Type Universes as Kripke Worlds

PAULETTE KORONKEVICH, University of British Columbia, Canada
WILLIAM J. BOWMAN, University of British Columbia, Canada

What are mutable references; what do they mean? The answers to these questions have spawned lots of important theoretical work and form the foundation of many impactful tools. However, existing semantics collapse a key distinction: which allocations does a reference depend on?

In this paper, we deconstruct the space of mutable higher-order references. We formalize a novel distinction—splitting the design space of references not only into higher-order vs (full-)ground references, but also dependency of an allocation on past vs future allocations. This distinction is fundamental to a thorny issue that arises in constructing semantic models of mutable references—the type-world circularity. The issue disappears for what we call predicative references, those that only quantify over past, not future, allocations, and for non-higher-order impredicative references. We design a syntax and semantics for each point in our newly described space. The syntax relies on a type universe hierarchy, a la dependent type theory, to kind the types of allocated terms, and stratify allocations. Each type universe corresponds to a semantic Kripke world, giving a lightweight syntactic mechanism to design and restrict heap shapes. The semantics bear a resemblance to work on regions, and suggest some connection between universe systems and regions, which we describe in some detail.

ACM Reference Format:
Paulette Koronkevich and William J. Bowman. 2025. Type Universes as Kripke Worlds. 1, 1 (June 2025), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Constructing a semantic model of typed higher-order mutable references is complex. Worse, we find the complexity of the semantics arises in two ways: gradually, then suddenly [23, Chapter 13]. The semantics of pure functional languages is simple. Adding mutable state that only accesses ground data is simple. Adding mutable state that can reference other references, including expressing cycles, is slightly more complex, but proportional to adding the cycles. Each of these small changes does not drastically change the complexity, and metatheoretic properties of the original functional language still hold.

However, once we add the ability to store functions that close over the heap—*higher-order* references[1]—the semantics get very complex indeed. We must throw out all the previous semantics and redo them from scratch. Some of the metatheoretic properties that held, such as termination, no longer hold.

But why?

Informally, the complexity arises from semantics that allow functions to rely on past *and all future* allocations on the heap, including allocation of the function itself[2]. Functions close over local variables, referred to as the function's environment. The environment can contain mutable state, so in the semantics, each function must have a notion of the local state it captures, to describe its behaviour as valid only in future evolutions of that initial state. Most semantics describe functions essentially as store passing: the semantic values expect a semantic heap (the type of which we call

[1]We use the term higher-order reference in the sense of Abramsky et al. [1].
[2]We interpret the function value as a heap allocated structure, a closure, in this interpretation.

Authors' Contact Information: Paulette Koronkevich, University of British Columbia, Vancouver, Canada, pletrec@cs.ubc.ca; William J. Bowman, University of British Columbia, Vancouver, Canada, wjb@williamjbowman.com.

2025. ACM XXXX-XXXX/2025/6-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

a *world*) to describe the local state closed over by a function. Then, for a function of type $\tau_1 \to \tau_2$, we describe the semantic value's type as follows.

$$[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \to \mathit{World} \to ([\![\tau_2]\!] \times \mathit{World})$$

The semantics model the behaviour of functions that close over and alter local state by using the world parameter to describe what locations must be allocated. However, the world can also contain future allocations, from the perspective of the function's allocation—allocations that had not happened when the function was first defined. For example, consider the following program.

$$\begin{array}{ll} r = \textbf{new}\ (\boldsymbol{\lambda}\ x.2) & : \textbf{Ref}\ ((\textbf{Ref Nat}) \to \textbf{Nat}) \\ f = (\boldsymbol{\lambda}\ n.(!\ r)(\textbf{new}\ !\ n)) & : (\textbf{Ref Nat}) \to \textbf{Nat} \\ s = \textbf{new}\ 3 & : \textbf{Ref Nat} \\ r := f; (f\ s) & \end{array}$$

The function f dereferences r, which means the semantic value of f takes a world with a semantic value for r's location to describe f's behaviour. At the point the function f is invoked, a new reference s has been allocated; to model this behaviour, the semantic value of f will take a future world from when f was first defined. However, allowing f to depend on this future world also means the interpretation of r includes f as one of the possible values of r, so the interpretation of r depends on f which depends on r, etc! Syntactically, this means we can update r to be a reference to f, implementing recursion through back-patching, and giving rise to potential non-termination. Semantically, when functions are *stored* in references, the world contains the semantic values of functions that quantify over future allocations! It is precisely the quantification over future allocations that causes complexity when trying to formally define a world, since relying on future allocations of the world includes the current world itself.

Formally, we can see this complexity in the *type-world circularity*, described in detail by Ahmed [5]. To model a type with a representing set of values, one needs a *world* to represent locations on the heap. We represent a type as a function over worlds. A world, a description of the heap, maps locations to all possible semantic values at a particular type, *i.e.,* the set defined by *Type*.

$$\begin{array}{rcl} \mathit{Type} & = & \mathit{World} \to \mathit{Set\ of\ Values} \\ \mathit{World} & = & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Type} \end{array}$$

Ahmed [5] notes that *Type* essentially has an impossibly large cardinality, in particular one larger than itself. One can also unroll the definition of *World* and see that the definition is a function with a self-reference in a negative position, $\mathit{World} = \mathit{Loc} \xrightarrow{\text{fin}} (\mathit{World} \to \mathit{Set\ of\ Values})$.

Much work on the semantics of higher-order references *approximates a solution* to the type-world equations, such as by introducing step-indexing [5], later modalities [8], or by using recursive domain equations [28]. We're interested in a different question: does there exist a non-approximate semantics for a class of higher-order references; that is, can we *avoid* the type-world circularity?

In this paper, we present how *type universes* can model a class of higher-order references and avoid the type-world circularity. Type universes are used to create a *type universe hierarchy* in dependent type theories to avoid the type-in-type inconsistency [22] where **Type** (the type of types) was considered to have type **Type**. To avoid the inconsistency, Martin-Löf [31] presented a solution that used a type universe hierarchy, where **Type** was stratified into an infinite hierarchy where $\textbf{Type}_0$ had type $\textbf{Type}_1$ which had type $\textbf{Type}_2$ and so on. The hierarchy eliminated the circularity that caused the original type theory's inconsistency, since type universes can only be contained in some larger universe, and never belong to themselves.

Formally, we use type universes to stratify the heap into *regions* of allocations, which allows us to distinguish allocations into regions in the future (higher universes) from the perspective of allocations in the past (lower universes). In our earlier example, s is logically in the past from

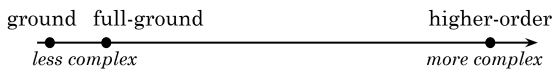
f—it could have been allocated first since it does not depend on f being allocated. This is reflected in its universe, which is lower than that of f's universe. Type universes allow us to describe the semantics of functions without quantifying over *all* future allocations but rather any future states of *existing* allocations. We define a semantics for a language where there is a correspondence between a type's universe and the level of the semantic Kripke world required to interpret the type. Viewed operationally, the type universe can be seen as the region where the expression's value will be allocated on the heap. (We discuss connections to regions à la region-based allocation in Section 5.)

Our contributions include:

- The decomposition of two axes of references previously conflated in the literature: what *order* of value a reference quantifies over (first-order/ground vs higher-order), and what *region of allocations*, represented by a universe, the reference quantifies over (past or future) (Section 2). We emphasize that quantifying over *future* regions introduces cycles.
- The decomposition of references gives rise to the language λ_{PR} with a type universe hierarchy to describe stratified higher-order references (Section 3), and acyclic full-ground references.
- A syntactic logical relations semantics for λ_{PR}, with stratified Kripke worlds that avoid the type-world circularity entirely (Section 3.2). We use the relation to prove termination, one of the metatheoretic properties lost when a terminating typed functional language is extended with higher-order references;
- An extension of λ_{PR}, $\lambda_{\mathrm{PR}}^{\circ}$, with an impredicative universe in the type universe hierarchy for cyclic full-ground references, with stratified higher-order references at all higher universes (Section 4). The language supports both stratified higher-order references and cyclic ground data structures in the heap.
- An extension of the semantics and proof of termination for cyclic full-ground references (Section 4.1); we conjecture this extension provides insight into open problems in previous work on semantics of full-ground references, such as the categorical semantics by Kammar et al. [25].

2 Background and Main Ideas

Past work typically divides references in three categories: *ground*, *full-ground*, and *higher-order* references [25, 34]. *Ground* references store ground (or base) type data, *full-ground* references store ground data and other full-ground references, and *higher-order* references store functions. The relative scale of the complexity of each kind of semantics can be informally visualized below.



One can also view the scale as the expressivity of the languages.

Possible worlds semantics allow local reasoning about the dynamic heap present with mutable references. The main idea is that the semantic interpretation of types is indexed by a possible world that describes the general layout, or shape, of the heap. The world describes the *locations allocated* and what possible values the locations could store, and the world grows like the heap when new locations are allocated.

For example, consider the possible worlds semantics for ground references, where the definition of a world restricts what can be stored in locations. A world is a map from a finite number of locations to the ground type stored at that particular location.

$$\mathit{World} = \mathit{Loc} \xrightarrow{\text{fin}} \mathcal{G}$$

$\mathcal{G}$ is the set of syntactic ground types, like $\mathcal{G} = \{\mathbf{Nat}, \mathbf{Bool}\}$. Given this definition of a world, the language modeled will have the same restriction in expressivity, where typed references can only store ground types. The language can restrict references with the following typing rule, where any new allocated reference is expected to store values of a ground type.

$$\frac{\Gamma \vdash e : \tau \qquad \tau \in \mathcal{G}}{\Gamma \vdash \mathbf{new}\ e : \mathbf{Ref}\ \tau}$$

The semantic interpretation of types is indexed by a world, that is, a semantic type is a function that expects a world and gives the semantic $\mathcal{P}(\mathit{Value})$ representing a type.

$$\mathit{Type} = \mathit{World} \longrightarrow \mathcal{P}(\mathit{Value})$$

For example, the semantics of ground types is defined as follows:

$$\begin{array}{rcl} [\![\mathbf{Nat}]\!](W) & = & \mathbb{N} \\ [\![\mathbf{Bool}]\!](W) & = & \{\mathbf{true}, \mathbf{false}\} \end{array}$$

The semantic values of **Nat** is the set of all natural numbers, and the semantic values of **Bool** is the set of values **true** and **false**. The interpretation of these ground types is simple in the sense that they do not depend on the current world W.

The world W is key for the semantics of mutable references. We model a type $\mathbf{Ref}\ \tau$ (where $\tau \in \mathcal{G}$) by the set of all locations in world W that map to type τ. The possible values for those locations are any syntactically well typed term, since they do not depend on any allocations. This means the interpretation of the values of locations in the world do not themselves depend on any world.

$$[\![\mathbf{Ref}\ \tau]\!](W) \quad = \quad \{\ell \mid W(\ell) = \tau\}$$

For example, given the world $W_1 = \{l_1 : \mathbf{Nat}, l_2 : \mathbf{Bool}\}$ the interpretation $[\![\mathbf{Ref\ Nat}]\!](W_1) = \{l_1\}$, since $l_1 : \mathbf{Nat} \in W_1$ and $[\![\mathbf{Ref\ Bool}]\!](W_1) = \{l_2\}$, since $l_2 : \mathbf{Bool} \in W_1$.

To describe possible worlds semantics for full-ground references, we can essentially follow the same formula. However, instead of $\mathcal{G}$, our worlds now map to syntactic full-ground types $\mathcal{F}$ that include types $\mathbf{Ref}\ \tau$ where $\tau \in \mathcal{F}$.

$$\mathit{World} = \mathit{Loc} \xrightarrow{\text{fin}} \mathcal{F}$$

The syntax is similarly restricted to only allow references to contain full-ground types. Cyclic references can allow the allocation of cyclic data structures, which may require more care in the semantics depending on the intended structure of the model. We describe a full-ground model in more detail in Section 4.

Mutable references that store functions that close over the heap—higher-order references—cause the semantics to suddenly increase in complexity; to see why, we review the semantics of functions in the presence of mutable references. Functions close over some local environment which can include values from the dynamic heap. The computation inside of a function can also affect the heap by updating or allocating more values. Furthermore, a function is a suspended computation with respect to a *particular* heap, but may be applied (and thus compute) in a later evolution of the heap. The semantics of functions require using a *future* world, *i.e.,* the world that exists later when the function is invoked compared to when the function was created. For example, a future world can be a world where some additional values have been allocated on the heap *after* the function was created. The semantics describe functions of type $\tau \longrightarrow \tau'$ in world W as semantic functions taking a semantic value τ in some future world from W and producing a semantic value τ' in another future world (because the function body is a computation that itself may perform some allocations). We describe the semantics formally, where $W' \sqsupseteq W$ describes a future world W' from W.

$$[\![\tau \longrightarrow \tau']\!](W) = \{\lambda x.e \mid \forall W' \sqsupseteq W, v \in [\![\tau]\!](W').\exists W'' \sqsupseteq W'.e[x/v] \in [\![\tau']\!](W'')\}$$

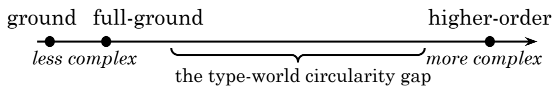
The semantic values of $\tau \to \tau'$ at world W are functions $\lambda x.e$ that given some $v \in [\![\tau]\!](W')$, *i.e.,* values in τ in any future world W', have a body e with x substituted for v in the interpretation of τ' at some future world W''.

To model higher-order references, we must change the definition of worlds. The world can no longer be a map from locations to syntactic types, but must instead map locations to sets of *semantic values*, because the semantic values of functions now rely on a *particular* world W. Not every function of type $\tau \to \tau'$ can be stored at a location expecting type $\tau \to \tau'$, but only *particular* functions that can be invoked from a future world of W. For example, given a world $\{\ell' : \mathbf{Nat} \to \mathbf{Nat}\}$, it may be safe to store a function $\lambda n : \mathbf{Nat}.(!\, \ell)$ in location ℓ'. However, determining whether the storage is safe requires that ℓ stores a semantic value that behaves like type **Nat**, and determining what ℓ stores requires a world. This means the interpretation of values of locations in the world depend on particular worlds, and it's possible for a world to depend on itself!

If we change the definition of a world to store semantic values and we still have semantic types rely on a world, a circularity emerges known as the *type-world circularity*.

$$\begin{array}{rcl} \mathit{Type} & = & \mathit{World} \to \mathcal{P}(\mathit{Value}) \\ \mathit{World} & = & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Type} \end{array}$$

The definition of worlds relies on itself in a negative position $\mathit{World} = \mathit{Loc} \to (\mathit{World} \to \mathcal{P}(\mathit{Value}))$. Ahmed [5] pointed out that such equations have no solution due to the cardinality of the *World* and *Type* sets, which are made inconsistently large by being defined in terms of each other (and thus, themselves). The type-world circularity is why the complexity gap between full-ground references and higher-order references is so large.



