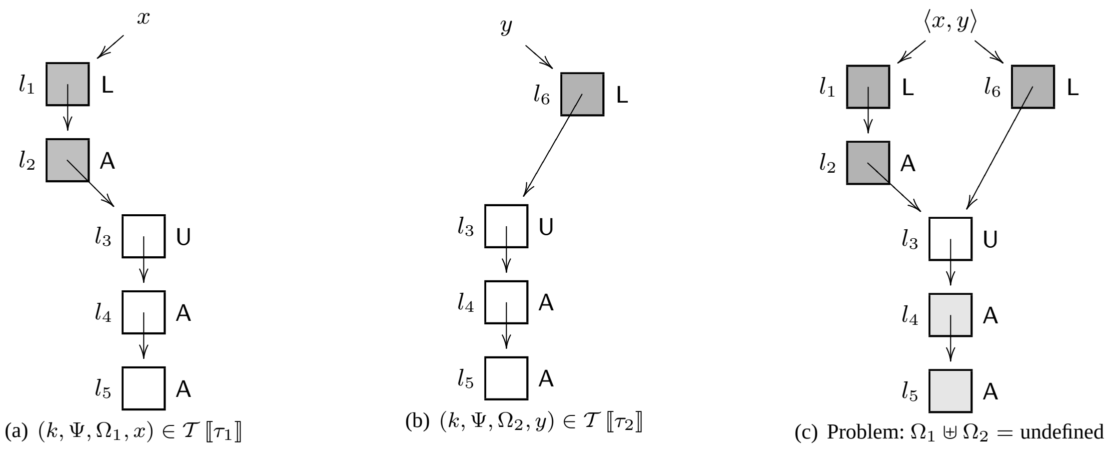
We prove the type soundness of $\lambda^{\mathsf{refURAL}}$ in a manner similar to that employed by Appel's Foundational PCC project [6]. The technique uses syntactic logical relations (that is, relations based on the operational semantics) where relations are further refined by an index that, intuitively, records the number of steps available for future evaluation. This stratification is essential for modeling the recursive functions (available via backpatching unrestricted references) and impredicative polymorphism present in the language.

4.1 Background: A Model of Unrestricted References

Our model is based on the indexed model of ML-style references by Ahmed, Appel, and Virga [1, 4], henceforth AAV. In their model, the semantic interpretation $\mathcal{T}[\![\tau]\!]$ of a (closed) type τ is a set of triples of the form (k, Ψ, v), where, k is a natural number (called the *approximation index* or *step index*), Ψ is a (global) store typing that maps locations to (the interpretation of) their designated types, and v is a (closed) value. Intuitively, $(k, \Psi, v) \in \mathcal{T}[\![\tau]\!]$ says that in any computation running for no more than k steps, v cannot be distinguished from values of type τ. Furthermore, since dereferencing a location consumes an execution step, in order to determine whether v has type τ for k steps it suffices to know the type of each store location for $k-1$ steps; hence, Ψ need only specify each location's type to approximation $k-1$. We use a similar indexing approach which is key to ensuring that our model is well-founded (as we shall demonstrate in Section 4.3).

4.2 Towards a Model of $\lambda^{\mathsf{refURAL}}$

Aliasing and Ownership Though our model is similar to AAV, the presence of shared and unique references places very different demands on the model, which we illustrate by considering the interpretation of product types in various settings. In a language with *only unrestricted* references (e.g. AAV), one would say $(k, \Psi, \langle v_1, v_2\rangle) \in \mathcal{T}[\![\tau_1 \otimes \tau_2]\!]$ if and only if $(k, \Psi, v_1) \in \mathcal{T}[\![\tau_1]\!]$ and $(k, \Psi, v_2) \in \mathcal{T}[\![\tau_2]\!]$, where the store typing Ψ describes *every* location allocated by the program thus far. In this setting, every location (in Ψ) may be *aliased;* hence, the model allows v_1 and v_2 to point to data structures that overlap in the heap.



x
l_1 L
l_2 A
l_3 U
l_4 A
l_5 A
(a) $(k, \Psi, \Omega_1, x) \in \mathcal{T}[\![\tau_1]\!]$
y
l_6 L
l_3 U
l_4 A
l_5 A
(b) $(k, \Psi, \Omega_2, y) \in \mathcal{T}[\![\tau_2]\!]$
$\langle x, y\rangle$
l_1 L
l_6 L
l_2 A
l_3 U
l_4 A
l_5 A
(c) Problem: $\Omega_1 \uplus \Omega_2$ = undefined

Figure 9. Unique References in Shared References: Aliased or Owned?

In a language with *only linear* references [23, 2], however, one must ensure that the set of (linear) locations reachable from v_1 is disjoint from the set of locations reachable from v_2. This mirrors the fact that we can only construct tree-like data structures in this setting. Furthermore, it guarantees the safety of strong updates by providing a notion of *exclusive ownership*. Hence, to model a language with only linear references, it is useful to replace the global store description Ψ with a description of only the *accessible* (reachable) locations in the store, say Ω. Intuitively, when we write $(k, \Omega, v) \in \mathcal{T}[\![\tau]\!]$, we intend for Ω to describe only the subset of store locations that are accessible from, and hence, "owned" by v. Thus, one would say $(k, \Omega, \langle v_1, v_2\rangle) \in \mathcal{T}[\![\tau_1 \otimes \tau_2]\!]$ if and only if $(k, \Omega_1, v_1) \in \mathcal{T}[\![\tau_1]\!]$ and $(k, \Omega_2, v_2) \in \mathcal{T}[\![\tau_2]\!]$, where the Ω is the disjoint union of Ω_1 and Ω_2.

For the $\lambda^{\mathsf{refURAL}}$-calculus, we tried to build a model that supports both aliasing and ownership as follows. We defined the semantic interpretation of a type $\mathcal{T}[\![\tau]\!]$ as the set of tuples of the form (k, Ψ, Ω, v) where Ψ describes every U and R location allocated by the program and Ω describes only those A and L locations that are reachable from (and owned by) v. The interpretation of $\tau_1 \otimes \tau_2$ then naturally yields: $(k, \Psi, \Omega, \langle v_1, v_2\rangle) \in \mathcal{T}[\![\tau_1 \otimes \tau_2]\!]$ if and only if $(k, \Psi, \Omega_1, v_1) \in \mathcal{T}[\![\tau_1]\!]$ and $(k, \Psi, \Omega_2, v_2) \in \mathcal{T}[\![\tau_2]\!]$, where the Ω is the disjoint union of Ω_1 and Ω_2.

Unfortunately, the above model did not suffice for $\lambda^{\mathsf{refURAL}}$, since it assumes that every unique location reachable from v is exclusively owned by v, which is not the case when unique references may be stored in shared references.

Unique References in Shared References: Aliased or Owned? Consider the situation depicted in Figure 9(a) where x maps to l_1 and locations l_1 through l_5 are reachable from x. Locations "owned" by x are shaded. Notice that l_1 and l_2 are unique locations owned by x, while l_4 and l_5 are unique locations that x must consider aliased, since they can be reached (from other program subexpressions) via the unrestricted location l_3. Figure 9(b) depicts such a subexpression, y. Note that y maps to l_6 whose contents alias l_3, making l_4 and l_5 reachable from y.

In $\lambda^{\mathsf{refURAL}}$ we may safely construct the pair $\langle x, y\rangle$ (shown in Figure 9(c)), but the interpretation of $\tau_1 \otimes \tau_2$ that we proposed above prohibits such a pair since locations l_4 and l_5 occur in both Ω_1 and Ω_2, violating the requirement that their domains be disjoint.

To model the $\lambda^{\mathsf{refURAL}}$-calculus, we tried to further refine our model so that the interpretation of a type $\mathcal{T}[\![\tau]\!]$ is a set of tuples of the form $(k, \Psi, \Omega, \Theta, v)$ where Ψ is as before, but now Ω describes unique *owned* locations, (i.e., those reachable from v without indirecting through a shared reference), while Θ describes unique *aliased* locations, (i.e., those that *cannot* be reached without indirecting through a shared cell). The intuition is that the interpretation of $\tau_1 \otimes \tau_2$ splits Ω into disjoint pieces for each component of the pair, but allows each component to use Ψ and Θ unchanged.

This proposal, however, is fraught with complications. In particular, whether a unique location belongs in Ω or Θ depends on the configuration of the entire program, rather than just the type of the location. This limits the compositionality of the model. For instance, consider l_5 in Figure 9(c). Clearly l_5 must appear in Θ as it is reachable from an unrestricted location. However, if locations l_1, l_2, l_3, and l_6 did not exist, then l_5 could appear in Ω. In the next section, we propose a far simpler solution that we consider one of the main technical contributions of our work.