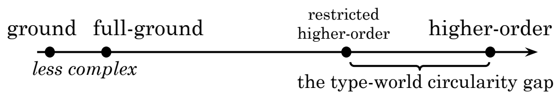
When encountering a circularity, one can try to instead *approximate* the circular definitions. Ahmed [5] does so by using *step-indexed logical relations* to model languages with higher-order references, where the type-world circularity is side-stepped using a decreasing number k. Using step-indexing, the definitions become:

$$\begin{array}{rcl} \mathit{Type}_{k+1} & = & \mathit{World}_k \to \mathcal{P}(\mathit{Value}) \\ \mathit{World}_k & = & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Type}_k \end{array}$$

Unrolling World_k, we have the following consistent definition $\mathit{World}_k = \mathit{Loc} \to (\mathit{World}_{k-1} \to \mathcal{P}(\mathit{Value}))$. The cardinality of Type_k no longer relies on itself but now relies on a smaller Type_{k-1}. The decreasing metric k comes from enumerating the number of steps of reduction that an expression takes. Type_k is an approximation of *Type*, so one can show a type is in *Type* by instead showing a type is in Type_k for all possible number of steps k. Other works use more advanced mathematics that allow reasoning about a solution to the circularity, like recursive domain equations [28].

2.1 Our contribution

In this paper, we ask (and answer): what kinds of functions can we store in references before the type-world circularity appears in the semantics?



We can start by only storing *simple* functions to check if the semantics still has a circularity. By simple, we mean constant functions or the identity function: in particular, functions are *closed* and *pure*, *i.e.,* do not rely on any local values or state in their environment. Semantically, we can store these functions on the heap without any regard to their behaviour in future worlds: closed functions will definitely not alter the state of the heap. This means we can pull our definitions of worlds back to what we had for full-ground and ground references: a world maps locations to a set of types C which includes types $\tau \to_c \tau'$ of *closed* functions.

$$\mathit{World} = \mathit{Loc} \xrightarrow{\text{fin}} C$$

We see that references storing closed functions do not create a circularity in the semantics like the usual higher-order references.

However, closed functions are highly restricted, and we would like to see how far we can expand the expressivity of functions we store in references. A desired feature of functions is that they can close over a local environment including state and perform computations that affect the state. We could include storing functions that may close over local *pure* values that do not rely on the heap, and functions that may close over only *ground* references.

We can ultimately induce a stratification: the semantics of functions that close over local state with only ground references (say of type $\tau \to_{\mathcal{G}} \tau'$) rely on our world definition from before.

$$\mathit{World}_1 = \mathit{Loc} \xrightarrow{\text{fin}} \mathcal{G}$$

The numbered index of World_1 has nothing to do with a semantic approximation, but rather describes what the world stores (in this case $\mathcal{G}$). We still index the semantic interpretation with a world, but this time with a ground heap World_1.

$$\mathit{Type} = \mathit{World}_1 \to \mathcal{P}(\mathit{Value})$$

But once we want to model *storing* functions $\tau \to_{\mathcal{G}} \tau'$ on the heap, we still have that the semantic values of $\tau \to_{\mathcal{G}} \tau'$ rely on World_1. To avoid a circularity, we define *another* world structure for heaps that specifically contain these kinds of semantic values.

$$\mathit{World}_2 = \mathit{Loc} \xrightarrow{\text{fin}} (\mathit{World}_1 \to \mathcal{P}(\mathit{Value}))$$

In these semantics, we still do not encounter a circularity, but have two different notions of worlds: World_1 storing only ground types, and World_2 storing the semantic values that close over World_1. We change our semantic interpretations of functions $\tau \to_{\mathcal{G}} \tau'$, as these functions do not need to behave in all future worlds W' (which would include World_2 and cause a circularity) but only future worlds from World_1.

$$[\![\tau \to_{\mathcal{G}} \tau']\!](W_1) = \{\lambda x.e \mid \forall W_1' \sqsupseteq_1 W_1, v \in [\![\tau]\!](W_1').\exists W_1'' \sqsupseteq_1 W_1'.e[x/v] \in [\![\tau']\!](W_1'')\}$$

Notice that these semantics do not require a step index, but rather the ground heap World_1, and the quantification over future worlds $\sqsupseteq_1$ only quantifies over future ground World_1. Functions of type $\tau \to_{\mathcal{G}} \tau'$ provably will not affect any location on the heap beyond locations storing ground values.

To generalize the stratified semantics, we ultimately must *enrich* the semantics of functions in the presence of mutable references to reason about what world the functions close over. The quantification over all possible future worlds W' from a particular W, *i.e.,* $W' \sqsupseteq W$, indicates what world a function closes over (W). We change this $W' \sqsupseteq W$ relationship in the semantics and restrict the quantification to future worlds at a particular level $W' \sqsupseteq_i W$. The quantification corresponds to behaviour in future states of *past* locations defined in W, but cannot include future locations in higher levels $i + 1$ that may include the function itself.

However, a question remains: how do we enrich the syntax to quantify over worlds? A function of type $\tau \to \tau'$ closing over world W is indistinguishable from a function $\tau \to \tau'$ closing over a different world W'.

We find that *type universes* are one elegant way of achieving this goal in both directions: type universes both enrich the semantics and the syntax. Type universes are used in dependent type theory to form a *type universe hierarchy*. In dependent type theory, a type universe hierarchy is used to circumvent a circularity with respect to the quantification of proofs, which can result in the unsoundness of the type theory. In our context, however, we use type universes to enforce a stratified relationship between types and worlds.

We show how type universes can correspond to worlds using our previous $World_1$ and $World_2$ definitions; we present the corresponding kinding rules for types in universes $\mathbf{Type}_1$ and $\mathbf{Type}_2$. Ground reference types depend on $World_1 = Loc \rightarrow \mathcal{G}$ in the semantics. We kind these ground reference types into $\mathbf{Type}_1$, given that they store $\mathbf{Type}_0$ types, the kind of ground types. Similarly, for reference types that store functions that close over ground values, *i.e.,* $\mathbf{Type}_1$ in the hierarchy, we classify these reference types into $\mathbf{Type}_2$, corresponding to the idea that the semantics of these types depend on $World_2$. Here we use :: to indicate kinding a type.

$$\frac{\tau :: \mathbf{Type}_0}{\mathbf{Ref}\ \tau :: \mathbf{Type}_1} \qquad\qquad \frac{\tau :: \mathbf{Type}_1}{\mathbf{Ref}\ \tau :: \mathbf{Type}_2}$$

In the semantics, we had that the function type $\tau \rightarrow_{\mathcal{G}} \tau'$ only relied on $World_1$ for its interpretation; this was possible because functions of this type only close over ground references, which are in universe $\mathbf{Type}_1$. To maintain the correspondence that $\mathbf{Type}_1$ types rely on $World_1$ in the semantics, $\tau \rightarrow_{\mathcal{G}} \tau' :: \mathbf{Type}_1$. To check *functions* of type $\tau \rightarrow_{\mathcal{G}} \tau'$, we can now check the environment for *kinds* $\mathbf{Type}_1$.

$$\frac{\Gamma, x : \tau \vdash e : \tau' \qquad 1 \geq \mathbf{max\text{-}level}(\Gamma)}{\Gamma \vdash \boldsymbol{\lambda}\ x : \tau_1.e : \tau \rightarrow_{\mathcal{G}} \tau'}$$

The typing rule includes the side condition that the maximum universe level of all the types must be less than or equal to 1. The maximum universe level is determined by kinding all the types in Γ to determine their level and taking the maximum.

Generalizing this approach, we intuitively have the following stratified semantic equations, with $Type_k$ corresponding to a type universe.

$$\begin{aligned} Type_k &= World_k \rightarrow \mathcal{P}(Value) \\ World_{k+1} &= Loc \xrightarrow{\text{fin}} Type_k \end{aligned}$$

The semantic type level and world level match, but now worlds that store semantic types at level k can only be accessed by semantic types at level $k+1$. A key difference in these equations compared to Ahmed [5]'s semantic approximation is that our equations describe a literal stratification seen in the language. In particular, our semantics of functions takes this stratification into account when quantifying over future worlds.

We augment the world definition to allow locations to map to types at levels *lower than* the level k, rather than at exactly k.

$$\begin{aligned} Type_k &= World_k \rightarrow \mathcal{P}(Value) \\ World_{k+1} &= Loc \xrightarrow{\text{fin}} (i : \mathrm{Fin}(k+1)) \times Type_i \end{aligned}$$

Here, each location in a world is associated with the index of the expected level of the type of values it stores. The index must be of finite type $k+1$, that is, any number i such that $i \in \{0, \ldots, k\}$. The world definition mirrors cumulativite type universes in dependent type theory, where a type in universe $\mathbf{Type}_i$ is implicitly also in $\mathbf{Type}_j$ when $j > i$.

One of the key kinding rules in the type universe hierarchy is the rule for determining the universe level of reference types. We generalize the previous kinding rules to determine that a

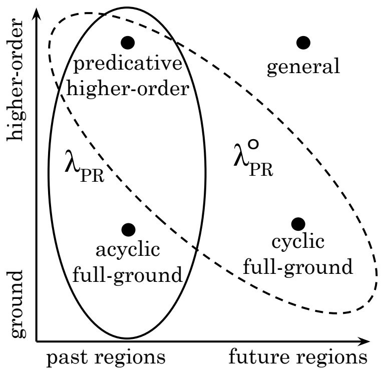


Fig. 1. The design space of references, including whether values can refer to past or future allocations.

reference type will be one level higher than the type it stores.

$$\frac{\tau :: \mathbf{Type}_i}{\mathbf{Ref}\ \tau :: \mathbf{Type}_{i+1}}$$

We also consider how to determine the universe level of a function type, to determine a function's world level for quantification over future worlds in the semantics. We saw previously $\tau \rightarrow_{\mathcal{G}} \tau'$ describes functions that close over ground reference types, *i.e.,* $\mathbf{Type}_1$ types, so $\tau \rightarrow_{\mathcal{G}} \tau'$ was in universe $\mathbf{Type}_1$ as well. Since a function can close over any dynamic values in the heap and possibly read or update them, the universes of these values must affect the universe of a function type. This means that the universe level of a function type is not only influenced by the domain and codomain types, but also the types captured in its environment!

To distinguish functions that rely on different worlds, we include an annotation on the function arrow. When kinding a function type, the only information available is the universes of the domain and codomain types, and not the (maximum) universe of the environment. We check that the universe annotation is consistent with the universe of the domain and codomain types. We separately check the annotation against the environment when the type is used to check a term.

$$\frac{\tau_1 :: \mathbf{Type}_i \qquad \tau_2 :: \mathbf{Type}_j \qquad k \geq i, j}{\tau_1 \xrightarrow{k} \tau_2 :: \mathbf{Type}_k} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad k \geq \mathbf{max\text{-}level}(\Gamma, \tau_1, \tau_2)}{\Gamma \vdash \boldsymbol{\lambda}\ x : \tau_1 . e : \tau_1 \xrightarrow{k} \tau_2}$$

The types $\mathbf{Nat} \xrightarrow{1} \mathbf{Nat}$ and $\mathbf{Nat} \xrightarrow{0} \mathbf{Nat}$ are both well kinded, but describe different kinds of functions: the former are permitted to capture references in universe $\mathbf{Type}_1$, while the latter cannot capture references at all. This is also reflected in the semantics: the semantics of type $\mathbf{Nat} \xrightarrow{1} \mathbf{Nat}$ describes functions that may be affected by changes in heaps of shape $World_1$ only, whereas $\mathbf{Nat} \xrightarrow{0} \mathbf{Nat}$ describes functions that are not affected (nor affect) the heap at all ($World_0$). The behaviour of semantic values of type $\mathbf{Nat} \xrightarrow{1} \mathbf{Nat}$ can only quantify over future states of $World_1$, and cannot access (or quantify over) $World_2$ or above.

Type universes classify stored values into *regions* of allocations separated by logical time. Higher universes correspond to regions of allocations that happen in the future from the perspective of allocations in the past (lower universes). A type at universe $\mathbf{Type}_i$ can only refer to types of values that could have been allocated in the past, which live in lower universes such as $\mathbf{Type}_{i-1}$. However, "future" and "past" here do not refer to the literal execution order of the program. Instead, they refer to a dependency order between when allocations could have occured given the other allocations they depend on. Because universes kind types, this does allow some values to "time travel" to

$$
\begin{array}{llll}
\text{Type} & \tau & ::= & \mathbf{Unit} \mid \mathbf{Nat} \mid \tau \xrightarrow{k} \tau \mid \mathbf{Ref}\ \tau \\
\text{Expr} & e & ::= & x \mid n \mid \langle\rangle \mid \boldsymbol{\lambda}\, x : \tau.e \mid e\ e \mid \mathbf{new}\ e \mid \mathbf{!}\ e \mid e \coloneqq e \\
\text{Value} & v & ::= & x \mid n \mid \langle\rangle \mid \boldsymbol{\lambda}\, x : \tau.e \mid \ell
\end{array}
$$

Fig. 2. λ_{PR} syntax

their logically correct region, irrespective of when they actually happen during program execution, since dependency restricts the kind of types, and not the literal time the allocation happens in the program.

Distinguishing between past and future allocations also enables distinguishing between full-ground references and *cyclic* full-ground references, which are commonly conflated in the literature [25, 34]. In our predicative universe hierarchy, a full-ground reference ℓ' can store another location ℓ, but ℓ cannot possibly store ℓ' to create a cycle. Viewed through the lens of past and future allocations, ℓ cannot rely on future allocations (which includes ℓ'), and ℓ' can only store ℓ because ℓ was allocated in the past. A cycle occurs when a reference can store a value from its current region of allocation.

In Section 3, we present a language λ_{PR} with "predicative" higher-order references: the stratified higher-order references that result from a predicative type universe hierarchy. The higher-order values (functions) stored in references can only quantify over past regions. This is made clear in the semantics of these functions, where the semantic values quantify over future worlds at a particular level rather than all possible future worlds (which corresponds to general higher-order references). Full-ground references, *i.e.,* reference to other references, are still permitted, but are also restricted to past regions, which disallows any cycles in the heap.

We modify the hierarchy in Section 4 to define $\lambda^{\circ}_{\text{PR}}$ which allows cyclic full-ground references and predicative higher-order references. The universe hierarchy allows one level of the universe where stored values can be in the same universe as the reference itself (allowing a limited notion of quantification over future allocations, *i.e.,* the current world), but functions must always live in higher universes.

$$
\frac{\tau :: \mathbf{Type}_0}{\mathbf{Ref}\ \tau :: \mathbf{Type}_0}
\qquad\qquad
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad k \geq \mathbf{max\text{-}level}(\Gamma, \tau_1, \tau_2, \boxed{1}\,)}{\Gamma \vdash \boldsymbol{\lambda}\, x : \tau_1.e : \tau_1 \xrightarrow{k} \tau_2}
$$

We present our final classification of references in Figure 1, not only by the order of types they store (first-order/ground vs higher-order), but also by regions of allocation (past vs future), which is approximated using our type universe hierarchy (the kind of the type). We also label our languages λ_{PR} and $\lambda^{\circ}_{\text{PR}}$ along the design space.

3 λ_{PR}: Predicative and Acyclic References

We present the language λ_{PR}, where PR stands for *predicative references*, inspired by predicative universe hierarchies. The type system is essentially standard, with a type universe hierarchy to enforce heap stratification, which also avoids the type-world circularity in the semantics. We do this by classifying reference types into a separate universe from the universe of types they store. The classification of reference types is not enough on its own—functions that may be stored in references can close over parts of the heap through local variables. We then classify function types by the maximum universe level of their environment, domain, and codomain types. The hierarchy maintains the invariant that the universe level is what part of the heap the values depend on. In the semantics, the invariant is reflected as the level of the Kripke *world*.

In Figure 2, we present the syntax of λ_{PR}, a simply typed λ-calculus with higher-order references. The language includes base types, **Nat** and **Unit**, represented by the metavariable n and value $\langle\rangle$ respectively. The rest of the syntax is standard, with **new**, **!**, and $\coloneqq$ for initializing, dereferencing,

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

$$\frac{}{\Gamma \vdash n : \mathbf{Nat}} \qquad \frac{}{\Gamma \vdash \langle\rangle : \mathbf{Unit}} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad k \geq \mathbf{max\text{-}level}(\Gamma, \tau_1, \tau_2)}{\Gamma \vdash \boldsymbol{\lambda}\, x : \tau_1 . e : \tau_1 \xrightarrow{k} \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{k} \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{new}\ e : \mathbf{Ref}\ \tau} \qquad \frac{\Gamma \vdash e : \mathbf{Ref}\ \tau}{\Gamma \vdash \mathbf{!}\, e : \tau} \qquad \frac{\Gamma \vdash e_1 : \mathbf{Ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \coloneqq e_2 : \mathbf{Unit}}$$

Fig. 3. λ_{PR} typing.

$$\boxed{\tau :: \mathbf{Type}_i}$$

$$\frac{\tau \in \{\mathbf{Nat}, \mathbf{Unit}\}}{\tau :: \mathbf{Type}_0} \qquad \frac{\tau :: \mathbf{Type}_i}{\mathbf{Ref}\ \tau :: \mathbf{Type}_{i+1}} \qquad \frac{\tau_1 :: \mathbf{Type}_i \qquad \tau_2 :: \mathbf{Type}_j \qquad k \geq \mathbf{max\text{-}level}(\tau_1, \tau_2)}{\tau_1 \xrightarrow{k} \tau_2 :: \mathbf{Type}_k}$$

Fig. 4. λ_{PR} kinding.

and updating references. The only exception is the annotation on the function type $\tau \xrightarrow{k} \tau$, where k indicates the universe of the function type and can be read as where to allocate a function of this type on the heap.

In Figure 3, we present the typing rules of λ_{PR}. The rules are standard, except for the function case. We check (or could infer) the annotation on the function type against the current environment, $k \geq \mathbf{max\text{-}level}(\Gamma, \tau_1, \tau_2)$. Determining the level annotation k relies on kinding the domain, codomain, and types in Γ. Then the side condition requires that the level k of the function type is greater than or *equal* to the maximum of type levels of variables in Γ and types τ_1 and τ_2. The equality constraint on k comes from the fact that functions are *immutable*, compared to references that are *mutable*. A function can only affect the heap through captured variables, which means the function can live at the same level as the maximum level of its environment, domain, and codomain types. If it were possible to update the computations inside functions, *i.e.*, functions were mutable, the type level would also need to increase by one to maintain the separation of past and future allocations. Notice also that the environment Γ could contain more variables (and thus universe levels) than the function actually captures, but through contraction one can easily determine the smallest level of a function type by only the variables captured.

We present the kinding of types in Figure 4. The simple base types **Nat** and **Unit** are of $\mathbf{Type}_0$. The universe level of a function is determined by the annotation, given that this annotation is greater than or equal to the universe levels of the domain and codomain types. And finally, a reference type is one level higher than the level of the type it stores. The combination of the reference type and function type kinding rules is key to avoid the circularity in the semantics, and forms the basis of the type universe hierarchy.

In this presentation of λ_{PR}, substitution and weakening do not hold; we discuss how to recover these properties for a programming language in Section 3.1.

In Figure 5, we present the operational semantics of λ_{PR}, which are standard for languages with mutable references. We represent the dynamic heap h by mapping locations ℓ to values v. Each expression e runs against a heap h, and steps to an expression or value with a corresponding heap. The heap may have been updated by evaluating some subexpression e_1 of e, and by threading the dynamic heap through evaluation, changes to the heap can be used by further steps. Type safety with respect to these operational semantics follows as a corollary of the fundamental lemma (Theorem 3.4).

$$\text{Heap} \quad h \quad ::= \quad \cdot \mid h[\ell \mapsto v]$$

$\boxed{\langle h \mid e\rangle \longrightarrow \langle h \mid e\rangle}$

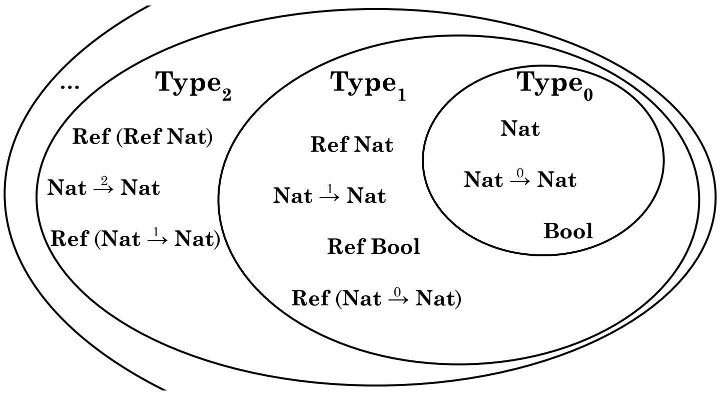
$$\begin{array}{rcll}
\langle h \mid (\boldsymbol{\lambda}\, x : \tau . e)\ v\rangle & \longrightarrow & \langle h \mid e[x/v]\rangle & \\
\langle h \mid \mathbf{new}\ v\rangle & \longrightarrow & \langle h[\ell \mapsto v] \mid \ell\rangle & \ell \text{ fresh in } h, v \\
\langle h \mid !\ \ell\rangle & \longrightarrow & \langle h \mid v\rangle & h(\ell) = v \\
\langle h \mid \ell := v\rangle & \longrightarrow & \langle h[\ell \mapsto v] \mid \langle\rangle\rangle &
\end{array}$$

$\boxed{\langle h \mid e\rangle \longrightarrow^* \langle h \mid e\rangle}$

$$\frac{\langle h \mid e\rangle \longrightarrow^* \langle h' \mid e'\rangle}{\langle h \mid e\ e_2\rangle \longrightarrow^* \langle h' \mid e'\ e_2\rangle} \qquad \frac{\langle h \mid e\rangle \longrightarrow^* \langle h' \mid e'\rangle}{\langle h \mid v\ e\rangle \longrightarrow^* \langle h' \mid v\ e'\rangle} \qquad \frac{\langle h \mid e\rangle \longrightarrow^* \langle h' \mid e'\rangle}{\langle h \mid \mathbf{new}\ e\rangle \longrightarrow^* \langle h' \mid \mathbf{new}\ e'\rangle} \qquad \cdots$$

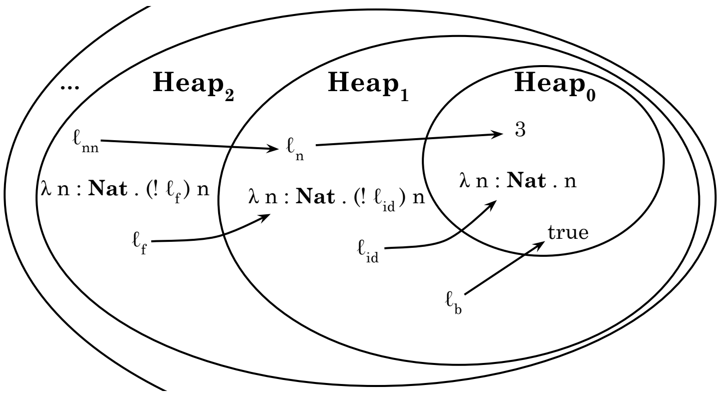
Fig. 5. λ_{PR} operational semantics.

The dynamic heap h in the operational semantics has no explicit stratification, but the worlds in the semantics do, so we can visualize the heap as stratified to see the correspondence. We visualize a few examples of types along the universe hierarchy.


...
$\mathbf{Type}_2$
$\mathbf{Ref\ (Ref\ Nat)}$
$\mathbf{Nat} \xrightarrow{2} \mathbf{Nat}$
$\mathbf{Ref\ (Nat} \xrightarrow{1} \mathbf{Nat)}$
$\mathbf{Type}_1$
$\mathbf{Ref\ Nat}$
$\mathbf{Nat} \xrightarrow{1} \mathbf{Nat}$
$\mathbf{Ref\ Bool}$
$\mathbf{Ref\ (Nat} \xrightarrow{0} \mathbf{Nat)}$
$\mathbf{Type}_0$
$\mathbf{Nat}$
$\mathbf{Nat} \xrightarrow{0} \mathbf{Nat}$
$\mathbf{Bool}$

Types climb the hierarchy by allocation on the heap (**Ref** types), but function types also climb the hierarchy by capturing values from the heap in their environment.

Now when we switch our thinking to values that *inhabit* the types along the hierarchy, and envision the kinds $\mathbf{Type}_0$, $\mathbf{Type}_1$, $\mathbf{Type}_2$ as parts of the heap, we then get a visual of the stratified heap.


...
$\mathbf{Heap}_2$
ℓ_{nn}
$\lambda\ n : \mathbf{Nat}\ .\ (!\ \ell_f)\ n$
ℓ_f
$\mathbf{Heap}_1$
ℓ_n
$\lambda\ n : \mathbf{Nat}\ .\ (!\ \ell_{id})\ n$
ℓ_{id}
ℓ_b
$\mathbf{Heap}_0$
3
$\lambda\ n : \mathbf{Nat}\ .\ n$
true

When values of $\mathbf{Type}_0$ are allocated on the heap, the *locations* pointing to these values are in $\mathbf{Type}_1$. Examples of such values are ℓ_n, ℓ_{id} and ℓ_b. One can consider ℓ_n as living in a separate region of the heap compared to its value 3. Similarly, when a function closes over $\textbf{Type}_1$ values, the function is considered of $\textbf{Type}_1$, like the function $\boldsymbol{\lambda}\ n : \textbf{Nat}.(!\ \ell_{id})\ n$. This trend continues up the hierarchy, where we can see examples like ℓ_{nn} and $\boldsymbol{\lambda}\ n : \textbf{Nat}.(!\ \ell_f)\ n$. Notice that all the locations must always point downwards in the heap, and there is no way to sneak backpatching through a higher-order reference, since the result is ill-kinded (and thus ill-typed). In the semantics, we can view these parts of the heap as realizing particular worlds in leveled worlds World_0, World_1, World_2, and so on.

We now present the logical relation for λ_{PR} to see how the semantics uses type universes to correspond to Kripke worlds, and prove termination as a result.

3.1 Weakening and Substitution

In λ_{PR}, the level of the context can change a function type when an unused variable is added, which means weakening and substitution do not hold. These properties are currently irrelevant in our semantic investigations, but we could adjust the function typing rule for a programming language. One alternative condition is to instead determine a function's type level by the *free variables* in the function body, rather than the context as a whole, so the function typing rule changes to:

$$\frac{\Gamma, y : \tau_1 \vdash e : \tau_2 \quad k \geq \textbf{max-level}(FV(e), \tau_1, \tau_2)}{\Gamma \vdash \lambda y : \tau_1.e : \tau_1 \rightarrow^k \tau_2}$$

The resulting programming language is similar to Swift [39], and other recent practical language designs we touch on further in Section 5.

With this function typing rule, weakening and substitution hold.

Lemma 3.1 (Weakening). *If* $\Gamma \vdash e : \tau$ *and* $x \notin e$ *and* $x \notin \Gamma$, *then* $\Gamma, x : \tau_x \vdash e : \tau$.

Proof. (Sketch). By induction on $\Gamma \vdash e : \tau$. In the function case $\lambda y : \tau_1.e$, the level k is determined by the maximum level of the free variables in the function body e. Since $x \notin \lambda y : \tau_1.e$, the free variables of e do not change, and the level k stays the same. □

Lemma 3.2 (Substitution). *If* $\Gamma, x : \tau_x, \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash v : \tau_x$, *then* $\Gamma, \Gamma' \vdash e[x/v] : \tau$.

Proof. (Sketch). By induction on $\Gamma \vdash e : \tau$. In the function case $\lambda y : \tau_1.e$, the level k is determined by the maximum level of the free variables in the function body e. There are two cases: either $x \in FV(e)$ and affects the maximum level k, or $x \notin FV(e)$ and the maximum level k remains the same as before. When $x \in FV(e)$ and affects the maximum level k, we may end up with a level $j < k$ after performing the substitution, since $x \notin FV(e)$ after substitution. In this case, we can still pick the previous higher k rather than the lower j to fulfill $k \geq \textbf{max-level}(FV(e), \tau_1, \tau_2)$. □

3.2 Logical Relation and Proof of Termination

Our logical relation relies on an index: the type universe level. Our relation interprets types as sets of expressions that step to a value, and we show the value only depends on the Kripke world dictated by its universe level. We then prove all well typed expressions in λ_{PR} are in the set associated with their type in Theorem 3.3. To be in the set associated with its type, an expression must step to a value, which allows us to conclude that all well typed expressions terminate. A full introduction of proving termination with logical relations can be found in standard textbooks [17].

Traditionally, logical relations are divided between an *expression* relation and a *value* relation—relations over syntactic types to semantic values. This divide corresponds nicely to the idea that expressions step and values do not. We define the expression relation for λ_{PR} with respect to the operational semantics presented in Figure 5. The value relation $\mathcal{V}[\![\tau]\!]$ gives the set of values that represent (or behave like) τ and the expression relation $\mathcal{E}[\![\tau]\!]$ gives the set of expressions that step to a $v \in \mathcal{V}[\![\tau]\!]$.

$$
\begin{array}{lcl}
\mathcal{V}[\![\mathbf{Nat}]\!]_0(W_0) & \overset{def}{=} & \mathbb{N} \\
\mathcal{V}[\![\mathbf{Unit}]\!]_0(W_0) & \overset{def}{=} & \{\langle\rangle\} \\
\mathcal{V}[\![\mathbf{Ref}\ \tau]\!]_{i+1}(W_{i+1}) & \overset{def}{=} & \{l \mid W_{i+1}(l) = (i, \mathcal{V}[\![\tau]\!]_i)\} \\
\mathcal{V}[\![\tau^i \xrightarrow{k} \sigma^j]\!]_k(W_k) & \overset{def}{=} & \{\boldsymbol{\lambda}\ x : \tau.e \mid \forall W_k' \sqsupseteq_k W_k, v \in \mathcal{V}[\![\tau]\!]_i(\lfloor W_k' \rfloor_i).e[v/x] \in \mathcal{E}[\![\sigma]\!]_j(\lceil W_k' \rceil)\} \\
\mathcal{V}[\![\tau]\!]_i(\mathbf{W}) & \overset{def}{=} & \mathcal{V}[\![\tau]\!]_i(\lfloor \mathbf{W} \rfloor_i)
\end{array}
$$

Fig. 6. Value relation for λ_{PR} types.