4.3 A Model with Local Store Descriptions

In our model of the $\lambda^{\mathsf{refURAL}}$-calculus, the semantic interpretation of a type $\mathcal{T}[\![\tau]\!]$ is a set of tuples of the form (k, q, ψ, v), where the *local store description* ψ describes only a part of the global store. Intuitively, ψ is the set of "beliefs" about the locations that appear as sub-expressions of the value v. Such locations are said to be *directly accessible* from the value v. Conversely, locations that are *indirectly accessible* from the value v are those locations that are reachable from v only by indirecting through one (or more) references. The local store description ψ says nothing about these indirectly-accessible locations. This enhances the compositionality of our model, making it straightforward to combine local store descriptions with one another.

4.3.1 Definitions

We use the meta-variable χ to denote sets of tuples of the form (k, q, ψ, v) and the meta-variable ψ to denote partial maps from locations l to tuples of the form (q, χ).[5] When χ corresponds to the semantic interpretation of a type and $(k, q, \psi, v) \in \chi$, we intend that q is the qualifier of v, ψ is the local store description of v, and v is a closed value. When ψ corresponds to a local store description and $\psi(l) = (q, \chi)$, we intend that q is the qualifier of the reference and χ is the semantic interpretation of the type of its contents.

[5] We write $\psi^{\mathsf{qual}}(l)$ and $\psi^{\mathsf{type}}(l)$ for the respective projections of $\psi(l)$.

(a)

$$\begin{array}{lcll} \textit{PreType/Type Interpretation (Notation)} & \chi & ::= & \{(k, q, \psi, v), \ldots\} \\ \textit{Local Store Description (Notation)} & \psi & ::= & \{l \mapsto (q, \chi), \ldots\} \end{array}$$

(b)

$$\begin{array}{rcll} \mathit{CandAtom}_k & \overset{\text{def}}{=} & \{(j, q, \psi, v) \in \mathbb{N} \times \mathit{Quals} \times \bigcup_{j<k} \mathit{CandLocalStoreDesc}_j \times \mathit{CValues} \mid & \\ & & \quad j < k \wedge \psi \in \mathit{CandLocalStoreDesc}_j\} & \\ \mathit{CandUberType}_k & \overset{\text{def}}{=} & 2^{\mathit{CandAtom}_k} & \\ \mathit{CandLocalStoreDesc}_k & \overset{\text{def}}{=} & \mathit{Locs} \rightharpoonup \mathit{Quals} \times \mathit{CandUberType}_k & \\ & & & \\ \mathit{CandAtom}_\omega & \overset{\text{def}}{=} & \bigcup_{k \geq 0} \mathit{CandAtom}_k & \\ \mathit{CandUberType}_\omega & \overset{\text{def}}{=} & 2^{\mathit{CandAtom}_\omega} & \supseteq \bigcup_{k \geq 0} \mathit{CandUberType}_k \\ \mathit{CandLocalStoreDesc}_\omega & \overset{\text{def}}{=} & \mathit{Locs} \rightharpoonup \mathit{Quals} \times \mathit{CandUberType}_\omega & \supseteq \bigcup_{k \geq 0} \mathit{CandLocalStoreDesc}_k \end{array}$$

(c)

$$\begin{array}{rcl} \lfloor \chi \rfloor_k & \overset{\text{def}}{=} & \{(j, q, \psi, v) \mid j < k \wedge (j, q, \psi, v) \in \chi\} \\ & \in & \mathit{CandUberType}_\omega \to \mathit{CandUberType}_k \\ & & \\ \lfloor \psi \rfloor_k & \overset{\text{def}}{=} & \{l \mapsto (q, \lfloor \chi \rfloor_k) \mid l \in \mathit{dom}(\psi) \wedge \psi(l) = (q, \chi)\} \\ & \in & \mathit{CandLocalStoreDesc}_\omega \to \mathit{CandLocalStoreDesc}_k \\ & & \\ \mathcal{P}(q, \psi) & \overset{\text{def}}{=} & \forall l \in \mathit{dom}(\psi).\ \psi^{\mathsf{qual}}(l) \preceq q \\ & \in & \mathit{Quals} \times \mathit{CandLocalStoreDesc}_\omega \to \mathbb{P} \\ & & \\ \mathcal{R}(\psi) & \overset{\text{def}}{=} & \forall l \in \mathit{dom}(\psi).\ (\psi^{\mathsf{qual}}(l) \preceq \mathsf{A} \Rightarrow \forall (\_, q', \_, \_) \in \psi^{\mathsf{type}}(l).\ q' \preceq \mathsf{A}) \\ & \in & \mathit{CandLocalStoreDesc}_\omega \to \mathbb{P} \end{array}$$

(d)

$$\begin{array}{rcll} \mathit{Atom}_k & \overset{\text{def}}{=} & \{(j, q, \psi, v) \in \mathit{CandAtom}_k \mid \psi \in \mathit{LocalStoreDesc}_j \wedge \mathcal{P}(q, \psi)\} & \subseteq \mathit{CandAtom}_k \\ \mathit{PreType}_k & \overset{\text{def}}{=} & \{\chi \in 2^{\mathit{Atom}_k} \mid \forall (j, q, \psi, v) \in \chi.\ \forall i \leq j.\ (i, q, \lfloor \psi \rfloor_i, v) \in \chi\} & \subseteq \mathit{CandUberType}_k \\ \mathit{Type}_k & \overset{\text{def}}{=} & \{\chi \in \mathit{PreType}_k \mid \exists q' \in \mathit{Quals}.\ \forall (\_, q, \_, \_) \in \chi.\ q = q'\} & \subseteq \mathit{CandUberType}_k \\ \mathit{LocalStoreDesc}_k & \overset{\text{def}}{=} & \{\psi \in \mathit{Locs} \rightharpoonup \mathit{Quals} \times \mathit{Type}_k \mid \mathcal{R}(\psi)\} & \subseteq \mathit{CandLocalStoreDesc}_k \\ & & & \\ \mathit{PreType} & \overset{\text{def}}{=} & \{\chi \in \mathit{CandUberType}_\omega \mid \forall k \geq 0.\ \lfloor \chi \rfloor_k \in \mathit{PreType}_k\} & \supseteq \bigcup_{k \geq 0} \mathit{PreType}_k \\ \mathit{Type} & \overset{\text{def}}{=} & \{\chi \in \mathit{CandUberType}_\omega \mid \forall k \geq 0.\ \lfloor \chi \rfloor_k \in \mathit{Type}_k\} & \supseteq \bigcup_{k \geq 0} \mathit{Type}_k \end{array}$$

Figure 10. $\lambda^{\mathsf{refURAL}}$ Model (Definitions)

Well-Founded & Well-Behaved Interpretations If we attempt to naïvely construct a set-theoretic model based on these intentions, we are led to specify:

$$\begin{array}{rcl} \mathit{Type} & = & 2^{\mathbb{N} \times \mathit{Quals} \times \mathit{LocalStoreDesc} \times \mathit{CValues}} \\ \mathit{LocalStoreDesc} & = & \mathit{Locs} \rightharpoonup \mathit{Quals} \times \mathit{Type} \end{array}$$

However, there is a problem with this specification: a simple diagonalization argument will show that the set Type of type interpretations has an inconsistent cardinality (i.e., it's an ill-founded recursive definition).

We can eliminate the inconsistency by stratifying our definitions, making essential use of the approximation index. To simplify the development, we first construct *candidate* sets, which are well-founded sets of our intended form. Next, we define some useful functions and predicates on these candidate sets. Finally, we construct our semantic interpretations by filtering the candidate sets, making use of the functions and predicates defined in the previous step. Our semantic interpretations impose a number of constraints (e.g., relating the qualifier of a reference to the qualifier of its contents) that are ignored in the construction of the candidate sets.

Figure 10(b) defines our candidate sets by (strong) induction on k. Note that elements of $\mathit{CandAtom}_k$ are tuples with approximation index j strictly less than k. Hence, our definitions are well-defined at $k = 0$:

$$\begin{array}{rcl} \mathit{CandAtom}_0 & = & \emptyset \\ \mathit{CandUberType}_0 & = & \{\emptyset\} \\ \mathit{CandLocalStoreDesc}_0 & = & \mathit{Locs} \rightharpoonup \mathit{Quals} \times \{\emptyset\} \end{array}$$

While our candidate sets establish the existence of sets of our intended form, our semantic interpretations will need to be well-behaved in other ways. There are key constraints associated with atoms, pre-types, types, and local store descriptions that will be enforced in our final definitions. Functions and predicates supporting these constraints are given in Figure 10(c).

For any set χ, we define the k-approximation of the set (written $\lfloor \chi \rfloor_k$) as the subset of its elements whose indices are less than k; we extend the notion pointwise to local store descriptions ψ (written $\lfloor \psi \rfloor_k$). Note that $\lfloor \chi \rfloor_k$ and $\lfloor \psi \rfloor_k$ necessarily yield elements of $\mathit{CandUberType}_k$ and $\mathit{CandLocalStoreDesc}_k$.

Figure 10(c) defines our semantic interpretations, again by (strong) induction on k. Note that our semantic interpretations can be seen as filtering their corresponding candidate sets. Next, we examine each of these filtering constraints.

Recall that we intend for Atom_k to define tuples of the form (j, q, ψ, v) where q is the qualifier of v and ψ is the local store

$$\mathcal{K}[\![\mathsf{QUAL}]\!] = \mathit{Quals} \qquad \mathcal{K}[\![\bar{\star}]\!] = \mathit{PreType} \qquad \mathcal{K}[\![\star]\!] = \mathit{Type}$$

$$\begin{aligned}
\mathcal{T}[\![\Delta \vdash \alpha : \kappa]\!]\,\delta &= \delta(\alpha) \\
\mathcal{T}[\![\Delta \vdash q : \mathsf{QUAL}]\!]\,\delta &= q \\
\mathcal{T}[\![\Delta \vdash \mathbf{1}_\otimes : \bar{\star}]\!]\,\delta &= \{(k, q, \{\}, \langle\rangle)\} \\
\mathcal{T}[\![\Delta \vdash \tau_1 \otimes \tau_2 : \bar{\star}]\!]\,\delta &= \{(k, q, \psi, \langle v_1, v_2\rangle) \mid \psi = (\psi_1 \odot_k \psi_2) \wedge \\
&\qquad (k, q_1, \psi_1, v_1) \in \mathcal{T}[\![\Delta \vdash \tau_1 : \star]\!]\,\delta \wedge q_1 \preceq q \wedge \\
&\qquad (k, q_2, \psi_2, v_2) \in \mathcal{T}[\![\Delta \vdash \tau_2 : \star]\!]\,\delta \wedge q_2 \preceq q\} \\
\mathcal{T}[\![\Delta \vdash \tau_1 \multimap \tau_2 : \bar{\star}]\!]\,\delta &= \{(k, q_c, \psi_c, \lambda x.\, e) \mid \psi_c \in \mathit{LocalStoreDesc}_k \wedge \mathcal{P}(q_c, \psi_c) \wedge \\
&\qquad \forall j < k, q_a, \psi_a, v_a. \\
&\qquad\quad (j, q_a, \psi_a, v_a) \in \mathcal{T}[\![\Delta \vdash \tau_1 : \star]\!]\,\delta \wedge (\psi_c \odot_j \psi_a) \text{ defined} \Rightarrow \\
&\qquad\quad \mathsf{Comp}(j, (\psi_c \odot_j \psi_a), e[v_a/x], \mathcal{T}[\![\Delta \vdash \tau_2 : \star]\!]\,\delta)\} \\
\mathcal{T}[\![\Delta \vdash \forall\alpha{:}\kappa.\, \tau : \bar{\star}]\!]\,\delta &= \{(k, q, \psi, \Lambda.\, e) \mid \psi \in \mathit{LocalStoreDesc}_k \wedge \mathcal{P}(q, \psi) \wedge \\
&\qquad \forall j < k, \mathcal{I} \in \mathcal{K}[\![\kappa]\!]\,. \\
&\qquad\quad \mathsf{Comp}(j, \lfloor\psi\rfloor_j, e, \mathcal{T}[\![\Delta, \alpha{:}\kappa \vdash \tau : \star]\!]\,\delta[\alpha \mapsto \mathcal{I}])\} \\
\mathcal{T}[\![\Delta \vdash \mathsf{ref}\ \tau : \bar{\star}]\!]\,\delta &= \{(k, q, \{l \mapsto (q, \chi)\}, l) \mid \chi = \lfloor\mathcal{T}[\![\Delta \vdash \tau : \star]\!]\,\delta\rfloor_k \wedge \\
&\qquad (q \preceq \mathsf{A} \Rightarrow \forall(\_, q', \_, \_) \in \chi.\, q' \preceq \mathsf{A})\} \\
\mathcal{T}[\![\Delta \vdash {}^{\xi}\bar{\tau} : \star]\!]\,\delta &= \{(k, q, \psi, v) \mid q = \mathcal{T}[\![\Delta \vdash \xi : \mathsf{QUAL}]\!]\,\delta \wedge \\
&\qquad (k, q, \psi, v) \in \mathcal{T}[\![\Delta \vdash \bar{\tau} : \bar{\star}]\!]\,\delta\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Comp}(k, \psi_s, e_s, \chi) \overset{\text{def}}{=}\ & \forall j < k, s_s, \psi_r, s_f, e_f. \\
& \quad s_s :_k (\psi_s \odot_k \psi_r) \wedge (s_s, e_s) \longmapsto^{j} (s_f, e_f) \wedge \mathit{irred}(s_f, e_f) \Rightarrow \\
& \quad \exists q_f, \psi_f. \\
& \qquad s_f :_{k-j} (\psi_f \odot_{k-j} \psi_r) \wedge (k-j, q_f, \psi_f, e_f) \in \chi
\end{aligned}$$

Figure 11. $\lambda^{\mathsf{refURAL}}$ Model (Interpretations)

description of v. Filtering $\mathit{CandAtom}_k$ by the predicate $\mathcal{P}(q, \psi)$ enforces the requirement that if v is a value with qualifier q, then each location directly accessible from v must have a qualifier q' such that $q' \preceq q$. We further require the local store description ψ to be a member of $\mathit{LocalStoreDesc}_j$.

We define $\mathit{PreType}_k$ as those $\chi \in 2^{\mathit{Atom}_k} \subseteq \mathit{CandUberType}_k$ that are closed with respect to a decreasing step-index. We define Type_k by further requiring that all values in χ share the same qualifier. Looking ahead, we will need to extend our semantic interpretations to a predicate $\mathsf{Comp}(k, \psi, e, \mathcal{T}[\![\tau]\!])$, where e is a (closed) expression. Intuitively, an expression e that is indistinguishable from a value of type τ for k steps must also be indistinguishable for $j < k$ steps. Since we will define the predicate $\mathsf{Comp}(\cdot,\cdot,\cdot,\cdot)$ on elements of Type, we incorporate this closure property into the definition of $\mathit{PreType}_k$.

Finally, we define $\mathit{LocalStoreDesc}_k$ using the predicate $\mathcal{R}(\psi)$, which requires that every unrestricted or affine location in ψ is mapped to a type with only unrestricted and affine values. The predicate $\mathcal{R}(\psi)$ disallows relevant or linear values as the contents of unrestricted or affine locations (recall Figure 7).

4.3.2 Semantic Interpretations

Figure 11 gives our semantic interpretation of kinds $\mathcal{K}[\![\kappa]\!]$, qualifiers $\mathcal{T}[\![q]\!]$, pre-types $\mathcal{T}[\![\bar{\tau}]\!]$, and types $\mathcal{T}[\![\tau]\!]$.[6] The interpretation of the kinds $\bar{\star}$ and $\star$ are the semantic interpretations $\mathit{PreType}$ and Type respectively, while the interpretation of the kind QUAL is the set of (constant) qualifiers Quals.

[6] Since our language supports polymorphic types, we must give the interpretations of type-level terms with free variables. While, technically, we should write $\mathcal{T}[\![\Delta \vdash \iota : \kappa]\!]\,\delta$, where the substitution δ is in the interpretation of the term context Δ (see $\mathcal{D}[\![\Delta]\!]$ in Figure 17), we will use the more concise notation $\mathcal{T}[\![\iota]\!]$ in the text.

Units: No Location Beliefs Consider the interpretation of the pre-type $\mathbf{1}_\otimes$. Clearly, no locations appear as sub-expressions of the value $\langle\rangle$; hence, the interpretation of $\mathbf{1}_\otimes$ demands an empty local store description $\{\}$. Furthermore, the value $\langle\rangle$ may be ascribed any qualifier q.

References: Single Location Beliefs Next, consider the interpretation of the pre-type $\mathsf{ref}\ \tau$. From the value l, the only directly-accessible location is l itself. Hence, the local store description ψ for the location l in the interpretation of $\mathsf{ref}\ \tau$ must take the form $\{l \mapsto (q, \chi)\}$. Furthermore, χ, the semantic interpretation of the type of l's contents, must match $\mathcal{T}[\![\tau]\!]$.