As summarized previously in Section 2, one has to model the heap shape to define a semantics for a language with mutable references, since expressions access and modify the heap. The shape of a heap is modelled using a *world*, a finite map from locations to sets of semantic values of types. Since each location is mapped to a set of all possible semantic values, the world can represent the shape of any particular heap.

To define worlds for the value relation, we refer back to our semantic equations from Section 2.

$$
\begin{array}{rcl}
Type_k & = & World_k \longrightarrow \mathcal{P}(Value) \\
World_{k+1} & = & Loc \xrightarrow{\text{fin}} (i : \text{Fin}(k+1)) \times Type_i
\end{array}
$$

The semantic values for a type τ rely on a particular world at level k, which is determined by the universe level of the type. The world at level k contains locations mapping to a type at level i such that $i < k$, which guarantees values at level k only depend on level k and below.

A heap that realizes a world $World_k$ is a heap that maps locations of type $\tau :: \mathbf{Type}_i$ such that $i < k$. However, according to our operational semantics in Figure 5, *expressions* step with a heap with no restrictions on what type of values are allocated. We need another kind of world to describe heaps where locations can be mapped to values at any level. This world, which we call a “general” world, is the world we use for the definition of the expression relation and consists of a product of worlds at individual levels.

$$\mathbf{World} = \forall i \in \mathbb{N}. World_i$$

We distinguish these general worlds $\mathbf{W} \in \mathbf{World}$ using a boldface font, and use names without indices.

Ultimately these *leveled* worlds $World_k$ support an interpretation of types that indicates *where* values are allocated, based on their universe level. If we alternatively allowed the value relation to be defined with respect to a general world $\mathbf{W}$, we would encounter the type-world circularity. Expressions are evaluated with heaps mapping locations at any level and are defined with respect to general worlds. This gives the property that e can be evaluated in any heap that may have locations beyond what the *value* of e depends on, *i.e.*, e may use locations only locally during evaluation, a property that can validate more program equivalences (*e.g.*, **let** $x =$ (**new** 3) **in** $e \equiv e$ when $x \notin e$).

However, because the expression e has a type level $\mathbf{Type}_i$ according to our type universe system, we know eventually its *value* can be restricted to the interpretation at $World_i$. Because the interpretation of a value can be restricted to $World_i$, the value relation is defined over leveled worlds. The expression relation is also indexed by the universe level to reflect that eventually, the final value is restricted to the leveled world at the index. These definitions reinforce that our type universe hierarchy describes where a value is allocated, and what prior allocations are depended on.

We present the value relation in Figure 6, where the index of the relation corresponds to the universe level of the type. We use italicized notation W to represent a particular world at a particular level, and usually include the level in the name, so $W_i \in World_i$. The resulting sets are simply sets of values, which is standard, but what is more interesting is the associated Kripke world structure. The value relation indexes the *world* at the same level as the universe, avoiding the type-world

Leveled World Extension

$W'_k \sqsupseteq_k W_k \iff \mathrm{dom}(W_k) \subseteq \mathrm{dom}(W'_k) \land \forall l \in W_k . W'_k(l) = W_k(l)$

General World Extension

$\mathbf{W}' \sqsupseteq \mathbf{W} \iff \forall i \in \mathbb{N}.\mathrm{dom}(\mathbf{W}i) \subseteq \mathrm{dom}(\mathbf{W}'i) \land \forall l \in (\mathbf{W}i).(\mathbf{W}'i)(l) = (\mathbf{W}i)(l)$

$$\begin{array}{llll}
\textbf{Leveled Lowering} & \lfloor \_ \rfloor_{\_} & : & \mathit{World}_k \to (i : \mathrm{Fin}(k+1)) \to \mathit{World}_i \\
 & \lfloor W_k \rfloor_i & = & \{(l, j, \mathcal{R}) \mid j < i \land (l, j, \mathcal{R}) \in W_k\} \\
\textbf{General Lowering} & \lfloor \_ \rfloor_{\_} & : & \mathbf{World} \to (i : \mathbb{N}) \to \mathit{World}_i \\
 & \lfloor \mathbf{W} \rfloor_i & = & (\mathbf{W}\ i) \\
\textbf{Lifting} & \lceil \_ \rceil & : & \mathit{World}_k \to \mathbf{World} \\
 & \lceil W_k \rceil & = & \lambda i. \begin{cases} \{(\ell, j, \mathcal{R}) \mid j < i \land (\ell, j, \mathcal{R}) \in W_k\}, & 0 < i \le k \\ W_k, & \text{otherwise} \end{cases}
\end{array}$$

Fig. 7. Additional definitions for worlds.

circularity and ensuring that values are not depending on locations beyond their world level. Values at $\mathbf{Type}_0$ are pure, so there is no need for a world for their interpretation. We define World_0 as the empty map (*i.e.*, a finite map mapping no locations).

The value relation for a reference type $\mathbf{Ref}\ \tau$ is indexed by a number $i + 1$, since the universe level of $\mathbf{Ref}\ \tau$ is always $i + 1$ for τ's level i. The world associated with the type $\mathbf{Ref}\ \tau$ is also at level $i + 1$. Then, the set of values for type $\mathbf{Ref}\ \tau$ is all the locations in the current world that map to the set of values associated with τ, *i.e.*, $\mathcal{V}[\![\tau]\!]_i$. The level of worlds associated with such values is necessarily lower than that of the current world W_{i+1}, since the set is indexed by i.

The value relation over a function type is defined over domain and codomain types annotated with their kind, τ^i and σ^j. Kinding types is easy as shown in Figure 4, and the levels are necessary for indexing the relations for τ and σ correctly. The set contains functions that given any value v in the value relation for the domain type τ, the body of the function e with the parameter x substituted with v is in the *expression* relation for σ.

The set of values for a function type $\tau \xrightarrow{k} \sigma$ relies on additional definitions for worlds, summarized in Figure 7. The first definition is *world extension* $\sqsupseteq_k$ over leveled worlds, which describes a *future world* W'_k with respect to the world W_k, and is only defined over worlds at the same level k. World extension is necessary because a function can be applied later in a future *heap*, and so the semantics must include function values valid in future *worlds*. The relation guarantees that W'_k has as many locations as W_k at the same types, but W'_k may have additional locations allocated. We also define world extension for general worlds $\mathbf{W}'$ and $\mathbf{W}$, which is essentially defined as $\sqsupseteq_k$ at all levels.

The next definition *lowers* a world W_k to level i ($\lfloor W_k \rfloor_i$), given that $i \le k$. Lowering a world is necessary since the values in the relation $\mathcal{V}[\![\tau]\!]_i$ rely on a world indexed at the same level i. However, the future world W'_k in the function case is at level k, which is potentially incompatible with level i. We use the lowering operation to remove all locations that map to types with levels *higher* than level $i - 1$. Each location l is associated with an index j, and lowering keeps mappings $(l, j, \mathcal{R})$, where $j < i$ (and so $\mathcal{R}$ is Type_j). The resulting world $\lfloor W'_k \rfloor_i$ is a World_i since all locations map to types with level j where $j < i$. The lowering operation can be considered a form of semantic garbage collection.

To mediate between worlds at a particular level (World_i) and general worlds ($\mathbf{World}$), we have an operation that *lifts* a world W_k to a general world shown in Figure 7. The lift operation can be considered a "cast" from a leveled world to a world that allows allocations at higher levels. For each level $i < k$, we reconstruct a World_i consisting of locations from W_k less than i. For levels higher than k, locations with higher levels have yet to be allocated, but all other locations lower remain the same as W_k.

$$\begin{array}{rcl}\mathcal{E}[\![\tau]\!]_i(\mathbf{W}) & \stackrel{def}{=} & \{e \mid \ \forall \mathbf{W}' \sqsupseteq \mathbf{W}, \mathbf{W}' \ \vdash\ h'.\langle h' \mid e\rangle \longrightarrow^* \langle h'' \mid v\rangle \wedge \\ & & \qquad \exists \mathbf{W}'' \sqsupseteq \mathbf{W}'.\mathbf{W}'' \ \vdash\ h'' \wedge v \in \mathcal{V}[\![\tau]\!]_i(\mathbf{W}'')\}\end{array}$$

Fig. 8. Expression relation.

The expression relation defined in Figure 8 describes the set of terminating expressions associated with a type τ at universe level i. The set consists of expressions that step to a value v in the value relation for τ at level i, that is, $\mathcal{V}[\![\tau]\!]_i$. We need to define how our heaps h realize the general worlds $\mathbf{W}$, which we write as $\mathbf{W} \ \vdash\ h$. The realization ensures all expressions step to a value with a heap that “matches” the world the relation is indexed over.

$$\mathbf{W} \ \vdash\ h \iff \mathrm{dom}(h) = \bigcup_{i \in \mathbb{N}} \mathrm{dom}(\mathbf{W}\ i) \wedge \forall l \in h.h(l) \in (\mathbf{W}i)(l)(\lfloor \mathbf{W} \rfloor_i)$$

Since the heap maps values at different levels, then the domain should match the domain of $\mathbf{W}$ at all possible levels. The value that the heap maps to a location should be in the expected set of values mapped by $\mathbf{W}$, given that $\mathbf{W}$ is lowered to the appropriate level. The level of the location in h used to index $\mathbf{W}$ correctly for a particular level is given by the index of the relation when needed in the subsequent proof.

There are three worlds $\mathbf{W}$, $\mathbf{W}'$, and $\mathbf{W}''$ in the expression relation. The world $\mathbf{W}$ is the model of the minimum or initial heap needed for evaluation. Expressions can be evaluated in heaps larger than the initial world $\mathbf{W}$, which corresponds to a heap h' realizing the future world $\mathbf{W}'$. The heap h' contains the same locations as $\mathbf{W}'$, but maps each location to a *single* value from the value relation mapped by $\mathbf{W}'$. There is a final heap at the end of evaluation h'', and there must exist a future world $\mathbf{W}'' \sqsupseteq \mathbf{W}'$ related to h''. Finally, the value v resulting from evaluation is in the value relation for type τ, with the final world $\mathbf{W}''$ lowered to i since v does not rely on any part of the heap higher than i. Lowering also maintains the stratification invariant since values such as functions are guaranteed not to depend on the heap at levels greater than i, and thus these levels can be “deallocated”.

We prove λ_{PR} terminating by proving the *fundamental lemma* (Theorem 3.3) for the relation, that is, every well typed expression in λ_{PR} is in the expression relation. Given every well typed expression is in the expression relation, every expression steps to a value. Because the relation is defined over closed expressions, we must define a relation-respecting substitution γ (mapping variables to values) with respect to a typing context Γ to close expressions. These kinds of substitutions γ are defined in the usual manner using a context relation $\mathcal{G}[\![\cdot]\!]$. Variables are mapped to values in the value relation, and in our case we must define the context relation with respect to a general world $\mathbf{W}$, as different variables have different types at various universe levels.

$$\begin{array}{rcl}\mathcal{G}[\![\cdot]\!](\mathbf{W}) & \stackrel{def}{=} & \emptyset \\ \mathcal{G}[\![\Gamma, x : \tau]\!](\mathbf{W}) & \stackrel{def}{=} & \{\gamma[x \mapsto v] \mid \tau :: \mathbf{Type}_i \wedge v \in \mathcal{V}[\![\tau]\!]_i(\mathbf{W}) \wedge \gamma \in \mathcal{G}[\![\Gamma]\!](\mathbf{W})\}\end{array}$$

The fundamental lemma states that if an expression e is well typed at type τ with universe level $\mathbf{Type}_i$, then e is in the expression relation for τ, *i.e.,* e steps to a value v at type τ. We extend the lemma to open terms by using a substitution γ from the context relation $\mathcal{G}[\![\cdot]\!](\mathbf{W})$.

Theorem 3.3. *If* $\Gamma \vdash e : \tau$ *and* $\tau :: \mathbf{Type}_i$, *then* $\forall \gamma \in \mathcal{G}[\![\Gamma]\!](\mathbf{W}).\gamma(e) \in \mathcal{E}[\![\tau]\!]_i(\mathbf{W})$.

We present the key cases of the proof, in particular the cases for functions and references, and the full proof is in the anonymous supplementary materials. Functions are particularly important, as we mediate between worlds at level i, j, and k, and prove that a function body does not reference a location beyond what is allowed by its universe level.

Proof. By induction on $\Gamma \vdash e : \tau$.

Case: [Lam] We have

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad k \geq \textbf{max-level}(\Gamma, \tau_1, \tau_2)}{\Gamma \vdash \boldsymbol{\lambda}\, x : \tau_1 . e : \tau_1 \xrightarrow{k} \tau_2}$$

and that $\tau_1 \xrightarrow{k} \tau_2 :: \textbf{Type}_k$, and have that $\tau_1 :: \textbf{Type}_i$ and $\tau_2 :: \textbf{Type}_j$.
The key insight is that the side condition allows us to determine that a lifted *leveled* world $\lceil W'_k \rceil$ is a future world of a *general* world $\mathbf{W}$. This is not true in general, but it is precisely the side condition $k \geq \textbf{max-level}(\Gamma, x : \tau_1, \tau_2)$ that allows us to conclude this in Theorem 3.10.
By unfolding definitions, we must show that given a $W'_k \sqsupseteq_k \lfloor \mathbf{W}' \rfloor_k$ and $v \in \mathcal{V}[\![\tau_1]\!]_i(\lfloor W'_k \rfloor_i)$, we have $\gamma(e)[v/x] \in \mathcal{E}[\![\tau_2]\!]_j(\lceil W'_k \rceil)$.
This follows by induction, if we show the following:
(1) $\gamma[x \mapsto v](e) = \gamma(e)[v/x]$
(2) $\gamma \in \mathcal{G}[\![\Gamma]\!](\lceil W'_k \rceil)$
(3) $v \in \mathcal{V}[\![\tau]\!]_i(\lfloor \lceil W'_k \rceil \rfloor_i)$
We show (1) by Theorem 3.11, which follows by the definition of the substitution γ.
We show (2) by Theorem 3.5 which states that, if $\gamma \in \mathcal{G}[\![\Gamma]\!](\mathbf{W})$ and $\mathbf{W}' \sqsupseteq \mathbf{W}$, then $\gamma \in \mathcal{G}[\![\Gamma]\!](\mathbf{W}')$, which is a property often referred to as *world monotonicity*. However, at this point we have to show a bit more, since we only have that $\mathbf{W}' \sqsupseteq \mathbf{W}$ and $W'_k \sqsupseteq_k \lfloor \mathbf{W}' \rfloor_k$. In this case, we must show $\lceil W'_k \rceil \sqsupseteq \mathbf{W}$ from these facts, which we do in Theorem 3.10. Notice that this does not hold in general, but because we have $k \geq \textbf{max-level}(\Gamma, x : \tau_1, \tau_2)$ from our typing rule, we can show it holds in this case for these particular worlds. We discuss this property further in the proof of Theorem 3.10.
Finally, we can show (3) by Theorem 3.8, which states that for $k \geq i$, $\lfloor \lceil W'_k \rceil \rfloor_i = \lfloor W'_k \rfloor_i$.
Together with (2) and (3), we conclude with $\gamma[x \mapsto v] \in \mathcal{G}[\![\Gamma]\!](\lceil W'_k \rceil)$.