Figure 12 graphically depicts the local store description $\psi = \{l \mapsto (q, \mathcal{T}[\![\tau]\!])\}$ (slightly abusing notation in the interest of brevity). Our intention is to express the idea that ψ "believes" that l is allocated with qualifier q and contents of type τ, but ψ "believes" nothing about any other location in the store, represented by "?".

$$(k, q, \psi = \{l \mapsto (q, \mathcal{T}[\![\tau]\!])\}, l) \in \mathcal{T}[\![\mathsf{ref}\ \tau]\!]$$

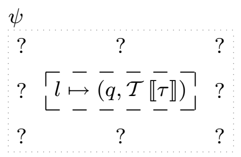


Figure 12. A Local Store Description in $\mathcal{T}[\![\mathsf{ref}\ \tau]\!]$

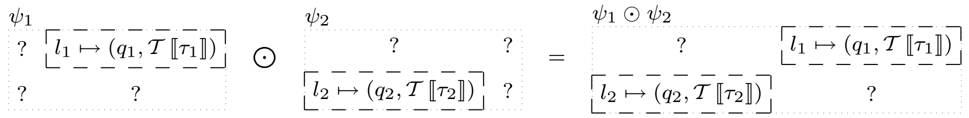
Note that the definition of $\mathcal{T}[\![\mathsf{ref}\ \tau]\!]$ requires that if l is an unrestricted or affine location, then χ should never contain local

$$\psi_1 \odot_k \psi_2 \overset{\text{def}}{=} \begin{cases} \begin{array}{l} \{l \mapsto \lfloor\psi_1\rfloor_k(l) \mid l \in dom(\psi_1) \cap dom(\psi_2)\} \\ \quad \uplus \{l \mapsto \lfloor\psi_1\rfloor_k(l) \mid l \in dom(\psi_1) \setminus dom(\psi_2)\} \\ \quad \uplus \{l \mapsto \lfloor\psi_2\rfloor_k(l) \mid l \in dom(\psi_2) \setminus dom(\psi_1)\} \end{array} & \begin{array}{l} \text{if } \forall l \in dom(\psi_1) \cap dom(\psi_2).\ \lfloor\psi_1\rfloor_k(l) = \lfloor\psi_2\rfloor_k(l) \\ \quad \text{and } \forall l \in dom(\psi_1).\ \mathsf{A} \preceq \psi_1^{\mathsf{qual}}(l) \Rightarrow l \notin dom(\psi_2) \\ \quad \text{and } \forall l \in dom(\psi_2).\ \mathsf{A} \preceq \psi_2^{\mathsf{qual}}(l) \Rightarrow l \notin dom(\psi_1) \end{array} \\ \\ \textbf{undefined} & \text{otherwise} \end{cases}$$

Figure 13. $\lambda^{\mathsf{refURAL}}$ Model (Join Partial Function)

(a)

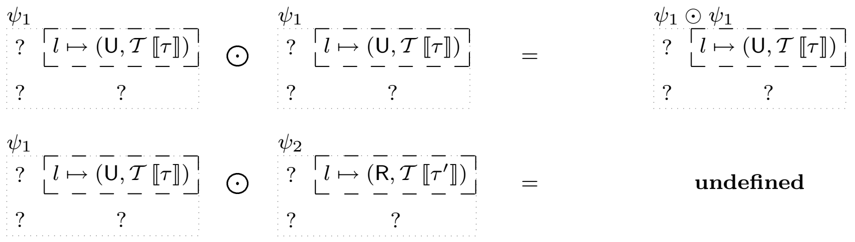
$$(k, q_1, \psi_1 = \{l_1 \mapsto (q_1, \mathcal{T}[\![\tau_1]\!])\}, l_1) \in \mathcal{T}[\![{}^{q_1}\mathsf{ref}\ \tau_1]\!]$$
$$(k, q_2, \psi_2 = \{l_2 \mapsto (q_2, \mathcal{T}[\![\tau_2]\!])\}, l_2) \in \mathcal{T}[\![{}^{q_2}\mathsf{ref}\ \tau_2]\!]$$



ψ_1
? $l_1 \mapsto (q_1, \mathcal{T}[\![\tau_1]\!])$
? ?
$\odot$
ψ_2
? ?
$l_2 \mapsto (q_2, \mathcal{T}[\![\tau_2]\!])$?
=
$\psi_1 \odot \psi_2$
? $l_1 \mapsto (q_1, \mathcal{T}[\![\tau_1]\!])$
$l_2 \mapsto (q_2, \mathcal{T}[\![\tau_2]\!])$?

(b)

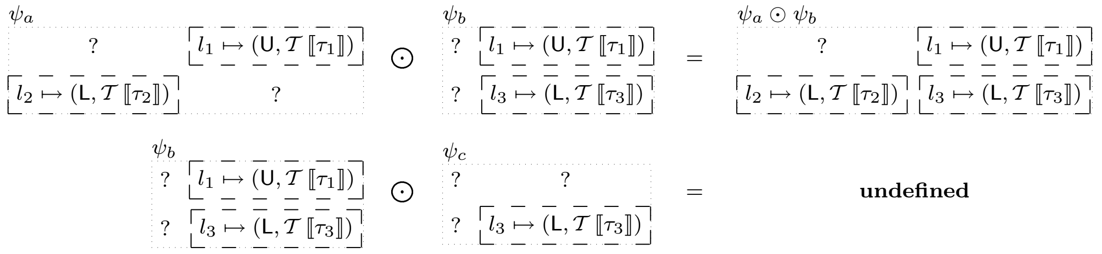
$$(k, \mathsf{U}, \psi_1 = \{l \mapsto (\mathsf{U}, \mathcal{T}[\![\tau]\!])\}, l) \in \mathcal{T}[\![{}^{\mathsf{U}}\mathsf{ref}\ \tau]\!]$$
$$(k, \mathsf{U}, \psi_2 = \{l \mapsto (\mathsf{R}, \mathcal{T}[\![\tau']\!])\}, l) \in \mathcal{T}[\![{}^{\mathsf{R}}\mathsf{ref}\ \tau']\!]$$



ψ_1
? $l \mapsto (\mathsf{U}, \mathcal{T}[\![\tau]\!])$
? ?
$\odot$
ψ_1
? $l \mapsto (\mathsf{U}, \mathcal{T}[\![\tau]\!])$
? ?
=
$\psi_1 \odot \psi_1$
? $l \mapsto (\mathsf{U}, \mathcal{T}[\![\tau]\!])$
? ?
ψ_1
? $l \mapsto (\mathsf{U}, \mathcal{T}[\![\tau]\!])$
? ?
$\odot$
ψ_2
? $l \mapsto (\mathsf{R}, \mathcal{T}[\![\tau']\!])$
? ?
=
undefined

(c)

$$(k, \mathsf{L}, \psi_a = \{l_1 \mapsto (\mathsf{U}, \mathcal{T}[\![\tau_1]\!]), l_2 \mapsto (\mathsf{L}, \mathcal{T}[\![\tau_2]\!]), v_a = \langle l_1, l_2\rangle) \in \mathcal{T}[\![{}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{ref}\ \tau_1 \otimes {}^{\mathsf{L}}\mathsf{ref}\ \tau_2)]\!]$$
$$(k, \mathsf{L}, \psi_b = \{l_1 \mapsto (\mathsf{U}, \mathcal{T}[\![\tau_1]\!]), l_3 \mapsto (\mathsf{L}, \mathcal{T}[\![\tau_3]\!]), v_b = \langle l_1, l_3\rangle) \in \mathcal{T}[\![{}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{ref}\ \tau_1 \otimes {}^{\mathsf{L}}\mathsf{ref}\ \tau_3)]\!]$$
$$(k, \mathsf{L}, \psi_c = \{l_3 \mapsto (\mathsf{L}, \mathcal{T}[\![\tau_3]\!]), v_c = \langle l_3, \langle\rangle\rangle) \in \mathcal{T}[\![{}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{ref}\ \tau_3 \otimes {}^{\mathsf{U}}\mathbf{1}_{\otimes})]\!]$$



ψ_a
? $l_1 \mapsto (\mathsf{U}, \mathcal{T}[\![\tau_1]\!])$
$l_2 \mapsto (\mathsf{L}, \mathcal{T}[\![\tau_2]\!])$?
$\odot$
ψ_b
? $l_1 \mapsto (\mathsf{U}, \mathcal{T}[\![\tau_1]\!])$
? $l_3 \mapsto (\mathsf{L}, \mathcal{T}[\![\tau_3]\!])$
=
$\psi_a \odot \psi_b$
? $l_1 \mapsto (\mathsf{U}, \mathcal{T}[\![\tau_1]\!])$
$l_2 \mapsto (\mathsf{L}, \mathcal{T}[\![\tau_2]\!])$ $l_3 \mapsto (\mathsf{L}, \mathcal{T}[\![\tau_3]\!])$
ψ_b
? $l_1 \mapsto (\mathsf{U}, \mathcal{T}[\![\tau_1]\!])$
? $l_3 \mapsto (\mathsf{L}, \mathcal{T}[\![\tau_3]\!])$
$\odot$
ψ_c
? ?
? $l_3 \mapsto (\mathsf{L}, \mathcal{T}[\![\tau_3]\!])$
=
undefined

Figure 14. $\psi_1 \odot \psi_2$ Examples

store descriptions that include relevant or linear locations; i.e., the definition of $\mathcal{T}[\![\mathsf{ref}\ \tau]\!]$ incorporates the predicate $\mathcal{R}(\cdot)$ specialized to $\{l \mapsto (q, \chi)\}$.

Pairs: Compatible Location Beliefs A pair $\langle v_1, v_2\rangle$ (such that $(k, q_1, \psi_1, v_1) \in \mathcal{T}[\![\tau_1]\!]$ and $(k, q_2, \psi_2, v_2) \in \mathcal{T}[\![\tau_2]\!]$) is in the interpretation of $\tau_1 \otimes \tau_2$ if and only if the pair is ascribed a qualifier greater than that of its components and the two sets of beliefs about the store, ψ_1 and ψ_2, can be combined into a single set of beliefs sufficient for safely executing k steps (written $\psi_1 \odot_k \psi_2$, see Figure 13). Informally, local store descriptions can be combined only if they are *compatible*; that is, if the beliefs in one local store description do not contradict the beliefs in the other store description.

Clearly, if ψ_1 and ψ_2 have disjoint sets of beliefs about the store, then $\psi_1 \odot_k \psi_2$ is defined and equal to the union of their beliefs (see Figure 14(a)). In the more general case, where the same location may be found in the domain of both ψ_1 and ψ_2, there are two requirements enforced by the definition of $\psi_1 \odot_k \psi_2$.

First, we require that for any location l that is described by both ψ_1 and ψ_2, it must be the case that ψ_1 and ψ_2 have identical beliefs about l to approximation k. Note that ψ_1 and ψ_2 must agree on both the qualifier of the location as well as the type of the location's contents (see Figure 14(b)).

The second requirement is more subtle, having to do with the notion of directly-accessible locations. Suppose that l_3 is a linear or affine location mapped by ψ_b. Therefore, a value v_b with local store description ψ_b must contain l_3 as a sub-expression. Since l_3 is linear or affine, this occurrence of l_3 in the value v_b must be the one (and only) occurrence of l_3 in the entire program state. Now, suppose that l_3 is also in the domain of a local store description ψ_c. As before, a value v_c with local store description ψ_c must contain l_3 as a sub-expression. If we were to attempt to form the value $\langle v_b, v_c\rangle$, then we would have a value with two distinct occurrences of l_3, violating the uniqueness of the location l_3. Hence, we consider ψ_b and ψ_c to represent incompatible (contradictory) beliefs about the current store (see Figure 14(c)).

Functions & Abstractions: Closure Location Beliefs Since functions and abstractions are suspended computations, their interpretations are given in terms of the interpretation of types as computations (see below). A function $\lambda x.\, e$ with qualifier q_c and local store description ψ_c (where ψ_c describes the locations directly accessible from the function's closure and, hence, must satisfy $\mathcal{P}(q_c, \psi_c)$) is in the interpretation of $\tau_1 \multimap \tau_2$ for k steps if, at some point in the future, when there are $j < k$ steps left to execute, and there is an argument v_a such that $(j, \_, \psi_a, v_a) \in \mathcal{T}\,[\![\tau_1]\!]$ and the beliefs ψ_c and ψ_a are compatible, then $e[v_a/x]$ looks like a computation of type τ_2 for j steps. The interpretation of $\forall \alpha{:}\kappa.\, \tau$ is analogous, except that we quantify over (type-level term) interpretations $\mathcal{I} \in \mathcal{K}\,[\![\kappa]\!]$.

Store Satisfaction: Tracing Location Beliefs The interpretation of types as computations (Comp) makes use of an auxiliary relation $s :_k \psi$ (given in Figure 15), which says that the store s satisfies local store description ψ (to approximation k). We motivate the definition of $s :_k \psi$ by drawing an analogy with the specification of a *tracing garbage collector* (see Figure 16). As described above, ψ corresponds to (beliefs about) the portion of the store directly accessible from a value (or multiple values, when ψ corresponds to $\odot_k$-ed store descriptions). Hence, we can consider $dom(\psi)$ as a set of root locations. In the definition of $s :_k \psi$, $\mathcal{S}$ corresponds to the set of reachable (root and non-root) locations in the store that would be discovered by the garbage collector. The function $\mathcal{F}_\psi$ maps each location in $\mathcal{S}$ to a local store description, while the function $\mathcal{F}_q$ maps each location to a qualifier. It is our intention that, for each location l, $\mathcal{F}_q(l)$ is an appropriate qualifier and $\mathcal{F}_\psi(l)$ is an appropriate local store description for the value $s^{\mathsf{val}}(l)$. Hence, we can consider $dom(\mathcal{F}_\psi(l))$ as the set of child locations traced from the contents of l.

Having chosen the set $\mathcal{S}$ and the functions $\mathcal{F}_\psi$ and $\mathcal{F}_q$, we require that they satisfy three criteria. The *congruity* criteria ensures that our choices are both internally consistent and consistent with the store s. The "global" store description ψ_* combines the local store descriptions of the roots with the local store descriptions of the contents of every reachable location; the implicit requirement that ψ_* is defined ensures that the local beliefs of the roots and individual store contents are all compatible. The clause $dom(\psi_*) = \mathcal{S}$ requires that $\mathcal{S}$ and $\mathcal{F}_\psi$ are chosen such that $\mathcal{S}$ includes *all* the reachable locations (and not just *some* of the reachable locations), while the clause $dom(s) \supseteq \mathcal{S}$ requires that all of the reachable locations are actually in the store. Finally, $(j, \mathcal{F}_q, \lfloor \mathcal{F}_\psi(l) \rfloor_j, s^{\mathsf{val}}(l)) \in \lfloor \psi_*^{\mathsf{type}}(l) \rfloor_k$ ensures that the contents of l, with the qualifier assigned by $\mathcal{F}_q$ and local store description assigned by $\mathcal{F}_\psi$, is in the type assigned by the global store description ψ_* (for $j < k$ steps).

The *minimality* criteria ensures that our choice for the set $\mathcal{S}$ does not contain any locations not reachable from the roots. For example, in Figure 16, including l_{11} in $\mathcal{S}$ would not violate congruity, but would violate minimality. Finally, the *reachability* criteria ensures that every linear and relevant location is reachable from the roots (and, hence, has not been implicitly discarded).

Computations: Relating Current to Future Beliefs Informally, the interpretation of types as computations $\mathsf{Comp}(k, \psi_s, e_s, \chi)$ (see Figure 11) says that if the expression e_s (with beliefs ψ_s, again, corresponding to the locations appearing as sub-expressions of e_s) reaches an irreducible state in less than k steps, then it must have reduced to a value v_f (with beliefs ψ_f) that belongs to the type interpretation χ. More precisely, we pick a starting store s_s such that $s_s :_k (\psi_s \odot_k \psi_r)$, where ψ_r is the set of beliefs about the store held by the rest of the computation (alternatively, the set of beliefs held by e_s's continuation). If (s_s, e_s) steps to an irreducible configuration (s_f, e_f) in $j < k$ steps, then the following conditions

$$
\begin{array}{rcl}
\mathcal{D}\,[\![\bullet]\!] & = & \{\emptyset\} \\
\mathcal{D}\,[\![\Delta, \alpha{:}\kappa]\!] & = & \{\delta[\alpha \mapsto \mathcal{I}] \mid \delta \in \mathcal{D}\,[\![\Delta]\!] \wedge \mathcal{I} \in \mathcal{K}\,[\![\kappa]\!]\} \\[1em]
\mathcal{G}\,[\![\Delta \vdash \bullet]\!]\,\delta & = & \{(k, q, \{\}, \emptyset)\} \\
\mathcal{G}\,[\![\Delta \vdash \Gamma, x{:}\tau]\!]\,\delta & = & \\
\multicolumn{3}{l}{\qquad\qquad \{(k, q, \psi, \gamma[x \mapsto v]) \mid} \\
\multicolumn{3}{l}{\qquad\qquad\quad \psi = (\psi_\Gamma \odot_k \psi_x) \wedge} \\
\multicolumn{3}{l}{\qquad\qquad\quad (k, q_\Gamma, \psi_\Gamma, \gamma) \in \mathcal{G}\,[\![\Delta \vdash \Gamma]\!]\,\delta \wedge q_\Gamma \preceq q \wedge} \\
\multicolumn{3}{l}{\qquad\qquad\quad (k, q_x, \psi_x, v) \in \mathcal{T}\,[\![\Delta \vdash \tau : \star]\!]\,\delta \wedge q_x \preceq q\}} \\[1em]
[\![\Delta; \Gamma \vdash e : \tau]\!] & \overset{\text{def}}{=} & \\
\multicolumn{3}{l}{\qquad\qquad \forall k \geq 0.\, \forall \delta, q_\Gamma, \psi_\Gamma, \gamma.} \\
\multicolumn{3}{l}{\qquad\qquad\quad \delta \in \mathcal{D}\,[\![\Delta]\!] \wedge (k, q_\Gamma, \psi_\Gamma, \gamma) \in \mathcal{G}\,[\![\Delta \vdash \Gamma]\!]\,\delta \Rightarrow} \\
\multicolumn{3}{l}{\qquad\qquad\quad \mathsf{Comp}(k, \psi_\Gamma, \gamma(e), \mathcal{T}\,[\![\Delta \vdash \tau : \star]\!]\,\delta)}
\end{array}
$$

Figure 17. $\lambda^{\mathsf{refURAL}}$ Model (Additional Interpretations)

hold. First, e_f must be a value with a qualifier q_f and a set of beliefs ψ_f such that $(k - j, q_f, \psi_f, e_f) \in \chi$. Second, the following two sets of beliefs must be compatible: ψ_f (what e_f believes) and ψ_r (what the rest of the computation believes — note that these beliefs remain unchanged). Third, the final store s_f must satisfy the combined set of these beliefs.