Case: [New] We have

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{new}\ e : \textbf{Ref}\ \tau}$$

and $\textbf{Ref}\ \tau :: \textbf{Type}_{i+1}$ for some i.
The key insight in this case is that we do not require world monotonicity, as is required for general references. This is because the only future world we consider is also in a higher universe, and lowering discards the new allocation, so the case follows from the induction hypothesis.
From induction and unfolding of definitions, we conclude that **new** $\gamma(e)$ steps to a value, in particular, a fresh location.

$$\frac{\langle h' \mid \gamma(e) \rangle \longrightarrow^* \langle h'' \mid v \rangle}{\langle h' \mid \textbf{new}\ \gamma(e) \rangle \longrightarrow^* \langle h''[l \mapsto v] \mid l \rangle}$$

All that is left to show is $\exists \mathbf{W_f} \sqsupseteq \mathbf{W}' . \mathbf{W_f} \Vdash h''[l \mapsto v] \wedge l \in \mathcal{V}[\![\textbf{Ref}\ \tau]\!]_{i+1}(\lfloor \mathbf{W_f} \rfloor_{i+1})$.
From induction, we have a particular $\mathbf{W}'' \sqsupseteq \mathbf{W}'$ such that $\mathbf{W}'' \Vdash h''$. We need $\mathbf{W_f} \sqsupseteq \mathbf{W}'$ such that $\mathbf{W_f} \Vdash h''[l \mapsto v]$, which means we need some $\mathbf{W_f}$ that maps location l to $(i, \mathcal{V}[\![\tau]\!]_i)$. We can extend the world $\mathbf{W}''$ with l to $(i, \mathcal{V}[\![\tau]\!]_i)$, which we represent as $\mathbf{W}''[l \mapsto (i, \mathcal{V}[\![\tau]\!]_i)]$. We must then show that $\mathbf{W}''[l \mapsto (i, \mathcal{V}[\![\tau]\!]_i)] \Vdash h''[l \mapsto v]$. We know that $\mathbf{W}'' \Vdash h''$, so all that is left to show is that $v \in \mathbf{W}''[l \mapsto (i, \mathcal{V}[\![\tau]\!]_i)](l)(\lfloor \mathbf{W}''[l \mapsto (i, \mathcal{V}[\![\tau]\!]_i)] \rfloor_i)$, which is to say $v \in \mathcal{V}[\![\tau]\!]_i(\lfloor \mathbf{W}''[l \mapsto (i, \mathcal{V}[\![\tau]\!]_i)] \rfloor_i)$.
By the lowering operation to level i, we also guarantee that v does not depend on l since $\lfloor \mathbf{W}''[l \mapsto (i, \mathcal{V}[\![\tau]\!]_i)] \rfloor_i = \lfloor \mathbf{W}'' \rfloor_i$, so we can conclude $v \in \mathcal{V}[\![\tau]\!]_i(\lfloor \mathbf{W}''[l \mapsto (i, \mathcal{V}[\![\tau]\!]_i)] \rfloor_i)$.

Finally, we must show $l \in \mathcal{V}[\![\textbf{Ref } \tau]\!]_{i+1}(\lfloor \mathbf{W}''[l \mapsto (i, \mathcal{V}[\![\tau]\!]_i)] \rfloor_{i+1})$, that is,

$$l \in \{l \mid \lfloor \mathbf{W}''[l \mapsto (i, \mathcal{V}[\![\tau]\!]_i)] \rfloor_{i+1}(l) = (i, \mathcal{V}[\![\tau]\!]_i)\}$$

Since the world is being lowered to $i+1$, the mapping $l \mapsto (i, \mathcal{V}[\![\tau]\!]_i)$ remains in the lowered world, and l maps to $(i, \mathcal{V}[\![\tau]\!]_i)$ as required. □

Corollary 3.4. *If* $\vdash e : \tau$ *and* $\tau :: \textbf{Type}_i$, *then* $\exists h. \langle \cdot \mid e \rangle \longrightarrow^* \langle h \mid v \rangle$, *that is, we encounter no type errors during evaluation.*

We now present the key lemmas, all of which follow a standard structure, but include small modifications to adjust for the changes in the world structure, in particular for leveled worlds.

World monotonicity is a standard lemma needed for possible worlds semantics, also called *closed under state extension* by Ahmed [5]. World monotonicity usually states that, given a value v in the value relation at world $\mathbf{W}$, then for all $\mathbf{W}' \sqsupseteq \mathbf{W}$, v is also in the value relation at world $\mathbf{W}'$. From an operational standpoint, world monotonicity is a desirable property: a value remains the same regardless of what additional locations may be allocated on the heap. As a corollary, we extend world monotonicity from values to contexts. World monotonicity already holds for the expression relation, because the notion of a possible future world $\mathbf{W}'$ is already built into the relation.

Lemma 3.5 (World Monotonicity). $\forall \mathbf{W}' \sqsupseteq \mathbf{W}$, *if* $v \in \mathcal{V}[\![\tau]\!]_i(\mathbf{W})$ *then* $v \in \mathcal{V}[\![\tau]\!]_i(\mathbf{W}')$.

Proof. We show the key cases of the proof for values v by cases on τ.

Case: $\tau = \textbf{Ref } \tau$

Given $v = l \in \mathcal{V}[\![\textbf{Ref } \tau]\!]_{i+1}(\lfloor \mathbf{W} \rfloor_{i+1})$ we must show $l \in \mathcal{V}[\![\textbf{Ref } \tau]\!]_{i+1}(\lfloor \mathbf{W}' \rfloor_{i+1})$, that is, $\lfloor \mathbf{W}' \rfloor_{i+1}(l) = (i, \mathcal{V}[\![\tau]\!]_i)$. Since $\mathbf{W}' \sqsupseteq \mathbf{W}$, we have that $\lfloor \mathbf{W} \rfloor_{i+1} \sqsupseteq_{i+1} \lfloor \mathbf{W}' \rfloor_{i+1}$ by Theorem 3.7, which means that $\forall l \in \lfloor \mathbf{W} \rfloor_{i+1}. \lfloor \mathbf{W} \rfloor_{i+1}(l) = \lfloor \mathbf{W}' \rfloor_{i+1}(l)$, in which case we can conclude, as $\lfloor \mathbf{W} \rfloor_{i+1}(l) = \lfloor \mathbf{W}' \rfloor_{i+1}(l) = (i, \mathcal{V}[\![\tau]\!]_i)$.

Case: $\tau = \tau_1 \xrightarrow{\mathbf{k}} \tau_1$

Given $v \in \mathcal{V}[\![\tau_1 \xrightarrow{k} \tau_2]\!]_k(\lfloor \mathbf{W} \rfloor_k)$ we must show $v \in \mathcal{V}[\![\tau_1 \xrightarrow{k} \tau_2]\!]_k(\lfloor \mathbf{W}' \rfloor_k)$, that is, $\forall W'_k \sqsupseteq_k \lfloor \mathbf{W}' \rfloor_k, v \in \mathcal{V}[\![\tau_1]\!]_i(\lfloor W'_k \rfloor_i).\gamma(e)[v/x] \in \mathcal{E}[\![\tau_2]\!]_j(\lceil W'_k \rceil)$

Since $v \in \mathcal{V}[\![\tau_1 \xrightarrow{k} \tau_2]\!]_k(\lfloor \mathbf{W} \rfloor_k)$, we already know that this property holds for $W'_k \sqsupseteq_k \lfloor \mathbf{W} \rfloor_k$. Since $\lfloor \mathbf{W}' \rfloor_k \sqsupseteq_k \lfloor \mathbf{W} \rfloor_k$, then any $W'_k \sqsupseteq_k \lfloor \mathbf{W}' \rfloor_k$ will also be larger than $\lfloor \mathbf{W} \rfloor_k$, that is, $W'_k \sqsupseteq_k \lfloor \mathbf{W} \rfloor_k$. Then this property holds for all $W'_k \sqsupseteq_k \lfloor \mathbf{W}' \rfloor_k$, and we have $v \in \mathcal{V}[\![\tau_1 \xrightarrow{k} \tau_2]\!]_k(\lfloor \mathbf{W}' \rfloor_k)$ □

Corollary 3.6. $\forall \mathbf{W}' \sqsupseteq \mathbf{W}$, *if* $\gamma \in \mathcal{G}[\![\Gamma]\!](\mathbf{W})$ *then* $\gamma \in \mathcal{G}[\![\Gamma]\!](\mathbf{W}')$.

World monotonicity for reference types relies on the fact that two related general worlds $\mathbf{W}' \sqsupseteq \mathbf{W}$ will remain related when lowered, which follows easily by the definitions of world extension. We also need to show that lowering a particular world at level k to level i is the same as lifting it to a general world, then lowering it to i. The proofs of these lemmas are easily shown by definition.

Lemma 3.7. *If* $\mathbf{W}' \sqsupseteq \mathbf{W}$, *then for any* i, $\lfloor \mathbf{W}' \rfloor_\mathbf{i} \sqsupseteq_i \lfloor \mathbf{W} \rfloor_\mathbf{i}$

Lemma 3.8. *For* $k \geq i$, $\lfloor \lceil W'_k \rceil \rfloor_i = \lfloor W'_k \rfloor_i$.

We also have that lowering preserves leveled world extension.

Lemma 3.9. *If* $W'_k \sqsupseteq_k W_k$ *and* $i < k$, *then* $\lfloor W'_k \rfloor_i \sqsupseteq_i \lfloor W_k \rfloor_i$.

Proof. Unrolling definitions, we have $W'_k \sqsupseteq_k W_k \iff \mathrm{dom}(W_k) \subseteq \mathrm{dom}(W'_k) \wedge \forall l \in W_k.W'_k(l) = W_k(l)$. We must show $\lfloor W'_k \rfloor_i \sqsupseteq_k \lfloor W_k \rfloor_i \iff \mathrm{dom}(\lfloor W'_k \rfloor_i) \subseteq \mathrm{dom}(\lfloor W'_k \rfloor_i) \wedge \forall l \in \lfloor W_k \rfloor_i.\lfloor W'_k \rfloor_i(l) = \lfloor W_k \rfloor_i(l)$. We have that $\lfloor W_k \rfloor_i = \{(\ell, j, R) \mid j < i \wedge (\ell, j, R) \in W_k\}$ and $\lfloor W'_k \rfloor_i = \{(\ell, j, R) \mid j < i \wedge (\ell, j, R) \in W'_k\}$. For all $\ell \in \mathrm{dom}(\lfloor W_k \rfloor_i)$, we know $\ell \in \mathrm{dom}(W_k)$ by definition of lowering, and $\ell \in \mathrm{dom}(W'_k)$ since $\mathrm{dom}(W_k) \subseteq \mathrm{dom}(W'_k)$. We need to show $\ell \in \mathrm{dom}(\lfloor W'_k \rfloor_i)$, and by definition of lowering, since $(\ell, j, R) \in \lfloor W_k \rfloor_i$, the same mapping $(\ell, j, R) \in \lfloor W'_k \rfloor_i$ since $j < i$, and so we conclude $\ell \in \mathrm{dom}(\lfloor W'_k \rfloor_i)$. A similar argument holds for checking all the mappings: $\forall l \in \lfloor W_k \rfloor_i.\lfloor W'_k \rfloor_i(l) = \lfloor W_k \rfloor_i(l)$.x □

When proving the function case of the fundamental lemma, we have a particular world at level k that we must use for an interpretation of a particular substitution γ. However, we have that this substitution γ is defined over a *general* world $\mathbf{W}$. We must show that it is safe to use the particular world at k, because the interpretation of this γ will never need levels higher than k. We know this because the annotation on the function type *comes from* the highest level in Γ.

Lemma 3.10. *Given* $\mathbf{W}' \sqsupseteq \mathbf{W}$, $W'_k \sqsupseteq_k \lfloor \mathbf{W}' \rfloor_k$, *and* $\gamma \in \mathcal{G}[\![\Gamma]\!](\mathbf{W})$ *with* $k \geq \mathbf{max\text{-}level}(\Gamma)$, *we have* $\gamma \in \mathcal{G}[\![\Gamma]\!](\lceil W'_k \rceil)$.

Proof. Since $\gamma \in \mathcal{G}[\![\Gamma]\!](\mathbf{W})$, we have that $\forall v \in \gamma.v \in \mathcal{V}[\![\tau]\!]_i(\lfloor \mathbf{W} \rfloor_i)$. By Theorem 3.5, we have that $\forall v \in \gamma.v \in \mathcal{V}[\![\tau]\!]_i(\lfloor \mathbf{W}' \rfloor_i)$ since $\mathbf{W}' \sqsupseteq \mathbf{W}$. Finally, since $W'_k \sqsupseteq_k \lfloor \mathbf{W}' \rfloor_k \sqsupseteq_k \lfloor \mathbf{W} \rfloor_k$ by Theorem 3.7, we can have that $\forall v \in \gamma.v \in \mathcal{V}[\![\tau]\!]_i(\lfloor \lceil W'_k \rceil \rfloor_i)$ because we know that $k \geq \mathbf{max\text{-}level}(\Gamma)$, we never need any locations mapped to types higher than k for the interpretations of any v. Because of the relationship $\sqsupseteq$ between W'_k and $\lfloor \mathbf{W}' \rfloor_k$, we have all the locations we do need at lower levels. So each v also exists in the interpretation with world $\lceil W'_k \rceil$ because v does not depend on any level higher than k. This gives $\gamma \in \mathcal{G}[\![\Gamma]\!](\lceil W'_k \rceil)$. □

Finally, we have that extending a substitution γ is the same as substituting into an expression e.