Note that since ψ_r is an arbitrary set of beliefs compatible with ψ_s, one instantiation of ψ_r is the local store description that includes all of the shared locations of ψ_s. By requiring that ψ_f and s_f are compatible with ψ_r, we ensure that the types and qualifiers and allocation status of shared locations are preserved.

Judgements: Type Soundness Finally, the semantic interpretation of a typing judgement $[\![\Delta; \Gamma \vdash e : \tau]\!]$ (see Figure 17) asserts that for all $k \geq 0$, if δ is a mapping from type-level variables to an element of the appropriate kind interpretation, and γ is a mapping from variables to closed values, and ψ_Γ is a local store description for the values in the range of γ, then $(k, \psi_\Gamma, \gamma(e))$ is in the interpretation of τ as a computation ($\mathsf{Comp}(k, \psi_\Gamma, \gamma(e), \mathcal{T}\,[\![\tau]\!])$).

Our extended technical report [3] gives the proof of the following theorem which shows the soundness of the $\lambda^{\mathsf{refURAL}}$ typing rules with respect to the model.

THEOREM 1. **($\lambda^{\mathsf{refURAL}}$ Soundness)**

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

An immediate corollary is type-safety of $\lambda^{\mathsf{refURAL}}$. Another interesting corollary is that if we evaluate a closed, well-typed term of base type (e.g., ${}^q\mathbf{1}_\otimes$) to a value, then the resulting store will have no linear or relevant references.

COROLLARY 2. **($\lambda^{\mathsf{refURAL}}$ Safety)**

If $\bullet; \bullet \vdash e_1 : \tau$ *and* $(\{\}, e_1) \longmapsto^* (s_2, e_2)$,
then either $\exists v_2.\, e_2 \equiv v_2$ *or* $\exists s_3, e_3.\, (s_2, e_2) \longmapsto (s_3, e_3)$.

COROLLARY 3. **($\lambda^{\mathsf{refURAL}}$ Collection)**

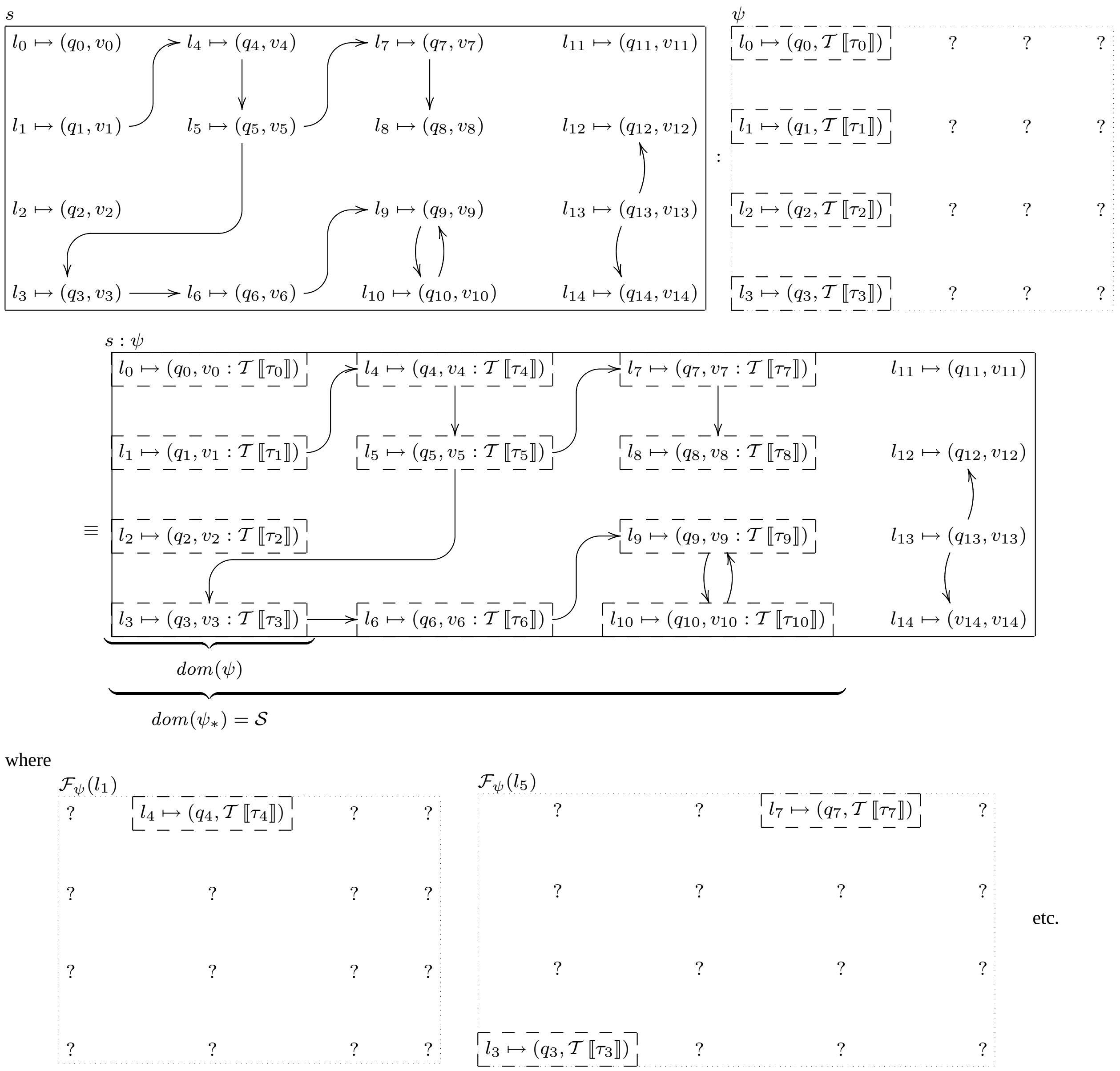
If $\bullet; \bullet \vdash e_1 : {}^q\mathbf{1}_\otimes$ *and* $(\{\}, e_1) \longmapsto^* (s_2, v_2)$,
then $\forall l \in dom(s_2).\, s_2^{\mathsf{qual}}(l) \preceq \mathsf{A}$.

Proof ($\lambda^{\mathsf{refURAL}}$ Safety)

Suppose $\bullet; \bullet \vdash e_1 : \tau$ and $(\{\}, e_1) \longmapsto^* (s_2, e_2)$.
If $\neg irred(s_2, e_2)$, then $\exists s_3, e_3.\, (s_2, e_2) \longmapsto (s_3, e_3)$.
If $irred(s_2, e_2)$, then $\exists i.\, (\{\}, e_1) \longmapsto^i (s_2, e_2)$.
Theorem 1 applied to $\bullet; \bullet \vdash e_1 : \tau$ yields $[\![\bullet; \bullet \vdash e_1 : \tau]\!]$.
$[\![\bullet; \bullet \vdash e_1 : \tau]\!]$ instantiated with $i + 1 \geq 0$, $\emptyset \in \mathcal{D}\,[\![\bullet]\!]$,
  and $(i + 1, \mathsf{U}, \{\}, \emptyset) \in \mathcal{G}\,[\![\bullet]\!]\,\emptyset$
  yields $\mathsf{Comp}(i + 1, \{\}, e_1, \mathcal{T}\,[\![\bullet \vdash \tau : \star]\!]\,\emptyset)$.
$\mathsf{Comp}(i{+}1, \{\}, e_1, \mathcal{T}\,[\![\bullet \vdash \tau : \star]\!]\,\emptyset)$ instantiated with $i < i{+}1$,
  $s_1 :_{i+1} (\{\} \odot_{i+1} \{\})$, $(\{\}, e_1) \longmapsto^i (s_2, e_2)$,