Lemma 3.11. *Given syntactically well formed* e, *closed value* v, *and a substitution* γ *mapping variables to values, we have* $\gamma[x \mapsto v](e) = \gamma(e)[v/x]$

4 $\lambda^{\circ}_{\mathbf{PR}}$: Predicative and Cyclic Full-Ground References

We change our type hierarchy to create a language $\lambda^{\circ}_{\mathrm{PR}}$ with *cyclic* full-ground references by allowing values in references that quantify over future allocations. These references enable creating mutable cyclic data structures like mutable linked lists, which were not possible in λ_{PR} with *acyclic* full-ground references. In λ_{PR}, a location ℓ' could store another location ℓ, but the predicative type hierarchy prevents ℓ from storing ℓ' and creating a cycle. The predicative hierarchy restricts ℓ, the new allocation, to be in a different universe from the past allocation ℓ'. We can create cyclic data structures by instead allowing references to be in the *same* universe as their stored type, *i.e.,* allow references to refer to future allocations, namely itself.

To combine cyclic full-ground references with stratified higher-order references, we take inspiration from *impredicative* type universe hierarchies. An impredicative type universe hierarchy allows one universe of types to impredicatively quantify over itself and higher universes, *i.e.,* the type $\forall A :: \mathbf{Type}_0.\mathbf{Type}_0$ remains in the impredicative universe $\mathbf{Type}_0$. The rest of the type universe hierarchy follows the usual predicative stratification. We use a similar notion in our semantics for full-ground references, where we create a single type universe level for cyclic full-ground references, but require the stratification of higher-order references.

This extension shows how stratified higher-order references can be integrated into other full-ground semantics; in particular, we closely follow the language presented by Kammar et al. [25]. The semantics developed by Kammar et al. [25] are categorical semantics for full-ground references, and

$$
\begin{array}{llll}
\text{Cell Sort} & c & \in & \mathsf{S} \\
\text{Type} & \tau & ::= & \ldots \mid \mathbf{Ref}\ \tau \mid \mathbf{Ref}\ c \mid \tau \times \tau \mid \tau + \tau \\
\text{Expr} & e & ::= & \ldots \mid \langle e, e \rangle \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e \mid \mathbf{inl}\ e \mid \mathbf{inr}\ e \mid \mathbf{match}\ e\ \{\ \mathbf{inl}\ x \mapsto e \mid \mathbf{inr}\ x \mapsto e\ \} \\
 & & \mid & \mathbf{letref}\ (x_1 : \mathbf{Ref}\ c_1) \coloneqq v_1\ \ldots\ (x_n : \mathbf{Ref}\ c_n) \coloneqq v_n\ \mathbf{in}\ e \\
\text{Value} & v & ::= & \ldots \mid \langle v, v \rangle \mid \mathbf{inl}\ v \mid \mathbf{inr}\ v
\end{array}
$$

Fig. 9. $\lambda^{\circ}_{\mathrm{PR}}$ syntax

they mention that extending to higher-order references would introduce more complexity because of the type-world circularity. We conjecture that our extension shows how predicative higher-order references could be integrated into their categorical model without excessively complicating their semantics.

Extending to cyclic full-ground references involves three parts: adding more data structures (sums and pairs) for expressing cyclic data, enabling the definition of cyclic references using a **letref** form and a *signature*, and tweaks to the type universe hierarchy to create a universe for cyclic full-ground types. Most of the extensions in the following figures involve adding sums and pairs to the language.

One major change is we parameterize the language over a signature Σ, following Kammar et al. [25]. A signature can be considered as list of top level data declarations, or sometimes called *type ascriptions*[3] [35], where a type name is defined in terms of other types. We restrict these type definitions to full-ground types in our case, which in turn restricts cyclic type references to full-ground types. A signature consists of a *countable set* of cell sorts $c \in \mathsf{S}$ and we describe the set of full-ground types $\mathcal{G}$ using these cell sorts c:

$$\mathcal{G} ::= \mathbf{Nat} \mid \mathbf{Unit} \mid \mathcal{G} \times \mathcal{G} \mid \mathcal{G} + \mathcal{G} \mid \mathbf{Ref}\ c$$

The second component of a signature is a function *ctype* S to $\mathcal{G}$ assigning cell sorts to their full-ground type definition.

As an example, we define a mutable (possibly) cyclic list by defining the following cell sorts.

$$S = \{\mathbf{linked\text{-}list}, \mathbf{list\text{-}cell}\}$$

And define $\mathit{ctype} : \mathsf{S} \rightarrow \mathcal{G}$ as:

$$\mathit{ctype}\ (\mathbf{linked\text{-}list}) = \mathbf{Unit} + \mathbf{Ref}\ (\mathbf{list\text{-}cell})$$
$$\mathit{ctype}\ (\mathbf{list\text{-}cell}) = \mathbf{Nat} \times \mathbf{Ref}\ (\mathbf{linked\text{-}list})$$

A signature gives our language the ability to describe cyclic data systematically that might otherwise require inductive or recursive types. We use a signature to avoid adding additional unnecessary machinery in the semantics to deal with such types. In particular, recursive types can require a different semantic interpretation: interpreting types into complete partial orders, or step-indexing [7]. But the kind of cyclic types we want to create are simple enough that a signature can define these structures without excessively complicating the semantics.

We present the extended syntax of $\lambda^{\circ}_{\mathrm{PR}}$ compared to λ_{PR} in Figure 9, assuming a fixed signature with cell sorts S and function *ctype*. We extend the types τ with an additional full-ground reference type **Ref** c which is a reference type referring to cell sorts c. To allow the parallel definition of cyclic references, we extend the expression syntax with a corresponding **letref** syntax [25, 29]. The definition of cyclic references is restricted to values and full-ground reference types. Each v_i has access to any other x_i in the definition block. We extend the language $\lambda^{\circ}_{\mathrm{PR}}$ with product $(\tau \times \tau)$ and sum $(\tau + \tau)$ types and include their corresponding introduction and elimination forms. We introduce products with the pair notation $\langle e, e \rangle$, and introduce sums with left and right injections **inl** e and **inr** e. We eliminate products with projections **fst** e and **snd** e, and eliminate sums with pattern matching $\mathbf{match}\ e\ \{\ \mathbf{inl}\ x \mapsto e \mid \mathbf{inr}\ x \mapsto e\ \}$.

[3]Usually, type ascription refers to ascribing a type to an expression e, *i.e.*, e **as** τ.

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

$$\ldots \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad k \geq \textbf{max-level}(\Gamma, \tau_1, \tau_2, 1)}{\Gamma \vdash \boldsymbol{\lambda}\, x : \tau_1 . e : \tau_1 \xrightarrow{k} \tau_2} \qquad \frac{\Gamma \vdash e : \mathit{ctype}\ c}{\Gamma \vdash \textbf{new}\ e : \textbf{Ref}\ c} \qquad \frac{\Gamma \vdash e : \textbf{Ref}\ c}{\Gamma \vdash\, !\, e : \mathit{ctype}\ c}$$

$$\frac{\Gamma \vdash e_1 : \textbf{Ref}\ c \qquad \Gamma \vdash e_2 : \mathit{ctype}\ c}{\Gamma \vdash e_1 \coloneqq e_2 : \textbf{Unit}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \textbf{inl}\ e_1 : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{inr}\ e_2 : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \textbf{match}\ e\ \{\ \textbf{inl}\ x \mapsto e_1 \mid \textbf{inr}\ x \mapsto e_2\ \}\ : \tau}$$

$$\frac{\begin{array}{c}\Gamma, x_1 : \textbf{Ref}\ c_1, \ldots, x_n : \textbf{Ref}\ c_n \vdash v_1 : \mathit{ctype}\ c_1 \cdots \\ \Gamma, x_1 : \textbf{Ref}\ c_1, \ldots, x_n : \textbf{Ref}\ c_n \vdash v_n : \mathit{ctype}\ c_n \qquad \Gamma, x_1 : \textbf{Ref}\ c_1, \ldots, x_n : \textbf{Ref}\ c_n \vdash e : \tau\end{array}}{\Gamma \vdash \textbf{letref}\ (x_1 : \textbf{Ref}\ c_1) \coloneqq v_1\ \ldots\ (x_n : \textbf{Ref}\ c_n) \coloneqq v_n\ \textbf{in}\ e : \tau}$$

Fig. 10. $\lambda^{\circ}_{\mathrm{PR}}$ typing.

$\boxed{\tau :: \textbf{Type}_i}$

$$\frac{}{\textbf{Nat} :: \textbf{Type}_0} \qquad \frac{\tau_1 :: \textbf{Type}_i \qquad \tau_2 :: \textbf{Type}_j \qquad k \geq \textbf{max-level}(\tau_1, \tau_2)}{\tau_1 \times \tau_2 :: \textbf{Type}_k} \qquad \frac{\mathit{ctype}\ c :: \textbf{Type}_0}{\textbf{Ref}\ c :: \textbf{Type}_0}$$

$$\frac{\tau :: \textbf{Type}_i \qquad i > 0}{\textbf{Ref}\ \tau :: \textbf{Type}_{i+1}} \qquad \frac{}{\textbf{Unit} :: \textbf{Type}_0} \qquad \frac{\tau_1 :: \textbf{Type}_i \qquad \tau_2 :: \textbf{Type}_j \qquad k \geq \textbf{max-level}(\tau_1, \tau_2)}{\tau_1 + \tau_2 :: \textbf{Type}_k}$$

$$\frac{\tau_1 :: \textbf{Type}_i \qquad \tau_2 :: \textbf{Type}_j \qquad k \geq \textbf{max-level}(\tau_1, \tau_2, 1)}{\tau_1 \xrightarrow{k} \tau_2 :: \textbf{Type}_k}$$

Fig. 11. $\lambda^{\circ}_{\mathrm{PR}}$ kinding.

We present the extended typing rules of $\lambda^{\circ}_{\mathrm{PR}}$ compared to λ_{PR} in Figure 10. The typing rules for sums and products are standard. The typing rules for allocating, referencing, and updating full-ground references defined with respect to a cell sort c requires the *ctype* function. The typing rule for parallel definition of cyclic references **letref** checks each v_i has type *ctype* c_i under a environment extended with each definition x_i, and checks e has type τ under the same extended environment.

We present the full type hierarchy of $\lambda^{\circ}_{\mathrm{PR}}$ in Figure 11, which now includes our impredicative universe $\textbf{Type}_0$. The type hierarchy stratifies reference types over types with level $i > 0$; otherwise, reference types storing types in $\textbf{Type}_0$ remain in $\textbf{Type}_0$. To stratify function types away from cyclic full-ground reference types, we classify function types starting at $\textbf{Type}_1$, as $\textbf{Type}_0$ is reserved for cyclic full-ground reference types. A function type that closes over no references $\boldsymbol{\lambda}\ x : \textbf{Nat}.x$ has type $\textbf{Nat} \xrightarrow{1} \textbf{Nat}$ at $\textbf{Type}_1$, and the stratification of function types then follows as before. Finally, pair and sum types are in the universe greater than or equal to the maximum of their elements. Like functions, pairs and sums cannot mutate their contents, but they can contain contents that are mutable. If our pairs and sums were themselves mutable, and had to maintain the stratification invariant, their type level would be one level higher than the contents, like stratified reference types.

We present the extended operational semantics for $\lambda^{\circ}_{\mathrm{PR}}$ compared to λ_{PR} in Figure 12. The rules for products and sums are standard. The only nonstandard rule is for the **letref** form where cyclic

$$\text{Heap} \quad h \quad ::= \quad \cdot \mid h[\ell \mapsto v]$$

$$\boxed{\langle h \mid e\rangle \longrightarrow \langle h \mid e\rangle}$$

$$\begin{array}{rcl}
 & \dots & \\
\langle h \mid \mathbf{fst}\ \langle v_1, v_2\rangle\rangle & \longrightarrow & \langle h \mid v_1\rangle \\
\langle h \mid \mathbf{snd}\ \langle v_1, v_2\rangle\rangle & \longrightarrow & \langle h \mid v_2\rangle \\
\langle h \mid \mathbf{match\ inl}\ v\ \{\ \mathbf{inl}\ x \mapsto e_1 \mid \mathbf{inr}\ x \mapsto e_2\ \}\ \rangle & \longrightarrow & \langle h \mid e_1[x/v]\rangle \\
\langle h \mid \mathbf{match\ inr}\ v\ \{\ \mathbf{inl}\ x \mapsto e_1 \mid \mathbf{inr}\ x \mapsto e_2\ \}\ \rangle & \longrightarrow & \langle h \mid e_2[x/v]\rangle \\
\langle h \mid \ \mathbf{letref}\ \ (x_1 : \mathbf{Ref}\ c_1) \coloneqq v_1 \quad \rangle & \longrightarrow & \langle h[\ell_i \mapsto sv_i]_1^n \mid e[x_i/l_i]_1^n\rangle \\
 \vdots & & \ell_1, \dots, \ell_n \text{ fresh in } h, sv_i = v_i[x_i/l_i]_1^n \\
(x_n : \mathbf{Ref}\ c_n) \coloneqq v_n & & \\
\mathbf{in}\ e & &
\end{array}$$

Fig. 12. $\lambda_{\mathrm{PR}}^{\circ}$ operational semantics.

$$\begin{array}{lcl}
\dots & & \\
\mathcal{V}[\![\mathbf{Ref}\ c]\!]_0(W_0) & \stackrel{def}{=} & \{l \mid W_0(l) = c\} \\
\mathcal{V}[\![\mathbf{Ref}\ \tau]\!]_{i+1}(W_{i+1}) & \stackrel{def}{=} & \{l \mid W_{i+1}(l) = (i, \mathcal{V}[\![\tau]\!]_i)\} \\
\mathcal{V}[\![\tau^i \times \sigma^j]\!]_k(W_k) & \stackrel{def}{=} & \{\langle v_i, v_j\rangle \mid v_i \in \mathcal{V}[\![\tau]\!]_i(\lfloor W_k\rfloor_i) \wedge v_j \in \mathcal{V}[\![\sigma]\!]_j(\lfloor W_k\rfloor_j)\} \\
\mathcal{V}[\![\tau^i + \sigma^j]\!]_k(W_k) & \stackrel{def}{=} & \{\mathbf{inl}\ v_i \mid v_i \in \mathcal{V}[\![\tau]\!]_i(\lfloor W_k\rfloor_i)\} \cup \{\mathbf{inr}\ v_j \mid v_j \in \mathcal{V}[\![\sigma]\!]_j(\lfloor W_k\rfloor_j)\}
\end{array}$$

$$\begin{array}{llcl}
\textbf{Lowering to 0} & \lfloor \_ \rfloor_0 & : & World_k \longrightarrow World_0 \\
 & \lfloor W_k\rfloor_0 & = & \{(l, c) \mid c \in \mathcal{S} \wedge (l, c) \in W_k\}
\end{array}$$

Fig. 13. Value relation for $\lambda_{\mathrm{PR}}^{\circ}$.

allocation occurs by allocating fresh locations $\ell_1, \dots, \ell_n$ assigned to values $sv_1, \dots, sv_n$, written as $[\ell_i \mapsto sv_i]_1^n$. Each sv_i corresponds to $v_i[x_i/l_i]_1^n$, which is $v_i[x_1, \dots, x_n/l_1, \dots, l_n]$ for each v_i. The whole **letref** block steps to the body e with each x_i substituted with location l_i for $1, \dots, n$, written as $e[x_i/l_i]_1^n$.

4.1 The Logical Relation

Our world definitions change with the type universe hierarchy. The stratified definitions of worlds for references stay the same as before, starting from $\mathbf{Type}_1$. However, with full-ground references in universe $\mathbf{Type}_0$, we change the definition of $World_0$. In λ_{PR} we used the empty map, but in $\lambda_{\mathrm{PR}}^{\circ}$ we instead have the following definitions.

$$\begin{array}{rcl}
World_{k+1} & = & Loc \xrightarrow{\text{fin}} (i : \mathrm{Fin}(k+1)) \times Type_i \\
World_0 & = & Loc \xrightarrow{\text{fin}} CellSort
\end{array}$$

As discussed in Section 2, we map locations to their cell sorts (types) rather than to semantic values. This is because the semantic values of full-ground references are still restricted enough to behave the same in *all future worlds*, unlike functions, which capture past worlds and could behave differently in future worlds. Instead of mapping locations to semantic values at $World_0$, we avoid a circularity by mapping to the cell sort, and later interpreting the cell sort with a future world. Because the behavior of the full-ground reference can be determined in any future world, all we need is the cell sort to determine the behavior later, *i.e.,* at points new allocations have been performed.

Because of the change in $World_0$, we interpret our type **Ref** c differently from **Ref** τ in the value relation shown in Figure 13. The semantic values of **Ref** c are locations mapping to the cell sort c, in contrast to a reference type **Ref** τ, where locations are mapped to the *semantic values* in the relation $\mathcal{V}[\![\tau]\!]_i$. We also have additional types $\tau^i \times \sigma^j$ and $\tau^i + \sigma^j$ added to the value relation. These are standard definitions except for the universe annotations, where each component type τ^i and σ^j are annotated with their universe level to guarantee correct indexing into the relation. All other definitions stay the same, except for lowering, which must account for lowering to World_0 where locations map to cell sorts c. Heaps realizing worlds must also account for locations that map to c, and ensure values are in the interpretation of $\mathit{ctype}\ c$, that is, $h(l) \in \mathcal{V}[\![\mathit{ctype}\ c]\!](\lfloor \mathbf{W} \rfloor_0)$ when $l \mapsto c \in \mathbf{W}$.

To see how cyclic data does not result in a cycle in the semantics, we show how to semantically type a linked list. Consider the following cyclic list, with the same $S = \{\textbf{linked-list}, \textbf{list-cell}\}$ and $\mathit{ctype} : \mathrm{S} \longrightarrow \mathcal{G}$ defined previously.

$$\begin{array}{lll} \textbf{letref} \ (\texttt{cyclic-list} : \textbf{Ref}\ (\textbf{linked-list})) & := & \textbf{inr}\ \texttt{head} \\ \qquad\quad (\texttt{head} : \textbf{Ref}\ (\textbf{list-cell})) & := & \langle 1, \texttt{cyclic-list} \rangle \\ \ \textbf{in}\ \texttt{cyclic-list} & & \end{array}$$

With these cyclic allocations, we have a (non-cyclic) world $W_0 = \{(\ell_c, \textbf{linked-list}), (\ell_h, \textbf{list-cell})\}$, where ℓ_c is the fresh location associated with variable `cyclic-list` and ℓ_h is the fresh location associated with the variable `head`. We show how the final (cyclic) heap $h = \{\ell_c \mapsto \textbf{inr}\ \ell_h, \ell_h \mapsto \langle 1, \ell_c \rangle\}$ resulting from the expression realizes W_0.

$$\begin{array}{rcl} \ell_c \mapsto \textbf{inr}\ \ell_h & \in & \mathcal{V}[\![\mathit{ctype}\ (\textbf{linked-list})]\!](W_0) \\ & \iff & \ell_h \in \mathcal{V}[\![\textbf{Ref}\ (\textbf{list-cell})]\!](W_0) \\ & \iff & (\ell_h, \textbf{list-cell}) \in W_0 \end{array}$$

And similarly:

$$\begin{array}{rcl} \ell_h \mapsto \langle 1, \ell_c \rangle & \in & \mathcal{V}[\![\mathit{ctype}\ (\textbf{list-cell})]\!](W_0) \\ & \iff & \ell_c \in \mathcal{V}[\![\textbf{Ref}\ (\textbf{linked-list})]\!](W_0) \\ & \iff & (\ell_c, \textbf{linked-list}) \in W_0 \end{array}$$

By restricting World_0 to store cell sorts, we have cut off any potential circularity in the semantic model, but can still ensure that values in the heap are of the expected semantic type.

Intuitively, $\lambda^{\circ}_{\mathrm{PR}}$ is still terminating because there is no mechanism for infinitely dereferencing one of these cyclic structures. We could extend the language further and maintain termination with guarded recursive functions, or extend the language with recursion explicitly but still disallow recursion through the heap using predicative references.

The current model and language only allow cycles in $\textbf{Type}_0$; we conjecture that another possible extension includes cycles that occur at higher levels, as long as the cycle stay at the same level. $\textbf{Type}_i$ cycles would stay in $\textbf{Type}_i$, but could not extend to higher levels, as dependencies on higher levels (future allocations) results in general references. The key is to allow cyclic data, but distinguish cyclic computation into higher levels by bumping the universe level whenever a function closes over a particular level. This means the interpretation of cycles in $\textbf{Type}_i$ to use a World_i that does not include functions closing over $\textbf{Type}_i$, avoiding circularities.

To extend $\lambda^{\circ}_{\mathrm{PR}}$ with (non-guarded) recursive functions and give up termination, we need to consider the level of recursive functions like $\textbf{rec} f(x) = \textbf{new}\ (f\ \ x)$. The kind of the function type would need some sort of polymorphism or recursive type, since the codomain type requires $\alpha = \textbf{Ref}\ \ \alpha$. Kinding such recursive functions is not currently possible with our predicative references system, since the level k of the recursive function would need to be determined in terms of the reference kinding level $k + 1$, and $k \neq k + 1$. We discuss the possible extension and expressivity of polymorphic universes in Section 5.

5 Related and Future Work

Stratified Region Type-and-Effect Systems. Our work resembles region type-and-effect systems [40], a type system equipped with an effect system that records reads and writes to regions of the heaps. In particular, our work resembles *stratified* region type-and-effect systems.

Boudol [14] (and improved by Amadio [6] and Boudol [15]) showed that the effects in a region type-and-effect system can be restricted to programs that respect a heap with *stratified* regions. The stratification is enforced by *constraints* over regions to ensure a reference cannot contain a value from its own region. Region type-and-effect systems annotate a function type with the regions the function accesses, and in the stratified case, the typical non-terminating program Landin's Knot [26] is not well typed. Consider the following attempt at Landin's Knot. The function f accesses a region σ where a reference r is allocated. Trying to update r with f fails to type check since the effect system requires that the value put into region σ cannot contain region σ.

$$\begin{array}{ll}
id = (\boldsymbol{\lambda}\ x.x) & : \mathbf{Nat} \xrightarrow{\emptyset} \mathbf{Nat} \\
r = \mathbf{new}\ id & : \mathbf{Ref}_{\ \sigma}(\mathbf{Nat} \xrightarrow{\emptyset} \mathbf{Nat}) \\
f = (\boldsymbol{\lambda}\ x.(!\ r)\ x) & : \mathbf{Nat} \xrightarrow{\{\sigma\}} \mathbf{Nat} \\
r := f & ;\ \text{not well typed since } \sigma \in \{\sigma\}
\end{array}$$

Our semantics differ, although there could be an interpretation of our universe system as an effect system. First, the reasoning principles differ. Region type-and-effect systems use effects to determine allocation and deallocation behaviour of references from reads and writes for static memory management. In contrast, our work designs an abstraction to describe the semantics of a fundamentally different kind of reference, in particular, predicative higher-order references. If we were to interpret our language as a type-and-effect system, the effect would differ. Region type-and-effect systems describe what regions are *accessed*, while our kind system describes where a value is *allocated*. In particular, functions in type-and-effect systems have no effect until they are applied. In contrast, our functions do have an (allocation) effect: they are allocated in a region in the heap that corresponds to the universe of their type.

Tranquilli [41] extends the stratified region type-and-effect system to allow references that obey a positive recursive discipline. They observe that translating effectful programs to "memory passing style" can result in a recursive type equation. A function over region r of type $A \xrightarrow{r} B$ would be translated to memory passing style $A \longrightarrow r \longrightarrow B$, similar to the semantics of functions in possible worlds models (though not explicitly stated in these terms). If the region r contains types $A \xrightarrow{r} B$, then the recursive type equation is akin to $r = A \longrightarrow r \longrightarrow B$. They create a region type-and-effect system that enforces a *positiveness* requirement instead of a stratification requirement, where the regions respect positive/negative occurrences like positive recursive types.

Our type universe hierarchy enforces a stratification in universes rather than a positiveness requirement. We are interested in whether a universe hierarchy could enforce a positiveness requirement rather than a stratification requirement. We conjecture that the positiveness requirement between universes should still avoid the type-world circularity, that is, a semantic value could refer to its own world given that the world occurs in a "positive" position.

Demangeon et al. [20] defines a stratified region type-and-effect system where the stratification is induced by using natural numbers for regions. Their work focuses on providing an alternative termination proof method based on projecting an impure calculus to a purely functional calculus developed in prior work [19]. Their calculus seems the most similar to ours out of all the stratified regions work, given that our presented universe hierarchy is also defined over natural numbers; however, the effects described by our system are different as we elaborated earlier.

Region type-and-effect systems tend to have more fine-grained regions than those described by universes in our system. For example, distinct parts of a data structure can be in different regions, whereas our type hierarchy currently requires an entire data structure to be in one universe together (Section 4). Fine-grained regions reduce memory usage, as unused regions of a data structure can be deallocated separately. Further work is needed to make this connection formal, and we conjecture that such connection may allow transferring results from region type-and-effect systems to our languages, such as using fine-grained *universes* for practical static memory management.

Our type universe hierarchy has a *level algebra* that gives rise to the heap structure, and we conjecture the algebra could be altered to get fine-grained universes. A level algebra is defined by:

- The set of levels $\mathcal{L}$ and operations over these levels.
- The relationship between a reference type universe level l_r and what universe it stores l_s defined by $l_r \mathcal{R} l_s$.
- A function's type universe level l_f, determined by the universe levels of the input l_i, output l_o, and context types l_c, ..., defined by $l_f \ \mathcal{F} \ (l_i, l_o, l_c, ...)$.

An algebra requires additional relationships between reference and function type levels since these are the types of expressions that allocate on the heap. In λ_{PR}, the level algebra is one with $\mathcal{L}$ as the natural numbers with an increment operation, $\mathcal{R}$ requires that a reference universe level is one higher than what it stores $l_s + 1 \mathcal{R} l_s$, and $\mathcal{F}$ is $\geq$. In $\lambda^{\circ}_{\text{PR}}$, the level algebra is one with $\mathcal{L}$ as the natural numbers with an increment operation, $0 \mathcal{R} 0$ and $l_s + 1 \mathcal{R} l_s$ when $l_s > 0$, and $\mathcal{F}$ is $\geq$ with 1.

We conjecture that one could divide the heap into fine-grained universes by using a different level algebra inspired by *universe polymorphism*. Universe polymorphism allows expressions in dependent type theories to be polymorphic with respect to universe levels by replacing explicit natural numbers with a *level variable*. Recent work by Hou [24] influenced by McBride [32]'s *crude-but-effective stratification* has found that using an explicit *displacement* operator is a simple and effective mechanism for universe polymorphism. To $\mathcal{L}$, we could add level variables to represent names for universes, and define $\mathcal{R}$ and $\mathcal{F}$ to work over level variables.

To implement McBride's *crude-but-effective stratification* version of universe polymorphism, the language needs an explicit displacement operator on both expressions and types. A value in one universe of the heap can be "lifted" to another universe using the displacement operator. For example, given a term e of type τ, suppose we would like to allocate the value of e not in universe $\textbf{Type}_0$ (where the current system might dictate), but in some region we call β. We would use displacement $\Uparrow_\beta$, an operator also defined over types that updates the kind level accordingly.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Uparrow_\beta e : \Uparrow_\beta \tau} \qquad \frac{\tau :: \textbf{Type}_\alpha}{\Uparrow_\beta \tau :: \textbf{Type}_{\alpha+\beta}}$$

Models of General References. There are many models of references, some that use a more operational approach by defining a logical relation as we have, and others that take a deeper mathematical or categorical approach. Step-indexed logical relations were introduced by Appel and McAllester [7] and extended by Ahmed [5] to model references. As discussed in Section 2, step-indexing logically approximates the semantic definitions one wants in the presence of the type-world circularity. We focus on how far we can push references before encountering the type-world circularity, which meant taking the approximation equations "literally" to describe a new model and language where the index enriches the types through type universes. Ahmed [5] suggested a stratification in the syntax with a hierarchy of types, but chose the approximation approach instead to account for quantified types. Notably, Ahmed [5] did not study how functions would be allocated in the hierarchy, as we do, and did not observe a distinction between dependencies on past vs future allocations.

We would like to extend our models to a binary logical relation. The binary relation would allow us to prove program equivalences by relating two expressions in the relation. We do not foresee any problems that would prevent this extension; however, the binary case requires more machinery. In particular, extending to the binary case requires defining related heaps [12, 36]. To enable local reasoning, the relation must describe disjoint parts of the heap, so that a freshly allocated location does not interfere with related heaps. These disjoint parts of the heap are more fine-grained than our current universes. Extending the local reasoning to higher-order references, the heap relation must be parameterized by a world, since relating two values in the heap requires the values be related by the relation itself, and a circularity arises again. Ahmed et al. [3] solve this again using step-indexing, but we conjecture our leveled worlds could avoid this circularity.

Ownership Systems. Ownership restricts references by restricting the *usage* of references.

Universe types (UT) [18, 21] are an ownership system enforcing an *ownership-as-modifier* discipline for object-oriented languages. Despite the similar names, UT do not use a type universe hierarchy as seen in our work or in dependent type theories, but rather the types enforce an *ownership hierarchy* by classifying types into different universes. Similar to a kinding system, each type in UT is preceded by a universe modifier indicating the relationship to the current reference. The hierarchy ensures all modifications of an object are done through the object's owner.

Unlike our kinding system, the universe modifier *depends* on the relationship relative to the perspective of the current object, *e.g.*, whether the current object owns another object. For example, given an object o that owns two other objects o_1 and o_2, viewing the object o_2 from the perspective of o_1 results in the universe relationship **peer** because o_1 and o_2 are both owned by the same object o. However, viewing the object o_2 from the perspective of o results in the modifier **rep**, meaning the object o owns o_2. In contrast, our system keeps the universe of a type fixed, and the universe does not change depending on perspective of different locations or objects.