$$
\begin{array}{lll}
s :_k \psi & \overset{\text{def}}{=} & \exists \mathcal{S} : 2^{Locs}. \\
& & \exists \mathcal{F}_\psi : \mathcal{S} \to \mathit{LocalStoreDesc}. \\
& & \exists \mathcal{F}_q : \mathcal{S} \to \mathit{Quals}. \\
& & \quad \left.\begin{array}{l}
\text{let } \psi_* = (\psi \odot_k \bigodot_k^{l \in \mathcal{S}} \mathcal{F}_\psi(l)) \text{ in} \\
\mathit{dom}(\psi_*) = \mathcal{S} \wedge \mathit{dom}(s) \supseteq \mathcal{S} \wedge \\
\forall l \in \mathcal{S}. \\
\quad \forall j < k.\ (j, \mathcal{F}_q(l), \lfloor \mathcal{F}_\psi(l) \rfloor_j, s^{\mathsf{val}}(l)) \in \lfloor \psi_*^{\mathsf{type}}(l) \rfloor_k \wedge \\
\quad s^{\mathsf{qual}}(l) = \psi_*^{\mathsf{type}}(l) \wedge
\end{array}\right\} \text{congruity} \\
& & \quad \left.\begin{array}{l}
\forall \mathcal{S}^\dagger \subseteq \mathcal{S}. \\
\quad \mathit{dom}(\psi) \subseteq \mathcal{S}^\dagger \wedge (\forall l \in \mathcal{S}^\dagger.\ \mathit{dom}(\mathcal{F}_\psi(l)) \subseteq \mathcal{S}^\dagger) \Rightarrow \mathcal{S} = \mathcal{S}^\dagger \wedge
\end{array}\right\} \text{minimality} \\
& & \quad \left.\begin{array}{l}
\forall l \in \mathit{dom}(s). \\
\quad \mathsf{R} \preceq s^{\mathsf{qual}}(l) \Rightarrow l \in \mathcal{S}
\end{array}\right\} \text{reachability}
\end{array}
$$

Figure 15. $\lambda^{\mathsf{refURAL}}$ Model (Store Satisfaction)



s
$l_0 \mapsto (q_0, v_0)$ $l_4 \mapsto (q_4, v_4)$ $l_7 \mapsto (q_7, v_7)$ $l_{11} \mapsto (q_{11}, v_{11})$
$l_1 \mapsto (q_1, v_1)$ $l_5 \mapsto (q_5, v_5)$ $l_8 \mapsto (q_8, v_8)$ $l_{12} \mapsto (q_{12}, v_{12})$
$l_2 \mapsto (q_2, v_2)$ $l_9 \mapsto (q_9, v_9)$ $l_{13} \mapsto (q_{13}, v_{13})$
$l_3 \mapsto (q_3, v_3)$ $l_6 \mapsto (q_6, v_6)$ $l_{10} \mapsto (q_{10}, v_{10})$ $l_{14} \mapsto (q_{14}, v_{14})$
:
ψ
$l_0 \mapsto (q_0, \mathcal{T}[\![\tau_0]\!])$? ? ?
$l_1 \mapsto (q_1, \mathcal{T}[\![\tau_1]\!])$? ? ?
$l_2 \mapsto (q_2, \mathcal{T}[\![\tau_2]\!])$? ? ?
$l_3 \mapsto (q_3, \mathcal{T}[\![\tau_3]\!])$? ? ?
$s : \psi$
$\equiv$
$l_0 \mapsto (q_0, v_0 : \mathcal{T}[\![\tau_0]\!])$ $l_4 \mapsto (q_4, v_4 : \mathcal{T}[\![\tau_4]\!])$ $l_7 \mapsto (q_7, v_7 : \mathcal{T}[\![\tau_7]\!])$ $l_{11} \mapsto (q_{11}, v_{11})$
$l_1 \mapsto (q_1, v_1 : \mathcal{T}[\![\tau_1]\!])$ $l_5 \mapsto (q_5, v_5 : \mathcal{T}[\![\tau_5]\!])$ $l_8 \mapsto (q_8, v_8 : \mathcal{T}[\![\tau_8]\!])$ $l_{12} \mapsto (q_{12}, v_{12})$
$l_2 \mapsto (q_2, v_2 : \mathcal{T}[\![\tau_2]\!])$ $l_9 \mapsto (q_9, v_9 : \mathcal{T}[\![\tau_9]\!])$ $l_{13} \mapsto (q_{13}, v_{13})$
$l_3 \mapsto (q_3, v_3 : \mathcal{T}[\![\tau_3]\!])$ $l_6 \mapsto (q_6, v_6 : \mathcal{T}[\![\tau_6]\!])$ $l_{10} \mapsto (q_{10}, v_{10} : \mathcal{T}[\![\tau_{10}]\!])$ $l_{14} \mapsto (v_{14}, v_{14})$
$\mathit{dom}(\psi)$
$\mathit{dom}(\psi_*) = \mathcal{S}$
where
$\mathcal{F}_\psi(l_1)$
? $l_4 \mapsto (q_4, \mathcal{T}[\![\tau_4]\!])$? ?
? ? ? ?
? ? ? ?
? ? ? ?
$\mathcal{F}_\psi(l_5)$
? ? $l_7 \mapsto (q_7, \mathcal{T}[\![\tau_7]\!])$?
? ? ? ?
? ? ? ?
$l_3 \mapsto (q_3, \mathcal{T}[\![\tau_3]\!])$? ? ?
etc.

Figure 16. $s : \psi$ Example

and $irred(s_2, e_2)$
yields q_2 and ψ_2 such that $s_2 :_1 (\psi_2 \odot_1 \{\})$
and $(1, q_2, \psi_2, e_2) \in \mathcal{T}\,[\![\bullet \vdash \tau : \star]\!]\,\emptyset$.
Recall that $\mathcal{T}\,[\![\bullet \vdash \tau : \star]\!]\,\emptyset \in Type$
and $Type \subseteq CandUberType_\omega = 2^{CandAtom_\omega}$.
Hence, $(1, q_2, \psi_2, e_2) \in CandAtom_\omega = \bigcup_{k \geq 0} CandAtom_k$,
which implies that $e_2 \in CValues$ and $\exists v_2.\ e_2 \equiv v_2$. □

Proof ($\lambda^{\mathsf{refURAL}}$ Collection)

Suppose $\bullet; \bullet \vdash e_1 : {}^q\mathbf{1}_\otimes$ and $(\{\}, e_1) \longmapsto^* (s_2, v_2)$.
By the reasoning above, $(1, q_2, \psi_2, v_2) \in \mathcal{T}\,[\![\bullet \vdash {}^q\mathbf{1}_\otimes : \star]\!]\,\emptyset$,
which implies that $q_2 = q$, $\psi_2 = \{\}$, and $v_2 = \langle\rangle$.
Recall that $s_2 :_1 (\{\} \odot_1 \{\}) \equiv s_2 :_1 \{\} \equiv \exists \mathcal{S}, \mathcal{F}_\psi, \mathcal{F}_q.\ \ldots$.
The minimality criteria of $s_2 :_1 \{\}$ instantiated with $\emptyset \subseteq \mathcal{S}$,
$dom(\{\}) \subseteq \emptyset$, and $(\forall l \in \emptyset.\ dom(\mathcal{F}_\psi(l)) \subseteq \emptyset)$
yields $\mathcal{S} = \emptyset$.
The reachability criteria of $s_2 :_1 \{\}$
yields $\forall l \in dom(s_2).\ \mathsf{R} \preceq s_2^{\mathsf{qual}}(l) \Rightarrow l \in \emptyset$,
which implies $\forall l \in dom(s_2).\ s_2^{\mathsf{qual}}(l) \preceq \mathsf{A}$. □

4.4 Discussion

A key difference in the model presented here, as compared to previous models of mutable state, is the *localization* of the store description. Recall that we identify the local store description of a value with those locations that are directly accessible from the value. This is in contrast to the AAV model of unrestricted references [1, 4], where the global store description of any value describes *every* location that has been allocated. It is also in contrast to our previous model of linear references [23, 2], where the store description of a value describes the *reachable* locations from that value.

The transition from a global store description to a local store description is motivated by the insight that storing a unique object in a shared reference "hides" the unique object in some way. Note that the shared reference must mediate all access to the unique object. The authors have found it hard to construct a model where the store description of a value (in the interpretation of a type) describes the entire store or even the store reachable from the value. When one attempts to describe the entire store, there is a difficulty identifying where the "real" occurrence of a unique location is to be found. When one attempts to describe the reachable store, there is a difficulty defining the $\odot$ relation; it cannot be defined point-wise, and one is required to formally introduce the notions of directly- and indirectly-accessible locations. Furthermore, the reachable store is a property of the actual store, not of the type; hence, it seems better to confine reachability to the store satisfaction relation. We further note that the model of mutable references given in this paper subsumes the models of mutable references cited above. Hence, the technique of localizing the store description subsumes the techniques used by previous approaches.