Our type system does not provide the same guarantees that ownership systems do, such as safety in the presence of aliasing. It would be interesting to investigate whether type universe hierarchy might be modified to reason about aliasing, perhaps taking ideas from UT.

Core L3 [4] achieves termination in the presence of higher-order references using linear capabilities, *i.e.*, permissions to modify and read contents of a mutable reference. L3 supports *strong* updates (updates to references can change the type), aliasing, and type safety, and termination is a side effect of restricting capabilities. Capabilities grant an expression exclusive permission to a location on the heap. Although not explicitly addressing the type-world circularity, their model avoids it.

$$\begin{aligned} \mathit{Type} &= \mathcal{P}(\mathit{Store} \times \mathit{Value}) \\ \mathit{Store} &= \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Value} \end{aligned}$$

Since capabilities grant restricted access to the heap, the model pairs up a particular *store* with each value based on its capabilities. Notice there is no longer a circularity in these definitions because the heaps are represented as particular stores.

We could view our type universes as an access restriction to parts of the heap, similar to the capabilities of L3, but the universe does not ensure exclusive access. A value in universe $\mathbf{Type}_i$ can access universes $\mathbf{Type}_{i-1}$ and below, and is duplicable, but access is restricted in the sense the value cannot access $\mathbf{Type}_i$ or above.

Dependent types. The language F⋆ [38] extends a dependent type theory with the ability to specify and verify effectful code. Ahman et al. [2] describe how the specification can be achieved through *Dijkstra monads.* They note that the predicative universe type hierarchy in the dependent type theory disallows storing stateful functions in the heap, because their heap type **heap** is in $\mathbf{Type}_{i+1}$ and any stateful functions (which require a heap) end up being in $\mathbf{Type}_{i+1}$ as well. They note that a stratified heap type should be possible, and we assume such an encoding would be similar to our presentation. The key difference between our work is how universes are used. Our universes describe the stratified heap, while their type universes are used in the usual way to stratify types for consistency, which indirectly stratifies the heap encoded as a type.

Future work could extend our language to a dependent type theory. We imagine two separate universe hierarchies: one for heaps, and one for types. δCPBV, a dependent type theory with effects separating *values* and *computations* (based on call-by-push-value (CBPV) [27]), features a similar separation of type universes, with *value* type universes and *effectful* type universes [37].

Other practical language designs. We focus on the semantics of predicative higher-order references to inform future language designs. Like early work on semantics of linear typing and references (like [4]), these calculi are not yet practical languages, but the semantics of such calculi served as a foundation for practical languages such as Rust. Nevertheless, recent language and type system designs equip languages with low-level memory reasoning similar to our calculus of predicative higher-order references, approaching the same problem from the other direction.

Reachability types, initially presented by Bao et al. [11] and extended by Wei et al. [42], enable reasoning about aliasing, specifically about the *separation* of values. Types are annotated with a *reachability qualifier*, that is, the set of reachable values/locations. Using these sets of reachable values and locations, the type system can statically decide separation of values, and can prove that expressions with disjoint qualifiers can evaluate in parallel. Similar to our type and kind system, the type system annotates function types with reachable locations, which is determined in part by the free variables of the function. Our current model is not concerned with aliasing, and cannot give as strong guarantees about separation as reachability types can. However, we might view our universes as a coarse-grained abstraction of reachability, and perhaps we could extend the system along the lines of reachability types to reason about aliasing.

The logical relations model of reachability types by Bao et al. [10] is similar to our's, but differs in key ways. The model has a restricted form of world extension $\sqsupseteq_L$, parameterized by a set L of reachable locations, similar to our level restricted world extension $\sqsupseteq_k$. The world is only extended in terms of locations L (all other locations are irrelevant). In contrast to our model, the current model of reachability types is not compositional. The model relies on unfolding the expression relation to define the semantics of a function, as the semantics of a function body relies on the reachable locations from a function's argument. Like our model, reachability types ensure termination in the presence of higher-order references, but unlike our model, do not support cyclic references.

Lorenzen et al. [30] equip OCaml's type system with *modes*, where types are associated with additional modes that describe their uniqueness, affinity, and locality, allowing for safe stack allocation and memory reuse. Similar to our calculus, function types are annotated, but modes rather than kinds. Functions have separate mode from their domain and codomain modes, and the mode of a function type is determined with respect to its context. The system uses a *lock* in the context to infer the required mode of the function, ensuring that, *e.g.,* a function that uses global variables is marked as global. Using a modality for a language based on our semantics would be an interesting avenue for future research. Our initial conjecture is that we could use a similar presentation, where the context is restricted to a certain level (rather than a certain mode). This could be similar to how universes are presented in StraTT [16]. The authors observe that the expressivity of StraTT is restricted compared to MLTT, so there may be also expressivity differences between kinds and modes for predicative references.

Milano et al. [33] present a type system for fearless concurrency, *i.e.,* freedom from destructive races and synchronizing only when threads explicitly communicate. Their type system is for an imperative language without higher-order references, but top-level function types require extra information to facilitate reasoning about affected regions. Function types $(\mathcal{H}; \Gamma) \Rightarrow (\mathcal{H}'; \Gamma'; r, \tau)$ require a context describing the parameters and their regions (Γ), along with the tracking context of these regions ($\mathcal{H}$), and the codomain type describes the effect to these regions in $\mathcal{H}'; \Gamma'$, the final region r, and value codomain type τ. Our function typing rule also relies on the context, but in contrast we record the necessary information with a level annotation on the function type. The authors expose a simpler user-facing syntax for functions based on the principles (1) the programmer should never explicitly mention regions and (2) "good defaults,", which we may also want to consider when designing a practical language from our semantics.

Boruch-Gruszecki et al. [13] present a type system where capabilities are tracked as program variables, *i.e.,* values are characterized by what variables they capture, to model effects. Their presentation is a promising direction for a practical language design for our semantics, as our semantics of functions relies on the universe level of what it captures. Types are annotated with the program variables they capture (capture sets), which delimites the extent of capabilities and can model the extent of effects. In contrast, we implicitly track captured variables by dividing types into universes and level annotations on function types, since the kind of a type can determine its dependency on the heap. However, we do lose precision on the exact locations values depend on in the heap because of our course-grained universes; in contrast, capturing types are precise about the extent of capabilities using capture sets.

Arvidsson et al. [9] use reference capabilities on types to divide a program's heap into independent regions, which allows the programmer to chose memory management strategies for each region of memory. In their region system Reggio, objects within a region can freely refer to each other, similar to $\lambda^{\circ}_{\mathrm{PR}}$ where $\mathbf{Type}_0$ references could be cyclic in $\mathbf{Type}_0$. Types statically enforce region isolation by restricting regions to having one accessible or "bridge" object, and various capabilities distinguish between bridge objects, intra-region objects, temporary objects within a scope, and immutable objects. In contrast, $\lambda^{\circ}_{\mathrm{PR}}$ only has $\mathbf{Type}_0$ intra-universe objects, and in both λ_{PR} and $\lambda^{\circ}_{\mathrm{PR}}$ all universes can have multiple reference objects to universes. When extending our semantics to enable static memory management, we could take inspiration from Reggio and enforce universe isolation by using a single point of reference for each universe.

6 Conclusion

In this paper, we have only begun an exploration into a new design space of references. We separate higher-order references and full-ground references into semantically distinct references by distinguishing dependencies on past and future allocations, which allow the semantics to avoid the type-world circularity. We use type universes to distinguish allocations into regions in the future (higher universes) from the perspective of allocations in the past (lower universes). Type universes allow us to describe *predicative* higher-order references and *acyclic* full-ground references; references that depend on past allocations. However, type universes do not only restrict us to the past! We also explore the future with *cyclic* full-ground references through limited impredicativity. The impredicativity was restricted to one universe to allow references to depend on current (future) allocation. We also observe some strong similarities to region type-and-effect systems, suggesting there might be a connection between type universes and regions, which creates another design space to explore, or that we can take advantage of different allocation patterns for predicative references.

7 Acknowledgments

We have many people to thank; research is a product of many voices, ideas, and interactions with our fellow researchers. We sincerely thank the reviewers from the Higher-Order Programming with Effects (HOPE) workshop associated with ICFP 2023 who reviewed and engaged with our initial idea for this current work. Our talk “One Weird Trick to Untie Landin’s Knot” at the HOPE workshop sparked many meaningful ideas and connections. We would like to especially thank Ohad Kammar who was excited to expand our initial idea presented at HOPE 2023; your excitement was (and still is) infectious. Thank you Ohad, for the many countless hours you spent helping us understand your own categorical model for full-ground references, and the confidence you instilled in the initial ideas of our model. Thank you Chris Casinghino, who also engaged in both the talk and earlier drafts of the work, for your enthusiasm and ideas for future work as they pertain to static memory management. Thank you Nik Swamy, who also attended the HOPE 2023 talk, for your ideas about reasoning about higher-order ghost state in F*, and thank you Sam Lindley, for expressing your own discovered connections to effect handlers, reinforcing confidence in our idea. We feel so grateful and honoured to have so many incredible researchers engaged from an initial idea (where there were no proofs or even a model).

Thank you Ugo Dal Lago, who pointed us to the stratified type-and-effect regions work after a conversation while attending POPL 2024.

Thank you to the reviewers of ESOP 2025; while the work was still in its initial stages, you gave us some ideas for future work and related work like the Universe types line of work.

Thank you Ron Garcia for suggesting the Ernest Hemingway quote to create a great opening line to our paper, and for your overall support over the years.

We would like to extend our greatest thanks to the ICFP 2025 reviewers; your level of engagement in our work was incredible, and your suggestions have vastly improved the quality, correctness, and clarity of our work.

References

[1] Samson Abramsky, Kohei Honda, and Guy McCusker. 1998. A Fully Abstract Game Semantics for General References. In *Symposium on Logic in Computer Science (LICS)*. doi:10.1109/LICS.1998.705669

[2] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra monads for free. In *Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/3009837.3009878

[3] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-Dependent Representation Independence. In *Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/1480881.1480925

[4] Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L3: A Linear Language with Locations. (2007). doi:10.1007/11417170_22

[5] Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State.* Ph. D. Dissertation. http://www.ccs.neu.edu/home/amal/ahmedthesis.pdf

[6] Roberto M. Amadio. 2009. *On Stratified Regions.* doi:10.1007/978-3-642-10672-9_16

[7] Andrew W. Appel and David McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *Transactions on Programming Languages and Systems* (2001). doi:10.1145/504709.504712

[8] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *Symposium on Principles of Programming Languages (POPL)*. ACM. doi:10.1145/1190216.1190235

[9] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. doi:10.1145/3622846

[10] Yuyan Bao, Songlin Jia, Guannan Wei, Oliver Bračevac, and Tiark Rompf. 2023. Modeling Reachability Types with Logical Relations. doi:10.48550/ARXIV.2309.05885

[11] Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. doi:10.1145/3485516

[12] Nick Benton and Benjamin Leperchey. 2005. *Relational Reasoning in a Nominal Semantics for Storage.* doi:10.1007/11417170_8

[13] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. 2023. Capturing Types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2023). doi:10.1145/3618003

[14] Gérard Boudol. 2007. Fair cooperative multithreading: typing termination in a higher-order concurrent imperative language. In *International Conference on Concurrency Theory (CONCUR)*.

[15] Gérard Boudol. 2010. Typing termination in a higher-order concurrent imperative language. *Information and Computation* (2010). doi:10.1016/j.ic.2009.06.007

[16] Jonathan Chan and Stephanie Weirich. 2025. Stratified Type Theory. In *European Symposium on Programming (ESOP)*. doi:10.1007/978-3-031-91118-7_10

[17] Karl Crary. 2005. Logical Relations and a Case Study in Equivalence Checking. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 6.

[18] Dave Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. 2008. *Universe Types for Topology and Encapsulation*. doi:10.1007/978-3-540-92188-2_4

[19] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. 2010. *Termination in Impure Concurrent Languages*. doi:10.1007/978-3-642-15375-4_23

[20] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. 2012. *Strong Normalisation in lambda-Calculi with References*. doi:10.1007/978-3-642-29320-7_9

[21] Werner Dietl and Peter Müller. 2005. Universes: Lightweight Ownership for JML. (2005). https://www.jot.fm/issues/issue_2005_10/article1.pdf

[22] Jean-Yves Girard. 1972. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph. D. Dissertation. Université Paris VII. https://www.worldcat.org/oclc/493768392

[23] Ernest Hemingway. 1926. *The Sun Also Rises*. Scribner.

[24] Kuen-Bang Hou (Favonia), Carlo Angiuli, and Reed Mullanix. 2023. An Order-Theoretic Analysis of Universe Polymorphism. In *Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/3571250

[25] Ohad Kammar, Paul B. Levy, Sean K. Moss, and Sam Staton. 2017. A Monad for Full Ground Reference Cells. In *Symposium on Logic in Computer Science (LICS)*. doi:10.48550/arXiv.1702.04908

[26] P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* (1964). doi:10.1093/comjnl/6.4.308

[27] Paul Blain Levy. 2001. *Call-by-push-value*. Ph. D. Dissertation. Queen Mary University of London, UK.

[28] Paul Blain Levy. 2002. Possible World Semantics for General Storage in Call-By-Value. In *International Workshop on Computer Science Logic (CSL)*. doi:10.1007/3-540-45793-3_16

[29] Paul Blain Levy. 2008. Global State Considered Helpful. *Electronic Notes in Theoretical Computer Science* (2008). doi:10.1016/j.entcs.2008.10.015

[30] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. In *International Conference on Functional Programming (ICFP)*.

[31] Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73, Proceedings of the Logic Colloquium*. Elsevier, 73–118. doi:10.1016/s0049-237x(08)71945-1

[32] Conor McBride. 2011. Crude but Effective Stratification. https://mazzo.li/epilogue/index.html%3Fp=857&cpage=1.html

[33] Mae Milano, Julia Turcotti, and Andrew C. Myers. 2022. A flexible type system for fearless concurrency. In *International Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3519939.3523443

[34] Andrzej S. Murawski and Nikos Tzevelekos. 2012. *Algorithmic Games for Full Ground References*. doi:10.1007/978-3-642-31585-5_30

[35] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press Ltd.

[36] Andrew Pitts and Ian Stark. 1998. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*.

[37] Pierre-Marie Pédrot and Nicolas Tabareau. 2020. The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects. In *Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/3371126

[38] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming (ICFP)*. doi:10.1145/2034773.2034811

[39] The Swift Development Team. [n. d.]. Swift Language Documentation: Strong Reference Cycles for Closures. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/automaticreferencecounting/#Strong-Reference-Cycles-for-Closures

[40] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* (1997). doi:10.1006/inco.1996.2613

[41] Paolo Tranquilli. 2011. Termination of Threads with Shared Memory via Infinitary Choice. (2011). https://hal.science/hal-00573690

[42] Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. In *Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/3632856