Although our model of substructural references is different from the previous model of unrestricted references, our model retains the spirit of the step-indexed approach used in Foundational PCC [6, 7] and may be applicable in future extensions of FPCC. This approach, in which the model mixes denotational and operational semantics, offers a number of distinct advantages over a purely syntactic approach to type soundness. One obvious advantage of this approach is that it gives rise to a simpler set of typing rules; note that our typing judgement requires neither a store description component nor a rule for locations. A less obvious advantage of this approach is that it gives rise to stronger meta-theoretic results. For example, the impredicative polymorphism of $\lambda^{\mathsf{refURAL}}$ implies a strong parametricity theorem: an element of $\mathcal{T}\,[\![\forall\alpha{:}\,\star\,.\,\tau]\!]$ behaves uniformly on *all* elements of $Type$, which includes elements that do not correspond to the interpretation of any syntactic type. This approach also naturally extends to union and intersection types and to an inclusion interpretation of subtyping. Finally, a (well-founded) set-theoretic model means that soundness and safety proofs are amenable to formalization in the higher-order logic of FPCC.

While we are partial to the step-indexed approach, we acknowledge that there is no fundamental difficulty in adopting a purely syntactic approach to proving the type soundness of substructural state. However, we believe that *any* proof of type soundness must adopt many of the insights presented here. For example, we conjecture that the typing rule for well-typed configurations would naturally take the form:

$$\dfrac{\dfrac{\begin{array}{c}\psi_* = \psi \odot \bigodot^{l \in \mathcal{S}} \mathcal{F}_\psi(l) \\ dom(\psi_*) = \mathcal{S} \qquad dom(s) \supseteq \mathcal{S} \\ \forall l \in \mathcal{S}.\ \cdot;\cdot;\mathcal{F}_\psi(l) \vdash s^{\mathsf{val}}(l) : \psi_*^{\mathsf{type}}(l)\ \wedge \\ s^{\mathsf{qual}}(l) = \psi_*^{\mathsf{qual}}(l)\end{array}}{\vdash s : \psi} \qquad \cdot;\cdot;\psi \vdash e : \tau}{\vdash (s, e) : \tau}$$

Note that the judgement $\vdash s : \psi$ mirrors the store satisfaction predicate given in Figure 15. The store typing component complicates the judgement $\Delta; \Gamma; \psi \vdash e : \tau$, which must further rely upon an operator $\psi_1 \odot \psi_2 = \psi$ to split the locations in ψ between the store typings ψ_1 and ψ_2. Splitting the store typing is necessary to ensure that a given unique location is used by at most one sub-expression. The $\odot$ operator in the syntactic approach would need to satisfy many of the same properties as the $\odot_k$ operator in the step-indexed approach (e.g., identical beliefs about locations in the common domain and no unique locations in the common domain).

5. Related Work

Our λ^{URAL} is most directly influenced by the presentation of substructural type systems by Walker [30], which in turn draws upon the work of Wansbrough and Peyton-Jones [33] and Walker and Watkins [32]. Relative to that work, we have added both relevant and affine qualifiers, which is necessary to account for the varied forms of linearity found in higher-level programming-language proposals.

A related body of work is that on type systems used to track resource usage [28, 22, 33, 21, 16, 19]. We note that the usage subsumption found in these systems (e.g., a "possibly used many times" variable may be subsumed to appear in a context requiring a "used exactly once" value) is not applicable in our setting (e.g., it is clearly unsound to subsume ${}^{\mathsf{U}}\mathsf{ref}\ \tau$ to ${}^{\mathsf{L}}\mathsf{ref}\ \tau$), due to differences in the interpretation of type qualifiers.

Section 1 noted a number of projects that have introduced some form of linearity to "tame" state. An underlying theme is that linearity and strong updates can be used to provide more effective memory management (c.f. [10, 18, 9, 8]).

More recent research has explored other ways in which unique and shared data may be mixed. For example, Cyclone's `alias` construct [17] takes a unique pointer and returns a shared pointer to the same object, which is available for a limited lexical scope. Vault's `focus` and CQuals's `restrict` constructs [14, 5] provide the opposite behavior: temporarily giving a linear view of an object of shared type. Both behaviors are of great practical significance.

Our model's semantic interpretations seem strongly related to the logic of Bunched Implications (BI) [20] and Reynolds' separation logic [25]. In particular, our interpretation of $\otimes$ and $\multimap$ resemble the resource semantics for the $*$ and $-\!\!*$ connectives in BI.

Finally, Boyland and Retert have recently proved the soundness of a variation of Vault by giving an operational semantics of "adoption" [11]. The authors note that adoption may be used to embed a unique pointer within another object; their notion of uniqueness most closely resembles our affine references, as access keys may be dropped.

6. Conclusion and Future Work

We have presented the $\lambda^{\mathsf{refURAL}}$-calculus, a substructural polymorphic λ-calculus with mutable references of unrestricted, relevant, affine, and linear sorts. We motivated the design decisions, gave a type system, and constructed a step-indexed model of $\lambda^{\mathsf{refURAL}}$, where types are interpreted as sets of store description / value pairs, which are further refined using an index representing the number of steps available for future evaluation.

In previous work [23, 2], we separated the typing components of a mutable object into two pieces: an unrestricted *pointer* to the object and a linear *capability* for accessing the contents of the object. We believe that we can extend the current language and model in the same way. The advantage of this approach is that separating the name of a reference from what it currently holds gives us a model of alias types [27, 31].

As noted in the previous section, allowing a unique pointer to be temporarily treated as shared (and vice versa) can be useful in practice. Understanding how to model these advanced features is a long-term goal of this research. A promising aproach is to model regions as a linear capability to access objects in the region and allow changes in reference qualifiers to be mediated by this capability.

Acknowledgments

We would like to thank Dan Grossman, Aleks Nanevski, and Dan Wang for their helpful comments.

References

[1] Amal Ahmed, Andrew W. Appel, and Roberto Virga. An indexed model of impredicative polymorphism and mutable references. Available at `http://www.cs.princeton.edu/~appel/papers/impred.pdf`, January 2003.

[2] Amal Ahmed, Matthew Fluet, and Greg Morrisett. $\mathbf{L}^3$: A linear language with locations. Technical Report TR-24-04, Harvard University, October 2004.

[3] Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. Technical Report TR-16-05, Harvard University, July 2005.

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

[5] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 129–140, June 2003.

[6] Andrew W. Appel. Foundational proof-carrying code. In *Proc. Logic in Computer Science (LICS)*, pages 247–258, June 2001.

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

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

[9] David Aspinall and Martin Hofmann. Another type system for in-place update. In *Proc. European Symposium on Programming (ESOP)*, pages 36–52, March 2002.

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

[11] John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. In *Proc. Principles of Programming Languages (POPL)*, pages 283–295, January 2005.

[12] James Cheney and Greg Morrisett. A linearly typed assembly language. Technical Report 2003-1900, Department of Computer Science, Cornell University, 2003.

[13] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 59–69, June 2001.

[14] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 13–24, June 2002.

[15] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[16] Jörgen Gustavsson and Josef Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In *Proc. International Workshop on Implementation of Functional Languages (IFL)*, pages 140–157, September 2001.

[17] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in Cyclone. In *Proc. International Symposium on Memory Management (ISMM)*, pages 73–84, October 2004.

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

[19] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proc. ACM Principles of Programming Languages (POPL)*, pages 331–342, January 2002.

[20] Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *Proc. Principles of Programming Languages (POPL)*, pages 14–26, January 2001.

[21] Naoki Kobayashi. Quasi-linear types. In *Proc. Principles of Programming Languages (POPL)*, pages 29–42, January 1999.

[22] Torben Æ. Mogensen. Types for 0, 1 or many uses. In *Proc. International Workshop on Implementation of Functional Languages (IFL)*, pages 112–122, 1998.

[23] Greg Morrisett, Amal Ahmed, and Matthew Fluet. $\mathbf{L}^3$: A linear language with locations. In *Proc. International Conference on Typed Lambda Calculi and Applications (TLCA)*, pages 293–307, April 2005.

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

[25] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. Logic in Computer Science (LICS)*, pages 55–74, July 2002.

[26] Sjaak Smetsers, Erik Barendsen, Marko C. J. D. van Eekelen, and Rinus 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.

[27] Fred Smith, David Walker, and Greg Morrisett. Alias types. In *Proc. European Symposium on Programming (ESOP)*, pages 366–381, March 2000.

[28] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proc. Functional Programming Languages and Computer Architecture (FPCA)*, pages 1–11, June 1995.

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

[30] David Walker. Substructural type systems. In Benjamin Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1, pages 3–43. MIT Press, Cambridge, MA, 2005.

[31] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Proc. Workshop on Types in Compilation (TIC)*, pages 177–206, September 2000.

[32] David Walker and Kevin Watkins. On regions and linear types. In *Proc. International Conference on Functional Programming (ICFP)*, pages 181–192, September 2001.

[33] Keith Wansbrough and Simon Peyton-Jones. Once upon a polymorphic type. In *Proc. Principles of Programming Languages (POPL)*, pages 15–28, January 1999.